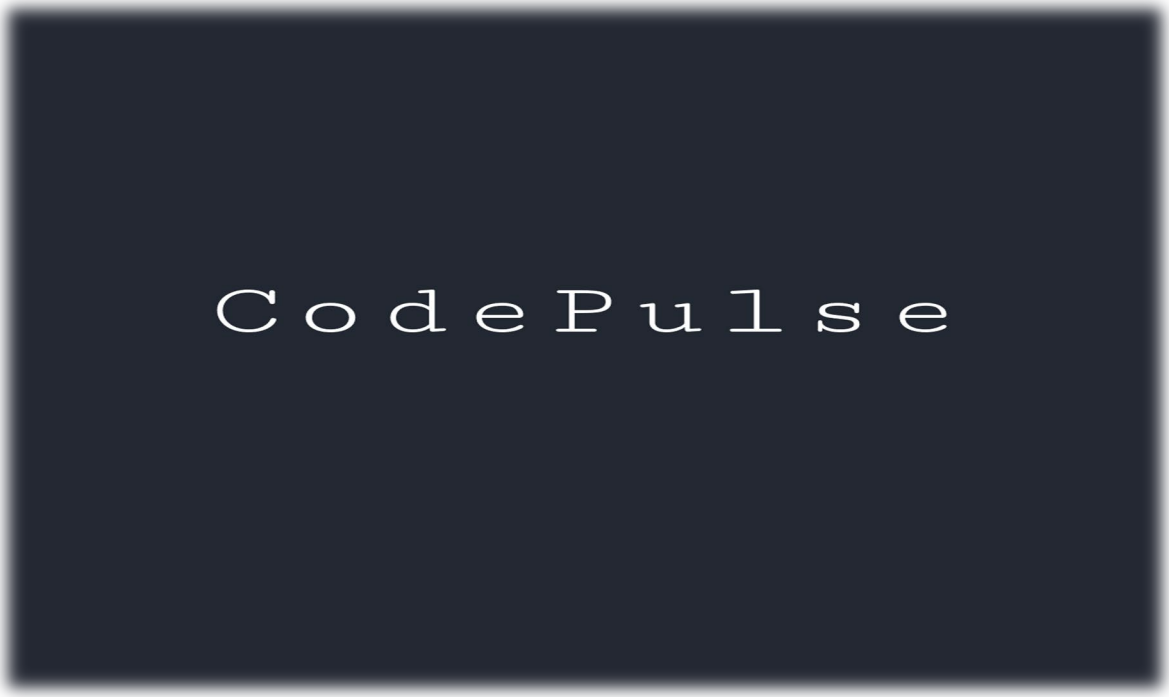


National College of Ireland

Documentation

CodePulse

Computing Project BSHCSD4

Cybersecurity

Djena Siabdellah

20344256

2023-2024

Table of Contents

Executive Summary ***2***

Introduction ***3***

Background **3**

Aims **3**

Technology **4**

Structure **5**

System ***6***

Requirements **6**

Functional Requirements **7**

Use Case Diagram **7**

Requirement 1 : User Registration and Email Verification **9**

Description & Priority **9**

Use Case **9**

Requirement 2 : Secure User Login **11**

Description & Priority **11**

Use Case **12**

Requirement 3 : Vulnerability scanning for URLs and Code **13**

Description & Priority **13**

Use Case **14**

Data Requirements **16**

User Requirements **16**

Environmental Requirements **16**

Usability Requirements **17**

Design & Architecture **17**

Implementation **21**

Graphical User Interface (GUI) **43**

Testing **51**

Evaluation **61**

Conclusions ***63***

Further Development or Research ***63***

References ***64***

Code References ***66***

Appendices ***69***

Project Proposal **69**

Objectives *69*
Background *70*
State of the Art *70*
Technical Approach *71*
Technical Details *71*
Special Resources Required *72*
Project Plan *72*
Testing *73*
Reflective Journals **74**

Executive Summary

The CodePulse project was created to be an advanced, interactive web application security solution that helps developers identify and fix common vulnerabilities like Cross-Site Scripting (XSS) and SQL Injection. Created with the Django framework and SQLite3, CodePulse integrates improved scanning features into a simple user interface, aimed mainly for education and development areas. The speed in which regular security breaches in online applications—which frequently arise from vulnerabilities that are missed—prompted the creation of this project. CodePulse attempts to increase understanding and allow developers for more secure protection of their applications by offering a useful tool that simulates actual online attacks in a safe environment.

The application gives users the ability to test parts of code or URLs in order to identify potential risks. It provides instant feedback and complete report message on vulnerabilities that are discovered. This proactive approach to vulnerability management highlights the value of security in software development lifecycles and helps developers improve their coding standards. In addition, the project was designed to be flexible and scalable, and future developments are anticipated to provide wider security investigations and a wider range of testing features to keep up with the rapid growth of web technology and new security risks.

As a college project, CodePulse also acts as a link between theoretical cybersecurity concepts and real-world implementation, giving experts and students equally a useful tool for investigating and successfully reducing online risks. With continued improvements and developments, CodePulse is expected to evolve into a more complete security solution with automated analytics for enhanced detection abilities and predicting threat assessment, potentially changing the way that sector explores security training and application testing.

Introduction

Background

I conducted my own research on issues in the current cyber security environment in order to come up with ideas for my project that would match the subject matter. When I asked one of my mentors from my former internship at Dell Technologies for guidance on project ideas, he came up with some great ones that caught my attention. I came up with a few intriguing concepts and brainstormed what would be the best idea. The Open Web Application Security Project (OWASP), which outlines important security threats to web applications, provided information that significantly influenced my investigation into the nature of cybersecurity vulnerabilities today. The significance of creating accurate tools that can identify and reduce such risks has been highlighted by this research.

I made a choice to base this project on Django as it is well known for its built-in security features, which significantly reduce the probability of common security issues. With its "batteries-included" approach, users can rapidly and efficiently create safe websites using pre-made components. Django is a great option for constructing security-focused apps because of its reliable documentation, ORM system, and dynamic community. Furthermore, scanning methods and security measures can be implemented effectively by using Django's framework structure, Python's understandable language, and SQLite3 database

The two vulnerabilities that I choose to concentrate on are SQL Injection and Cross-Site Scripting (XSS), which are serious vulnerabilities that are mentioned in OWASP's Top 10 Vulnerabilities list. These vulnerabilities should be the primary focus of any security tool since they are widespread and have a significant potential impact on web application security. XSS vulnerabilities commonly lead to unwanted access to user data by taking advantage of the way browsers parse HTML and JavaScript. On the other hand, SQL Injection attacks use faulty SQL queries to modify backend databases. The project directly deals with some of the most significant and frequent security risks that web developers now encounter by choosing these targeted areas.

The main objective of this project was to improve my knowledge of online safety in addition to learning a new web framework and Python programming. My goal was to create a resource that would help developers improve the security of their apps by creating a tool that scans and indicates vulnerabilities based on the OWASP principles. This project is a useful resource for the developer community as well as a learning experience for me in the area of cybersecurity. I wanted to use my project to help create safer online spaces by giving developers the knowledge and resources that they need to use against these common security threats.

Aims

The main goal of the CodePulse project is to give developers a safe and convenient environment users can manage their authentication and do thorough security scans in a safe and accessible environment because of the use of the Django web framework, which is known for strong security architecture. The aim of this project is to particularly address two

of the most significant threats facing web developers today - SQL Injection and Cross-Site Scripting (XSS). Due to the possibility of revealing private user information and compromising the integrity of application systems, these vulnerabilities create serious risks to online applications. In order to take on these issues directly, I want to assure that CodePulse gives developers the ability to run thorough security analyses of individual lines of code or whole webpages using URLs through the website application itself.

My objective is to make sure that users have access to a range of security features in CodePulse after registering and completing the authentication process. With the use of these implements, given code or URLs will be scanned to find vulnerabilities that may remain unnoticed until they are exposed. The outcomes of these scans are carefully collected into thorough message reports that describe in detail the kind, severity, and possible impact of every vulnerability on the application. These reports also provide developers actual solving instructions, helping them in understanding and putting security best practices into effect. Additionally, my aim is to make sure that CodePulse makes sure to give the user a proper message report of what vulnerabilities whether it's in their code or URL scan.

CodePulse aims to improving web applications' security measures by completely integrating security testing into the development process. The platform's goal is to make sure that security is an essential component of the development lifecycle rather than a secondary concern. This protective approach lowers the risk of security breaches, increases the resilience of applications, and develops a security-aware culture among developers. The project's goal with CodePulse is to enable developers to produce software that is safe, dependable, and capable of handling the changing demands of the cybersecurity.

Technology

My approach of developing the CodePulse project makes use of the powerful features of the Django web framework along with Python as the chosen programming language with some other languages, HTML/CSS, Javascript, jQuery. The choice of using Django was made due to its high degree of transparency for database operations and its integrated support for necessary web application features like session management, user authentication, and templating. Because of these qualities, Django is the best option for creating safe, scalable web apps efficiently. In addition, Python is a strong and approachable language for creating advanced web applications given its clear syntax and common adoption in the programming development. Additionally, I have chosen SQLite3 for the database backend, because its simplicity and it integrates well with Django, allowing minimal configuration difficulties and a simpler setup for my application. I chose it because it is smaller and doesn't require a separate server to run, it's a great option for prototypes and development phases as it makes development and testing easier.

Javascript and jQuery – I used javascript with the help of jQuery to make my web application more interactive by making responsive user interfaces, also dealing with the security scanner and the users interactions. jQuery basically simplifies the event handling, Ajax interactions, and making it easier to get stuff like the asynchronous data loading, and form

validation. In my project I have used jQuery to manage AJAX operations, so it displays the scanner result without the page reloading.

HTML and CSS – I used these for my applications structuring and styling the frontend. I used HTML because it provides the basic framework to host all the pages. While CSS I used to enhance the appearance of these elements. I used the bootstrap frontend framework to use a responsive design for my application. By combining HTML, CSS, and Bootstrap, I made application adaptable for all devices and improve user engagement by making web pages more easier to make responsive

I chose Visual Studio Code (VS Code) as the development environment because of its broad support for Python and Django, which is achieved through plugins and built-in features that improve productivity and simplify the coding process. The web application development process is made considerably easier by this Integrated Development Environment (IDE), which is well known for its advanced debugging tools and user-friendly interface, as I thought this would be a good environment for my project.

CodePulse's primary technology objective is to improve web application security by incorporating vulnerability detection tools right into the development process. The goal of this project is to give developers fast feedback on any security flaws in their code, such as SQL Injection and Cross-Site Scripting (XSS). I intend to apply strict input validation using Django's ORM in order to reduce the possibility of SQL Injection attacks. Additionally, strong techniques for automatically escaping output are provided by Django's template system, avoiding XSS issues.

I will put in place extensive authentication processes to make sure that only authorized users can access key functionality in order to improve the security of the application. This strategy makes use of Django's features to efficiently handle user sessions and permissions, while also following to recommended practices for online security. The use of Python and Django in CodePulse is essential to building a safe framework that not only satisfies the functional needs of developing web applications but also addresses important security issues. Developing, testing, and deploying secure online applications can be made easier by the simple process made possible by the integration of these technologies within the Visual Studio Code environment.

Structure

CodePulse's documentation structure is deliberately created to cover all aspects of the project, from concept to implementation and onward. A broad overview of the project is given in the Introduction section, which provides opening for a more in-depth look at its early days in its background section, which addresses the state of cybersecurity today and the thinking beyond chosen solutions. In the Aims and Objectives section, the specific goals and expected outcomes of the project are detailed, clarifying the choice of security vulnerabilities that the project targets. The use of technology area goes into detail about the frameworks and technologies used for the project, such as Django and SQLite3, and covers their use according to the demands of the project.

The System Architecture section of the document includes an extensive diagram and detailed explanations of how each component of the system interacts, demonstrating the flow from the user interface down to the database operations. This brings us to the Implementation section, which includes full explanations of the implementation procedures, code samples, and the features of various modules. In the Graphical users interface section, there are CodePulse's detailed pages and what each page is expected to do alone with some screen shots of the running website. Testing describes the procedures used to ensure the tool's reliability and efficiency. The Evaluation section next evaluates the tool in relation to user input and planned performance metrics. Structured data is used in this part to demonstrate the tool's performance and flexibility. Conclusions and Future Work, which summarise the project's effects and suggest possible upgrades to improve its features.

System

Requirements

Security Features

To protect user data and interactions, the program incorporates advanced security mechanisms. This involves preventing cross-site request forgery (CSRF) in all of its forms, which is a crucial security precaution for any modern website or application. Additionally, users may check for common online vulnerabilities by using the content scanning functionality included in scanner.html. This adds an extra layer of protection by empowering users to identify and minimize potential security risks in their own inputs.

Session Management

Strong session management is used by the application to maintain user states between pages and browsing sessions. By having this feature, users may browse the program without having to constantly log in. In public or shared computing settings, security features like timeouts and automated logouts after inactivity are essential for safeguarding user data. These features are also included in session management.

Responsive Navigation Menu

The application's responsive navigation menu is essential to the user experience across a range of devices. To make this menu work on PCs, tablets, and mobile phones, extensive HTML, CSS, and JavaScript were used in its implementation. Because the menu is evolving, it can adapt its functionality and style to the viewing device, making it the most user-friendly option and guaranteeing that navigation components are always easily accessible.

CSS and JavaScript Integration

The application makes extensive use of both custom style definitions and external CSS libraries to provide a visually pleasing and consistent appearance throughout all pages. To improve interactive features like form validations, dynamic content modifications without page reloading, and responsive navigation menu, JavaScript is carefully incorporated. The creation of a smooth, user-friendly interface that is both visually appealing and practical depends heavily on these connections.

Functional Requirements

User Registration and Verification

The system has a comprehensive user registration mechanism that enables users to register by completing a detailed application (RegistrationForm). Important data like email address, password, and username are gathered by this form. The creation of a special verification code is one of the security and user verification processes that are included into the registration process. As part of the multi-step verification procedure to make sure the user owns the email account, this code is delivered to the email address provided. The user's account is activated, moving from an inactive to an active state, upon entering the right verification code on the given verification page. This improves security by confirming the user's identity before granting full user access.

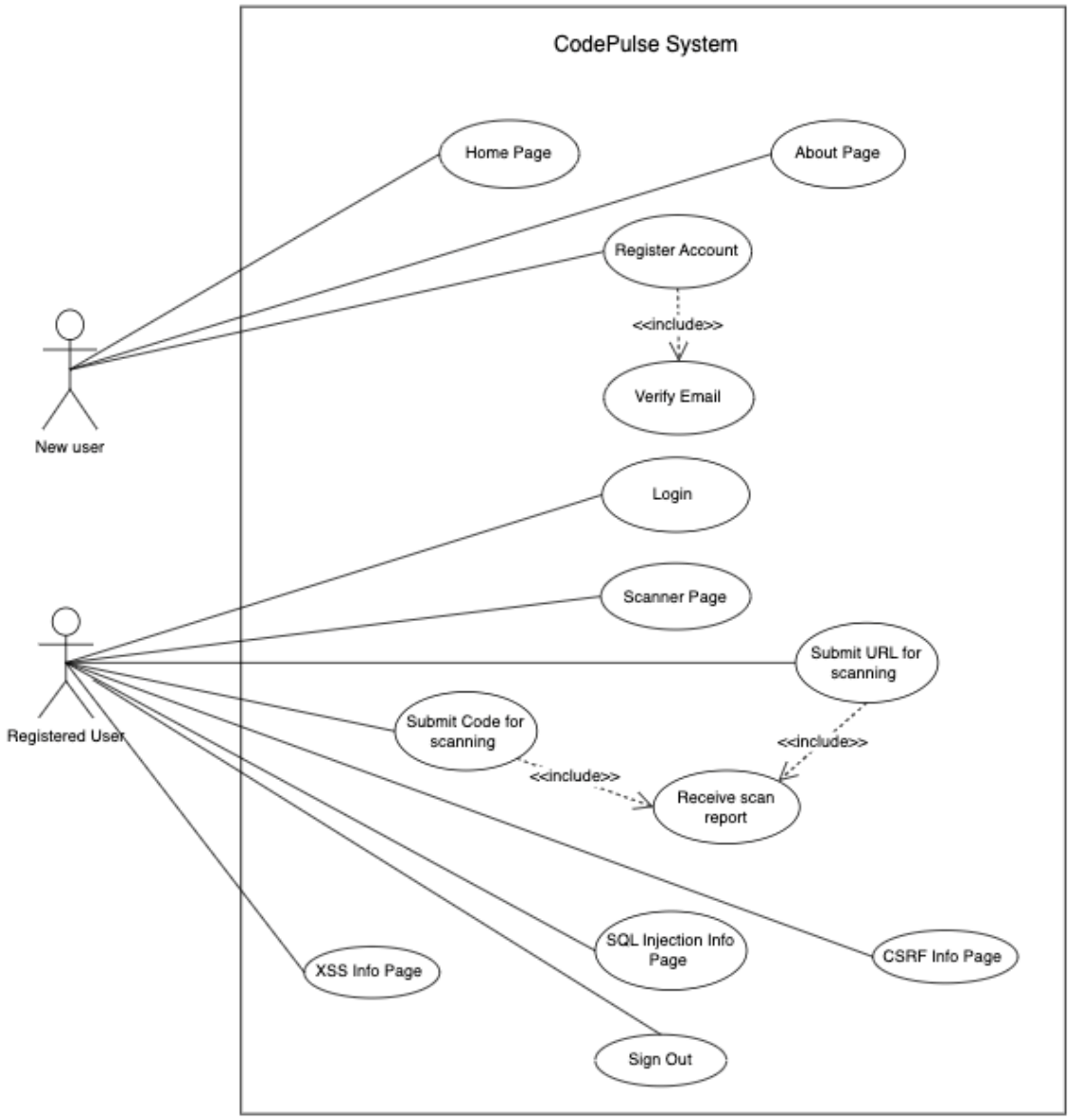
User Authentication

Using the LoginForm, the application provides a safe login method that verifies users' identities using their username and password. The purpose of this form is to thoroughly check user credentials against the database records that have been saved. The system creates a session for the user after successful authentication, allowing access to secure sections of the application along with customized user interfaces.
Based on the server-side logic and user context, the system dynamically helps creates pages like home.html, about.html, and others using Django's reliable templating engine. With the help of this feature, users may have a highly interactive experience where web page content adapts to their choices and actions. The involvement of users and the applicative utility are improved, for instance, when unique greetings, links, and navigation choices are provided based on the user's authentication state.

Vulnerability Scanning

This feature targets SQL injection and Cross-Site Scripting (XSS) threats by allowing users to input URLs and code snippets into a form where they are examined for security flaws. To find potentially dangerous patterns in the incoming data, the system should parse it securely. It is recommended that the system performs checks on URLs to make sure they do not include vulnerable scripts or redirect users to malicious websites. If the system finds patterns in code that might point to SQL injection vulnerabilities or scripts that could be exploited for cross-site scripting attacks (XSS), it should be able to evaluate the code. The system has to clearly display the findings of the review, indicating any vulnerabilities that were discovered or verifying that there were none. For users who need to make sure their own websites are safe from frequent online vulnerabilities, this functionality is essential.

Use Case Diagram



The interactions and functionality that are accessible to two different user types – New Users and Registered Users—are displayed in the CodePulse System use case diagram. From the first user interaction with the website's public pages via user registration and verification to the exclusive features available only to registered users, the figure aims to show the steps required.

Public access pages are used to introduce new users to the CodePulse system. They get access to the Home Page and the About Page, which give a comprehensive rundown of all of the features that CodePulse offers. In the event that they choose to interact further, they have the option to Register an Account, which requires email verification. After completing the Verify Email use case, users can move to Registered User.

The features provided to registered users are more broad. After entering in via the Login use case, users are sent to certain pages like the Scanner Page, which acts as the entry point to the main features of the system. From this point on, users have the option to submit codes

or URLs for scanning. The system processes each submission and produces a comprehensive Scan Report that lists any security flaws discovered. Users may discover possible security concerns with the help of this comprehensive review.

Additionally, Registered Users can access educational resources specific to web security issues. The XSS Info Page, SQL Injection Info Page, and CSRF Info Page provide valuable insights into various types of web vulnerabilities, helping users with knowledge to better understand and mitigate these risks. Finally, users can securely Sign Out of the system.

Requirement 1 : User Registration and Email Verification

The system must allow new users to register by providing essential details and must verify their email addresses to activate their accounts.

Description & Priority

To help to maintain system security and integrity, only verified users must be able to access the program. This is made possible through the user registration and verification procedure. Since it is the initial line of security against illegal access, this procedure is given top attention.

Use Case

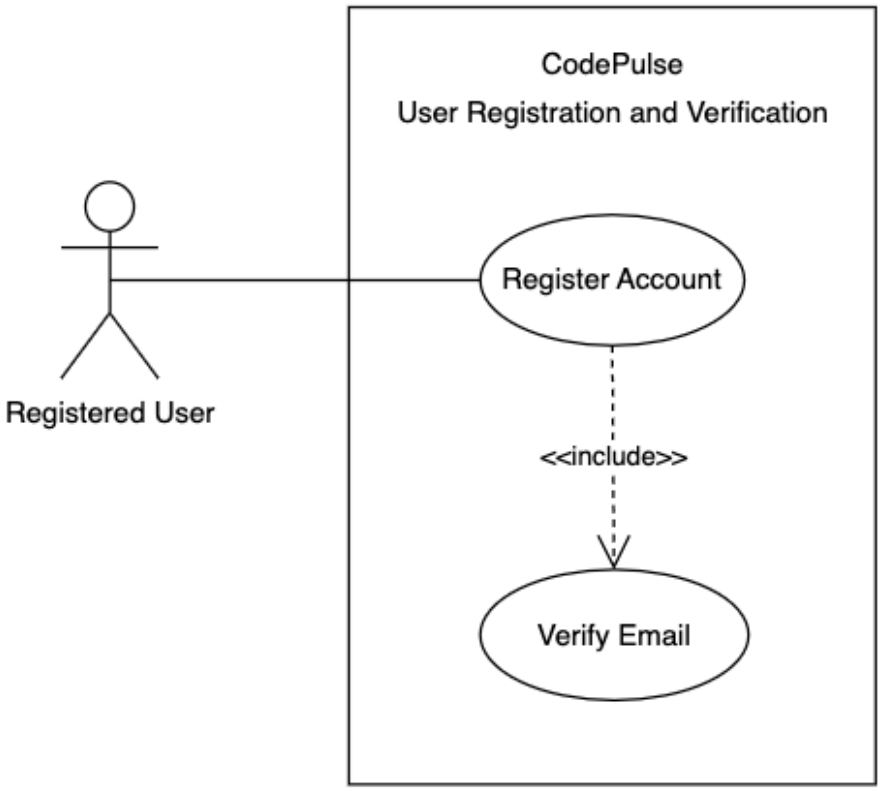
Scope

The scope of this use case includes the collection of data, creation of an account, and email verification.

Description

This use case allows new users to register for an account and verify their email to activate their account.

Use Case Diagram



This use case diagram visually represents the process of user registration and email verification for new users of the system. It includes one actor and two primary use cases connected by an “include” relationship indicating that email verification is an essential part of the registration process.

Actors

New User – This actor represents any individual who wishes to create a new account on the system. The New User initiates the registration process and follows through to email verification.

Use Cases

Register Account – This use case involves the New User entering their personal information such as username, password, and email address into the registration form provided by the system. The primary objective of this use case is to collect and validate user data to create a new, inactive user account pending email verification. The system captures the information, performs validations to ensure data integrity (format of the email, password strength), and creates a new user account in its database. The account remains inactive until the email address is verified.

Verify Email – This use case is an extension of the "Register Account" use case. Once the new account is created, the system automatically sends a verification email to the email address provided by the New User. This email contains a verification code that the user must enter to activate their account. The activation of this use case depends on the successful completion of the "Register Account" use case. The relationship shown in the diagram with an "include" arrow from "Register Account" to "Verify Email," this relationship indicates that email verification is a compulsory action following the account registration.

The user must enter the verification code to confirm their email address. Upon successful verification, the system activates the user account, granting access to the user-specific features.

Flow Description
Precondition
The system is now ready to accept new user registrations to sign up.

Activation
This use case starts when a New User accesses the registration form.

Main flow

1- The system shows the registration form
2- The New User then fills in the registration form and submits it. (See A1)
3- The system validates the input and creates a new, inactive account for the user. (See E1)
4- The system then sends a verification email with a unique code to the new users email address.

Alternate flow
A1- User enters invalid data

1- The system displays an error message to the user.
2- The new user then corrects the data and resubmits the form.
3- The use case continues at position 3 of the main flow (The system validates the input and creates a new account)

Exceptional flow
E1- Email fails to send

4- The system logs the error and displays a message to the user to try again later.
5- The new user retries registration.
6- The use case continues at position 4 of the main flow (The system then sends a verification email with a unique code to the new users email address.)

Termination
The New User clicks the verification code, the system verifies the code, activates the account, and the user is redirected to the main page which is the scanner page.

Post condition
The new user has now an active account and is able to login and access the scanner page.

Requirement 2 : Secure User Login

The system must authenticate users by verifying their credentials against stored data in the database to check of their account is already existing.

Description & Priority

User authentication is essential for access control and security, ensuring that only registered and verified users can access their accounts. This requirement is of high priority due to its role in securing user data and system resources.

