Bobby School of Cyber Security

Summer Term 2025

Arundas Mohandas, Ganga Sunil and Sai Bhargav Thoomu

October 19, 2025

Contents

1 Introduction

The importance of secure software systems has grown significantly in recent years, as vulnerabilities in applications continue to be a leading cause of data breaches, financial loss, and erosion of user trust. Despite this, many applications are still developed with insufficient attention to security principles, leading to exploitable flaws that could have been avoided through awareness and secure coding practices.

The primary aim of this project is to design and implement a deliberately insecure web-based system, and subsequently use it as an educational tool to demonstrate common security flaws and their impact. By first exposing students to insecure implementations, the project emphasizes the importance of security in the software development life cycle and builds awareness of how seemingly minor oversights can result in severe vulnerabilities.

To introduce the concept of insecure systems in a relatable manner, the well-known *Bobby Tables* cartoon was presented as a starting point. This example highlights the risks of improper input validation and serves as a memorable entry point to the broader field of software security.

For the literature review and hands-on exploration, the project made use of the *OWASP Juice Shop* platform, which is widely recognized as one of the most comprehensive and intentionally vulnerable web applications available for educational purposes. Students were instructed to attempt exploitation of its vulnerabilities by relying only on the hints provided, rather than directly consulting the solutions. This approach encouraged independent problem-solving, critical thinking, and a deeper understanding of how security weaknesses can be identified and exploited in practice.

Building upon these foundations, the project focuses on developing an insecure web-based *School Management System* with three distinct configurations, each demonstrating a different stage of software security. This implementation allows for a comparative analysis of vulnerabilities, mitigations, and secure development practices.

System 1 represents the *fully Vulnerable Configuration*, which intentionally includes common web security flaws such as CAPTCHA token reuse, unrestricted voting mechanisms, and unfiltered PDF uploads containing hidden metadata. This version serves as a controlled environment to observe and understand real-world exploitation scenarios.

The two non vulnereble systems, the *Non-Vulnerable Configuration I* and *Non-Vulnerable Configuration II*, introduces key defensive measures aimed at mitigating these specific vulnerabilities. These enhancements demonstrate how targeted mitigations improve overall application robustness.

By progressing through these three configurations, the project provides practical insight into how layered security mechanisms can prevent exploitation. Through this foundation, the project lays the groundwork for developing a school management system that demonstrates both insecure and secure

implementations. The insecure version serves as a practical learning environment, while the secure versions highlights industry-recommended best practices, thereby bridging the gap between theory and application in secure software engineering.

2 Related Work

The foundation of this project draws heavily from the **OWASP Juice Shop**, an intentionally vulnerable web application developed and maintained by the Open Web Application Security Project (OWASP). Juice Shop is widely recognized as one of the most comprehensive training platforms for web application security, as it incorporates vulnerabilities that map directly to the **OWASP Top 10** categories, as well as numerous other security flaws commonly encountered in real-world systems. It has been widely used in security education and training, offering a realistic, hands-on environment for us to explore and understand the impact of vulnerabilities.

In this project, the Juice Shop served not only as a reference but also as a practical learning tool. We were explicitly instructed by the course instructors to read each challenge, study the provided hints, and attempt exploitation without consulting the official solutions. This pedagogical constraint required learners to invest significant time in trial-and-error, experimentation, and collaborative problem-solving. Through this process, participants developed a deeper conceptual understanding of the vulnerability classes they encountered (for example, sql injection, IDOR, XXE, and security misconfiguration), and gained practical skills in reconnaissance, exploitation techniques, and evidence collection. The deliberate emphasis on working through challenges from hints fostered perseverance, improved debugging skills, and a stronger mental model of how real-world attackers probe and exploit application weaknesses.

Through this engagement, we encountered a wide range of vulnerabilities, including *SQL injection*, *broken authentication*, *insecure direct object references (IDOR)*, *security misconfigurations*, and *XML External Entity (XXE) attacks*. These exercises provided both theoretical grounding in web security concepts and practical experience in identifying and mitigating risks, bridging the gap between abstract security principles and real-world application.

Beyond the Juice Shop, prior research emphasizes the importance of staged or multi-level educational environments for learning secure software development. Studies have shown that learners gain deeper comprehension when exposed first to vulnerable systems, followed by guided mitigation exercises that illustrate the effect of security improvements. In line with this, the project implements three distinct configurations: **System 1**, the vulnerable system, serves as a baseline for observing exploitation; **System 2** and **System 3**, the partially secured system, demonstrates the impact of resolving selected vulnerabilities; This staged approach allows us to directly observe the effect of security interventions on system behavior and resilience, reinforcing lessons about secure coding and access control.

The project also aligns with other educational initiatives and platforms that combine vulnerability exploration with remediation, such as WebGoat and Hackademic, while contributing a distinct emphasis on comparative anal-

ysis across multiple system versions. By intentionally implementing and then progressively hardening the same application, the **Bobby School of Cyber Security** enables learners to observe not only how vulnerabilities are exploited, but also how targeted and layered mitigations alter the attack surface and system behavior.

By incorporating the OWASP Juice Shop methodology and enforcing a challenge-driven learning model, this project contributes to security education by combining hands-on exploitation, focused mitigation development, and reflective analysis. The approach emphasizes both technical skill acquisition and the development of security reasoning — preparing us to apply secure software engineering practices in real-world development and assessment contexts.

3 School Management System Architecture

The School Management System is a Django-based web application designed to manage students, teachers, classes, timetables, and announcements. The system implements role-based access control, where each participant (Admin, Teacher, Student) interacts with the system through a web interface. All communication occurs over standard HTTP/HTTPS, with JSON used for asynchronous operations such as voting on announcements. The system leverages Django sessions to maintain authentication and authorization state.

The architecture is modular, separating concerns into dedicated Django apps and views. Each module handles specific functionality, such as student registration, timetable management, and announcements. Data is stored in a relational database (e.g., SQLite, PostgreSQL) using Django's ORM. Security-critical modules, such as CAPTCHA validation and file upload handling, incorporate measures to prevent automated attacks, although some vulnerabilities are intentionally included for learning purposes.

3.1 Roles and Participants

3.1.1 Administrator (Admin)

The Admin is responsible for managing the overall school system. Admins have the highest level of access and can perform the following actions:

- Add, edit, or delete subjects, teachers, classrooms, and timeslots.
- Generate the class timetable based on teacher availability, subjects, and class requirements.
- View all timetable entries with filters for class, subject, teacher, and day.
- Manage system-level configurations and monitor data integrity.

The Admin interacts with the system via the web interface and is authenticated using the `Admin_login` session decorator. All requests are verified against the session to ensure authorized access.

3.1.2 Teacher

Teachers are responsible for delivering lessons and interacting with student-related data. Their interactions include:

- Viewing their assigned timetable entries for specific days.
- Accessing subjects and class information relevant to their teaching assignments.

- Posting announcements that students can view and vote on.
- Uploading documents or resources, with file type and size validations.

Teachers are authenticated using the `Teacher_login` session decorator. They only see data related to their assignments, ensuring separation of concerns and data privacy.

3.1.3 Student

Students are the end users who consume educational content and interact with the system. Student actions include:

- Registering via a web form with CAPTCHA validation to prevent automated submissions.
- Viewing their class timetable and filtering entries by day.
- Viewing teacher announcements and voting via upvote/downvote mechanisms.
- Tracking personal information, class level, and timetable assignments.

Students are authenticated using the `Student_login` session decorator. Access is restricted to data relevant to their class and assignments.

3.2 Workflow and Data Flow

The system operates in the following structured workflows:

- **Registration:** Students and Teachers submit registration forms, validated with CAPTCHA and optional file uploads. The system stores validated data in the corresponding `StudentReg` or `TeacherReg` tables.
- **Authentication:** Upon login, the system sets session variables to identify the user role (Admin, Teacher, Student) and restrict access to views accordingly.
- **Timetable Generation:** Admin adds subjects, teachers, rooms, and timeslots, then generates the timetable using an algorithm that assigns teachers and rooms to periods. Conflicts are detected and reported.
- **Timetable Viewing:** Students and Teachers fetch timetable entries filtered by their class or assigned schedule. Queries are optimized using Django ORM with related objects.
- **Announcements:** Teachers post announcements. Students view announcements and cast votes, which are tracked in the `AnnouncementVote` table. JSON is used for asynchronous vote updates.