Use Case

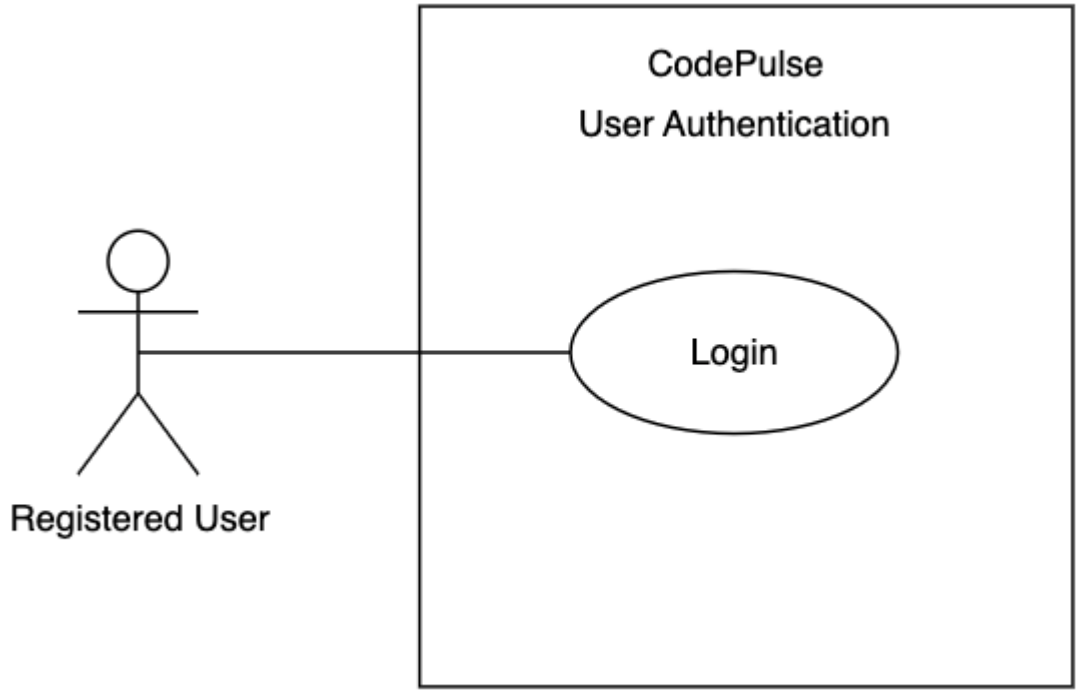
Scope

The scope of this use case is to authenticate the registered users to grant them access to their accounts.

Description

This use case describes the process by which users that log into the system using their username and password.

Use Case Diagram



The use case diagram represents the "User Authentication" process within the CodePulse System. It depicts the interactions between the system and the external actor, the Registered User, specifically detailing the login functionality.

Actors

Registered User – This is a user who has already completed the registration and email verification processes. The Registered User wants to gain access to their account and the system's features by logging into the system.

Use cases

Login – This use case enables the Registered User to enter their credentials (username and password) to access their account. The purpose is to authenticate the user's identity against the system's stored credentials and give or deny access based on the validation outcome. After receiving the credentials, the system verifies them against its database. If the

credentials match, the system grants access to the user, allowing them to interact with various secure features. If the credentials do not match, the system denies access and offer the user options to retry logging in.

Flow Description
Precondition
The user is registered and has verified their email account.

Activation
This use case starts when a registered user accesses the login page to login.

Main flow

1- The system identifies the login form
2- The Registered user then enters their username and password and submits the form. (See A1)
3- The system verifies the credentials the registered user adds against the system. (See E1)
4- If the user is verified, then the system grants them access to the scanner page.

Alternate flow
A1- Incorrect credentials

1- The system displays an error message about incorrect credentials.
2- The Registered User has to re-enters their credentials.
3- The use case continues at position 3 of the main flow (The system verifies the credentials the registered user adds against the system.)

Exceptional flow
E1- Account cannot be accessed

4- The system will reload the page for the user to try again.
5- The user needs to enter the valid credentials for their account.
6- The use case ends without a successful login.

Termination
The user successfully accesses the scanner page.

Post condition
The user is finally logged in and can interact with the system as per their access levels.

Requirement 3 : Vulnerability scanning for URLs and Code

The system must allow users to submit URLs and code snippets for scanning to identify potential security vulnerabilities like XSS and SQL Injection.

Description & Priority

Vulnerability scanning is a core function of the application, critical for detecting and reporting potential security threats in user-submitted URLs and code snippets. This is a high-

priority requirement given its direct impact on the application's value proposition in enhancing cybersecurity.

Use Case

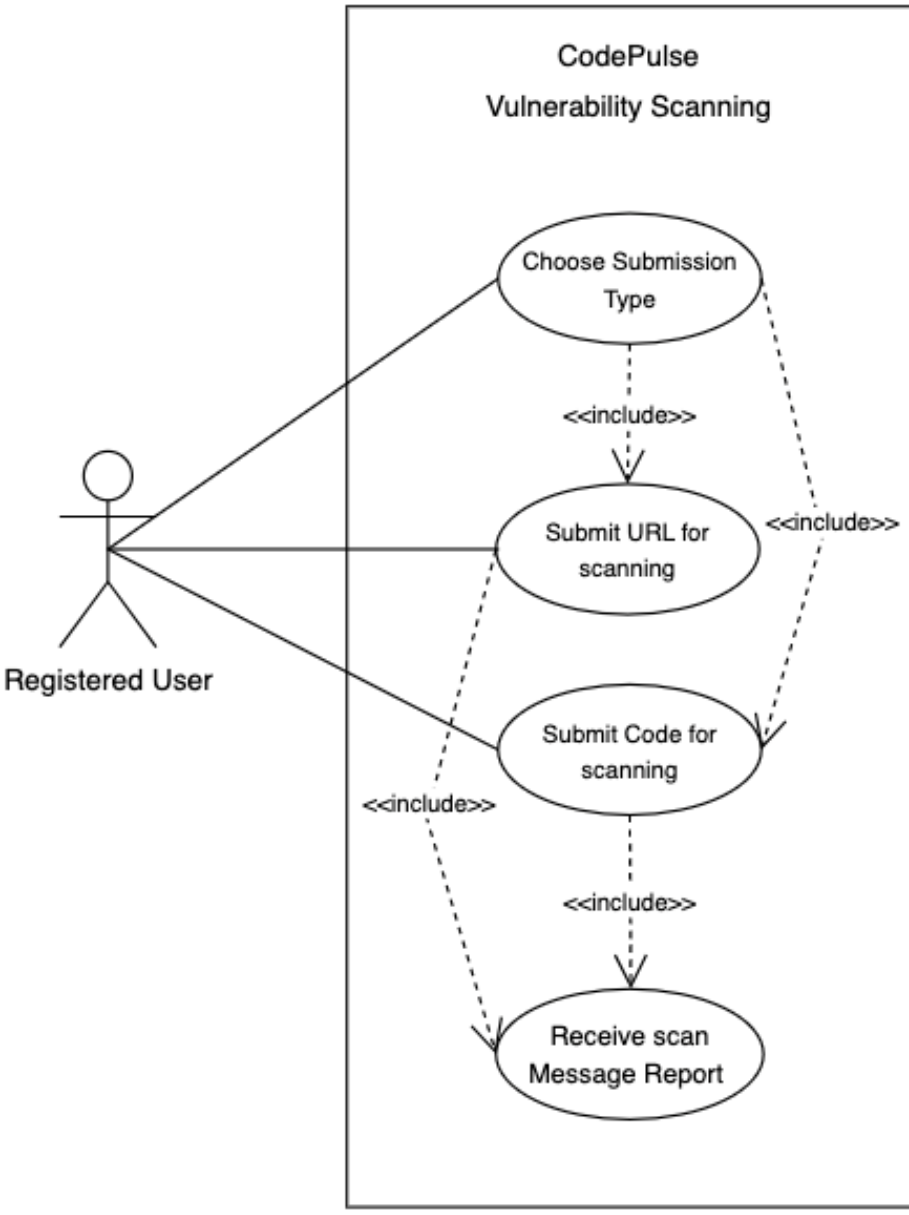
Scope

The scope of this use case is scanning the URLs and code snippets to detect security Vulnerabilities.

Description

This use case describes the scanning of the users submitted data for vulnerabilities.

Use Case Diagram



This use case diagram shows the steps involved in the vulnerability scanning procedure that a Registered User initiates within the CodePulse System. The diagram shows the direct

relationship between selecting a submission type and the next required steps and report output.

Actors

Registered User – An authenticated user of the system, who has the ability to submit data for vulnerability scanning whether it's a URL or Code submission.

Use Cases

Choose Submission Type – The user selects whether to submit a URL or code for vulnerability scanning. This is the starting point for further actions.
Submit URL for Scanning and Submit Code for Scanning – Depending on the user's choice, they proceed to submit either a URL or code. These submissions are necessary steps that follow the user's initial choice.
Receive Scan Message Report – This use case is automatically triggered after a submission is made, generating a detailed report of the scan results. This step is essential for providing the user with feedback on potential vulnerabilities identified during the scan.

Flow Description

Precondition

The users logged in and on to the scanning page to scan their URLs/code.

Activation

This use case starts when the user submits a URL or code snippet for scanning.

Main flow

1- The user enters the URL or code snippet into the system.
2- The system processes the input and performs the scan. (See A1)
3- The system identifies any vulnerabilities and complies a message to the user based on what has been detected. (See E1)
4- The user views the message report detailing the vulnerabilities.

Alternate flow

A1- No vulnerabilities found

1- The system informs the user that no vulnerabilities were detected.
2- The user decides whether to submit more data for scanning.
3- use case ends with the user taking further action or signing out.

Exceptional flow

E1- Scanning error

4- The system encounters an error during the scanning process.
5- The system logs the error and informs the user to try again (if the URL in invalid).
6- The user then has to resubmit the data for scanning (valid URL/code.).

Termination

The user reviews the vulnerability message report.

Post condition

The user resubmits any data (URLs/code) for scanning.

Data Requirements

The CodePulse website's data requirements cover every aspect required for maintenance and functioning of the website's primary features. User data management, including safely storing registration data such usernames, emails, and secure passwords, is an essential for the system. To ensure user validity, verification mechanisms like email verification codes need to be maintained and essentially maintained.

The website also requires that vulnerability scan data is handled with accuracy This includes scan results, and supplied URLs and code stored. Strong encryption and access restrictions are necessary to ensure the data's integrity and confidentiality. The system must also deal with a range of contents inside its educational pages in order to improve user engagement and offer educational value. This calls for a dynamic content management system that administrators can use to update information on vulnerabilities such as XSS, CSRF, SQL Injection, and others vulnerabilities that are added as a new feature every few months for an update.

User Requirements

The features and accessibility that various user types look for from the CodePulse website are outlined in the User Requirements. The registration process should be simple for new users, with step-by-step instructions and quick feedback on requirements like email verification. After registering, users should have no trouble accessing the website, and security features like 2FA should improve account security.

The scanning features should be simple to use for registered users, who can submit URLs and code snippets straight through a simple interface. The outcomes of these scans must to be provided in an understandable and helpful way, outlining any possible vulnerabilities and providing guidance or connections to other resources. Additionally, comprehensive instructional pages that offer insightful information on a variety of web security risks should be accessible to registered users.

Environmental Requirements

The CodePulse website's environmental requirements focus on the operating environments in which the web application has to function. High availability and capacity should be prioritised with the goal to ensure that the website is capable of handling large user loads without experiencing performance decrease. To be useful to a wide range of users, it should function on many platforms and technology, assuring responsive design and compatibility across all main browsers.

Strong security measures should be in place in the hosting environment to stop illegal access and data breaches. To fix new vulnerabilities as they appear, regular security audits and upgrades are necessary. Additionally, the system needs to be scalable in order to change resources in response to user usage and data load, ensuring effective operation both during periods of high demand and low demand.

Usability Requirements

Usability Requirements for the CodePulse website ensure that the site is user-friendly and accessible to users with different levels of technical knowledge and easy to use is the goal of its requirements. To help users in performing security scans and gaining access to educational content, the website should have a clear and user-friendly interface. Accessible menus, regular page layouts, and clear labelling should all be features of simple navigation.

Design & Architecture

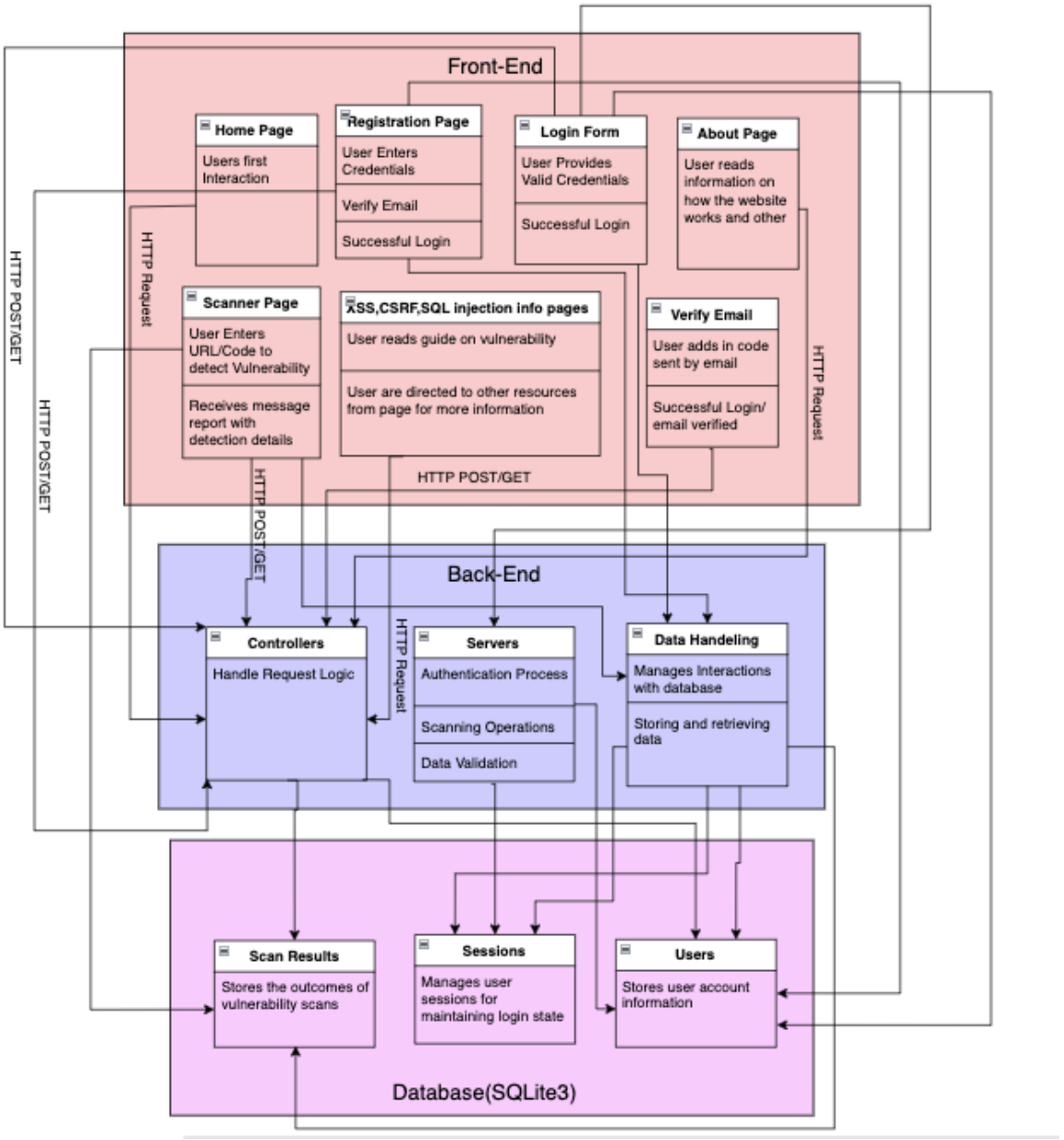


Diagram Description

Front-End Layer

In the Front-End layer there contains multiple of pages, home, about, registration, Login, scanner, verification, and three informational pages (XSS, SQL Injection, CSRF). This diagram shows the user interaction with the application.

Home page – functions as the user's first point of contact and offers navigation options and basic information.

Registration Page – functions by allowing new users to create an account by entering their credentials and verifying their email, this is important to know that they have a valid email.

Login Form – This form allows existing users to access their accounts by entering valid credential's, which they are then are authenticated by the back-end.

About page – Gives users detailed information about how the website operates and other relevant information, informs users about the security measures and technologies used.

Scanner page – important component where users are able to input URLs and codes to detect vulnerabilities, receiving feedback on the detection.

Vulnerability information pages (XSS, SQL Injection, CSRF) – These provide educational content about the vulnerabilities, guiding the users on how to mitigate these risks in their pages.

All of these pages are connected to the back-end through HTTP POST/GET requests, showing that the data submitted and received gets the necessary information web content delivery and user interaction.

Back-End Layer

In the Back-End, the applications logic is being processed.

Controllers – These handle the incoming requests by implementing the websites response logic. It determines what type of response is returned to the front-end based on the users action.

Servers – These are responsible for the authentication process during user login and the overall maintenance of the user sessions, ensuring secure and stable connectivity.

Data Handling – This manages all the interactions with the database, with data storage, retrieval, and validation. This ensures data integrity and security.

Database Layer

In the Database (SQLite3), this is where all the applications information is stored.

Users – This stores the users account information, which supports the registration and login processes.

Sessions – This manages user sessions, keeping the users logged In and maintain their state across different pages of the application.

Scan Results – this captures and keeps the outcome of vulnerability scans. this is important for the further if users want to review their past scans.

Connections

The application's operating flow depends on the connections made between these components. The HTTP POST/GET arrows show how the front-end and back-end interact by allowing data submission through forms (POST) and page retrieval (GET). Based on the logic specified in the controllers, the back-end handles these requests, interacts with the database, and delivers the appropriate responses.

Server and Client Interaction – The program uses a client-server architecture, and to provide flexibility and convenience of testing new features, the Django development server is used throughout the development and testing stages. Through an HTML, CSS, and JavaScript-built web interface, clients communicate with the server. The flexible design of this interface allows it to respond to HTTP requests and present the user with server-side data in a simple and interactive approach.

Backend Logic and Data Handling – Python and the Django framework, which control operations, user application, and session management, are the backend building blocks of CodePulse. The backend of the program handles data processing, control logic implementation, and database interaction to store and retrieve user data, including session information, login credentials, and scan results.

User Authentication and security features – Django's inbuilt user authentication mechanism handles authentication in CodePulse, ensuring that passwords are hashed and not kept in plain text in the database. Django views, which process POST requests with user credentials, provide the login functionality. This function demonstrates how the system authenticates users by comparing their credentials to the database. The user is logged in and taken to the scanner after successful authentication; if not, an error message appears.

For my frontend design I created a responsive and user-friendly design for my user interface using Django templates alongside with HTML, CSS, and JavaScript. I have designed CSS style ensures an appealing and easily navigable user experience on a range of screens and devices. Additionally, I have handled asynchronous requests like real-time vulnerability scanning results by using interactive components like JavaScript and AJAX, which minimize the need to reload the website. This improves efficiency and user involvement.

Components

The data interaction in this project is based on Django models. To make the special requirements of this application better, I have developed a ‘CustomUser’ model that builds on ‘AbstractUser’. Another important model is ‘ScanResult’, which is intended to record vulnerability scan outcomes, including URLs searched for detection time, and vulnerability kind.

The application logic, request processing, and response generation are all handled by the views component. For example, the ‘verify_email’ view handles user email verification using a code given to the user's address, whereas the register view handles user registration.

These views work along with templates and models to provide an overall functionality. Views such as ‘url_scanner’ and scanner, which handle URL content scanning and code snippet scanning for vulnerabilities, respectively, are used to start security scans.
Django uses forms for both data validation and input. Before any data is processed or stored in this application, forms like ‘RegistrationForm’ and ‘UrlForm’ make sure that the information users enter follows the specified security standards and formats. This is essential to keeping harmful data from interfering with the security or functionality of the system.

As for my templates, I’ve created HTML files to create templates that allow dynamic data display based on views' context. It creates a structured and user-friendly display of forms and information is made available by templates such as ‘register.html’ and ‘scanner.html’. Additionally, they include links to CSS files that specify how these pages are visually presented and interact.

Main Algorithms

Cross Site Scripting (XSS Detection)

The application's powerful Cross-Site Scripting (XSS) detection mechanism searches input and retrieved HTML text for possible XSS vulnerabilities. Using regular expression (regex) patterns, this technique finds popular XSS vectors like:

Inline Scripts – An attacker can directly insert malicious JavaScript into web pages by using the regex pattern .*? to identify any inline script elements.
JavaScript URI: Any situation in which a JavaScript URI may be maliciously used to run code when a user interacts with a link or an embedded resource is caught by the pattern javascript:[^\s]*.
Event Handlers – Regular expression (on\w+=['\"]?)(?!http|https)[^\s>]*) is used to find inline event handlers (e.g., onclick, onerror) that may unexpectedly cause JavaScrpt code execution.
Suspicious Attributes – Using (src|href)=['\"]?I (!http|https|\/)[^\s>]*['\"]?This algorithm step looks for features that might be used to launch cross-site scripting (XSS) attacks, especially when such attributes differ from standard, secure URLs.

Every pattern is assessed, and in the event that a possible vulnerability is found, the system classifies the threat's severity and offers recommendations for fixing it. This guidance might involve applying safe coding standards to reduce the risk of XSS, sanitize inputs to remove or escape harmful characters, and putting Content Security Policies (CSP) into place to stop the execution of inline scripts.

SQL Injection Detection

The system's similarly effective SQL Injection detection feature looks for malicious SQL code in user inputs or transactions within the database. It makes use of certain regex patterns to identify common SQL injection techniques:

SQL Injection using tautologies – Identifies patterns such as OR 1=1, which is frequently used in SQL injections to change query logic so that it always returns true, possibly revealing private information.

The insertion of Malicious SQL Code – In order to detect direct attempts to insert SQL code that could alter or obtain data without authority, the pattern (SELECT|INSERT|DELETE|UPDATE).* is used.

Authentication

Authentication and Session Management: To protect user data and ensure that access is only provided to verified users, CodePulse uses advanced algorithms for user authentication, including password hashing and session management.

CSS and Design

My Website application incorporates CSS to improve the user interface and give it an appealing, modern appearance. It has a dark theme, which works especially well for apps that are security-focused. Animated GIFs are used in interactive components and navigation bars to provide a dynamic user experience that increases user engagement and improves interface simplicity. Using responsive design principles, the application is made to work and be available across a range of screens and devices. The navigation bar's integration of media components, such as GIFs, is managed by CSS, which defines both the look and function of these aspects. This improves the application's visual communication of its functions and boosts its aesthetics, all of which contribute to a more engaging user experience.

Implementation

Views

User Registration and Verification

The register function manages user registration. It checks if the request method is POST, then retrieves user details from the form data. It creates a new user using Django's User model but sets 'is_active' to 'False' to prevent user from logging in before email verification. A verification code is generated and sent via the 'send_verification_email' function. Starting with the function definition, 'Request', an object with all of the information of the 'request' is sent to the server, including form data and HTTP headers, this is the only parameter accepted by the function register. Regarding the Method Check The code initially determines whether 'POST' is the requested method. When a user submits data through an HTML form, the 'POST' method is used. To make sure that the function only handles form submissions and not 'GET' or other request types, this check is crucial.

```
# This handels the registration process of users via form submissions.
def register(request):
    # This check if the form was submitted using POST method.
    if request.method == 'POST':
        # This Extract data from form fields submitted by the user.
        username = request.POST.get('username')
        password = request.POST.get('password1')
        email = request.POST.get('email')
        confirm_password = request.POST.get('password2')
```

Using request, the code extracts the email address, password, and username from the 'request.POST.get('name_field')'. If the key is missing, this function returns 'None'. Otherwise, it returns the value from the form data that corresponds to the supplied key. 'password1' in this case refers to the main password field from a registration form.

```
# This validate that all fields contain data.
if not (username and password and email):
    messages.error(request, "Missing fields in the form.")
    return render(request, 'register.html')

# Check if passwords match
if password != confirm_password:
    messages.error(request, "Passwords do not match.")
    return render(request, 'register.html')

# Initialize and apply custom password validator
password_validator = CustomPasswordValidator()
try:
    # This will raise a ValidationError if the password fails any checks
    password_validator.validate(password)
except ValidationError as e:
    messages.error(request, str(e))
    return render(request, 'register.html')
```

This section verifies that all of the form's mandatory fields have been completed. The system creates an error message and reloads the registration page to request the user to complete all needed forms if any of the fields password, email address, or username are missing. This is a simple validation step to make sure that there are no blanks in any of the important fields. This verifies that the password entered matches the confirmation. If not, the registration form is reloaded and an error message indicating that the passwords submitted do not match is sent back to the user using the messaging system. This is essential to prevent user mistakes when several passwords might be accidentally set, affecting future login attempts.

This part of the code establishes an extra security measure by comparing the password to rules that have been specifically set. Most likely, 'CustomPasswordValidator' is a class that verifies that a password satisfies a set of requirements (e.g., length, includes specified characters). The Custom Password Validator has the ability to verify passwords for a number of security factors, including minimum length, digit and symbol inclusion, and more. A 'ValidationError' is raised in the event that the password does not match the requirements specified. The 'except' block is where a 'ValidationError' is caught if it happens. The form is then shown again to provide the user the opportunity to change the password when the error message from the exception is sent to them through the messaging system.

```
# This attempt to create a new user and send a verification email.
try:
    user = User.objects.create_user(username=username, email=email)
    user.set_password(password)  # This ensures that the password is correctly set & Securely set user's password.
    user.is_active = False  # The user will not be active until they verify their email.
    user.save() # This saves the user object in the database.
```

The Django function called 'User.objects.create_user()' is used to create a new user object. This approach is useful since it manages some of the backend tasks required for managing users, such as hashing the password. 'User.is_active = False' indicates that the user's

account is originally set to inactive. This security measure makes sure that the user can't access the account until their email address has been validated.

```
    # This simply generate a random verification code to be sent to the user.
    verification_code = random.randint(100000, 999999)
    request.session['user_id'] = user.id
    request.session['verification_code'] = verification_code

    if send_verification_email(user, verification_code): # This sends the verification email function call.
        messages.success(request, 'Please check your email to confirm your registration.')
        return redirect('verify_email')  # This will then redirect to the email verification page
    else:
        messages.error(request, 'Failed to send verification email. Please try registering again.')
        user.delete()  # this deletes the user if email sending fails
except Exception as e:
    messages.error(request, f'Registration failed: {e}')
    # If not a POST request, just show the registration form.
    return render(request, 'register.html')
```

For my website's Generating Verification Codes and Sending Emails I Used 'random.randint(100000, 999999)', a random integer between 100000 and 999999 is created, and this is used to construct a verification code. The user will receive an email with this code to validate their account. The custom method 'send_verification_email(user, verification_code)', this is supposed to send the user an email with the verification code. The function takes the user to a different page (one where they must enter the verification code) if the email is successfully sent. The user record is erased and the registration form is reloaded so the user may attempt registering again if the email cannot be delivered (the method returns False).

```
else:
    return render(request, 'register.html')
```

The registration form is shown again if the request type is not POST (for example, the user has simply switched to the registration page without submitting the form), or if the form submission fails for any reason and does not meet all requirements. This enables the user to try completing the form once again or to see it when they go to the registration page for the first time.