- **Data Integrity and Security:** The system enforces session-based access control. Modules implement additional validation (CAPTCHA, file type checks). Certain vulnerabilities are intentionally included for educational purposes (CAPTCHA replay, duplicate voting).

3.3 System Communication and Integration

All modules communicate internally through Django's ORM and request-response cycle. Asynchronous operations, such as voting on announcements, use AJAX with JSON payloads. Session management ensures role-specific access control.

- Admin, Teacher, and Student communicate via HTTP requests to Django views.
- JSON is used for client-server interactions in interactive features (e.g., voting, timetable updates).
- ORM queries ensure referential integrity across models like `Student`, `Teacher`, `TimetableEntry`, and `AnnouncementVote`.
- Security-sensitive operations, including registration and voting, are logged and validated against session credentials.

3.4 Security Considerations

The system-level security considerations include:

- Role-based access control using session decorators (`Admin_login`, `Teacher_login`, `Student_login`).
- Input validation for forms and file uploads.
- CAPTCHA validation for automated bot prevention (with intentional replay vulnerability for study purposes).
- Vote integrity concerns, as duplicate votes are currently possible (educational vulnerability).
- Optimized ORM queries to prevent unintentional data leaks.

4 Implementation

In this chapter, the implementation of the Django-based School Management System is briefly summarized and explained. Detailed annotations to the code are provided as comments directly in the source files and are therefore not repeated here. Instead, the general concepts, design choices, and module-level implementations are highlighted and discussed.

4.1 Django School Management System Implementation

The implementation of the Django School Management System focuses on delivering a modular web application for managing students, teachers, classes, timetables, and announcements. The system emphasizes role-based access control, ensuring that Administrators, Teachers, and Students interact with the application according to their permissions. The core of the implementation is based on Django's MVC architecture, augmented with third-party libraries for CAPTCHA validation, data parsing, and UI enhancements.

The project is organized into multiple Django apps, each responsible for a specific module:

- `login`: handles user authentication and session management.
- `students`: manages student registration, profiles, and applications.
- `teachers`: handles teacher registration, profiles, and timetable assignments.
- `timetable`: generates and displays class schedules.
- `announcements`: manages posting, viewing, and voting on announcements.

4.1.1 Setup and Environment Requirements

The project requires Python 3.10+, Django 4.2+, and the following key third-party libraries:

- `django-simple-captcha` for CAPTCHA verification on registration forms.
- `pycountry` to populate nationality choices dynamically.
- `Pillow` for handling profile photo uploads.
- `django-crispy-forms` for form rendering and UI improvements.

Database configuration uses SQLite by default, but PostgreSQL is supported for production. All migrations must be applied in order, and a superuser must be created for admin-level access. The application runs on the built-in Django development server for testing purposes, or via Gunicorn/NGINX in production.

4.1.2 Differences from Standard Django Conventions

While the system follows Django's standard MVC patterns, several deviations were introduced to meet the learning objectives:

- Custom session decorators were implemented (`session_required`) to enforce role-based access.
- Certain security vulnerabilities were intentionally included for educational purposes:
  - IDOR vulnerability in the Teacher profile view.
  - CAPTCHA reuse detection demonstration with a limited time window.
  - Duplicate voting on announcements.
- Asynchronous interactions, such as upvoting announcements, are handled via AJAX with JSON payloads instead of traditional page reloads.
- Some forms include read-only fields for sensitive data, visually styled to indicate non-editable state.

4.1.3 Supported Functionality and Scope

The system implements the following key features:

- **Admin:** register new users, approve/reject student and teacher applications, generate class timetables, post announcements, and manage overall system configurations.
- **Teacher:** view assigned timetable, post announcements, upload teaching documents, and update personal profile (with optional photo/document upload).
- **Student:** submit applications, view timetables, access teacher announcements, vote on announcements, and manage personal profile.
- **Security and Validation:** session-based role verification, CAPTCHA validation, input validation on forms, and file type/size checks.