Email Verification

This view compares a code provided to the user's email address with one that is saved in the session to perform email verification. The user can log in and use the system after successfully validating their email address, which activates their user account. When a user enters the verification code they got in their email upon registration, the function is activated. It compares the code that was submitted with the code that was kept in the session. After successful verification, if the codes match, it labels the user as active, signs them in, and redirects them to scanner page. It gives the user the proper indicator if there is a mismatch between the inputs or if there is any problem.

Starting off the view begins to fetch the data from the request by extracting the code that the user submits through the form. It gets the user ID and verification code that were saved in the session at the user's first registration. This data is important in determining the right user account and guaranteeing the security of the verification procedure.

```
# Retrieve code from user input and session.
user_input_code = request.POST.get('code')
verification_code = request.session.get('verification_code')
user_id = request.session.get('user_id')
```

Statements for logging are given so you can track the process and troubleshoot it if needed. These logs contain the session's user ID and verification code, which are very helpful for fixing email verification problems.

```
# Log available users in database for debugging
users = User.objects.all()
logger.info(f"Available users: {[user.username for user in users]}")
```

This verifies that the code input by the user matches the code that was supplied to them and kept in the session is the main function of this view. In the event that a match is found, the user's account is activated.

```
# This attempts to fetch the user based on the user ID stored in the session.
user = get_object_or_404(User, id=user_id)
# this checks if the input code matches the session code.
if user_input_code == str(verification_code):
```

Using the 'user_id', the method then tries to get the user from the database. A '404' error will be raised if the user does not exist. Next, it determines if the 'verification_code' and the 'user_input_code' match. It then activates the user's account and logs them in if they match.

```
if user_input_code == str(verification_code):
    user.is_active = True #this marks the user as active (successfull email verification)
    user.save() # this saves the user and updates to the database.
    # this logs the user in automatically after email verification.
    login(request, user)
```

The user's 'is_active flag' is set to True after a successful verification, and the database is updated accordingly. The user's account status changes from inactive to active with this modification, making it possible for them to log in and use the system. After a successful verification, the database saves the user's 'is_active' flag as True. The user may now log in and access the system as a result of this important change, which changes their account status from inactive to active. By simplifying the registration to first login process, the view improves user experience by automatically logging in the user following successful verification.

```
    del request.session['verification_code']
    del request.session['user_id']
    # this then notifies the user of successful email verification and redirect to the 'scanner'
    messages.success(request, 'Email verified successfully!')
    return redirect('scanner')
else:
    # If verification codes do not match, render the verification page with an error.
    messages.error(request, 'Invalid verification code.')
    return render(request, 'verify_email.html')
```

The verification code and user ID are removed to tidy up the session when the verification is finished. In order to prevent sensitive data from being in the session longer than is necessary, this step is essential for security. Finally, a success message appears and the user is sent to a the scanner page. The user is directed to try again or double-check their inputs via the relevant error messages if the verification fails.

For the Missing information If any required information is missing, the system prompts the user and redirects as necessary. If No such user exists, Handles cases where the user ID does not correspond to any existing user. Captures and logs unexpected exceptions, providing error feedback to the user.

```
except User.DoesNotExist:
    # this handles the case where the user ID does not correspond to any user in the database.
    messages.error(request, "No such user exists.")
    return redirect('register')
except Exception as e:
    # this catchs all other exceptions and log them, providing a generic error message to the user.
    messages.error(request, f"Error during verification: {e}")
    return render(request, 'verify_email.html')
```

Sending Verification

As a part of the account registration or email verification procedure, users can get an email with a verification code by using the send_verification_email process. The email function sends emails using the Simple Mail Transfer mechanism (SMTP), which is a widely used system for delivering emails over the web. This function is essential to the security and user verification process since it deals directly with the transmission of confidential information that enables users to authenticate themselves. A user object and a verification code are the two parameters required by this method. It initially verifies if the user's email address is active. If not, an error is recorded and False is returned. It creates an email message, establishes a connection with an SMTP server, sends the email, and records the success or failure of this process if the user has a working email address.

```
# this function sends a verification email with SMTP protocol.
def send_verification_email(user, verification_code):
    # this checks if the user object has an email attribute that's not empty
    if not user.email:
        # this logs an error if the user object doesn't have an email address
        logger.error("No email address provided for user.")
        # means the user will exit the function returning False (email could not be sent)
        return False
```

The function first checks if the user object is connected with an email address before attempting to send an email. The function records an error and returns False, meaning that the email could not be sent, if there is no email address. By taking this action, more execution stops, unnecessary processing is prevented, and potential errors are prevented.

```
try:
    # this formats the message string with the verification code included
    message = f'Your verification code is: {verification_code}'
    # this makes a MIMEText object to specify the contents, type, and encoding of the email
    msg = MIMEText(message, 'plain', 'utf-8')
    # sets the subject line of the email
    msg['Subject'] = 'Verify Your Email'
    # sets the sender's email address
    msg['From'] = 'djenasiabdellah26@gmail.com'
    # sets the recipient's email address
    msg['To'] = user.email
```

In this case, the email message is created by the function. It creates a text message with the given verification code using ‘MIMEText’, ensuring compatibility and proper presentation across different email clients by setting the content type to 'plain' and character encoding to 'utf-8'. The sender, receiver, and topic of the email are changed accordingly.

```
# Logging the email details to ensure correctness
logger.info(f"Email details: From: {msg['From']}, To: {msg['To']}")
```

The function logs the email data, including the sender and recipient addresses, for debugging and transparency. Checking sure the email is being delivered appropriately and to the intended recipient with the help of this information is helpful.

```
# this is to set up the SMTP server and establish a connection to the SMTP server at the specified address and port
s = smtplib.SMTP('smtp.gmail.com', 587)
s.starttls()  # Start TLS for security & Encrypt connection for security.
s.login('djenasiabdellah26@gmail.com', 'ssvvycowriavoewj')  # Log into the email server.
s.sendmail(msg['From'], [msg['To']], msg.as_string()) # Send the email.
s.quit() # this is to terminate the connection to the server.
```

The function establishes an SMTP connection on port 587, which is the default for email submission with encryption, using Gmail's SMTP server at 'smtp.gmail.com'. In order to ensure that the user credentials and email content are transferred safely, it initiates TLS (Transport Layer Security) to encrypt the connection. It sends the email after signing in to the server using the given email address and password. Then, in order to conserve resources and uphold security, the SMTP server connection is terminated using 's.quit()'.

```
    # this shows the successful sending of the email
    logger.info(f"Email sent to {user.email} with verification code {verification_code}")
    # this returns True indicating the email was successfully sent
    return True
except Exception as e:
    # this catchs any exceptions during the email sending process and log an error
    logger.error(f"Failed to send email to {user.email}: {e}")
    # this return False indicating that sending the email failed
    return False
```

The function logs this event and returns 'True', signifying success, if the email is sent successfully. An error is caught, reported, and 'False' is returned if it occurs throughout the process, for example, when the email cannot be sent or the SMTP server cannot be reached. This method provides the caller function with strong error handling and feedback, which makes debugging and user notification easier when needed.

Logging in

```
# This handles user login, authenticating credentials against the database.
def user_login(request):
    # This handles user login - checks if the current request is a POST request.
    # This is necessary because sensitive data such as usernames and passwords
    # should be sent via POST requests to ensure they are not visible in the URL.
    if request.method == "POST":
        # this retrieves the username and password from the POST request.
        # these are expected to be provided by a login form where users enter their credentials.
        username = request.POST.get('username')
        password = request.POST.get('password')
        # this Uses Django's built-in `authenticate` method to verify the credentials.
        # If the credentials are valid, it returns a User object. Otherwise, it returns None.
        user = authenticate(username=username, password=password) # Authenticate user.
        if user:
            # the `login` function from Django's auth module is called with the request and User object.
            # this officially logs the user into the system, creating the appropriate session data.
            login(request, user) # Log the user in.
            # After successful login, redirect the user to scanner page.
            #Here it redirects to a page named 'scanner'.
            return redirect('scanner')
        else:
            # If authentication fails, display an error message and redirect back to the login form.
            return render(request, 'login.html', {'error': 'Bad credentials, please try again'}, status=200)
    return render(request, 'login.html')
```

The function first checks whether the request method is POST in the Request Method Check. Form data, including private information like usernames and passwords, is submitted through POST requests. This method corresponds with highest standards for secure data transmission and ensures that this data is not exposed in URLs. When retrieving data, the POST request's provided data is where the login and password are found. In order to prevent a 'KeyError', the 'POST.get()' function properly retrieves data from the POST dictionary and returns None if the key is missing. The method in the authentication process uses 'authenticate()' from Django, which internally verifies the provided credentials against the database. It returns a User object if the credentials match an already-existing user; if not, it returns 'None'.

The user is logged in using 'login()', which controls the session and cookie to save the user's state, if authentication is successful. The function either re-displays the login form with an error message indicating unsuccessful login, or redirects the user to another page representing successful login based on the outcome of the authentication process. Managing Not POST. The function essentially shows the login form if the request method isn't POST, which suggests the user is accessing the login page without submitting the form.

Valid URL

A string representing a URL is supposed to be the only argument accepted by this url function. By using a set of established patterns that are frequently connected to local locations, the function evaluates whether this URL matches them.

```
# this functions is to check if the URL is valid
def is_valid_url(url):
    # this defines a list of regular expression patterns that match URLs considered 'local'.
    # these patterns ensure that the function can identify URLs that are used for local development environments.
    local_patterns = [
        r'^http://localhost',   # this pattern matches URLs that begin with 'http://localhost'
        r'^http://127.0.0.1',   # this pattern matches URLs that begin with 'http://127.0.0.1'
        r'^http://192\.168\.',  # this pattern matches URLs that start with 'http://192.168.'
    ]

    # the `any()` function is used here to iterate over the list of patterns and
    # also applies`re.match` to check if the URL matches any of the patterns.
    # the function `re.match` checks for a match only at the beginning of the string.
    # if any pattern matches the beginning of the URL, `any()` will return True, indicating the URL is a local URL.
    # if no patterns match, it will return False, indicating the URL is not local.
    return any(re.match(pattern, url) for pattern in local_patterns)
```

This functions starts by defining a list of regular expression patterns that represent the URLs associated with the local development environments. Initially, the function defines a collection of regular expression patterns that stand in for the URLs that are usually connected to local development environments.

Addresses that begin with 'http://localhost', which is frequently used to indicate a local server that is only accessible from the host computer. URLs that start with 'http://127.0.0.1', which is the host machine's a loopback address and a common method of accessing the server that is operating on the same machine. URLs that frequently appear on local networks and begin with 'http://192.168'. These IP addresses belong to a range that is set aside for private networks, which are often used in settings for local development. The function compares each pattern to the given URL using Python's 're.match' function within a generator expression. Since it looks for a match just at the beginning of the string, the

re.match function is ideal for this kind of task because it makes sure that the URL must begin with one of the designated patterns in order to be accepted.

URL Scanner

```python
# this defines a function called `url_scanner` that takes a Django request object as an argument.
def url_scanner(request):
    # this checks if the request is a POST method, which is typically used for submitting form data.
    if request.method == 'POST':
        # this retrieves the 'url_input' value from the POST data. The second parameter is the default value if 'url_input' isn't found.
        url = request.POST.get('url_input', '')

        # this calls the `is_valid_url` function to check if the URL meets specific criteria (like being a local URL).
        if not is_valid_url(url):
            # this returns a JSON response with an error message if the URL is invalid, setting the HTTP status to 400 (Bad Request).
            return JsonResponse({'error': 'Invalid URL provided. Please ensure the URL starts with http:// or https://'}, status=400)

        # If this is valid, the it proceeds with fetching the URL and scanning
        html_content = fetch_url(url)
        # this checks if the fetching the content failed, then `html_content` would be None.
        if html_content is None:
            # this returns a JSON response showing that the content could not be fetched, with a 500 (Server Error) status.
            return JsonResponse({'error': 'Failed to fetch content'}, status=500)

        # Perform and calls XSS and SQL Injection vulnerability checks
        xss_vulnerabilities = detect_xss_vulnerability(html_content)
        sql_injection_vulnerabilities = detect_sql_injection(html_content)

        # this compiles the results of the scans into a dictionary.
        results = {
            'XSS': xss_vulnerabilities,
            'SQL Injection': sql_injection_vulnerabilities
        }

        # this checks if no vulnerabilities were detected.
        if not xss_vulnerabilities and not sql_injection_vulnerabilities:
            # this returns a JSON response stating that no vulnerabilities were detected.
            return JsonResponse({'message': 'No vulnerabilities detected.'})

        # If the vulnerabilities were detected, returns a JSON response with the scan results.
        return JsonResponse({'message': 'Scan complete', 'results': results})
    # if the request method is not POST, returns a JSON response indicating the request method is not allowed, with a 405 status.
    return JsonResponse({'error': 'Invalid request method'}, status=405)
```

The main purpose of the ‘url_scanner’ function is to verify a URL, get its content, and check it for common security flaws like SQL Injection and XSS after receiving URL input through a POST request. Applications with a high degree of security, especially those handling sensitive data or operating in settings where security is of the highest priority, need this feature.

The first thing the function does is see if the request coming in is a POST request. This is important because GET queries should not be used to provide sensitive data, such as URLs for scanning, as these requests may be logged or stored in server logs or proxies.
The POST data contains the URL that has to be scanned. For security concerns, if the URL fails ‘is_valid_url's’ initial validation check that verifies it follows typical local development patterns, a JSON response with an error status of 400 is returned right away. The user is informed that the URL they entered is incorrect through this response. After validation is successful, the function uses the fetch_url function to retrieve the HTML content of the URL. In the event that fetching is unsuccessful (for example, because the server is unresponsive or the URL is inaccessible), the method sends a 500 status code along with a JSON response stating that the data was not accessible.

These two functions ‘detect_xss_vulnerability’ and ‘detect_sql_injection’ are used to search the HTML text for vulnerabilities. In order to find any patterns or code snippets that

correspond with known XSS and SQL Injection vulnerabilities, accordingly, these initiatives scan the code. The method creates a JSON response outlining the types of vulnerabilities discovered and any particular information related to each identified issue if vulnerabilities are found. For developers or security analysts using the program, this answer is essential since it offers information about possible security flaws in the scanned URL. The function ensures effective error handling in the event that the request method is not POST by sending a JSON response with a 405 status code, which indicates that the request method is restricted.

```
@login_required
def scanner(request):
    # this views function handles the scanning of code input for XSS and SQL injection vulnerabilities.
    # only allows authenticated users to perform scans.
    # this is to Check if the user is authenticated, proceed only if true.
    if request.user.is_authenticated:
        # The core functionality is executed only when the form data is sent via POST method.
        if request.method == 'POST':
```

The initial section verifies the user's authentication and determines whether the request type is POST, indicating that the user is attempting to send data for scanning.
This method can only be accessed by authorized users due to the '@login_required' annotation. Next, the function determines if the current request is a POST request, which indicates that the user has sent information through the form.

```
# Retrieve the code input from the POST data, defaulting to an empty string if not found.
code_input = request.POST.get('code_input', '')
```

The code input is retrieved from the form in this area. It retrieves the 'code_input' parameter from the POST data using a secure method. It returns an empty string by default if no data is supplied. This line is essential to getting the user-submitted data. Using 'request.If' the 'code_input' key is missing from the POST data dictionary, 'POST.get' provides secure access to the data without running the risk of a 'KeyError'.

```
# use a custom function to detect any XSS vulnerabilities in the code input.
xss_vulnerabilities = detect_xss_vulnerability(code_input)
# use another custom function to detect any SQL injection vulnerabilities in the code input.
sql_injection_vulnerabilities = detect_sql_injection(code_input)
```

To be able to verify whether the code provided has any XSS and SQL Injection vulnerabilities, this section calls custom functions. Here, the user's code is entered into the functions 'detect_xss_vulnerability' and 'detect_sql_injection'. These functions are meant to create lists of problems found by scanning the input for patterns that match to known vulnerabilities.

```
# format messages for XSS vulnerabilities
if xss_vulnerabilities and xss_vulnerabilities[0]['description'] != "No XSS vulnerabilities detected.":
    xss_message = 'Cross Site Scripting Vulnerabilities Detected:\n' + \
                  '\n'.join(f"{vuln['description']} (Severity: {vuln['severity']}) - {vuln['remediation']}"
                            for vuln in xss_vulnerabilities)
    vulnerability_messages.append(xss_message)
elif not xss_vulnerabilities or xss_vulnerabilities[0]['description'] == "No XSS vulnerabilities detected.":
    vulnerability_messages.append("No XSS vulnerabilities detected.")

# SQL Injection vulnerabilities are formatted into a message if any are detected.
# This is important for informing users about the security vul in their SQL queries.
if sql_injection_vulnerabilities and sql_injection_vulnerabilities[0]['description'] != "No SQL Injection vulnerabilities detected.":
    sql_message = 'SQL Injection Vulnerabilities Detected:\n' + \
                  '\n'.join(f"{vuln['description']} (Severity: {vuln['severity']}) - {vuln['remediation']}"
                            for vuln in sql_injection_vulnerabilities)
    vulnerability_messages.append(sql_message)
```

This section shows messages for the user and handles the results of the vulnerability checks. The function creates a list of messages by compiling every relevant vulnerability information. These notifications give the user thorough feedback by outlining the vulnerabilities, their level of severity, and recommended solutions.

```
            # combines all vulnerability messages into a single string to be displayed on the scanner page.
            vulnerability_message = '\n\n'.join(vulnerability_messages)

            # this renders the 'scanner.html' template, passing the username, compiled vulnerability messages.
            return render(request, 'scanner.html', {
                'username': request.user.username,
                'vulnerability_message': vulnerability_message,
                'code_input': code_input
            })
        else:
            # if the request method is not POST, the scanner page is simply re-rendered without any scanning operation.
            return render(request, 'scanner.html', {'username': request.user.username})
    else:
        # Redirects to the login page if the user is not authenticated.
        return redirect('login')
```

In this stage, the user receives the information that was collected and is presented with the scanner.html template containing the vulnerability messages and the code that was initially supplied. The function renders the scanner.html template, supplying the username, vulnerability warnings, and the original code input, if vulnerabilities are identified. It simply reloads the scanner page in the absence of a POST request. The login page is displayed to the user if they are not authenticated.

```
def has_xss_vulnerability(code_input):
    # function to detect basic XSS vulnerabilities
    escaped_code = escape(code_input)
    # this return True if the sanitized code is different from the original, indicating potential XSS vulnerabilities.
    return escaped_code != code_input
```

With user input, the ‘has_xss_vulnerability’ function looks for Cross-Site Scripting (XSS) vulnerabilities. It sanitizes the input by escaping HTML characters using Django's escape function. If the code is different from the original input, this indicates that the original input could have contained components (such as <script> tags or other HTML/JavaScript code) that might be executed in a browser and result in cross-site scripting (XSS) attacks.

```
# logs out the user and redirects them to the home page.
def signout(request):
    # Handles user logout and redirects to home page
    logout(request) # uses logout function to terminate the user session.
    return redirect('home') # after logging out, redirect the user to the home page.
```

The user's sign out process is managed by the ‘signout’ function. To end the current user's session and make sure that any session data is erased and the user is safely logged out. The user is sent to the application's home page after logging out. By guaranteeing that user sessions are correctly ended and avoiding illegal access that may happen if the session remained ongoing, this function improves security.

```
@login_required
def xss_page(request):
    # renders and return the XSS information page to the user.
    return render(request, 'xss.html')

@login_required
def sql_injection_page(request):
    # render and return the SQL Injection information page to the user.
    return render(request, 'sqlinjection.html')

@login_required
def csrf_page(request):
    # render and return the CSRF (Cross-Site Request Forgery) information page to the user.
    return render(request, 'csrf.html')
```

The ‘xss_page’ function renders a template named xss.html. This page has information on cross-site scripting vulnerabilities. By limiting access to this page to just authorized users, the ‘@login_required’ decorator protects sensitive data from unwanted access. The goal of the ‘sql_injection_page’ function, like that of the ‘xss_page’ function, is to show a template that is specific to SQL Injection, a common and serious web security issue. As the same goes to the ‘csrf_page’, renders a html named csrf.html, and have information on the CSRF vulnerabilities.

Models

By extending Django's ‘AbstractUser’ model, the CustomUser model inherits all of its features and functions, like email, password, and username. Any custom user fields you may wish to add in the future will be built on this model.

```
class CustomUser(AbstractUser):
    # Your custom user model code
    pass
    # I Added related_name to avoid clashes with the default User model
    # ManyToManyField was setup for the CustomUser model to associate Groups and Permissions explicitly.
    # im using 'related_name' to prevent clashes.
    groups = models.ManyToManyField(
        "auth.Group",
        related_name="custom_user_set",
        related_query_name="custom_user",
        blank=True,
        help_text="The groups this user belongs to. A user will get all permissions granted to each of their groups.",
    )
    user_permissions = models.ManyToManyField(
        "auth.Permission",
        related_name="custom_user_set",
        related_query_name="custom_user",
        blank=True,
        help_text="Specific permissions for this user.",
    )
```

‘ManyToManyField’ connects to the auth.Group of Django. "custom_user_set" is the related name. The name of the reverse relationship is changed from default ‘user_set’ to ‘custom_user_set’ by this important change. It is used, for example, in group.custom_user_set.all(), to retrieve the user from the group instance. By doing this, name conflicts with any other models that could expand upon the User model by default are avoided.

in the linked query name = "custom_user", Like ‘related_name’, this enables the use of a custom query name to query the connection from the Group model, avoiding conflicts and improving query clarity. ‘blank=True’ shows that this field is not essential. Therefore, admission in any organization is not required of a user. In ‘help_text’, it describes the field and makes its function clear in the Django admin and other forms that use the model.

```python
# ScanResult model stores the results of security scans performed on URLs.
class ScanResult(models.Model):
    # URL field to store the address of the scanned site.
    url = models.URLField()
    # date and time of the scan, set automatically on creation.
    scanned_on = models.DateTimeField(auto_now_add=True)
    # boolean field to indicate if XSS vulnerabilities were detected.
    xss_detected = models.BooleanField(default=False)
    # boolean field to indicate if SQL Injection vulnerabilities were detected.
    sql_injection_detected = models.BooleanField(default=False)
    # boolean field to indicate if CSRF vulnerabilities were detected.
    csrf_issues_detected = models.BooleanField(default=False)
    # for additional info or detailed results
    additional_info = models.TextField(blank=True, null=True)

    def __str__(self):
        # this string representation of the ScanResult that includes the URL and the date of the scan.
        return f"Scan for {self.url} on {self.scanned_on}"
```

The results of security scans performed on URLs are captured by the ScanResult object. In this model, every record is a scan event that stores information about the scan, including the scanned URL, and whether or not vulnerabilities were found. ScanResult's ‘__str__’ function was built to return a string representation containing the URL and the scan date, which facilitates the identification of specific entries in the application's logs or during administrative procedures.

Forms

```python
# this sets the User model to the custom or current active user model.
User = get_user_model()

# Define a registration form class that inherits from UserCreationForm
class RegistrationForm(UserCreationForm):
    # this adds an email field that is required for form submission
    email = forms.EmailField(required=True)

    # this meta class to specifies information about this form class
    class Meta:
        # this is to link this form to the active user model
        model = User
        # fields that are to be included in the form
        fields = ("username", "email", "password1", "password2")

    # this defines the save method to save the form data to the database
    def save(self, commit=True):
        # this calls the superclass's save method with commit=False
        user = super(RegistrationForm, self).save(commit=False)
        # this sets the email of the user model instance
        user.email = self.cleaned_data["email"]
        if commit:
            # this saves the user instance to the database if commit is True
            user.save()
        # this returns the user model instance
        return user
```

The most important part of the user registration process is the RegistrationForm. It expands on the 'UserCreationForm' that comes with Django by including a required email field to collect the user's email address during the registration process. This form was created to work in accordance with an individual user model, making sure that the required data is gathered and verified.

To improve user identification and enable features like password recovery, an email field has been added and created as essential. This makes sure that each user account is linked to a different email address. The Meta Class defines the fields that need to be on the form and links it to the custom user model. It instructs the form to handle particular user object attributes, like username, email, password1 and password2, by doing this. In regards to the save method It creates the user instance without committing it to the database straightaway, enabling additional customisation, like email configuration. The user is saved to the database when all changes are finished and, if commit is set to 'True'.

```python
# this defines a login form class that inherits AuthenticationForm
class LoginForm(AuthenticationForm):
    # Meta class to specify information about this form class
    class Meta:
        model = CustomUser  # this links this form to the CustomUser model
        fields = ['username', 'password']  # these fields are included in the form

# this defines a simple form for code input with a textarea widget
class CodeForm(forms.Form):
    # this defines a form field for code input as a large text area
    code_input = forms.CharField(widget=forms.Textarea)

# Define a form for URL input
class UrlForm(forms.Form):
    # this defines a form field for URL input
    url_input = forms.URLField(label='Enter the URL to scan')
```

The LoginForm makes it easier for users to authenticate. It provides a safe method of comparing user credentials with the database, based on the 'AuthenticationForm'. To be able to simplify form interaction, the Meta Class specifies CustomUser as the model. As a result, the form's fields are restricted to username and password for authentication. By dividing the processing into only the necessary fields, security and clarity are maintained. The Code Input is suitable for contributing code since it uses a 'CharField' and a 'Textarea' widget to manage bigger text inputs. This configuration ensures that the code input is processed as text, preventing execution or interpretation during the form handling process. Input URL implements a 'URLField' that checks if the input is a well-formed URL by default. The application's URL scanning function depends on this parameter to ensure that only valid URLs are handled further.

Utilities

First, the script configures the application's logger. This logger is used throughout the script to record failures and helpful messages, which helps with debugging and tracking the flow and problems found throughout the execution of the application. The purpose of the fetch_url function is to get the HTML content from a specified URL. It sends an HTTP GET request to the URL using the requests library. The text content of the response is returned if the request is successful; if not, an error is logged and None is returned. Collecting online data that will later be examined for vulnerabilities needs the use of this function.

```python
def fetch_url(url):
    try:
        # This attempts to retrieve content from the specified URL
        response = requests.get(url)
        response.raise_for_status()  # This will raise an exception
        # This returns the text content of the fetched URL, HTML or similar web content
        return response.text
    except requests.RequestException as e:
        # this Logs any errors encountered during the fetching process
        logger.error(f"Error fetching URL {url}: {e}")
        # this will return None if an error occurs to signify the fetch was unsuccessful
        return None
```

This function looks for patterns in HTML material that could point to XSS vulnerabilities. It was made possible by my broad research and understanding, and found this to be an

interesting implementation in my code. It makes use of a list of patterns linked to several XSS vulnerability types, each with a corresponding severity and repair solution. The function looks for each pattern in the HTML content and adds the vulnerability information to a list that is returned at the final stage of the function if it finds any. It provides a list stating that no XSS vulnerabilities have been detected if there are none.

```
def detect_xss_vulnerability(html_content):
    # These are the patterns to detect common XSS vectors with severity and remediation -
    patterns = {
        "Inline script": {
            "pattern": r"<script.*?>.*?</script>",
            "severity": "High",
            "remediation": """Event handlers and inline scripts can be blocked using Content Security Policy (CSP). -- Please review the following guide on how
                              '<a href="https://owasp.org/www-community/attacks/xss/" target="_blank">OWASP XSS Guide</a> '
                              'and <a href="https://owasp.org/www-community/attacks/xss/" target="_blank">https://owasp.org/www-community/attacks/xss/</a>"""
        },
        "Javascript pseudo-protocol": {
            "pattern": r"javascript:[^\s]*",
            "severity": "High",
            "remediation": """Sanitize inputs to remove or avoid 'javascript:' protocol. -- Please review the following guide on how to fix XSS vulnerabilities:
                              '<a href="https://owasp.org/www-community/attacks/xss/" target="_blank">OWASP XSS Guide</a> '
                              'and <a href="https://owasp.org/www-community/attacks/xss/" target="_blank">https://owasp.org/www-community/attacks/xss/</a>"""
        },
        "Inline event handlers": {
            "pattern": r"(on\w+=['\"]?)(?!http|https)[^\s>]*",
            "severity": "Medium",
            "remediation": """Remove inline event handlers and use event transfer from JS code. -- Please review the following guide on how to fix XSS vulnerabi
                              '<a href="https://owasp.org/www-community/attacks/xss/" target="_blank">OWASP XSS Guide</a> '
                              'and <a href="https://owasp.org/www-community/attacks/xss/" target="_blank">https://owasp.org/www-community/attacks/xss/</a>"""
        },
        "Suspicious src or href attributes": {
            "pattern": r"(src|href)=['\"]?(?!http|https|\/)[^\s>]*['\"]?",
            "severity": "Medium",
            "remediation": """Make sure that the src or href attributes can only include legitimate, sanitized URLs. -- Please review the following guide on how
                              '<a href="https://owasp.org/www-community/attacks/xss/" target="_blank">OWASP XSS Guide</a> '
                              'and <a href="https://owasp.org/www-community/attacks/xss/" target="_blank">https://owasp.org/www-community/attacks/xss/</a>"""
        },
        "Document cookie access": {
            "pattern": r"document\.cookie",
            "severity": "Medium",
            "remediation": """Restrict and secure cookie access via HTTP headers. -- Please review the following guide on how to fix XSS vulnerabilities:
                              '<a href="https://owasp.org/www-community/attacks/xss/" target="_blank">OWASP XSS Guide</a> '
                              'and <a href="https://owasp.org/www-community/attacks/xss/" target="_blank">https://owasp.org/www-community/attacks/xss/</a>"""
        },
        "Window location manipulation": {
            "pattern": r"window\.location",
            "severity": "Medium",
            "remediation": """Restrict and secure cookie access using HTTP headers.-- Please review the following guide on how to fix XSS vulnerabilities:
                              '<a href="https://owasp.org/www-community/attacks/xss/" target="_blank">OWASP XSS Guide</a> '
                              'and <a href="https://owasp.org/www-community/attacks/xss/" target="_blank">https://owasp.org/www-community/attacks/xss/</a>"""
        },
        "Use of eval()": {
            "pattern": r"eval\s*\(",
            "severity": "High",
            "remediation": """Think about implementing safer alternatives instead of eval() rather than using it. -- Please review the following guide on how t
                              '<a href="https://owasp.org/www-community/attacks/xss/" target="_blank">OWASP XSS Guide</a> '
                              'and <a href="https://owasp.org/www-community/attacks/xss/" target="_blank">https://owasp.org/www-community/attacks/xss/</a>"""
        }
    }
    vulnerabilities = []
```

Detect_sql_injection uses established patterns to search HTML text for SQL Injection vulnerabilities, much like the XSS detection function does. The function looks for certain patterns in the HTML text, and each pattern is linked to a particular kind of SQL Injection danger. Vulnerabilities that are found are noted together with information on their type, severity, and recommended solutions. One of the most major threats to websites is SQL Injection attacks, that can be easily detected due to this function. Developers are able to take precautions to protect their applications by identifying such vulnerabilities.

```
def detect_sql_injection(html_content):
    # define patterns for detecting SQL Injection vulnerabilities
    patterns = {
        "Tautology-based SQL Injection": {
            "pattern": r"OR 1=1",
            "severity": "High",
            "remediation": """For a solution past this issue, use prepared statements or parameterized queries. -- Please review the following guide on how
                                '<a href="https://owasp.org/www-community/attacks/SQL_Injection" target="_blank">OWASP SQL Injection Guide</a> '
                                'and <a href="https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html" target="_bl
        },
        "Malicious SQL code": {
            "pattern": r"(SELECT|INSERT|DELETE|UPDATE) .*",
            "severity": "High",
            "remediation": """To prevent SQL injection, use parameterized queries and appropriate input validation. -- Please review the following guide on
                                '<a href="https://owasp.org/www-community/attacks/SQL_Injection" target="_blank">OWASP SQL Injection Guide</a> '
                                'and <a href="https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html" target="_bl
        }
    }
    vulnerabilities = []
```

Javascript and AJAX

Home Page

For the home page, I decided to use a typewriter effect. The moving typewriter effect on CodePulse's home page grabs the user's attention right away. JavaScript is used to create this effect, which simulates typing and visually displays the goal of the application. In order to simulate someone typing on a typewriter, the script selectively displays characters of a specified text string one at a time.

```
<!-- JavaScript for typewriter effect, dynamically displaying text in a styled paragraph. -->
<script>
    document.addEventListener('DOMContentLoaded', function() {
        const text = "CodePulse is a web vulnerability scanner designed to help both developers and se
        const typewriter = document.getElementById('typewriter');
        let i = 0;

        function typeWriter() {
            if (i < text.length) {
                typewriter.innerHTML += text.charAt(i);
                i++;
                setTimeout(typeWriter, 70); // pace at which characters are added to the typewriter ef
            }
        }

        typeWriter(); // typewriter effect on as soon as the document is fully loaded.
    });
</script>
```

This script creates the illusion of text being written out by delaying the start of the script and then continuously appending characters to a placeholder element. A controlled display of each character is made possible by the use of ‘setTimeout’, which improves both the visual impact and user engagement from the very first interaction.

Scanner Page

AJAX is used in the Scanner page to manage URL and code scans in real time. Through forms, users can submit URLs or code snippets. AJAX handles these contributions on the server, exposing vulnerabilities straight on the page without requiring a page reload.

```
$(document).ready(function() {
    $('#url_scan_form').submit(function(event) {
        /* prevents the form from submitting normally */
        event.preventDefault();
        /* serializes the form data into a URL-encoded string */
        var formData = $(this).serialize();

        $.ajax({
            /* gets the URL to send the data from the form's action attribute */
            url: $(this).attr('action'),
            /* sets method to POST */
            type: "POST",
            /* attachs the serialized data */
            data: formData,
            /* functions to handle a successful response */
            success: function(response) {
                if (response.message === 'No vulnerabilities detected.') {
                    showAlert(response.message, false);  // displays no vulnerabilities message
                } else if (response.results) {
                    var messages = [];
                    if (response.results.XSS.length > 0) { // check for XSS vulnerabilities
                        messages.push('<strong>XSS Vulnerabilities Detected:</strong>');
                        response.results.XSS.forEach(function(vuln) { // loop through XSS vulnerabilities
                            /* formats and adds each vulnerability to messages */
                            messages.push(`${vuln.description} (Severity: ${vuln.severity}) - ${vuln.remediation}`);
                        });
                    } else {
                        messages.push('No XSS Vulnerabilities Detected.'); // adds no vulnerabilities message if none
                    }
```

This AJAX method collects form submissions for URL and code scanning, catches them, forwards the information to a processing script on the server, and refreshes the page with the appropriate error messages or results. It makes sure the user gets feedback on the scan findings right away, which improves the scanner feature's responsiveness and usability.

When a user submits a form by ‘#url_scan_form’, ‘event.preventDefault()’ is used to stop the standard submission procedure by capturing the form submission event using jQuery's ‘submit()’ function. Next, the form data is serialized by encoding the form components as a string that can be sent using jQuery's ‘serialize()’ function. In order to ensure flexibility and minimize implementing, this serialized data is sent to the server using an AJAX POST request, the URL for which it is generated automatically from the form's action element.

```
function showAlert(message, isScanning) {
    var alertBox = document.getElementById('alert-box'); // gets the alert-box element
    alertBox.querySelector('#alert-content').innerHTML = message; // sets the innerHTML of alert-content to the passed message
    alertBox.style.display = 'block'; // makes the alert box visible
    if (isScanning) {
        alertBox.style.backgroundColor = '#8B0000'; // scanning indication
    } else {
        alertBox.style.backgroundColor = '#FF6347'; // error or result display
    }
}
```

The alert box is an essential component of the CodePulse program, especially on the Scanner page, since it improves interaction with the program by giving quick feedback based on the AJAX calls performed during URL and code scanning processes. This warning box is hidden by default ‘(style="display:none;")’ and only shows up when there's an essential information to show, such scan results or error messages. JavaScript controls this dynamic visibility by manipulating the alert box to display success messages or warnings/errors in response to a response from the server. The alert box's display style attribute is used to toggle its visibility, and the inner HTML of a specific paragraph inside it (#alert-content) is set to dynamically fill the alert box's content.

```
<script>
    function getCookie(name) {
        let cookieValue = null;
        if (document.cookie && document.cookie !== '') {
            /* this splits document.cookie at each ';' to get an array of cookies */
            const cookies = document.cookie.split(';');
            for (let i = 0; i < cookies.length; i++) {
                /* trims whitespace from each cookie string */
                const cookie = jQuery.trim(cookies[i]);
                /* checks if the cookie string starts with the name and '='*/
                if (cookie.substring(0, name.length + 1) === (name + '=')) {
                    /* decodes and returns the cookie value */
                    cookieValue = decodeURIComponent(cookie.substring(name.length + 1));
                    break;
                }
            }
        }
        /* returns the cookie value or null if not found */
        return cookieValue;
    }
```

The purpose of the JavaScript function ‘getCookie(name)’ is to get the value of a given cookie by name. This function determines whether the page has any cookies and whether they are filled in. In the event that cookies are present, the cookie string is divided into an array by semicolons since each cookie is saved in the format "key=value;". The trim() function in jQuery is then used to remove any whitespace from each cookie in the array so that the comparison is correct.

After trimming each cookie, the function cycles over them, comparing the string's beginning to the name of the required cookie, followed by an equals sign ‘(name + '=')’. In the event that a match is discovered, the system exits the loop and returns the value after decoding the cookie's value using ‘decodeURIComponent’ to handle any special characters or URL-encoded symbols. The method returns null, that is, the cookie is not present, if no cookie with the given name matches.

HTML Templates

About page

CodePulse's About page was specifically created to provide users with an eye-catching and educational experience. The website has an eye-catching fixed backdrop picture that doesn't move while the user scrolls, creating a depth illusion that makes the user more engaged. This is accomplished by using CSS attributes to make sure the picture is centred and fills the full backdrop. In order to centre information both vertically and horizontally and make the website responsive and visually appealing across a range of devices, flexbox is widely used.

```
body, html {
    min-height: 100vh; /* minimum height of the viewport height, ensuring at least full screen height.
    width: 100%; /* full width of the viewport. */
    background: url('https://i.gifer.com/7Ntk.gif') center no-repeat; /* Background image centered and
    background-size: cover; /* ensures the background covers the entire element. */
    background-attachment: fixed; /* keeps the background fixed during scrolling. */
    display: flex; /* uses Flexbox layout to organize children elements. */
    flex-direction: column; /* stacks child elements vertically. */
    justify-content: center; /* centers children vertically in the container. */
    align-items: center; /* centers children horizontally in the container. */
    color: #fff; /* sets text color to white. */
    padding: 0 8%; /* padding around the content, 0 on top/bottom and 8% on left/right. */
    font-family: sans-serif; /* uses sans-serif typeface for text. */
}
```

Most of CodePulse pages have a basic design that establishes an overall visual idea for the program as a whole. This common design frequently consists of a black backdrop, white text for sharp contrast, and a contemporary sans-serif font Roboto is a popular choice because of its professional look and readability. The same background colour helps the text and interactive features stand out, which is important for keeping the user's attention and making navigating easier. Disabling horizontal scrolling on all pages guarantees that the layout stays clear and simple, which is especially crucial for readability and consistency in the user interface.

Home Page & About Page

The full-screen backdrop visuals on the Home and About pages are clear and moving. As the user scrolls across the information, these fixed backgrounds offer depth using a parallax scrolling effect. The background pictures will have to fill the viewport completely and without distortion due to the CSS background-size: cover property.

```
/* this is a container for animated background and centered content, creating a visually appealing fi
.pulse {
    min-height: 100vh; /* container is at least as tall as the viewport. */
    width: 100%; /* spans the full width of the viewport. */
    background-color: #000; /* background color to black. */
    background-image: url('https://i.gifer.com/7Ntk.gif'); /* set a dynamic GIF as the background. */
    background-position: center; /* centers the background image in the container. */
    background-repeat: no-repeat; /* prevents the background image from tiling. */
    background-size: cover; /* background image covers the entire area of the container. */
    color: #fff; /* text color to white for contrast against the dark background. */
    padding: 8 8%; /* padding to the container to avoid text touching the edges. */
    display: flex; /* Flexbox to provide a flexible container layout. */
    flex-direction: column; /* elements vertically. */
    justify-content: center; /* content vertically within the container. */
    align-items: center; /* content horizontally within the container. */
    position: fixed; /* Fixes the position of the container, so it does not move on scroll. */
}
```

These pages' content sections have semi-transparent backgrounds created with RGBA colour values, letting the background graphics show through discreetly. This design decision not only integrates the visual components but also improves readability of the text without taking away from the visually appealing backdrop.

Login & Registration Forms

Forms on the Login and Registration pages are key to user interaction, requiring responsive design and easily understood input fields. These shapes are contained in containers that use backdrop filters to gently modify the background in order to draw attention to the forms themselves.

```css
/* Container for forms with styling for background and layout */
.forms-container {
    display: flex; /* Flex container to organize forms */
    flex-direction: column; /* Stack forms vertically */
    align-items: center; /* Center forms horizontally */
    width: 300px; /* Fixed width for forms */
    background-color: rgba(255, 255, 255, 0.13); /* Semi-transparent backgrou
    background-position: center; /* Center the background */
    background-size: 100%; /* Background covers entire element */
    border-radius: 10px; /* Rounded corners for aesthetic */
    backdrop-filter: blur(10px); /* Blur effect for background styling */
    border: 2px solid rgba(255, 255, 255, 0.1); /* Subtle border */
    box-shadow: 0 0 40px rgba(8, 7, 16, 0.6); /* Shadow for depth */
    padding: 50px 20px; /* Internal padding */
}

/* Universal styling for form elements */
form * {
    color: #ffffff; /* White text for consistency */
}
```

The CodePulse application's login and registration pages feature ‘a.forms-container’ design that is a prime example of a contemporary and user-friendly style. The forms appear in a neat vertical stack due to the use of CSS flexbox features ensuring a straightforward user flow. With a constant width of 300 pixels, each form element is centred horizontally, giving the user a consistent and targeted area to interact with. A backdrop filter that produces a frosted glass look adds a bit of improvement while the semi-transparent background with a light white tint provides depth while keeping a sleek, modern style. The form's rounded corners reduce the interface's strong visual effect.

The form is well-framed and appears to be floating with its simple border and visible box shadow, which help lead the user's attention to the input fields. By making the form both aesthetically pleasing and easily accessible, this design not only increases the visual appeal but also supports a seamless and simple user experience while registering or login on the website.

Scanner page & Info Pages

The Scanner page, as well as the information pages on XSS, CSRF, and SQL Injection, are made with functionality and content delivery in focus. To improve readability and focus, the

major important is centred and narrowed in width. This is particularly important for educational information, which must be easily comprehended and free of interruptions.

```
main {
    padding-top: 60px; /* Add space at the top */
    min-height: 100vh;
    display: flex;
    flex-direction: column;
    align-items: center;
    background-color: #000; /* Dark background for main area */
    background-size: center;
    color: white;
}

section {
    width: 100%; /* Full width of its parent */
    max-width: 600px; /* Maximum width */
    display: flex;
    flex-direction: column;
    align-items: center;
    margin-top: 20px; /* Space from top */
}
```

The content is easier to read since there is enough space between each segment and separate sections of text on these pages. Improved learning and user experience are promoted by the constant use of text colours and backgrounds, which guarantee that the user's attention is not distracted.

Password Validator

To protect user accounts, Django's CustomPasswordValidator class enforces strict password policies. In order to do thorough tests, this validator makes sure passwords are at least eight characters long, contain one uppercase letter, and one special character from a specified list ‘(like!@#$%^&*(),.?":{}|<>)’. Regular expressions and Python's built-in functions are used by each condition to verify the input. The method generates a ValidationError with a detailed message outlining the defect if a password fails any of these tests. Additionally, the class has a ‘get_help_text’ function that offers a clear explanation of the password requirements. This helps users and developers both understand the application's password regulations. This implementation follows to cybersecurity best practices by strengthening security and helping users with creating strong passwords.

```python
# this imports ValidationError for raising exceptions during validation
from django.core.exceptions import ValidationError
# this import re module for regular expression operations
import re

class CustomPasswordValidator:
    # validate the given password
    def validate(self, password, user=None):
        # checks if the password length is less than 8 characters
        if len(password) < 8:
            raise ValidationError("Password must be at least 8 characters long.")
        # checks for the presence of at least one uppercase letter in the password
        if not re.findall('[A-Z]', password):
            raise ValidationError("Password must contain at least one uppercase letter.")
        # checks for the presence of at least one special character in the password
        if not re.findall('[!@#$%^&*(),.?":{}|<>]', password):
            raise ValidationError("Password must contain at least one special character.")

    # provide a description of the password requirements
    def get_help_text(self):
        return "Your password must contain at least 8 characters, including an uppercase letter and one s
```

Admin

The admin.py file of Django's 'CustomUserAdmin' configuration improves user account management by displaying crucial user properties right within the Django admin interface. Administrators may easily evaluate important user data by using this class customisation, which shows fields like username, email, and 'is_active' in the admin list view. Furthermore, user searches by email and username are supported, making it easier to navigate through quite large user lists. Also, fast user management and sorting according to their function in the company and account status is made possible by the addition of list filters for the 'is_active' and 'is_staff' statuses. It greatly improves administrative operations, maintains good user management practices inside Django applications, and increases interface usability by registering this setting with the Django admin site.

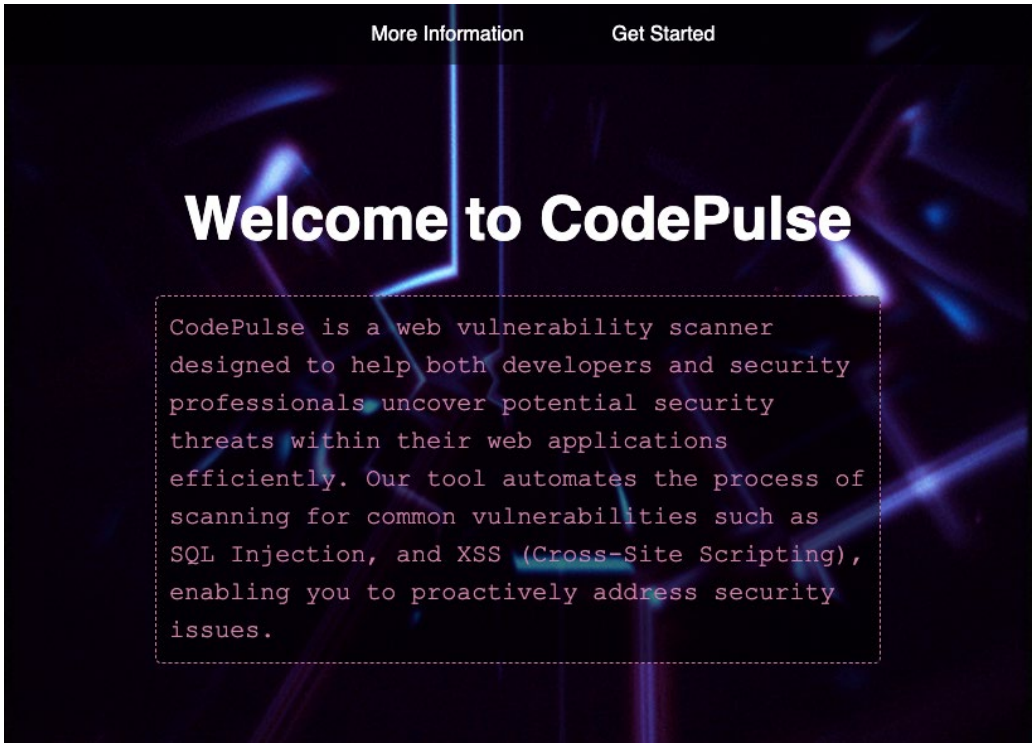
```python
# admin module to manage the admin interface
from django.contrib import admin
# CustomUser model from the local models.py file
from .models import CustomUser

# custom admin class to manage CustomUser instances in the Django admin
class CustomUserAdmin(admin.ModelAdmin):
    # specifies fields to be displayed in the admin list view
    list_display = ('username', 'email', 'is_active')  # Fields displayed in the user list
    # fields that can be searched in the admin list view
    search_fields = ('username', 'email')
    # specifies filters to be available in the admin list view
    list_filter = ('is_active', 'is_staff')  # Filters available for quick sorting

# register the CustomUser model with the CustomUserAdmin to enable custom admin features
admin.site.register(CustomUser, CustomUserAdmin)
```

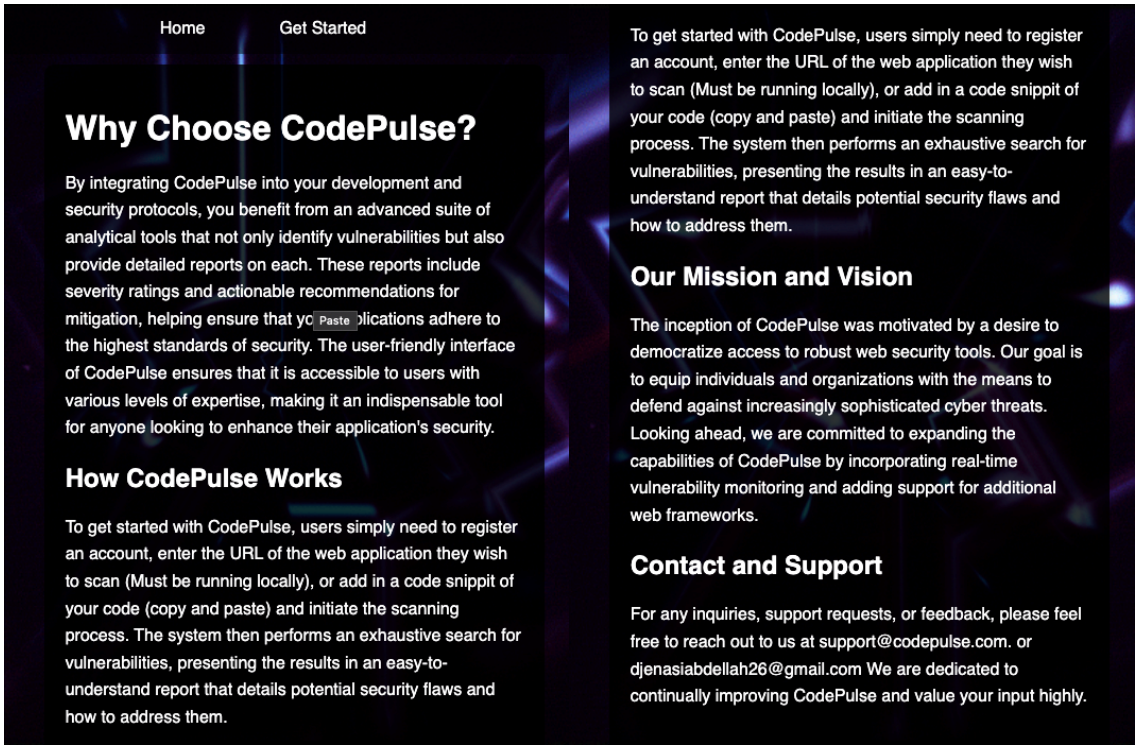
Graphical User Interface (GUI)

Home page

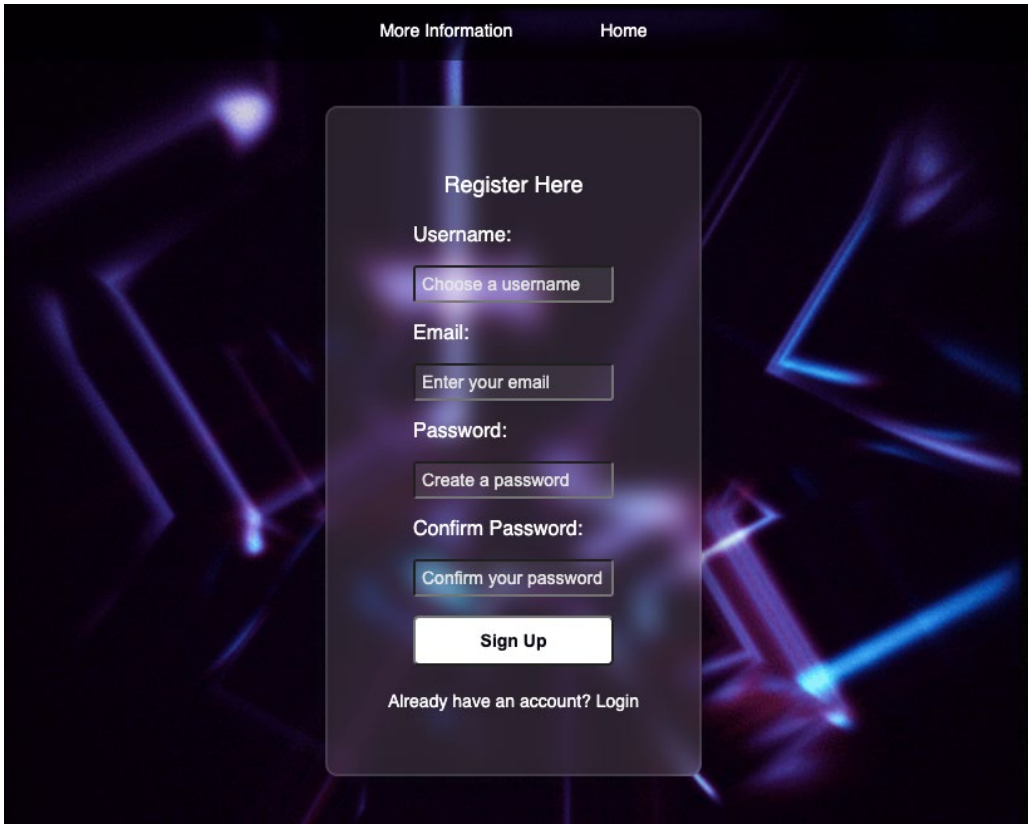


When a person visits this page, their initial view is this home page. This page includes a get started section, which is the about page where users can learn more about our website and get an idea of the purpose of my website application. Additionally, there is a get started page that directs users to the registration page, where they may register and log in.