- **Error Handling and Logging:** exceptions during profile updates, file uploads, and downloads are logged to the server console for monitoring.

Features partially implemented or left for future work include:

- Automated email notifications for student approval or announcement updates.
- Multi-class timetable optimization algorithms.
- Comprehensive audit logging for all user actions.

4.1.4 Running the System and Known Issues

To run the system locally:

1. Clone the repository and navigate to the project folder.
2. Install dependencies via `pip install -r requirements.txt`.
3. Apply migrations: `python manage.py migrate`.
4. Create a superuser for administrative access: `pythonmanage.pycreatesuperuser`.
5. Start the development server: `python manage.py runserver`.

Known limitations and issues include:

- CAPTCHA reuse is detected only within a limited time window.
- File uploads for photos and documents are limited by default media folder settings; larger files may fail.
- Duplicate voting on announcements is possible due to intentional design.
- Certain views may raise exceptions if session variables are missing or malformed; these are caught and logged, but some edge cases may result in user-facing errors.
- No automated email or notification system has been implemented; notifications rely on manual review.

4.2 Assignment Management Module

The `assignment.py` module implements the core functionality for handling assignments within the school application. It supports both teacher and student workflows: teachers can create questions, review submissions, and grade them, while students can view available questions, submit assignments, and track their submission history. The main code module is located at */login/assignment.py*.

In order to deliberately demonstrate insecure software practices, this module introduces an exploitable XML parsing routine. The function `parse_xml()` employs the `lxml` parser with `load_dtd` and `resolve_entities` enabled, which makes the system vulnerable to XML External Entity (XXE) attacks such as the "Billion Laughs" denial-of-service vector. This vulnerability highlights how insecure handling of user-uploaded files can compromise system integrity.

4.2.1 Dependencies

The module depends on multiple Django components and third-party libraries:

- `django.shortcuts` for rendering templates and managing redirects.
- `django.contrib.messages` for user-facing notifications.
- `django.db.models.Q` for complex query filtering.
- `django.utils.timezone` for timestamp management.
- `lxml.etree` for XML parsing (insecurely configured).
- Custom models from *login.models*: `Student`, `TeacherAvailability`, `ClassSection`, `Subject`, `AssignmentQuestion`, `AssignmentSubmission`.
- Custom forms from *login.forms*: `AssignmentQuestionForm`, `AssignmentSubmissionForm`, `GradeSubmissionForm`, `SubmissionFilterForm`.

4.2.2 Session Decorator

The decorator `session_required` enforces role-based access control by checking session keys. It ensures that only authenticated users with the correct role (teacher or student) can access assignment functionality. Invalid sessions are flushed, and the user is redirected to the application index.

4.2.3 Main Functions

The module contains several core functions, which can be grouped by teacher and student responsibilities:

Teacher Workflows

- **Create Question** (`teacher_create_question`): Allows teachers to post questions restricted to their assigned subjects and class sections.
- **List Questions** (`teacher_questions_list`): Displays all questions authored by the teacher.
- **Review Submissions** (`teacher_review_submissions`): Provides filtering by subject, class section, and status. Implements ordering by submission time.
- **Mark Seen** (`teacher_mark_seen`): Updates submission status to `seen`.
- **Grade Submission** (`teacher_grade_submission`): Assigns marks to a student submission and updates its status to `graded`.

Student Workflows

- **List Questions** (`student_questions_list`): Shows available questions for a student's class and subjects.
- **Submit Assignment** (`student_submit_for_question`): Enables students to upload XML files or typed content as assignment answers. Contains the XXE vulnerability in the XML parsing routine.
- **Track Submissions** (`student_my_submissions`): Lists a student's past submissions with their current status and marks.

4.2.4 Security Weakness

This module intentionally demonstrates insecure coding practices. The XML parsing routine is configured with both `load_dtd` and `resolve_entities` enabled, making the application susceptible to XML External Entity (XXE) attacks. An attacker could upload a malicious XML file to perform denial-of-service or exfiltrate server-side files. This highlights the importance of secure file-handling practices in web applications.