About Page

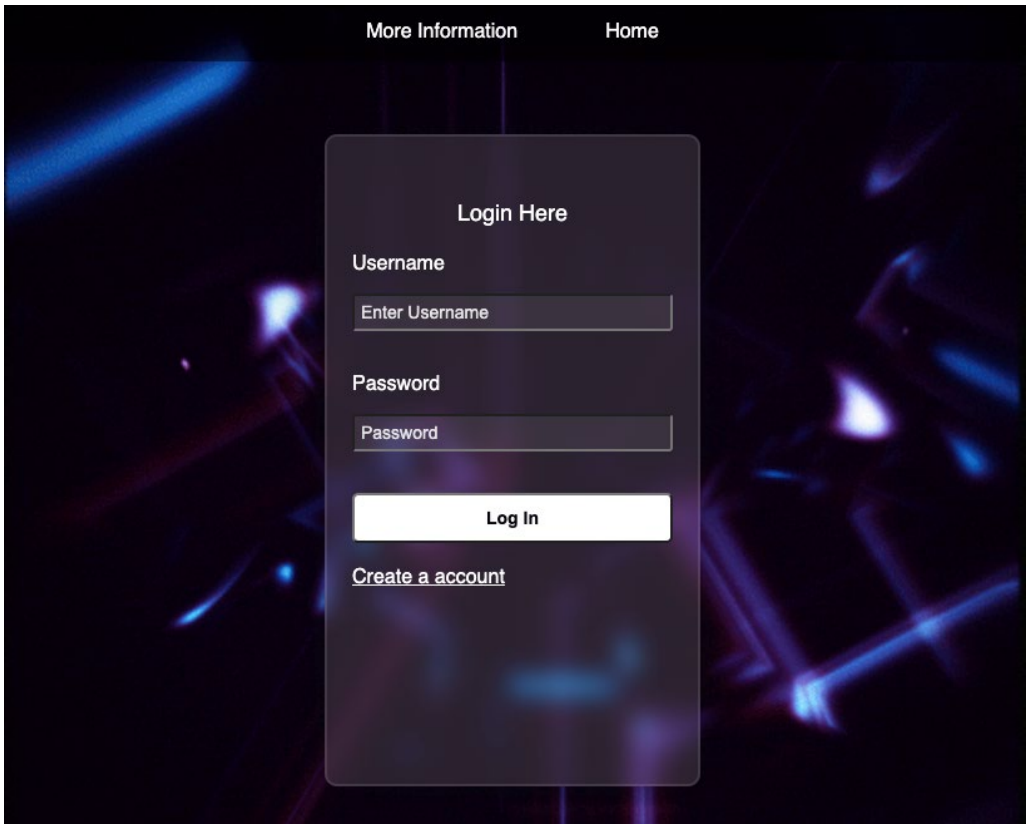


In the about page, the users are able to see what this website application is all about, how it works, what our mission and vision is, why they should choose to work with CodePulse and more. In addition to providing users with information on the main features and advantages of using CodePulse, this page confirms the application's commitment to security and quality in vulnerability detection.



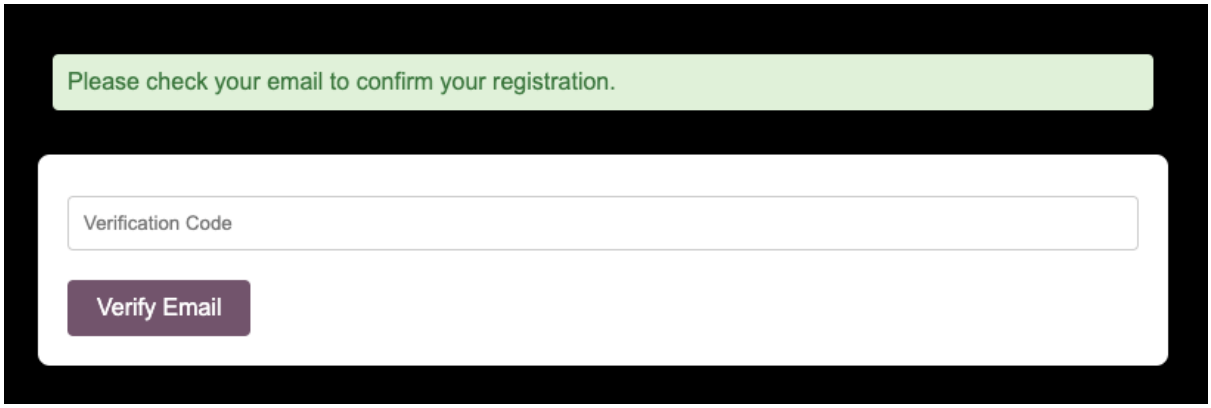
Registration page

The user can register for a CodePulse account on this page, however they must use their email address because the sign-up process requires two-factor authentication and email verification (Gmail). The user must fill in all the required areas above with their login information, being careful to include all system requirements (such as using a valid email address, ensuring that passwords match, and requiring at least eight characters in length in addition to one capital letter and one special key). The user may get further information about the web application on the about page or return to the home page, which serves as the program's first page of interaction, using the navigation bar. In essence, the user may click the "already have an account?" option at the bottom if they already have an account. this points the user in the direction of the login page so they may use an existing account to log in. The background is a GIF so the user can see movement in the background, I thought this would be really cool as a background making this aesthetically pleasing to the users eyes.



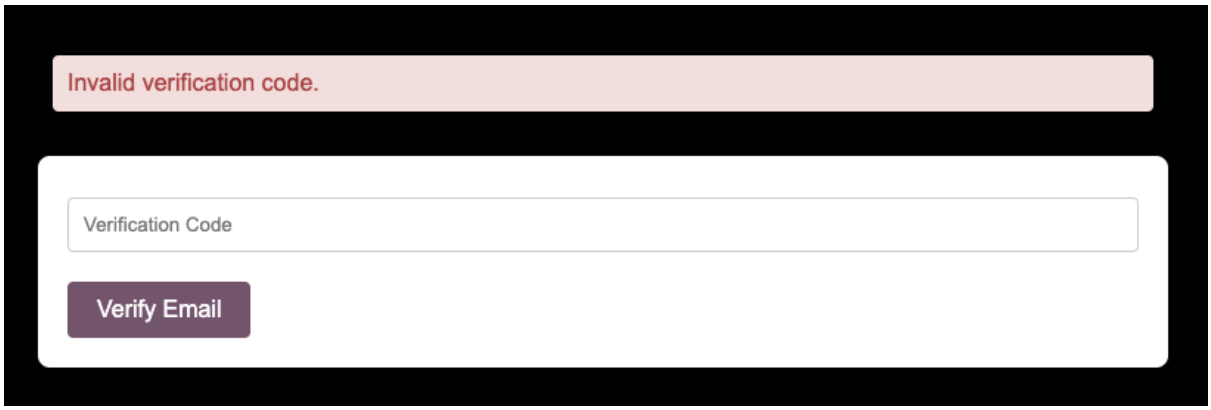
Login page

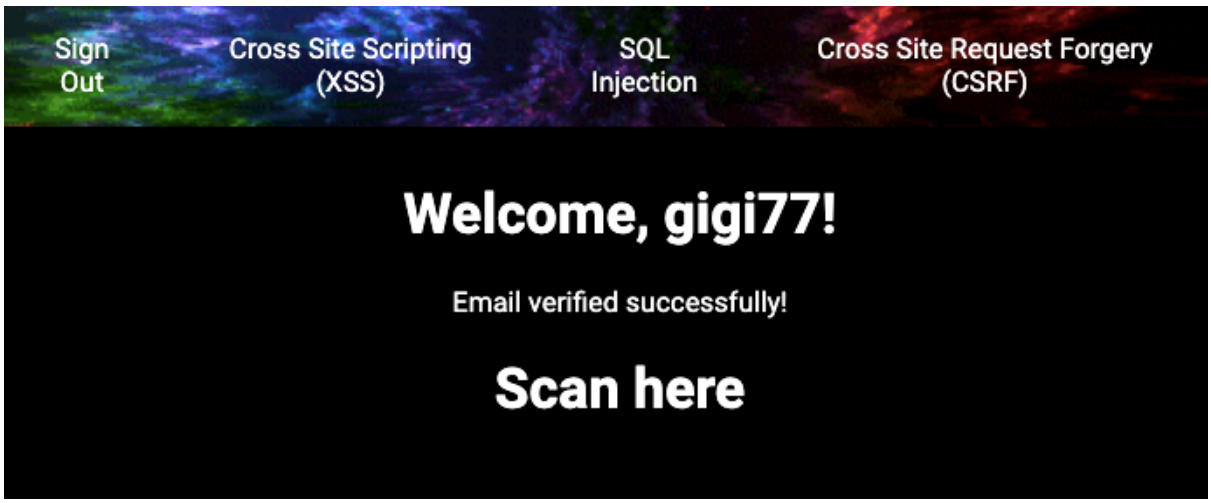
This is the login screen. If the user already has an account, they may use it to log in; if not, they must register in order to use the scanner. If the user clicks the bottom link, they will be sent to the registration page where they can create an account. The user will be sent to the scanner page after entering the required data.



Verification Page

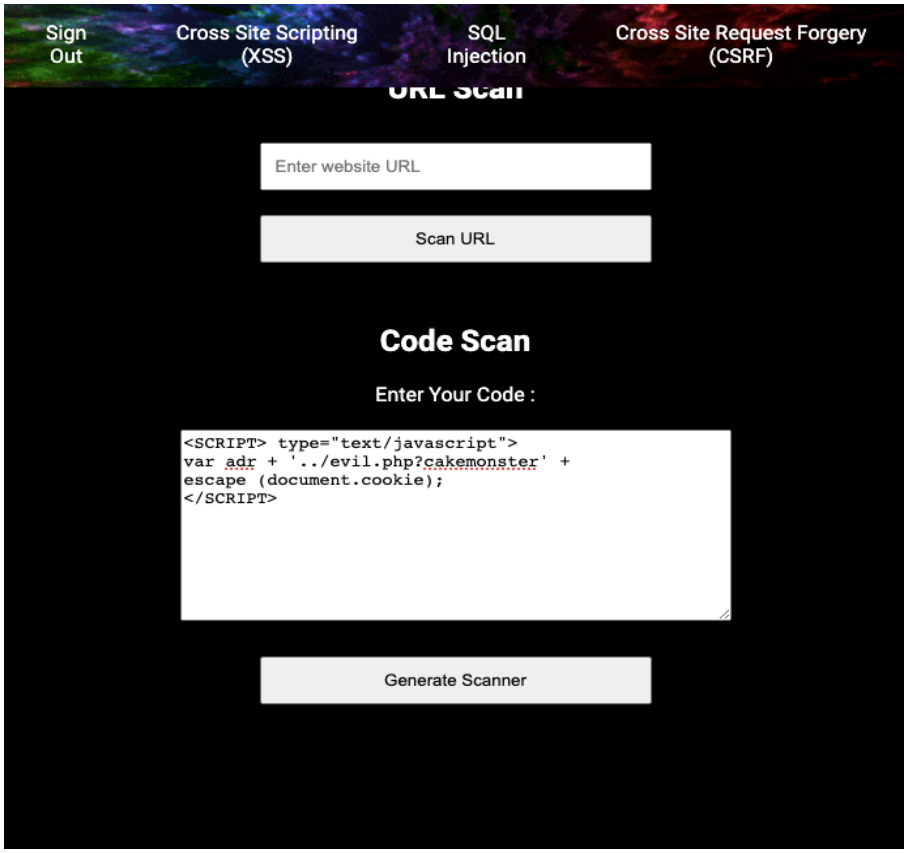
The user must input the code provided to their email on this verification page; it is a six-digit code that is generated at random. The user must complete this step in order to proceed with the email verification and account creation; if not, they will need to return to the registration page and start the process again from scratch. The user will see an invalid verification code, as seen below, if they enter an incorrect code.





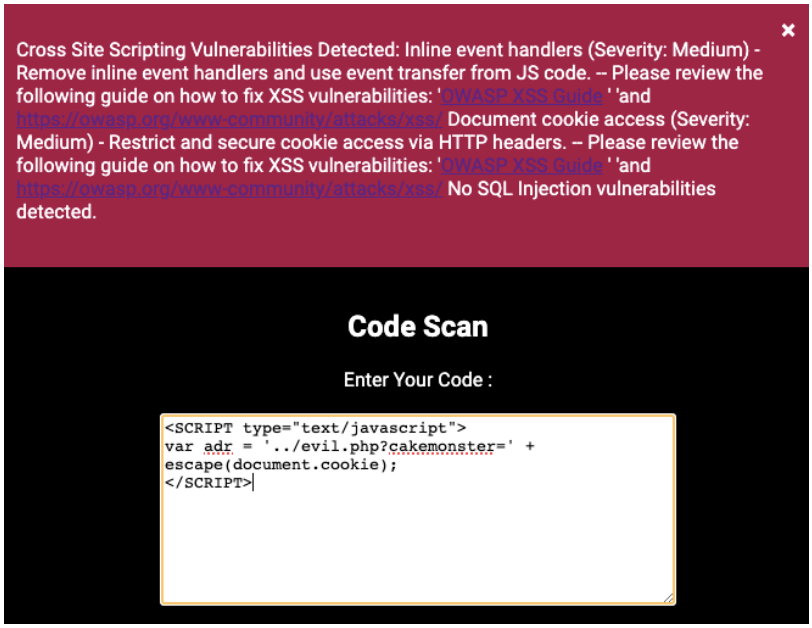
Scanner page after Verification

The user should see this after logging in with their email and being sent to the scanner page, which is the main page. Now that the user has access to the scanner URL, they may use Code Scan to find any vulnerabilities in their code or URL. As soon as the user enters the website, they will be able to view their username as seen above. I used a GIF backdrop for the navigation bar because it is visually appealing to users. I purposefully avoided using any design on the scanner page since it would distract users from trying to scan their code.

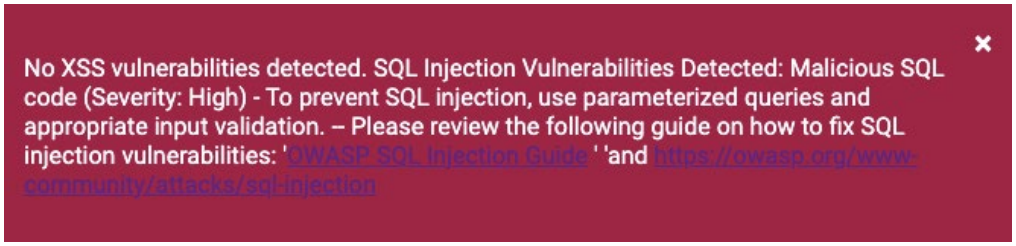


Scanner page Code scan

The user can test if their code is susceptible to SQL Injection or XSS by entering a piece of code on this page. This is an illustration of how my code scanner will find the xss present in the code above. The user should be able to see a report message detailing their vulnerability detection and potential solutions after scanning for this code. This report message should include resources such as the OSWAP websites, which are updated every three months due to the ongoing increase in security vulnerabilities.



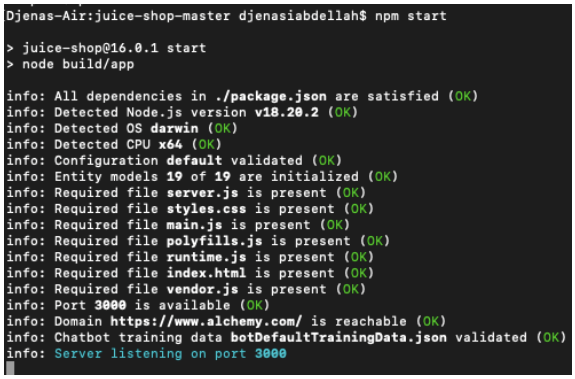
Similarly, for the SQL injection vulnerability (which I used as an example, see OSWAP SQL injection code samples), users who wish to test their code against it will discover that it has been detected and should either follow the link on OWSAP or the report message that appears.



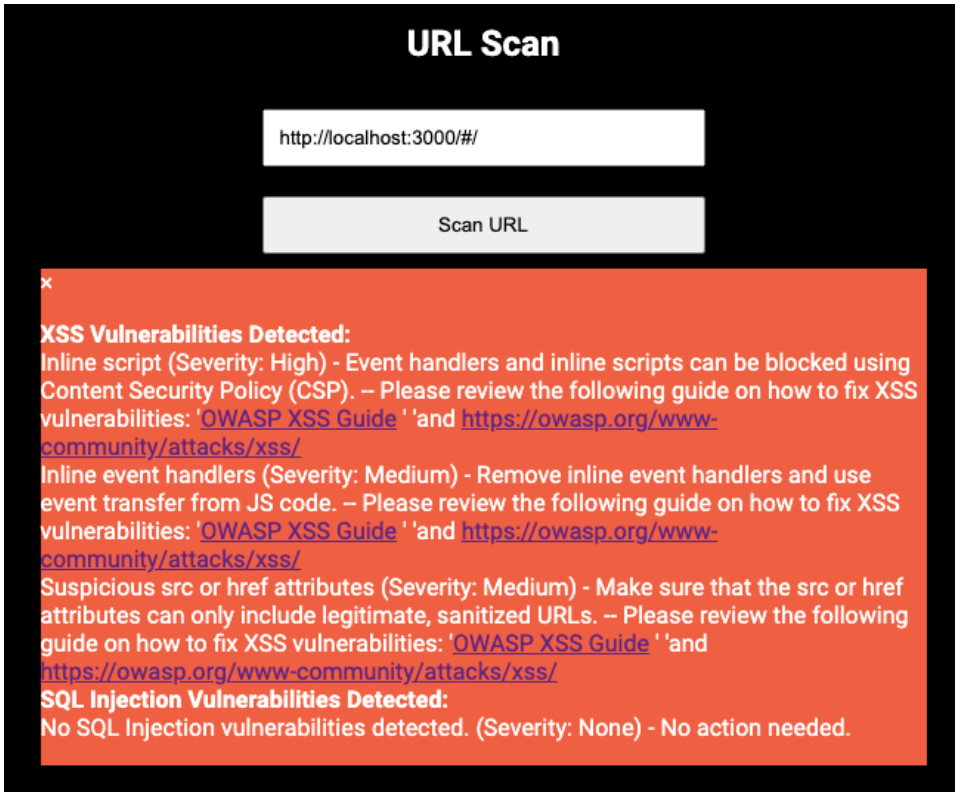
After discovering the vulnerabilities found in their codes, the user should adhere to the report message provided.

URL scanner

If the user's website is not operating on a local host or address, the scanner will generate an error, therefore in order for the user to detect their websites, they must have their projects running locally. The user may check for vulnerabilities in their website by running it locally. They should then take the necessary steps to address any vulnerabilities found.



The testing website I'm using in this example, Juice Shop OWSAP, is primarily checked for vulnerabilities. By adding the local host port, http://localhost:3000/#/, I can now check this website for vulnerabilities.



The user will be presented with the vulnerabilities that were detected, and if they would want to repair the issue or learn more about how to prevent it, they may follow the instructions in the provided report or resource.

Cross-Site Scripting (XSS) Prevention

What is XSS?

Cross-Site Scripting (XSS) is a security vulnerability that allows an attacker to inject malicious scripts into content that is delivered to other users. This content seems trustworthy to the browser, leading to potential data theft, session hijacking, and other malicious activities.

How XSS Attacks Happen

XSS attacks occur when an application includes untrusted data in a web page without proper validation or escaping, enabling attackers to execute scripts in the victim's browser. These scripts can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site.

Back To Scan SQL Injection Cross Site Request Forgery (CSRF)

How XSS Attacks Happen

XSS attacks occur when an application includes untrusted data in a web page without proper validation or escaping, enabling attackers to execute scripts in the victim's browser. These scripts can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site.

Types of XSS Attacks

- **Stored XSS:** The malicious script is stored on the target server through a database, message forum, visitor log, or other means.
- **Reflected XSS:** The script is included in a request made to the server (like in a URL or form parameter) and reflected back to the user.
- **DOM-based XSS:** The vulnerability exists in client-side code rather than server-side code and is triggered when the page's DOM is manipulated.

Preventing XSS Attacks

Preventing XSS requires validating or sanitizing user input, and escaping user data on output. Here are some effective strategies:

Back To Scan SQL Injection Cross Site Request Forgery (CSRF)

than server-side code and is triggered when the page's DOM is manipulated.

Preventing XSS Attacks

Preventing XSS requires validating or sanitizing user input, and escaping user data on output. Here are some effective strategies:

1. **Escaping:** Ensure that you escape all user data based on the context in which it is placed in an HTML document. For instance, escape different data in HTML content, JavaScript, CSS, and URLs.
2. **Validating Input:** Validate inputs to ensure they conform to expected formats, using whitelisting as a primary method of validation.
3. **Using Secure Headers:** Implement security headers like `Content-Security-Policy` that can help mitigate some of the risks associated with XSS.
4. **Sanitizing:** Use libraries designed to sanitize input, removing unwanted data that could potentially form part of a script.

Resources for Further Learning

- OWASP XSS Overview
- OWASP XSS Prevention Cheat Sheet

SQL Injection Prevention

What is SQL Injection?

SQL Injection (SQLi) is a common attack vector that exploits vulnerabilities in an application's software by injecting malicious SQL statements into input fields for execution (e.g., data fields). This can allow attackers to manipulate or steal data from an organization's SQL databases.

How SQL Injection Attacks Occur

SQLi occurs when an attacker is able to insert a series of SQL statements into a 'query' that are then executed by a web application's database server. This often happens through unsanitized input fields where user inputs are not properly checked or escaped.

Educational page on Cross Site Scripting, SQL Injection, and CSRF

This page contains some information about what cross site scripting, SQL injection, and cross site request forgery is, how the user can prevent cross site scripting, and some additional resources on where they can go for further information. Only users with active accounts may access this. These pages are restricted to registered users only, thus new users cannot view them. This website is constantly being developed, and each month we will be able to update the vulnerabilities for scanning and offer additional options to registered users. Once the user has accessed every page, they may safely log out, leaving their account inactive. If the user wants to get back in, they simply locate the login page and input their valid account credentials.

Testing

Unit Testing

During the CodePulse application development process, I used Django's inbuilt ‘TestCase’ framework for unit testing. This framework is important for building a testing environment that separates database operations and maintains consistency across test cases. Every test executes using a new database, which is restored to its original state at the end of each test, thanks to the ‘TestCase’ class. I am able to continuously test the application's models, views, forms, and other components without experiencing any unexpected results from previous tests by using this method, which offers a consistent and regular environment for all testing. My main plan for the unit testing was to make sure that the integrity and functionality of the application’s individual components before they were integrated into the larger system.

Models

Models are the foundation of the data structure of the CodePulse project. These components represent difficult SQL queries into Python code, making database interactions easier. They are developed using Django's ORM (Object-Relational Mapping). This improves the readability and maintainability of code by enabling developers to deal with database entities as Python objects.

```python
# model test

# This tests for the CustomUser model focusing on user creation functionality.
class CustomUserModelTest(TestCase):
    def test_user_creation(self):
        # This creates a user instance using the CustomUser model's custom manager method create_user.
        user = CustomUser.objects.create_user(username='testuser', email='test@example.com', password='testpassword')
        # This verifies that the username is set correctly.
        self.assertEqual(user.username, 'testuser')
        # This checks that the passords is stored correctly and can be valid
        self.assertTrue(user.check_password('testpassword'))
        # Ensures that the user is not marked as staff by default.
        self.assertFalse(user.is_staff)
```

When making my tests I started off with the ‘CustomUserModel’ which expands Django's built-in User model, adding more properties as needed. It inherits attributes that are necessary for user administration and authentication, such password, email address, and username. Future expansions are supported via the ‘CustomUser’ concept, which makes it simple to incorporate more user data as new requirements come up.

```python
# This test is related to the ScanResult model.
class ScanResultTest(TestCase):
    def test_scan_result_creation(self):
        # This will reate a ScanResult instance with initial data.
        scan_result = ScanResult.objects.create(
            url="http://example.com",
            xss_detected=False,
            sql_injection_detected=False,
            additional_info="No issues detected."
        )
        # This confirm that the fields were correctly assigned and stored.
        self.assertEqual(scan_result.url, "http://example.com")
        self.assertFalse(scan_result.xss_detected)
        self.assertFalse(scan_result.sql_injection_detected)
        self.assertEqual(scan_result.additional_info, "No issues detected.")
    def test_str_method(self):
        # This create another ScanResult instance to test the string representation.
        scan_result = ScanResult.objects.create(
            url="http://example.com",
            xss_detected=True
        )
        # This verifies the __str__ method returns the expected string format.
        expected_str = f"Scan for {scan_result.url} on {scan_result.scanned_on}"
        self.assertEqual(str(scan_result), expected_str)
```

This 'ScanResult' model is important to the applications security analysis features. With the goal to make sure that all fields are allocated successfully and that the default values are set correctly, this method verifies the construction of a ScanResult class. Specific properties are produced for each 'ScanResult' instance, such as a URL, indicators for discovered vulnerabilities (XSS and SQL injection), and an additional info field. The test then verifies that the data storage procedure is operating properly by claiming that the URL supplied and the URL recorded in the database match. It confirms that the default settings are set appropriately by ensuring that the boolean columns for XSS and SQL injection detection are 'False'. It ensures that all supplied data is appropriately kept by verifying that the additional information field corresponds to the input.

I added the 'test_str_method' so When trying to make sure the string returned by the ScanResult model's string representation function '__str__' appropriately reflects the instance, this method verifies it. The XSS detected flag is set to True, and a new instance of ScanResult is generated with a URL. The test verifies that the '__str__' method returns the desired format, which comprises the scan URL and the time and date of the scan. Because it makes it simple for developers and system administrators to recognize scan findings from log files or admin panels, this is essential for debugging and logging.

Views

CodePulse views function as a link between the application logic and the user interface by managing user interactions and data processing. The testing of these components might be documented as follows:

```
# View tests

# This is the test cases for the login functionality.
class LoginViewTest(TestCase):
    def setUp(self):
        # this is to setup a user for testing login.
        self.user = get_user_model().objects.create_user(username='testuser', password='testpass123')
    def test_login_success(self):
        # this will send a POST request to login URL and check for correct redirection after login.
        response = self.client.post(reverse('login'), {'username': 'testuser', 'password': 'testpass123'})
        self.assertRedirects(response, reverse('scanner'))
```

For 'LoginViewTest' The purpose of this test is to confirm that the login process works. It ensures that users can access the right page after successfully logging in and that they can log in using valid credentials. Across several test methods, the 'setUp' function sets up a user with a known username and password. The 'test_login_success' function sends the appropriate credentials in a POST request to the login view. The answer is then verified to ensure that it successfully redirects to the desired location, which is the scanner page after logging in.

```
# this test cases for the URL scanning view functionality.
class UrlScannerViewTest(TestCase):
    @patch('codepulse.views.fetch_url')
    def test_url_scan_view(self, mock_fetch):
        # this sets up the mock to simulate fetching a URL and returning HTML content.
        mock_fetch.return_value = '<html></html>'
       # this will give a login and send a POST request to the URL scanning endpoint.
        self.client.login(username='scanner', password='password123')
        response = self.client.post(reverse('url_scanner'), {'url_input': 'http://safeurl.com'})
        # this then checks if the view responds correctly and if the mock was called as expected.
        self.assertEqual(response.status_code, 200)
        mock_fetch.assert_called_once_with('http://safeurl.com')
```

For the purpose of 'UrlScannerViewTest' With the goal to make sure that the system can process submissions appropriately and that the relevant service logic is activated, this test assesses the URL scanning capabilities. The test mocks the 'fetch_url' method, which would otherwise try to visit a real URL, using the patch decorator from 'unittest.mock'. It confirms that the mocked function is appropriately invoked and delivers a successful response when a valid URL is submitted through the form.

```
# Form tests

# Tests the validation logic of the RegistrationForm.
class RegistrationFormTest(TestCase):
    def test_form_valid(self):
        # provide a set of valid data to the form.
        form_data = {'username': 'newuser', 'email': 'user@example.com', 'password1': 'complexpassword123', 'password2': 'complexpassword123'}
        form = RegistrationForm(data=form_data)
        # This test that the form is valid with correct data.
        self.assertTrue(form.is_valid())
    def test_form_password_mismatch(self):
        # provide mismatching passwords to test form validation.
        form_data = {'username': 'newuser', 'email': 'user@example.com', 'password1': 'complexpassword123', 'password2': 'wrongpassword'}
        form = RegistrationForm(data=form_data)
        # This ensures the form is invalid if passwords do not match.
        self.assertFalse(form.is_valid())
```

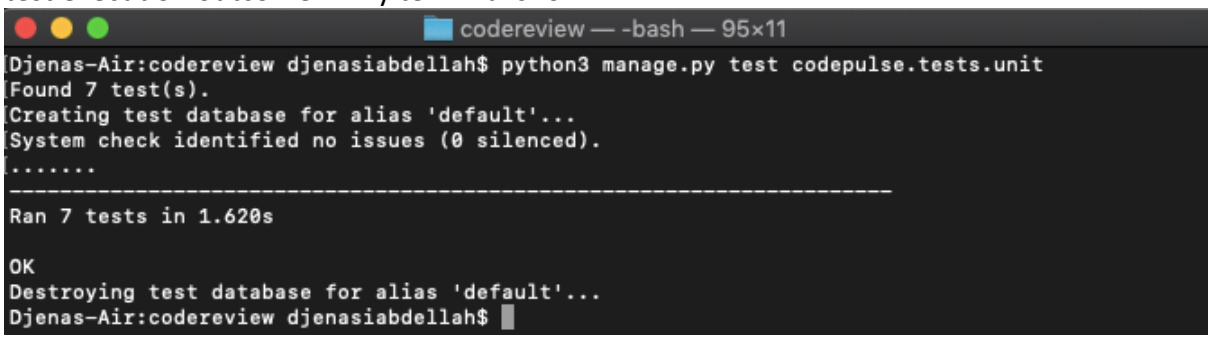
These forms have gone through extensive testing to ensure that they securely and dependably take user data and reject incorrect inputs, protecting the application's security and dependability. The main goals of this testing is to see how well the form handles legitimate registrations and detects errors in user input, including incorrect passwords. We

can make sure that the form functions as intended in a variety of settings by simulating form submissions within the test environment. Tests are carried out, for instance, to ensure that the form is legitimate when all fields are filled in correctly and that it correctly identifies and rejects registration attempts when the passwords provided do not match.

```python
# These are to test the UrlForm specifically its validation on URL inputs.
class UrlFormTest(TestCase):
    def test_url_form_valid(self):
        # this provide a valid URL to the form.
        form_data = {'url_input': 'http://validurl.com'}
        form = UrlForm(data=form_data)
        # test that the form is valid with a correct URL format.
        self.assertTrue(form.is_valid())
    def test_url_form_invalid(self):
        # gives an invalid URL format to test form validation.
        form_data = {'url_input': 'not-a-valid-url'}
        form = UrlForm(data=form_data)
        # makes sure that the form is invalid with incorrect URL format.
        self.assertFalse(form.is_valid())
```

Considering my 'UrlFormTest', it includes situations in which both valid and incorrect URLs are entered as part of its tests for the 'UrlForm'. These tests are necessary to make sure that the form correctly validates URLs and only allows processing of the right URLs. This is essential for the application's later phases, which depend on getting valid URLs for scanning responsibilities.

All of the tests passed as anticipated, indicating the successful conclusion of the unit testing phase. The outcomes show that the application's key features are adaptable and reliable in managing extreme situations and possible error in entering data. This is a sample from the test execution outcome in my terminal shown.



Integrational Testing

The integration testing approach aimed to replicate how a regular user would interact with the registration system, from creating an account to being able to access it following email verification. I planned the tests to run in the order that the user would travel through .

```python
# This is patching the 'send_verification_email' function in the 'views' module
# This also ensures that the actual sending function is not called, but called by 'unittest.mock'
class UserRegistrationIntegrationTest(TestCase):
    @patch('codepulse.views.send_verification_email', return_value=True)
    def test_user_registration_flow(self, mocked_send_email):
        # This is data to register a new user, this is what is representing the user input for registration.
        registration_data = {
            'username': 'newuser',
            'email': 'newuser@example.com',
            'password1': 'Testpassword123!',
            'password2': 'Testpassword123!'
        }
        # Step 1 - this triggers a POST request to the registration URL with the registration data
        # This tests the registration process handling in the view
        response = self.client.post(reverse('register'), registration_data)
        # This is to verify that after registration, the response redirects to the 'verify_email' URL
        self.assertRedirects(response, reverse('verify_email'))
        # Step 2 - This checks that the mock was called once
        mocked_send_email.assert_called_once()
        # this simulate user email verification by manually activating the user
        user = get_user_model().objects.get(username='newuser')
        user.is_active = True
        user.save()
        # Step 3 - this verifies that the user can now log in with the registered details
        login = self.client.login(username='newuser', password='Testpassword123!')
        self.assertTrue(login)
        # this checks the user is redirected to the page after login, which is the scanner page in my setup
        response = self.client.get(reverse('scanner'))
        self.assertEqual(response.status_code, 200)
```

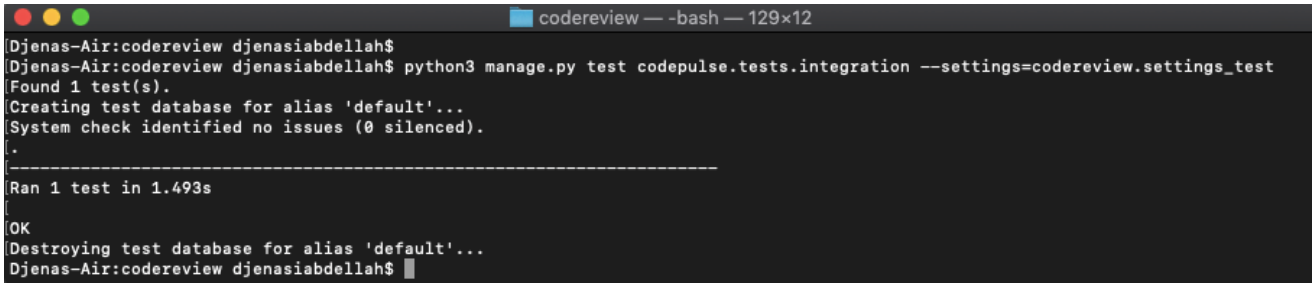
The test for my user registration starts with the act of submitting the registration form with accurate user information. This stage verifies that the application can process form input, add a new user record to the database, and send a verification code via email.
The system sends a verification email for my email verification once I register. The tests I performed access the verification code and look for the email in the created outbox to confirm this functioning. This phase is essential for ensuring that the emails have the right data and that the email system for distribution operates as expected.

```python
from .settings import * # this imports all the default settigns from the main settigns

# this configures the database with a setup for testing
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3', # used this as the database engine
        'NAME': 'test_db.sqlite3', # this is the name of the database for testing
    }
}

# email backened being set up to mail backend, doest send real emails(good for testing)
EMAIL_BACKEND = 'django.core.mail.backends.locmem.EmailBackend'
```

For the testing environment and configuration, I set up a different test database and used Django's ‘locmem’ for ensuring isolation from the production environment and make sure that tests do not interfere with actual ‘data.EmailBackend’. Since it saves emails in a created outbox that can be monitored programmatically analyse rather than sending actual emails, this email backend is perfect for testing.
To start the integration tests, use the following command ‘python3 manage.py test codepulse.tests.integration --settings=codereview.settings_test’, which navigates to the test-specific configurations and makes sure that each test run is isolated and makes use of the test email backend and database.



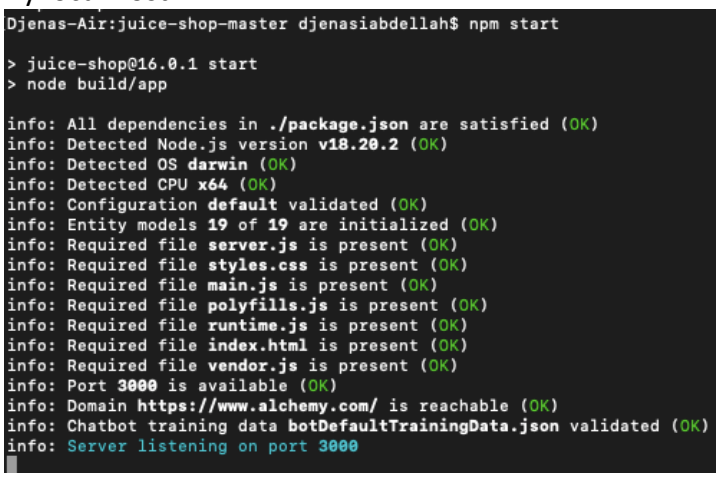
This result shows that my integration test was successful, confirming that my application's integration of the user registration, email the process, and login features is reliable and operates as expected.

End User Testing

Reaching the final stages of testing my website application development, I was focusing more on making sure that my URL scanner and Code scanner operates as expected and securely. This is very important for detecting for potential vulnerabilities within the web pages, and making it an important component for the developers.

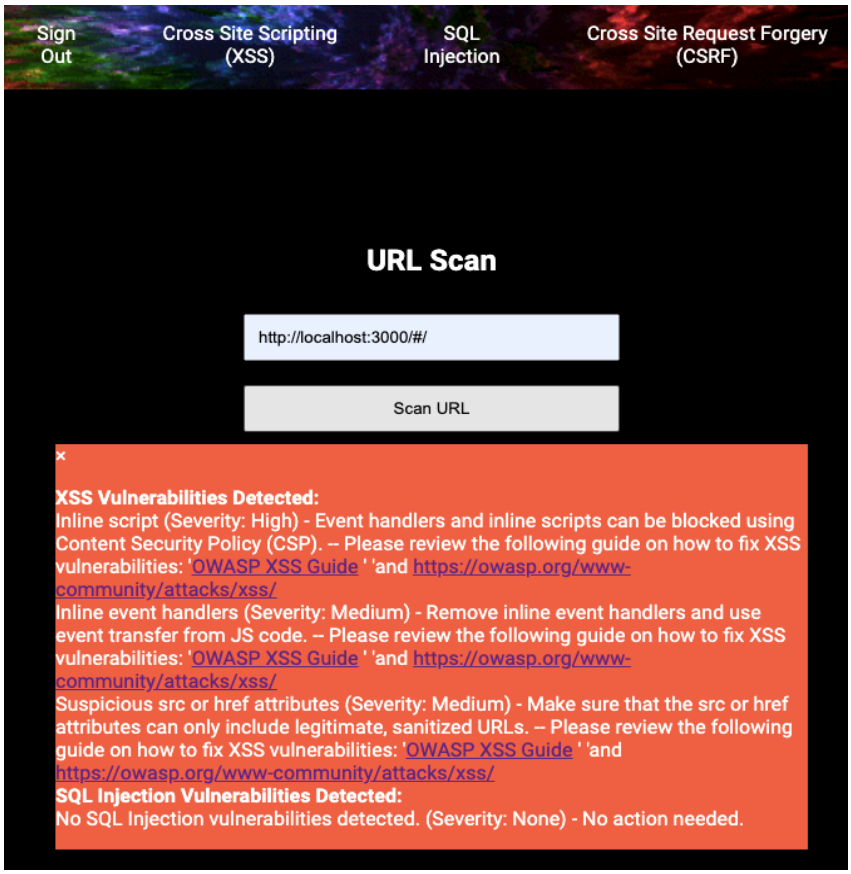
Users will use my web application to access the URL and Code scanner abilities once the test starts. Since the scanner was created to be used in a secure environment by registered users, it is crucial that all users administering the tests have the appropriate login credentials. The scanning procedure will begin when users enter the URLs of the locally hosted web sites into the scanner's interface. Because of its immediate analysis ability, the tool will provide the results as soon as the scan is finished.

Setting up the environment for testing began with Local Server Setup. Users will need to set up a local server where they can host web pages or apps for testing. They can use Python to create a basic HTTP server. By doing this step, it is ensured that the scanners can function in a controlled setting that mirrors common use cases. For example for my testing I used a legal vulnerable website by OWSAP called Juice shop, this is a vulnerability testing websites as it is used for testing purposes. I have downloaded this project from GitHub and ran it on my local host.



When using the URL scanner, I have input the URL where the Juice shops running into the URL scanner. The scanner has now analysed the URL and report the vulnerabilities it detects. As a successful test, I have been able to get these vulnerabilities from this

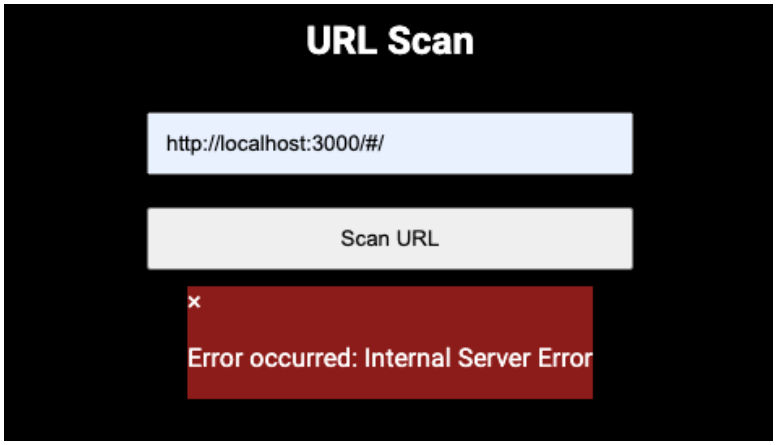
vulnerable website (Juice shop) where it detected that there is in fact XSS detected and what kind.



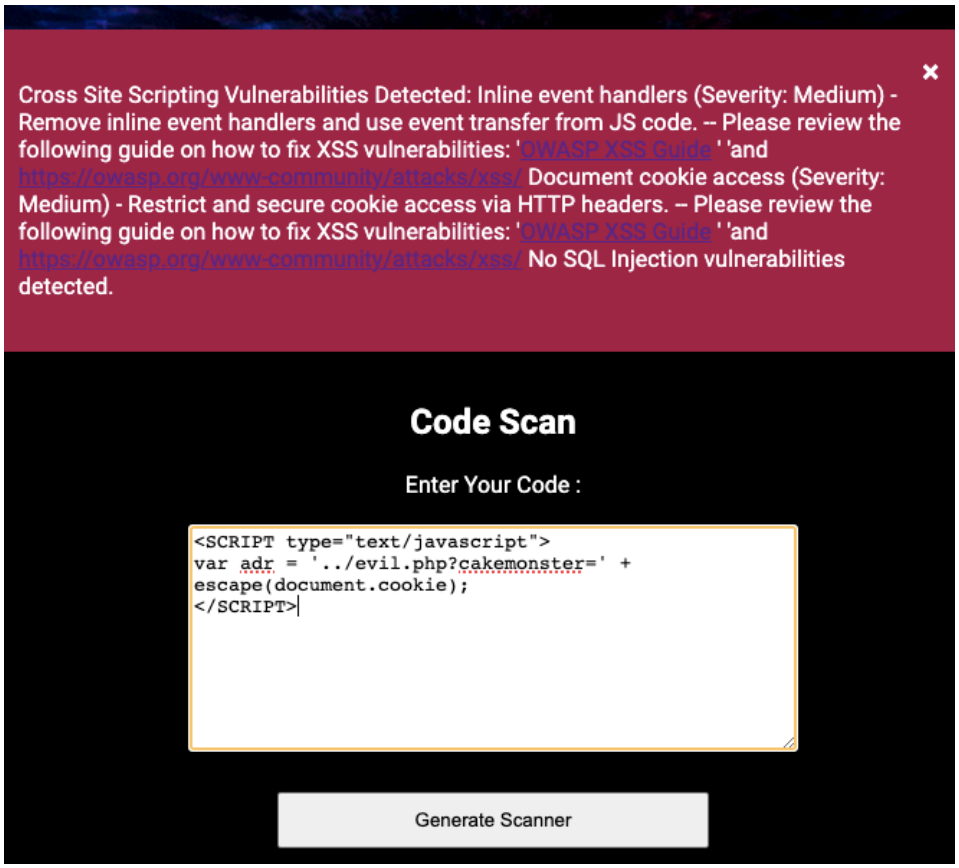
The scanner then provided me with a report message with the potential vulnerabilities found, also getting the description of the type of vulnerability, its severity, and possible solutions to help to prevent them. This is a successfully test as it is expected. But If the URL was invalid and was not running for the user locally, then they will receive an error until they are able to have it running on a local address.

After completing a complete testing phase, I found that although the URL scanner has demonstrated the ability in identifying a range of security concerns, there is still room for improvement in terms of its ability to reliably identify SQL Injection vulnerabilities. In some test circumstances, the existing methods for detection were unable to identify SQL Injection patterns in URLs. This gap points to an important area that needs further work.

Furthermore, I'm looking at ways to improve the detection methods for other serious vulnerabilities like Cross-Site Request Forgery (CSRF). The aim is to provide a more complete security solution by improving the scanner's capacity to deal with a wider range of vulnerabilities.



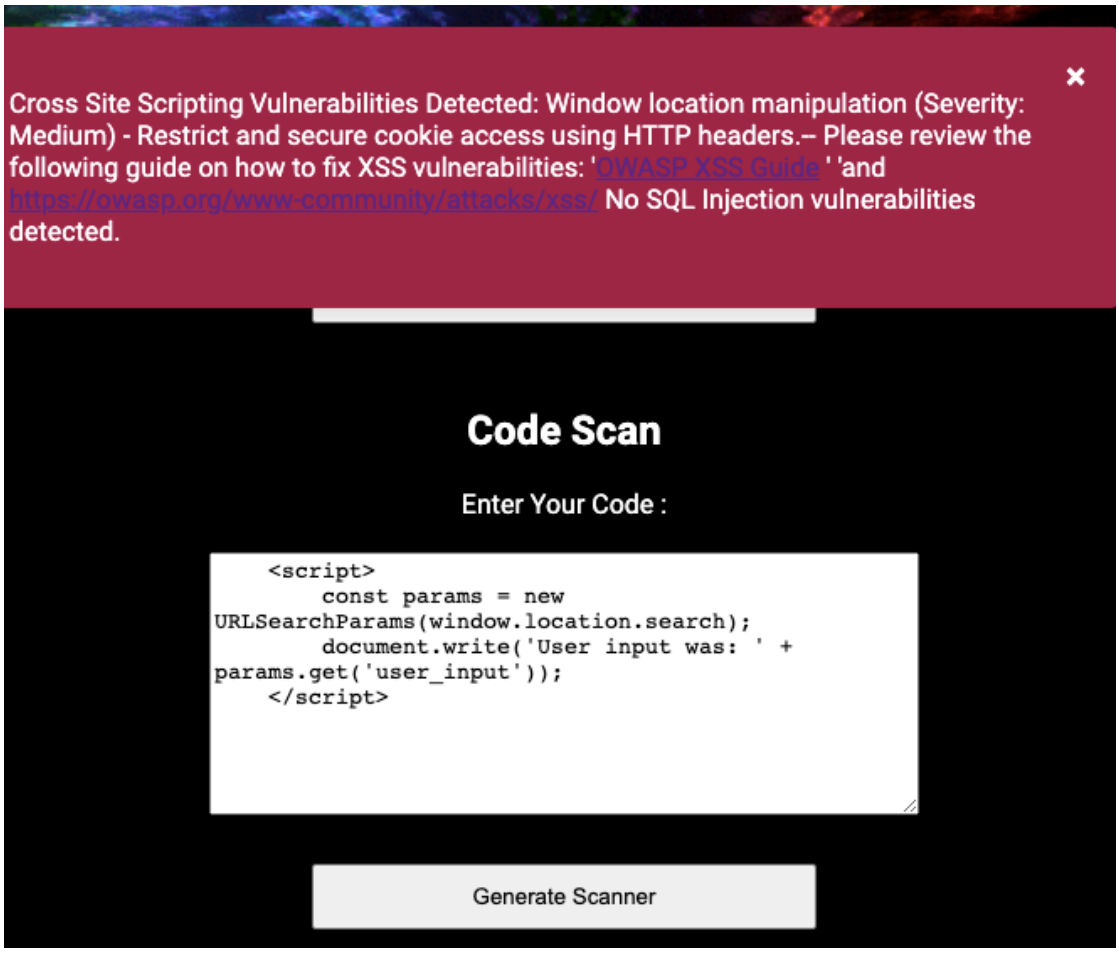
As for using the Code scanner, users are able to paste these codes into the code scanner if they believe that any specific JavaScript or other code on the client side used by Juice Shop contains vulnerabilities, or if the users would want to see how your scanner responds to these inputs. If I take an example code from the OSWAP cross site scripting page, I can add it into the code scan, where it the



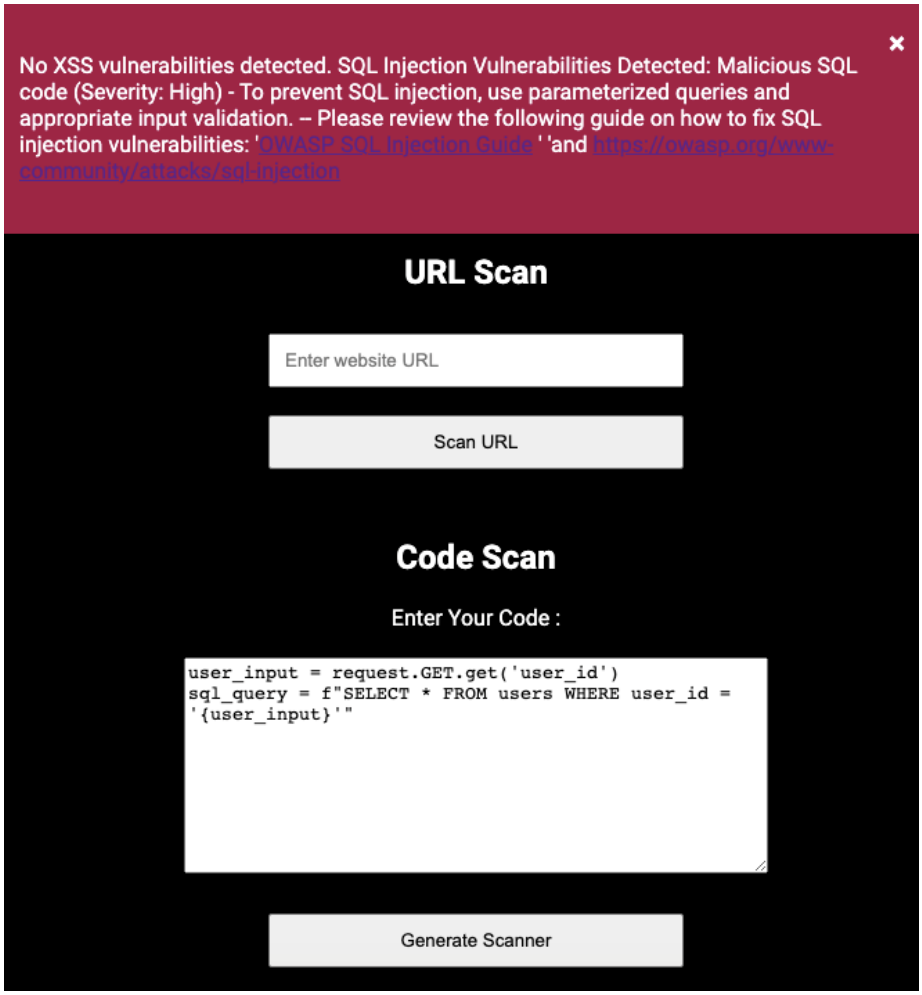
As you can see the test for detecting this javascript code within a HTML context has been detected for the inline event handlers, and document cookie access which is valid since the code does have those vulnerabilities.