4.3 Announcement Management Module

The *announcement.py* module manages announcements in the school application. It enables teachers to post announcements for their assigned subjects and class sections, while students can view announcements relevant to their enrolled classes. The main code module is located at */login/announcement.py*.

This module also demonstrates insecure coding practices by introducing a Cross-Site Scripting (XSS) vulnerability. User-submitted announcement

content is not properly sanitized, allowing malicious users to inject scripts into announcements, which can then be executed in the browser of other users.

4.3.1 Dependencies

The module depends on several Django components and custom models/forms:

- `django.shortcuts` for rendering templates and managing redirects.
- `django.contrib.messages` for user-facing notifications.
- `django.utils.timezone` for handling announcement timestamps.
- Custom models from *login.models*: `Announcement`, `TeacherAvailability`, `Student`, `ClassSection`, `Subject`.
- Custom forms from *login.forms*: `AnnouncementForm`, `AnnouncementFilterForm`.

4.3.2 Session Decorator

The decorator `session_required` enforces role-based access control. It ensures that only authenticated teachers can create announcements and only authenticated students can view them. If the session is invalid, it is flushed, and the user is redirected to the index page.

4.3.3 Main Functions

Teacher Workflows

- **Create Announcement** (`teacher_create_announcement`): Allows teachers to post announcements for their assigned subjects and class sections. The function validates that teachers cannot post outside of their scope.
- **List Announcements** (`teacher_announcements_list`): Displays announcements created by the currently logged-in teacher, ordered by creation time.

Student Workflows

- **View Announcements** (`student_announcements_list`): Shows announcements relevant to a student's enrolled class section and subjects. Announcements are retrieved from the database, optionally filtered by subject, and displayed in reverse chronological order.

4.3.4 Security Weakness

Announcement Vote Tampering

This module introduces an intentional security weakness, improper input sanitization leading to Cross-Site Scripting (XSS). Malicious users could craft announcements containing embedded JavaScript code, which would then execute in the browsers of other users viewing the announcement. This highlights the risks of failing to escape or sanitize user-generated content in web applications. Announcement vote tampering occurs when the application relies on client-side controls to enforce a one-vote-per-user policy and fails to enforce the same rule on the server. Without server-side uniqueness checks, an attacker can submit repeated votes for the same announcement from the same account, inflating vote counts and corrupting the integrity of feedback mechanisms.

Exploit In the vulnerable configuration the frontend disabled voting buttons after a single vote, but the backend did not validate whether a vote from the same user on the same announcement already existed. An attacker can:

1. Log in as a valid student and cast one vote via the UI.
2. Capture the POST request to the vote endpoint using browser DevTools or a proxy.
3. Replay the captured request multiple times, or script repeated POSTs via a small JavaScript loop with the same *announcement_ id* and *vote_ type*.
4. Observe that each request increments the announcement's vote count, producing an abnormally high total and misleading popularity metrics.

This attack undermines integrity and trust in the announcement system, and it can also be used to bias outcomes or enable social engineering.

Possible Mitigations for the vulnerebility

- **Server-side uniqueness check:** The vote handler can checks for an existing vote record for the requesting user and announcement before inserting. If a record exists, the server either updates the existing vote or returns a suitable response indicating the user has already voted.
- **Database-level constraint:** A composite unique constraint can be added to the votes table. This ensures duplicate inserts are rejected at the database level even under concurrency.

- **Authentication and CSRF protection:** The vote endpoint requires an authenticated session and validates CSRF tokens to prevent cross-site request forgery or forged POSTs.
- **Rate limiting and throttling:** Per-user and per-IP rate limits need to be enforced on the voting endpoint to prevent rapid automation or replay floods.
- **Audit logging and monitoring:** All vote attempts (successful and rejected) should be logged with user id, announcement id, timestamp and request fingerprint. Alerts are configured for abnormal voting patterns.
- **Client feedback and idempotency:** The API returns clear, idempotent responses so clients can handle state correctly without relying on disabled UI elements.

PDF Metadata and Hidden Annotation Exposure

PDF Metadata and Hidden Annotation Exposure occurs when uploaded PDF documents retain embedded metadata or annotations that are not removed before the file is served. These hidden attributes can leak sensitive information about authorship, internal comments, or workflow history even when the visible document content is benign.

Exploit In the vulnerable configuration teachers could upload PDF attachments to announcements and the application served the original PDF unchanged. An attacker (or any user with access to the announcement) can:

1. Locate the PDF URL from the announcement (for example via the `data-pdf-url` attribute or direct link).
2. Fetch the PDF using a browser, `pdf.js`, Postman, or a PDF reader.
3. Read embedded metadata via `pdf.js` API or extract it with tools (e.g., `exiftool`, `pikepdf`) and enumerate annotations/comments by inspecting page annotations.
4. Discover sensitive attributes such as author names, internal comments, timestamps, and document revision history information that may expose identities, internal notes, or development traces.

This compromises confidentiality and privacy and may reveal operational details useful for social engineering or further attacks.

Possible Mitigations for the vulnerebility

- **Server-side metadata stripping:** During upload, all PDFs need to be processed server-side to remove embedded metadata and annotations. By integrating libraries such as `pikepdf` or `PyPDF2` to open the PDF, clear the `/Info` dictionary and remove annotation objects, and then write a sanitized copy.
- **Sanitization pipeline:** Implement a multi-stage pipeline that validates MIME type and file size, converts the PDF to a canonical form (render-to-PDF or linearize) if necessary, strips metadata and embedded files, and performs an integrity check on the sanitized output before storage.
- **Immutable sanitized storage:** Only the sanitized PDF is stored and served; the original uploaded file is not kept or is stored transiently in a secure quarantine area pending sanitization and then securely deleted.
- **Secure storage and delivery:** Sanitized files should be stored under randomized filenames in a directory outside the web root and are delivered via authenticated endpoints that enforce access control.
- **Access control:** Access to announcement attachments is enforced by server-side authorization checks, only authorized users (e.g., enrolled students) may download attachments.
- **Automated scanning and logging:** Uploaded PDFs should be scanned for suspicious embedded objects (JavaScript, embedded files) and for residual metadata; events are logged and flagged for manual review if sanitization fails or suspicious content is detected.

4.4 Dashboard Management Module

The `dashboard.py` module implements the role-based dashboards of the school application. It defines separate dashboards for administrators, teachers, and students, with each view presenting functionalities specific to the logged-in role. The main code module is located at */login/dashboard.py*.

This module demonstrates secure session validation practices by ensuring that each dashboard can only be accessed by an authenticated user of the correct role. Although no deliberate vulnerabilities are embedded in this file, misconfiguration of role checks or session handling could potentially expose sensitive functionalities.

4.4.1 Dependencies

The module depends on the following Django components:

- `django.shortcuts` for rendering templates and handling redirects.
- `django.views.decorators.cache.never_cache` to prevent cached dashboard responses.
- `functools.wraps` to preserve metadata when applying decorators.

4.4.2 Session Decorator

The decorator `session_required` validates that the user session contains the correct role key (`Admin_login`, `Teacher_login`, or `Student_login`). If the session is invalid, it is flushed and the user is redirected to the application index. This enforces strict role-based access control to dashboards.

4.4.3 Main Functions

Administrator Dashboard

- **Admin Dashboard** (`admin_dashboard`): Provides access to administrative features such as managing timetables, approving users, viewing logs, managing pinboards, and profile management.

Teacher Dashboard

- **Teacher Dashboard** (`teacher_dashboard`): Offers functionality for teachers including posting questions, reviewing submissions, managing marks, viewing timetables, posting announcements, managing pinboards, and accessing their profile.

Student Dashboard

- **Student Dashboard** (`student_dashboard`): Allows students to view marks, access questions and submissions, manage profiles, interact in Q&A forums, view announcements and timetables, and use pinboards.