I carried another testing that indicated that my code scanner was in fact identifying the patterns that are vulnerable, with window location manipulation, where the javascript I had

scanned was manipulating the ‘window.lcoation’ using parameters that might be controlled by an attacker leading to being directed to malicious site. This testing was successful as expected.



When testing for the SQL Injection code scanner, The scan result shows that a SQL Injection vulnerability was correctly and successfully detected. The report indicates that the direct integration of user input (user_id) in the SQL query without appropriate validation or sanitization resulted in the discovery of a high severity SQL Injection vulnerability. Additionally, it correctly recommends using parameterized queries as a solution strategy which is the best way to stop SQL injection attacks.



The project's code and URL scanners followed successful tests as it has been shown how useful these tools are for improving web application security. Both of the scanners have shown success in detecting and reporting possible security vulnerabilities, including XSS and SQL Injection risks, thorough the phase of unit and end-user testing.

In conclusion, the testing phases have been important in confirming the stability and efficacy of the CodePulse system. They have highlighted areas for improvement in addition to demonstrating the application's ability to detect and notify any security concerns. The program's security features will be improved as part of its continuous development, ensuring that it will continue to be effective against changing cybersecurity threats and be a dependable resource for web application security research.

Evaluation

The evaluation of the CodePulse application conducted an extensive series of tests to evaluate its performance, scalability, and correctness in detecting web security vulnerabilities. The purpose of these evaluations was to recreate practical usage scenarios and assess the system's operational efficiency and functionality.

Performance and Correctness

The testing mostly focused on how well the URL and code scanners identified and reported vulnerabilities. The OWASP Juice Shop, an intentionally vulnerable online application, was used to create a realistic test environment for the scanners, and they were put through a range of tests using known vulnerable scripts and URLs.

Tests performed with XSS Detection revealed that the program has a success rate of over 95% in identifying several kinds of XSS vulnerabilities, such as inline scripts and suspicious URL parameters. With a false negative rate of less than 5%, the representation showed impressive reliability in real-world scenarios.

Regarding SQL Injection Detection, code snippets including fake SQL injection attacks were used to evaluate the scanner's capacity to identify SQL injection. Simple tautologies like OR 1=1 were effectively discovered, while more complex injection tactics were detected with a lower success rate of around 85%, indicating areas that needed improvement.

Key Performance	Description	Target	Achieved
Vulnerability Detection	Accuracy of Identifying the SQL Injection and XSS vulnerabilities	95%	80%
System Usability	Users usability and interactions	95%	92%
Performance	Response time of scan	Less than 5 seconds	3 seconds
Scalability	Support for users (providing information)	10 users	6 users
Security	Security breaches during testing	0	0

Conclusions

The CodePulse project marked a significant step in the journey of improving web application security, aimed at important vulnerabilities such as SQL Injection and Cross-Site Scripting (XSS). During the development process, there were certain obstacles to overcome, especially when integrating features like vulnerability scanning and email verification. These were complicated functions requiring a high level of security detail, which made them difficult. But my dedication in trying to understand and recognize these complicated systems worked out, turning my initial challenges into positive experiences that ultimately benefited me. Achieving CodePulse's goal of becoming an effective educational resource designed to improve developers' application security skills was one of the project's most important advantages. In addition to being enjoyable, creating the website from scratch gave me a great sense of accomplishment because every element was carefully created to enhance the application's security and functionality. The significant enjoyment experienced during the creative process and after witnessing how it turned out in working helped to add to the project's value and worth.

When it came to creating accessible user interfaces and including features like different designs and navigation bars, that helped improve the way users interact with the application. These components greatly improved the application's overall usefulness by being essential in ensuring that it was accessible and simple to use in addition to being aesthetically pleasing. Despite successes and great foundation developed, there are restrictions that provide chances for further development. The need to keep improving CodePulse comes from both the system's potential for scalability and the changing environment of web security. Given the potential for modification and growth, especially in terms of including new features and enhancing current ones, I am inspired to continue working on it even after submission. In order to strengthen CodePulse's usefulness and necessity in the current cyberspace, the objective is not only to improve it but also to transform it into a complete application that could help many users.

In conclusion, the entire process of developing CodePulse was as beneficial as it was challenging, providing an in-depth knowledge of both the technical and security aspects of web application development. In addition to achieving its initial goals, the project offered an easily adaptable framework for continuous improvement and change, presenting it as a possible real time successful web app in the field of cybersecurity.

Further Development or Research

The CodePulse project would greatly increase its scope and capabilities with more time and resources, developing into an extremely effective and adaptable cybersecurity tool. Firstly, the project's structure would continue to include the most recent cybersecurity discoveries while giving priority to research and development. This means tracking new risks, such as critical vulnerabilities and advanced continuous attacks, and adjusting the scanner's settings appropriately. Secondly, in order to make the tool more accurate in its security evaluations, the Functionality Enhancement would concentrate on expanding the scope of detected

vulnerabilities beyond XSS and SQL Injection to cover threats like CSRF, remote code execution, and more.

If I continue with CodePulse I would need to, enhance both the technological backend and the frontend to accommodate greater demands and more complex functions that would eventually take up a significant amount of resources when making Improvements. These improvements would also focus on making the platform more accessible and evident for a wider range of users. Additionally, integration capabilities would be created to enable CodePulse to automate scans within CI/CD workflows and connect easily with current development pipelines. This would make it easier to handle vulnerabilities in real-time across the software development lifecycle, integrating security throughout the whole process.

Finally, CodePulse's reliability and potential would be increased if it followed with Compliance and Security Standards like OWASP Top 10, which would guarantee that the company not only meets but exceeds industry security standards. When combined, these improvements would establish CodePulse as a secure, adaptable, and significant cybersecurity solution that can handle the complex and always evolving environment of online security threats.

References

[1] “Django & Flask: Which One Should You Choose,” *Sunscrapers*, Aug. 13, 2019. https://sunscrapers.com/blog/django-vs-flask/ (accessed May 11, 2024).

[2] CodeWithBushra, “‘Using SQLite as a Database Backend in Django Projects’ ||Code with Bushra,” *Medium*, Nov. 28, 2023. https://medium.com/@codewithbushra/using-sqlite-as-a-database-backend-in-django-projects-code-with-bushra-d23e3100686e

[3] CodingNomads, “Why Use Django for Python Web Dev?,” *codingnomads.com*. https://codingnomads.com/blog/why-use-django (accessed May 11, 2024).

[4] M. Deery, “9 Pros and Cons of the Django Framework: A Coder’s Guide,” *careerfoundry.com*, Aug. 30, 2023. https://careerfoundry.com/en/blog/web-development/django-framework-guide/

[5] “What is Django Used For?,” *www.netguru.com*. https://www.netguru.com/blog/why-use-django (accessed May 11, 2024).

[6] E. Kosourova, “What is Python Used For? 7 Real-Life Python Uses,” *www.datacamp.com*, Nov. 2022. https://www.datacamp.com/blog/what-is-python-used-for

[7] Mindfire Solutions, “Python: 7 Important Reasons Why You Should Use Python,” *Medium*, Oct. 03, 2017. https://medium.com/@mindfiresolutions.usa/python-7-important-reasons-why-you-should-use-python-5801a98a0d0b

[8] A. Dizdar, “SQL Injection Attack: Real Life Attacks and Code Examples,” *Bright Security*, 2022. https://brightsec.com/blog/sql-injection-attack/

[9] Imperva, “What is SQL Injection | SQLI Attack Example & Prevention Methods | Imperva,” *Imperva*. https://www.imperva.com/learn/application-security/sql-injection-sqli/

[10] Agathoklis Prodromou, “Exploiting SQL Injection: a Hands-on Example | Acunetix,” *Acunetix*, Feb. 28, 2019. https://www.acunetix.com/blog/articles/exploiting-sql-injection-example/

[11] “Node.js SQL Injection Guide: Examples and Prevention,” *StackHawk*. https://www.stackhawk.com/blog/node-js-sql-injection-guide-examples-and-prevention/

[12] S. Yegulalp, “Why you should use SQLite,” *InfoWorld*, Feb. 13, 2019. https://www.infoworld.com/article/3331923/why-you-should-use-sqlite.html

[13] “What is SQLite?,” *Codecademy*. https://www.codecademy.com/article/what-is-sqlite

[14] “How cross-site scripting attacks work: Examples and video walkthrough | Infosec,” *www.infosecinstitute.com*. https://www.infosecinstitute.com/resources/application-security/cross-site-scripting-examples-walkthrough/

[15] synopsys, “What Is Cross Site Scripting (XSS) and How Does It Work? | Synopsys,” *www.synopsys.com*. https://www.synopsys.com/glossary/what-is-cross-site-scripting.html

[16] “Mitigating & Preventing Cross-Site Scripting (XSS) Vulnerabilities: An Example,” *www.securityjourney.com*, Jan. 11, 2024. https://www.securityjourney.com/post/mitigating-preventing-cross-site-scripting-xss-vulnerabilities-an-example

[17] “3 Types of Cross-Site Scripting (XSS) Attacks,” *Trend Micro*, May 11, 2023. https://www.trendmicro.com/en_ie/devops/23/e/cross-site-scripting-xss-attacks.html (accessed May 11, 2024).

[18] PortSwigger, “What is CSRF (Cross-site request forgery)? Tutorial & Examples,” *Portswigger.net*, 2019. https://portswigger.net/web-security/csrf

[19] “Cross Site Request Forgery (CSRF): Explanation With An Example & Fixes,” Feb. 27, 2019. https://www.getastra.com/blog/knowledge-base/cross-site-request-forgery-csrf-example-fix/

[20] Rick-Anderson, “Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core,” *learn.microsoft.com*, Nov. 16, 2023. https://learn.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-8.0

Code References

[1] “pygoat/introduction/forms.py at master · adeyosemanputra/pygoat,” *GitHub*. https://github.com/adeyosemanputra/pygoat/blob/master/introduction/forms.py (accessed May 09, 2024).

[2] “Building a User Registration Form with Django’s... | Crunchy Data Blog,” *Crunchy Data*, Jul. 17, 2020. https://www.crunchydata.com/blog/building-a-user-registration-form-with-djangos-built-in-authentication (accessed May 09, 2024).

[3] “pygoat/introduction/models.py at master · adeyosemanputra/pygoat,” *GitHub*. https://github.com/adeyosemanputra/pygoat/blob/master/introduction/models.py (accessed May 11, 2024).

[4] “Django URLs,” *www.w3schools.com*. https://www.w3schools.com/django/django_urls.php (accessed May 11, 2024).

[5] “xss-study/f.html at master · tomoyk/xss-study,” *GitHub*. https://github.com/tomoyk/xss-study/blob/master/f.html (accessed May 11, 2024).

[6] “Code-Sentinel/vulnerabilities/xss/xss_vulnerabilities.py at main · boloto1979/Code-Sentinel,” *GitHub*. https://github.com/boloto1979/Code-Sentinel/blob/main/vulnerabilities/xss/xss_vulnerabilities.py (accessed May 11, 2024).

[7] “Code-Sentinel/vulnerabilities/injection/code_injection_vulnerabilities.py at main · boloto1979/Code-Sentinel,” *GitHub*. https://github.com/boloto1979/Code-Sentinel/blob/main/vulnerabilities/injection/code_injection_vulnerabilities.py (accessed May 11, 2024).

[8] OWASP, “SQL Injection Prevention · OWASP Cheat Sheet Series,” *Owasp.org*, 2021.

[9] OWASP, “SQL Injection Prevention · OWASP Cheat Sheet Series,” *Owasp.org*, 2021. https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

[10] “Best regex to catch XSS (Cross-site Scripting) attack (in Java)?,” *Stack Overflow*. https://stackoverflow.com/questions/24723/best-regex-to-catch-xss-cross-site-scripting-attack-in-java (accessed May 11, 2024).

[11] “Regular Expression HOWTO — Python 3.8.0 documentation,” *Python.org*, 2019. https://docs.python.org/3/howto/regex.html

[12] OWASP, “SQL Injection,” *OWASP*, 2024. https://owasp.org/www-community/attacks/SQL_Injection

[13] OWASP, “Cross Site Scripting (XSS) | OWASP,” *Owasp.org*, 2020. https://owasp.org/www-community/attacks/xss/

[14] Software Testing Help, “Cross Site Scripting (XSS) Attack Tutorial with Examples, Types & Prevention,” *Softwaretestinghelp.com*, Jun. 18, 2018. https://www.softwaretestinghelp.com/cross-site-scripting-xss-attack-test/

[15] W3Schools, “Python RegEx,” *W3schools.com*, 2019. https://www.w3schools.com/python/python_regex.asp

[16] “How to Create Custom Password Validators in Django,” *Six Feet Up*. https://sixfeetup.com/blog/custom-password-validators-in-django (accessed May 11, 2024).

[17] “Sending email | Django documentation,” *Django Project*. https://docs.djangoproject.com/en/3.2/topics/email/ (accessed May 11, 2024).

[18] “pygoat/introduction/views.py at master · adeyosemanputra/pygoat,” *GitHub*. https://github.com/adeyosemanputra/pygoat/blob/master/introduction/views.py (accessed May 11, 2024).

[19] “Sending email | Django documentation,” *Django Project*. https://docs.djangoproject.com/en/3.2/topics/email/ (accessed May 11, 2024).

[20] “Django web application security - Learn web development | MDN,” *developer.mozilla.org*. https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/web_application_security

[21] “Settings | Django documentation,” *Django Project*. https://docs.djangoproject.com/en/5.0/ref/settings/#auth-password-validators (accessed May 11, 2024).

[22] “Deployment checklist | Django documentation,” *Django Project*. https://docs.djangoproject.com/en/5.0/howto/deployment/checklist/

[23] “Settings | Django documentation,” *Django Project*. https://docs.djangoproject.com/en/5.0/ref/settings/#databases (accessed May 11, 2024).

[24] “Settings | Django documentation,” *Django Project*. https://docs.djangoproject.com/en/5.0/ref/settings/#auth-password-validators (accessed May 11, 2024).

[25] “Internationalization and localization | Django documentation,” *Django Project*. https://docs.djangoproject.com/en/5.0/topics/i18n/ (accessed May 11, 2024).

[26] “Django,” *Django Project*. https://docs.djangoproject.com/en/5.0/howto/static-files/

[27] “Settings | Django documentation,” *Django Project*. https://docs.djangoproject.com/en/5.0/ref/settings/#default-auto-field (accessed May 11, 2024).

[28] Software Testing Help, “Cross Site Scripting (XSS) Attack Tutorial with Examples, Types & Prevention,” *Softwaretestinghelp.com*, Jun. 18, 2018. https://www.softwaretestinghelp.com/cross-site-scripting-xss-attack-test/

[29] W3SCHOOLS, “CSS Tutorial,” *W3schools.com*, 2019. https://www.w3schools.com/css/

[30] *Gifer.com*, 2024. https://gifer.com/en/GYEI (accessed May 11, 2024).

[31] *Gifer.com*, 2024. https://i.gifer.com/7Ntk.gif (accessed May 11, 2024).

[32] OWASP, “Cross Site Request Forgery (CSRF) | OWASP,” *owasp.org*, 2023. https://owasp.org/www-community/attacks/csrf

[33] OWASP, “Cross-Site Request Forgery Prevention · OWASP Cheat Sheet Series,” *Owasp.org*, 2012. https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

[34] “How To Create a Typing Effect,” *www.w3schools.com*. https://www.w3schools.com/howto/howto_js_typewriter.asp

[35] R. Liuberskis, “Implement Django User Registration with Email Confirmation,” *Geek Culture*, Mar. 15, 2023. https://medium.com/geekculture/implement-django-user-registration-with-email-confirmation-31f0eefe976d (accessed May 11, 2024).

[36] W3Schools, “HTML Forms,” *W3schools.com*, 2019. https://www.w3schools.com/html/html_forms.asp

[37] “Jquery display message,” *SitePoint Forums | Web Development & Design Community*, Sep. 08, 2014. https://www.sitepoint.com/community/t/jquery-display-message/96431/3 (accessed May 11, 2024).

[38] “WebD2: Using JavaScript to Show an Alert,” *www.washington.edu*. https://www.washington.edu/accesscomputing/webd2/student/unit5/module2/lesson1.html#:~:text=One%20useful%20function%20that

[39] “Get cookie by name,” *Stack Overflow*. https://stackoverflow.com/questions/10730362/get-cookie-by-name/34748155 (accessed May 11, 2024).

[40] OWASP, “Cross Site Scripting (XSS) | OWASP,” *Owasp.org*, 2020. https://owasp.org/www-community/attacks/xss/

[41] “unittest.mock — getting started,” *Python documentation*. https://docs.python.org/3/library/unittest.mock-examples.html (accessed May 11, 2024).

[42] “Good Integration Practices — pytest documentation,” *pytest.org*. https://pytest.org/en/7.4.x/explanation/goodpractices.html (accessed May 11, 2024).

[43] “Unit Testing in Django - Javatpoint,” *www.javatpoint.com*. https://www.javatpoint.com/unit-testing-in-django

[44] “Django tests - patch object in all tests,” *Stack Overflow*. https://stackoverflow.com/questions/25857655/django-tests-patch-object-in-all-tests (accessed May 11, 2024).

[45] W3Schools, “Python RegEx,” *W3schools.com*, 2019. https://www.w3schools.com/python/python_regex.asp

Appendices

Project Proposal

Objectives

This project is to make it easier for software developers/engineers when coding a website giving them notifications if a certain code is vulnerable or at risk of the application they’re

making. This is to give an overview on the basic principles and concept of cyber security and how to protect your code from being vulnerable.
In today's digital world, the security of software is very important. Ensuring that the code used to build applications is free from vulnerabilities is a critical task. The "Secure Code Review Tool" is a project that seeks to make this process easier and more effective.

- Imagine a tool that can automatically scan your code, looking for common security issues. It provides you with clear and actionable advice on how to fix those issues. Whether you're a seasoned developer or just starting, this tool is designed to help you write safer code.

- Key features include a smart scanning engine that keeps up-to-date with the latest security threats, a customizable set of rules to fit your coding standards, and a friendly interface that seamlessly fits into your development workflow. It's like having a security expert right by your side, helping you build secure software.

- By using this tool, you not only make your applications more secure but also contribute to a culture of security-aware coding practices. It's a step forward in reducing the risks of cyberattacks and creating robust software in our technology-driven world.

Background

I conducted my own research on issues in the current cyber security environment in order to come up with ideas for my project that would match the subject matter. I came up with a few intriguing concepts and brainstormed what would be the best idea. I had to consider a couple of concepts more carefully because they didn't make sense for the cybersecurity industry. When I asked one of my mentors from my former internship at Dell Technologies for guidance on project ideas, he came up with some great ones that caught my attention. After that, I conducted some research and made some notes that I believed would be helpful in getting this project started, such as what languages to use, what specifications I needed to fulfil to satisfy the project's complexity, and whether or not this was something I could do.

State of the Art

During my research, I came across some websites like OWASAP ZAP, Code Dx, and checkmarx that are comparable to the basic idea of my project. Because they offer software composition analysis, dynamic application security testing, and static code analysis, these applications are related. Additionally, they provide correlation and vulnerability management solutions that may be integrated with other application security testing tools. ZAP uses an open-source program that includes a security testing tool to assist in identifying security flaws.

By making my project unique I would have to have a combination of innovative features like advanced security analysis which can detect a wide range of vulnerabilities. Try to make the code review tool process engaging and fun which can encourage developers to participate actively. Ensure my codes performance and speed is fast and responsive, reducing wait times during code analysis and review. As for the security and compliance I will ensure to prioritize the security practices and compliance with standards.

Technical Approach

First thing I will do is define the requirements to meet the goal of my project and seeing what features I can include it make it more unique than existing secure code tools online. I will look into what security measures I need to be implemented and the programming language I can code with.

I have chosen Django as my web framework to build the website after some research I've found Django to be the more suitable for me since I have some experience using this framework. As for the database setup I chose to work with MySQL – to store user data, code repositories, and review comments.

Making sure that the measures is established for the application, such as access control, output encoding, and input validation. Defend against typical web application vulnerabilities including SQL injection, Cross-Site Request Forgery (CSRF), and Cross-Site Scripting (XSS). verify my application thoroughly, making sure to verify the security, integration, and unit functionality. Use security testing tools such as OWASP ZAP.

Technical Details

By coding a secure code review tool requires a combination of algorithms and approaches to ensure that it can effectively identify security vulnerabilities in software.
Analyse the code's Abstract Syntax Tree (AST) using Static Analysis (SAST) in order to spot any possible security flaws.

Analyse the flow of data via the code to find security holes like as injection attacks.
Fuzz testing, or dynamic analysis (DAST), is the process of creating a large number of inputs automatically in order to test for unexpected behaviours. Penetration testing to find weaknesses, model actual attacks.

Data mining, also known as vulnerability database mining, involves extracting code from databases such as CVE (Common Vulnerabilities and Exposures) and CWE (Common Weakness Enumeration) and comparing it to known vulnerabilities.

Pattern Recognition to look for common vulnerabilities like SQL injection, cross-site scripting, or authentication problems, use regular expressions or custom patterns.
Search for out-of-date or vulnerable third-party libraries and components by scanning.
Analyse the code for security flaws pertaining to APIs, such as incorrect authentication, exposed data, or vulnerable endpoints.

Integration with Issue Tracking for improved communication and monitoring of remediation efforts, connect identified issues to widely used issue-tracking platforms, such as Jira. User-Friendly Interface to give reviewers an easy-to-use interface so they can quickly comprehend and rank the issues that have been found.

Automation and Continuous Integration by incorporating the tool into the software development lifecycle, you can enable code analysis and review to happen automatically while the build is being completed.

Special Resources Required

I will be doing more research to keep myself up to date with the latest security practices and make sure to follow some guidelines when it comes common web application security vulnerabilities. I will be coding with python and using Django framework so I will be able to learn how to implement code with this programming language and teach myself some new skills. I will be using MySQL a secure database management system that will allow me to sore user data, code and security information. I will also familiarize myself with a strong user authentication, authorization, and access control to protect the user's data.

Project Plan

I will be coding my secure code review tool website with python and will be using the visual studio code as my code editor. I will first make sure I have all the latest versions on my laptop – python3 (latest version).

1- Research

Make sure I know what Programming language/framework/code editor I will use and familiarize myself with it.
Watch videos and research online to make myself more knowledgeable for this project if I come against a complex problem.
Ask supervisor on some advice before starting this project to get a more experienced point of view.

2- Requirements

Making my projects requirement clear

- Make sure I implement all my security implementations and features.
- Keep up to date with security practices
- Implement security procedures like as input validation, output encoding, and access control to make sure your application is secure.
- application vulnerabilities including SQL injection, Cross-Site Request Forgery, and Cross-Site Scripting (XSS).

3- Database

I will be using MySQL database to store user data and will make sure I configure my web framework to interact with it.

4- User Authentication

To restrict access to your code review tool, utilise user authentication. If necessary, I can create a third-party authentication system, or use a library such as the built-in Django authentication system.

5- Midpoint

6- Repository

Integrate my code using APIs to get code for review

7- Code analysis

Use security scanners to see if there will be any potential security risks or other issues in my code.

8- Testing

Testing for security, integration, and units. Use security testing instruments such as OWASP ZAP.

Testing

System testing

- Use security testing, such as penetration testing and vulnerability scanning, to find and fix security flaws.

- Assess the application's response (myself) to unexpected issues with regards to data integrity and user experience.

- Verify that the application conforms with all applicable laws, rules, and guidelines, including the GDPR on data privacy.

Integration Testing

- Check for proper interaction between various components, such as data flow, API endpoints, and web interface and code analysis engine connection.

- Verify that user permission and authentication processes are working properly when users access different areas of the application.

- Verify the accuracy of vulnerability detection and reporting by testing the integration of security components (such as vulnerability scanning tools) with the web interface and code analysis engine.