4.4.4 Security Considerations

The module itself does not introduce explicit vulnerabilities. However, because dashboards aggregate access to critical features, any weakness in the session validation mechanism could expose privileged functionality to unauthorized users. Ensuring secure role checks and proper session handling is essential.

4.5 Error Management Module

The `errormanagement.py` module centralizes error handling and session validation across the school application. It provides utilities for rendering templates safely and enforcing robust role-based session management. The main code module is located at */login/errormanagement.py*.

By introducing structured error handling, this module reduces the likelihood of uncaught exceptions propagating to end users, and improves resilience against faulty templates, database errors, or session misuse.

4.5.1 Dependencies

The module depends on both Django core components and Python's standard logging framework:

- `django.shortcuts` for rendering templates and managing redirects.
- `django.http.HttpResponseServerError` for returning server error responses.
- `django.core.exceptions.SuspiciousOperation` for handling invalid requests.
- `django.db.DatabaseError` for handling database-related failures.
- `django.contrib.messages` for user-facing notifications during session failures.
- `django.views.decorators.cache.never_cache` to prevent caching of session-sensitive views.
- `django.template.TemplateDoesNotExist` for catching template resolution errors.
- Python's `logging` module for structured error logging.

4.5.2 Safe Rendering

The function `safe_render()` wraps Django's `render()` to provide fault-tolerant template rendering. It catches specific exceptions:

- **TemplateDoesNotExist**: Logs the error and returns a generic server error response.
- **Unhandled Exceptions**: Attempts to render a fallback `errors_500.html` template with a user-friendly message.

If rendering the fallback template also fails, a plain `HttpResponseServerError` is returned.

4.5.3 Session Decorator

The decorator `session_required` extends the role-based access control mechanism by including error handling for:

- **Expired or Invalid Sessions**: Flushes the session, notifies the user, and redirects to the index page.
- **SuspiciousOperation**: Logs the incident as a warning, resets the session, and forces re-authentication.
- **DatabaseError**: Logs the exception and renders an error page indicating a database failure.
- **Other Exceptions**: Logs the error and displays a generic fallback message.

This approach ensures that critical issues are caught gracefully while preserving security and user awareness.

4.5.4 Security Considerations

This module strengthens the overall security posture of the system by:

- Preventing sensitive exception traces from being exposed to end users.
- Handling suspicious sessions proactively to reduce session fixation and tampering risks.
- Logging all unexpected errors for post-incident investigation.

Although no deliberate vulnerability is introduced here, improper configuration of error messages or logging could still leak sensitive information if not handled correctly.

4.6 Error Handler Module

The `error_view.py` module defines custom error handlers that provide user-friendly responses for common HTTP error conditions. It replaces default Django error pages with tailored templates that display meaningful messages to end users. The main code module is located at */login/error_ view.py.*

This improves the user experience by avoiding exposure of raw error traces and ensures that system errors are logged for developers.

4.6.1 Dependencies

The module relies on:

- `django.shortcuts.render` for rendering error templates.
- `django.http.HttpResponseServerError` for returning server error responses when rendering fails.
- `django.conf.settings` for toggling debug-mode messages.
- Python's `logging` module for capturing error details during rendering failures.

4.6.2 Safe Error Rendering

The helper function `_safe_error_render()` ensures resilience during error page rendering:

- Attempts to render the requested error template with a user-friendly message.
- Falls back to the `login/errors_500.html` template if rendering fails.
- As a last resort, returns a minimal `HttpResponseServerError`.

This layered approach guarantees that a response is always provided, even if templates are missing or misconfigured.

4.6.3 Custom Error Views

The module defines four handlers for common HTTP error codes:

- **400 Bad Request** (`custom_bad_request_view`): Displays a message when the client request is malformed.
- **403 Forbidden** (`custom_permission_denied_view`): Alerts the user that they lack permission to access the requested resource.
- **404 Not Found** (`custom_page_not_found_view`): Returns a user-friendly page when the requested resource does not exist.
- **500 Internal Server Error** (`custom_server_error_view`): Provides a fallback error page for unhandled server-side exceptions.