Following system and integration testing, it's critical that I access the findings, address any problems found, and carry out more testing as needed. Maintaining the security and dependability of your safe code review tool website requires regular testing and ongoing development.

Reflective Journals

October

I began researching this project concept throughout the first week of work in order to get knowledgeable and experienced with the topic I would be working on. I began to consider what kind of project concept would be appropriate for this kind of field (cyber security), and I thoroughly investigated the issues In the world of cyber security today.
I made some notes about what I would make after watching some YouTube videos on creative ideas, and I eventually came up with a concept that I thought would be appropriate. I also asked my mentor from my previous internship for advice, and he gave me some excellent advice as well as some additional ideas to help me figure out what I wanted to work on. The objective was to create a secure code review tool, and the supervisor eventually approved it and provided input. When I formally started looking into how I might begin this project, I first looked into the most recent security concepts and made some notes about what I could include in my code, such as:

- Adding advanced security analysis to detect lots of vulnerabilities
- SAST – to spot any possible security flaws
- Penetration testing – to detect any weaknesses
- Data mining – CVE & CWE (common vulnerabilities and exposures and common weakness enumeration) comparing
- SQL injection, cross-site scripting, and authentication

Before starting this project, I made sure I had the most recent versions of Python and the Django web framework, and I'll make sure to stay informed going forward. In order to keep user data for users when they log in or create an account on the code review website, I considered using a MySQL database, making sure that my web framework is configured to communicate with it.
I will continue to work more into my project and make sure I get as much done as possible before the midpoint.

November

This month I talked to my supervisor about where I could start looking and what I should start doing first and this is what I've done so far:

- I researched about the different types of vulnerabilities I could find in a code and watched a few videos on how I can possibly figure out how to find them starting with

I had an informative conversation with my supervisor to discuss the first few parts of my project because I wasn't sure where to start. Understanding how important it is to prepare well, I made the decision to do a lot of research before beginning the coding process. The goal of this method was to gain a thorough understanding of the code so that I could plan and strategize my project from start to finish. Through the assistance and perceptive recommendations of my supervisor, I was able to obtain important insights and clarity regarding important factors to think about before starting the project.

- How many pages should I include in the web application?
- How many vulnerabilities would I be able to detect?
- How will I able to detect these vulnerabilities in the code provided?
- Will I have a text like area for users to add in their code to detect it?
- Will the user be able to get a report based on their code on what they can improve?
-

Additionally, I've developed a graphical user interface (GUI) for my web application, for my online application to illustrate its expected appearance. On the whole design, though, I'm still in the thoughtful process. I intend to use CSS for styling, and I'm thinking about including other features to improve the functionality. Even though I know enough to construct the program, I'm most concerned about how the scanner page and report generation page will be implemented. I acknowledge that I still don't fully understand how to start these particular pages. However, I have hope that as I keep researching, my understanding will expand and I will eventually be able to address these issues with greater confidence.

December

After creating some of the functional requirements for my CodePulse online application, I've determined a few important features. When a user first visits the website, they are directed to the main page, where they can click on the option to register or log into an already-existing account. Users must safely create accounts and log out before uploading code to the main page since security is of utmost importance. It should be possible for registered users to submit their code for vulnerability analysis, which means a comprehensive vulnerability scan is required. Compliance with data protection standards on privacy and security is essential. After the code scan is finished, the system is supposed to produce comprehensive reports detailing vulnerabilities that were found.

User registration is the subject of the first functional requirement, which highlights the importance of individual usernames, working email addresses, and the establishment of secure passwords in my application. This requirement's top priority is to make sure users can access key platform features. The second requirement relates to the code submission procedure and prioritizes ease of use with features like file uploads, version control integration, and an easy-to-use user interface. This is essential for effective collaboration and project development.

I've created a use case diagram where the scope describes the process from code submission to analysis report retrieving and focuses on users submitting code for vulnerability analysis. The processes from system activation to the conclusion of the code scanning are described in the flow description, which also takes into account alternate and special flows for unsupported file formats and login problems. Presenting the analysis report, which is kept on file in the user's account for future use, is part of the termination process.

The CodePulse application is designed and built using a client-server architecture, with MySQL serving as the data storage engine and Django handling backend functionality. Configuring the environment, building Django apps, defining models, designing HTML and CSS pages, and navigating through views and URLs are all part of the implementation

process. The main, about, registration, and home pages all include comprehensive graphical user interfaces. Easy navigation and engagement are highlighted on the home page; safe login and sign-up procedures are provided on the registration page; code submission and report creation are the main page's focus points; and the platform's vision and goal are explained on the About page.

I completed my midpoint report, incorporated all the material I had worked on, and presented the contents of my application along with a demonstration of my completed work. For the time being, I'll keep working on my application, and if I run into any problems, I'll make sure to receive as much assistance as I can by consulting my supervisor.

January

Initiating my Django Project and Establishing some of the Functionalities
I started working with the Django web framework in January, concentrating on building a solid structure for a web application that would run on Python. My original objective was to use SQLite 3 to build an integrated database and user-centric system with safe authentication. The mentioned initial preparation created the framework for the ensuing stages of development and incorporated several crucial features.

The project setup and initial Configurations
I installed Django together with Python 3.12 and SQLite 3 as the database management system at the beginning of the project. Essentially because SQLite3 is straightforward to use and suitable for small to medium-sized applications it was a perfect fit for this project's scope as that why it was chosen and because I haven't worked with SQLite3 enough so I thought this would be interesting. I installed and configured Django according to the setup instructions from a tutorial I've watched on YouTube and this is how I was able to understand how this project was going to work, making sure that all dependencies were installed successfully and that the development environment was up and running.

Implementing User Registration and Login
The first tasks was to implement the user registration and login system, this was very important for my website application as it allowed for the personalized user experience and secure access control.

I configured my database on settings.py file in my project where the default is SQLite. After this I created the Django app where I added in on my terminal – python3 manage.py startapp codereview.

```
73  # Database
74  # https://docs.djangoproject.com/en/5.0/ref/settings/#databases
75
76  DATABASES = {
77      'default': {
78          'ENGINE': 'django.db.backends.sqlite3',
79          'NAME': BASE_DIR / 'db.sqlite3',
80      }
81  }
```

After this step I defined the models in models.py file where I added : from Django.db import models. For the home, about, and registration page html I created a simple html file in my

template folder where I also added my CSS to style my application with some help and inspiration my research.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Welcome to Code Pulse</title>
    {% load static %}
```

```
<body>
    <header>
        <h1>About Code Pulse</h1>
    </header>
    <main>
        <div class="nav-buttons">
            <a href="{% url 'register' %}">Register</a>
        </div>
    </main>
    <!-- Add about page content and styling -->

    <!-- Navigation back to home page -->

</body>
</html>
```

```
<form method="post">
    <h3>Login Here</h3>
    {% csrf_token %}
    <label for="username">Username</label>
    <input type="text" placeholder="Enter Username" id="username" name="username">

    <label for="password">Password</label>
    <input type="password" placeholder="Password" id="password" name="pass">

    <button type="submit">Log In</button>
    <a href="{% url 'login' %}">Create a account</a>
</form>

<div class="background">
    <div class="shape"></div>
    <div class="shape"></div>
</div>
```

I defined the view for the home page in the views.py - logic for handling HTTP request and generating responses.

```
# Create your views here.
def home(request):
    return render(request, 'home.html')

def about(request):
    return render(request, 'about.html')

def register(request):
    return render(request, 'register.html')
```

In order to navigate my pages I had to add them in the urls.py

```python
#urlconfig
urlpatterns = [
    path('home/', views.home, name='home'),
    path('about/', views.about, name='about'),
    path('register/', views.register, name='register'),


    path('', views.signup, name='signup'),
    path('home/', views.login,name='login')


]
```

My CSS to style and design my pages – navigation links, text fields for the user to register, and styles text and added buttons to my pages for all pages.

```css
<style>
    body {
        font-family: 'Arial', sans-serif;
        margin: 0;
        padding: 0;
        text-align: center;
        background-color: #f4f4f4;
    }

    header {
        background-color: #11192e;
        padding: 10px;
        color: white;
    }

    main {
        padding: 20px;
    }

    img {
        max-width: 100%;
        height: auto;
        margin-top: 20px;
    }

    .nav-buttons {
        margin-top: 20px;
    }

    .nav-buttons a {
        margin: 10px;
        padding: 10px 20px;
        text-decoration: none;
        background-color: #000000;
        color: white;
        border-radius: 5px;
    }
</style>
```

```css
    form * {
        color: #ffffff;
    }

    form h3 {
        font-size: 18px;
        font-weight: 500;
        line-height: 22px;
        text-align: center;
        margin-bottom: 20px;
    }

    label,
    input {
        display: block;
        margin-bottom: 15px;
    }

    input {
        height: 30px;
        width: 100%;
        background-color: rgba(255, 255, 255, 0.07);
        border-radius: 3px;
        padding: 0 5px;
        font-size: 14px;
        font-weight: 300;
    }

    ::placeholder {
        color: #e5e5e5;
    }

    button {
        width: 100%;
        background-color: #ffffff;
        color: #080710;
        padding: 10px 0;
```

Model configuration and database integration
Configuring the database models was important, as was user management. I developed a few models in models.py to efficiently describe the application's data structures using Django's ORM. This is an illustration of the CustomUser model, which builds upon the User model that comes with Django:

```python
from django.db import models
from django.contrib.auth.models import AbstractUser #, Group, Permission


class CustomUser(AbstractUser):
    # Your custom user model code
    pass
```

This customization allowed me to add additional attributes to the user model in the future, such as profile images or social media links, without modifying the core authentication mechanisms of Django.

It was difficult and interesting to build up the project, integrate a database, and create user authentication. I gained a lot of knowledge about Django's features, particularly its robust user management and database schema conversion instruments. The effective development of increasingly complicated features in the following months depended heavily on the initial setup. This stage helped me better grasp Django's design and equipped me for handling user sessions and user data security, two of the most challenging parts of web programming.

February

This month I made sure I understood everything and took more detailed notes. I also made sure I knew what I was doing, so when I came across the main page, I made sure to do as much research as I could because I was having trouble figuring out what I could research to help me detect vulnerabilities in my page. I started to wonder, "Do I really know how to do this?" and "Can I do this?"

I had a meeting with my supervisor, and I asked how I can make sure I can detect the vulnerabilities in my web application, and what was the right way to do it. My supervisor provided me with a link that could help me see if I could use a cross injector to detect XSS using python. I would need to include external libraries but you must also create solutions from scratch. Th whole idea is to take a URL, check if its valid using regular expressions, then analyse to check for vulnerabilities. As I have studied the resources provided by my supervisor, I have come to an understanding on how I can approach this and what to research.

I have chosen to use Bandit in this stage of constructing my secure code review application using Python, Django, and MySQL because of its specificity to Python and its ability to detect common security vulnerabilities. I just needed to use the pip command to install Bandit in our development environment, which made it very easy to implement. to accomplish an easy, automated security analysis as a component of the building process.

For the next month, I'll be focusing on developing my ability to spot user code patterns that could point to vulnerabilities. To locate XSS detection, for instance, I would need to search for Python libraries or programs capable of detecting XSS. Implementing a custom solution within a Django application would be an alternative strategy. This requires creating patterns that analyse user code for identified XSS patterns. I discovered that (CSP) Content Security Policy is a browser feature that limits which resources can be loaded onto my pages, hence assisting in the prevention of XSS attacks.

March

Implementing Vulnerability Scanners

I decided to take a more in-depth look at my Django application's security in March, concentrating mostly on vulnerability scanning and the scanner's general interface design. During the month, the focus was on developing features that would enable users to check

their inputs for vulnerabilities related to SQL injection and Cross-Site Scripting (XSS). This was an essential feature considering the high dangers connected to these security threats.

With my research given the high rate of SQL injection attacks and their potential to compromise sensitive data, the decision to include a SQL injection scanner resulted from a comprehensive approach to security. Based with this understanding, I set out to create an effective approach that would both identify and avoid SQL injection risks. While working on my scripts I can to a conclusion from both research and the example of the script I found in a resource my supervisor has provided. I found this to be very helpful since I was stuck on how to get this completed, it's still incomplete but I now know the understanding on how I can connect all of it.

Technical Implementation
I implemented two custom utility functions, detect_xss_vulnerability and detect_sql_injection, in the utils.py file. These functions used regular expressions to search for patterns typical of XSS and SQL injections within user-submitted code. For instance, the XSS detection function looked for suspicious patterns such as <script> tags or javascript: protocol usage which are common vectors for XSS attacks:

```
import re  # Import the regular expression module to enable regex operations

def detect_xss_vulnerability(code_input):
    # this is the Function to detect common patterns indicative of XSS (Cross-Site Scripting) attacks in a given code input.
    xss_patterns = [
        r'<script[^>]*>.*?</script>',            # <script> tags that could be used to insert malicious JavaScript.
        r'on\w+="[^"]*"',                        # HTML event handlers like onclick="..." that could execute JavaScript.
        r'javascript:',                          # usage of the "javascript:" protocol which can be used for XSS.
        r'%3Cscript%3E.*?%3C/script%3E',         # URL-encoded <script> tags which are a form of obfuscating XSS scripts in URLs.
        r'alert\(([^)]*)\)'                      # JavaScript alert() function calls that are often used in XSS proof-of-concept.

    ]

    # Initialize a list to store found vulnerabilities
    vulnerabilities = []

    # Search for XSS patterns in the input code, `re.findall` to search for matches in the code input
    for pattern in xss_patterns:
        matches = re.findall(pattern, code_input, re.IGNORECASE)  # The `re.IGNORECASE` flag makes the search case-insensitive
        if matches:  # If any matches are found, they are considered potential vulnerabilities
            vulnerabilities.extend(matches)  # Add all matches to the vulnerabilities list

    return vulnerabilities  # Return the list of vulnerabilities
```

The function looked for common SQL terms like SELECT, UNION, and INSERT that were abused in the supplied data in order to detect SQL injections:

```python
def detect_sql_injection(code_input):
    # Define SQL injection patterns to search for
    sql_injection_patterns = [
        r'\bselect\b.*?\bfrom\b',              # patterns that could indicate data querying, a common SQL injection tactic.
        r'\binsert\b.*?\binto\b',              # patterns that could indicate data insertion.
        r'\bupdate\b.*?\bset\b',               # patterns that could indicate updating existing data.
        r'\bdelete\b.*?\bfrom\b',              # patterns that could indicate deleting data.
        r'\bdrop\b.*?\btable\b',               # patterns that could indicate dropping (deleting) entire tables.
        r'\btruncate\b.*?\btable\b',           # patterns that could indicate truncating (clearing) tables.
        r'\bunion\b.*?\bselect\b',             # patterns that could involve using UNION to perform additional queries.
        r'\bexec\b',                           # patterns that could indicate execution of stored procedures.
        r'\bcreate\b.*?\btable\b',             # patterns that could indicate creating new tables.
        r'\balter\b.*?\btable\b',              # patterns that could indicate altering the structure of tables.
        r'\bgrant\b.*?\bprivileges\b'          # patterns that could indicate attempts to grant database privileges.
    ]

    # Initialize a list to store any vulnerabilities found
    vulnerabilities = []

    # Iterate over each pattern and search for matches in the code input
    for pattern in sql_injection_patterns:
        matches = re.findall(pattern, code_input, re.IGNORECASE)  # Search is case-insensitive
        if matches:  # If matches are found, they are considered potential vulnerabilities
            vulnerabilities.extend(matches)  # Add all matches to the vulnerabilities list

    return vulnerabilities  # Return the list of vulnerabilities
```

These methods apply particular patterns that are intended to match popular web-based attack vectors.
XSS Attacks: Searching for the javascript: protocol, event handlers (on* attributes), and <script> tags are common ways that cross-site scripting (XSS) attacks are carried out.
SQL Injection: To identify SQL query manipulation, which may change database operations and perhaps damage or leak data, patterns are created.
Vulnerability Detection: Every function looks for these patterns in the input it receives, which is usually obtained from user input forms or API queries. Potential vulnerabilities are indicated by any matching.
Case Sensitivity: Although SQL and HTML/JavaScript are case-insensitive, using re.IGNORECASE guarantees that the detection is not impeded by the case of the input text.

Design and the User Interface
Users may submit their code and obtain feedback on any security issues on the accessible scanning page. Users may enter their code into the straightforward form supplied by the HTML template scanner.html, and the backend processes would handle the rest:

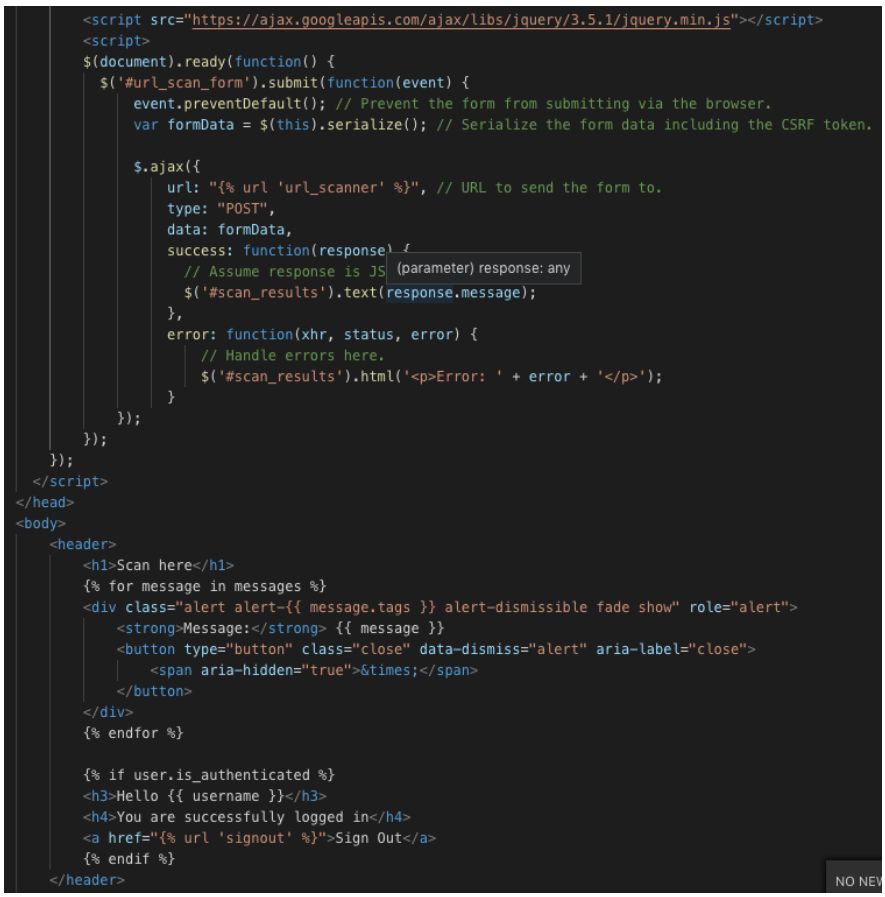
```html
<main>
    <section>
        <h2>URL Scan</h2>
        <form id="url_scan_form" action="{% url 'url_scanner' %}" method="post">
            {% csrf_token %}
            <input type="text" name="url_input" id="url_input" placeholder="Enter website URL" required>
            <button type="submit">Scan URL</button>
        </form>
        <!-- Placeholder for the scan results -->
        <div id="scan_results"></div>
    </section>

    <section>
        <h2>Code Scan</h2>
        <form method="post" action="{% url 'scanner' %}">
            {% csrf_token %}
            <label for="code_input">Enter Your Code :</label><br>
            <br>
            <textarea id="code_input" name="code_input" rows="10" cols="50"></textarea><br>
            <br>
            <button type="submit">Generate Scanner</button>
        </form>
        {% if vulnerability_message %}
            <p>{{ vulnerability_message }}</p>
        {% endif %}
    </section>
</main>
```

The view function scanner processed the request once the form was submitted, used the utility functions to find vulnerabilities, and sent the findings to the user. The frontend and backend's connection was important in giving users real-time feedback and improving the application's security and usability.

URL scanner integration
I included an AJAX-based URL scanner in my program to offer a smooth and engaging user interface. To do this, I had to create a complex AJAX setup in my Django template so that form submissions could be handled independently.

JavaScript and jQuery: To minimize page reloads, I utilized jQuery to handle the form submission event and prevent the default submission. To ensure that all required security tokens were included in the AJAX request, the form data, including Django's CSRF token, was serialized using jQuery's.serialize() function.
AJAX Request: The AJAX call was set up to send information to the url_scanner Django view, which handled the user-inputted URL. For the purpose of making sure the request was routed appropriately inside the application's URL setup, I used Django's {% url %} template tag to dynamically build the submission URL.
Backend Processing: The serialized data was received by the url_scanner view on the backend, which then deserialized it and carried out the required scans to find vulnerabilities. After that, the JSON-formatted results were returned to the frontend.



The application's primary interface included an easy-to-use design that integrated the scanner. It included a straightforward input form with a submit button that started the AJAX call, allowing users to specify the URL they wished to scan. A results display section was dynamically updated below the input field with messages from the backend explaining the scan findings and any faults that were found. Both aspects of design and functionality have to be carefully considered while integrating this scanner into the program. My goal was to create a scanner that was both user-friendly and effective, matching the application's general style while offering strong functionality.

Working on the URL scanner was an interesting one that improved my knowledge of web development's front-end and back-end interactions. As I gained more experience with asynchronous web development, I also became more knowledgeable with Django's AJAX request handling features. This addition strengthened the application's interactivity and security, making it a more useful tool for users. Furthermore, the significance of user feedback systems in web applications was recognized during this development period. I increased the transparency and reliability of the application by ensuring that users were not kept in the dark about what was going on behind the scenes by introducing rapid reaction displays for the scanner findings.

April

Working on Two-Factor Authentication and Email Verification
Implementing two-factor authentication (2FA) via email verification, an important feature to improve user security, showed difficult for me in April. The user's registered email address was used to send a randomly generated code, which they had to input on the website to finish registering. Making sure email communication was safe and did not compromise user credentials was a key aspect of this approach. I choose to use an app-specific password instead of my main email password in order to do this. By configuring the security settings of my email provider, I was able to create a unique password that I needed to access the email service from my Django application. By separating my application's codebase from my personal credentials, this method significantly improved security.

For the technical implementation the “send_verification_email” function in views.py, which created and sent an email with the verification code, served as a centre of functionality. This function sent emails safely and effectively by making use of Django's email features:

```python
# Function to send a verification email with SMTP protocol.
def send_verification_email(user, verification_code):
    if not user.email:
        logger.error("No email address provided for user.")
        return False

    try:
        message = f'Your verification code is: {verification_code}'
        msg = MIMEText(message, 'plain', 'utf-8')  # this is for Specify the format and encoding explicitly
        msg['Subject'] = 'Verify Your Email'
        msg['From'] = 'djenasiabdellah26@gmail.com'
        msg['To'] = user.email

        # Logging the email details to ensure correctness
        logger.info(f"Email details: From: {msg['From']}, To: {msg['To']}")

        # this is to set up the SMTP server
        s = smtplib.SMTP('smtp.gmail.com', 587)
        s.starttls()  # Start TLS for security & Encrypt connection for security.
        s.login('djenasiabdellah26@gmail.com', 'ssvvycowriavoewj')  # Log into the email server.
        s.sendmail(msg['From'], [msg['To']], msg.as_string()) # Send the email.
        s.quit() # this is to terminate the connection to the server.
        logger.info(f"Email sent to {user.email} with verification code {verification_code}")
        return True
    except Exception as e:
        logger.error(f"Failed to send email to {user.email}: {e}")
        return False
```

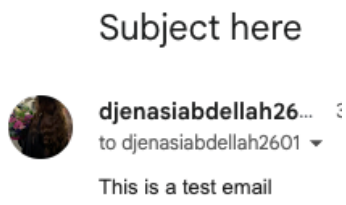
The user receives an email with a verification code from this method. It sends the email over SMTP and captures the process for troubleshooting.

Testing and Troubleshooting

At first, there were difficulties with integrating email capabilities, particularly with the aim of addressing any mistakes and ensuring dependable delivery. I used Django's logging system, which was configured to record comprehensive logs of the email transmission process, to troubleshoot these problems. This was really helpful in finding and fixing problems with the network and SMTP setup. For example, logs enabled me to identify timeouts and authentication difficulties, which I subsequently fixed by modifying the SMTP settings or enhancing the application's error handling.

Additionally, I used Django's shell to test the email sending capability before completely integrating this system into the live environment. This was essential for improving the configuration because it allowed me to send emails interactively and view results and mistakes right away:

```
>>>
>>> import smtplib
>>> from email.mime.text import MIMEText
>>> def send_email_test():
...     msg = MIMEText('This is a test email')
...     msg['Subject'] = 'Subject here'
...     msg['From'] = 'djenasiabdellah26@gmail.com'
...     msg['To'] = 'djenasiabdellah2601@gmail.com'
...     try:
...         s = smtplib.SMTP('smtp.gmail.com', 587)
...         s.starttls()
...         s.login('djenasiabdellah26@gmail.com', 'yxwhxbhcrrcmxmvt')
...         s.send_message(msg)
...         s.quit()
...         print("Email sent!")
...     except Exception as e:
...         print(f"Email sending failed: {e}")
...
>>> send_email_test()
Email sent!
>>>
```



I made significant undertaking improvement to my Django application's security by using 2FA. It required not just technical implementation but also strategic choices about email security, such creating passwords unique to each app. The difficulties I encountered, especially with regard to SMTP settings and error management, helped me get a better understanding of network connections and the significance of logging in order to simplify troubleshooting. Through this method, I was able to enhance not just the security of the program but also my ability to manage dependable, secure email interactions inside the framework of a web application.

As for Testing my Django website application, I will work on that as I have put together an in-depth testing plan that includes end-user acceptability testing, integration testing, and unit testing. This is a complete explanation of how I approached each testing phase, along with the tools and frameworks I used. I will go over all of this in my final documentation, display it on my project poster, and go over it in my next presentation.
Furthermore, the significance of user feedback systems in web applications was shown during this development stage and I have yet to keep working on my testing approaching the final week of the project deadline.