4.6.4 Security Considerations

This module enhances security by:

- Preventing internal server details and stack traces from being exposed to end users.
- Ensuring all errors are logged for developers without leaking sensitive system data.
- Providing consistent and user-friendly error messages that minimize confusion for non-technical users.

Improper configuration of templates could still degrade user experience, but the layered fallback approach minimizes risk of blank or insecure error responses.

4.7 Password Reset Module

The `password_reset.py` module implements the multi-step password recovery process for users of the school application. It validates usernames, enforces security question checks, and allows verified users to securely reset their password. The main code module is located at */login/password_ reset.py*.

This module demonstrates how user identity verification is enforced before allowing a password change, but also contains an intentional workflow weakness to illustrate poor security practices.

4.7.1 Dependencies

The module depends on:

- `django.shortcuts` for rendering templates and redirects.
- `django.contrib.messages` for providing user feedback.
- `django.views.decorators.cache.never_cache` to prevent caching of sensitive pages.
- Models `Login` and `Login2` from *login.models*.
- Forms `ForgotPasswordForm`, `SecurityAnswerForm`, and `ResetPasswordForm` from *login.forms*.
- Utility function `simple_hash` from *login.views* for password hashing.

4.7.2 Session Decorator

The `session_required` decorator is reused in this module to enforce that intermediate steps (username verification and security question validation) are completed before accessing subsequent steps in the password reset flow. If the required session key is missing, the user is redirected back to the application index.

4.7.3 Main Functions

Step 1: Enter Username

- Function: `forgot_password_step1`
- Clears any previous session state, accepts a username, and checks if it exists.
- If valid, stores the username in the session and redirects to the security question step.

Step 2: Verify Security Question

- Function: `forgot_password_step2`
- Fetches the stored username and prompts the user for the correct security question answer.
- On successful validation, sets a session flag (`forgot_verified`) and allows access to the reset page.
- If no security question is configured, the user is redirected to the index with a warning.

Step 3: Reset Password

- Function: `forgot_password_step3`
- Ensures that the username is valid and security verification is complete.
- Accepts new password and confirmation, validates them, and applies a hash before saving.
- Updates both `Login` and `Login2` model records for consistency.
- On success, clears the session flags and redirects the user to the index page.

4.7.4 Security Weakness

This module intentionally demonstrates a flawed password reset workflow:

- The system may allow users without configured security questions to bypass the verification step, reducing overall account protection.
- Passwords are saved both hashed (in `Login`) and in plaintext (in `Login2`), which is highly insecure and could lead to data exposure.
- Custom hashing via `simple_hash` is weaker than Django's built-in password hashing framework, making it easier for attackers to crack stolen credentials.

Security Misconfiguration

Security misconfiguration occurs when systems, frameworks, or servers are deployed with insecure default settings, unnecessary services, outdated software, or improper permissions. Such misconfigurations can leave sensitive operations exposed, allowing attackers to bypass intended security controls.

Exploit In the vulnerable version of the system, the "Forgot Password" functionality did not properly enforce verification steps when a user lacked a security question. An attacker could:

1. Navigate to the Forgot Password page and enter the target username.
2. If no security question existed for that user, the system redirected to the login page without validation.
3. Manually access the URL `/forgot-password/step3` to bypass the security question step.
4. Reset the target user's password without answering any verification question.

This vulnerability allowed unauthorized users to reset passwords and take control of accounts, compromising confidentiality and integrity of user data.

Possible Mitigations for the vulnerebility

- **Session validation:** Each step of the password reset process should validates that the user has completed the previous steps and answered their security question correctly.
- **Mandatory verification:** Users without a security question are required to complete an alternative verification method before password reset.

- **Timed session tokens:** Password reset flows generate unique, expiring session tokens for each request to prevent direct URL manipulation.
- **Server-side enforcement:** All steps of the reset process should be validated on the server, and bypass attempts result in `403 Forbidden` or redirection to the login page.
- **Audit and monitoring:** Reset attempts are logged with user ID and timestamp. Repeated failed attempts should trigger alerts for potential brute-force or unauthorized access attempts.

4.8 Login Module

The `login.py` module implements the authentication system for the application. It provides user login, logout, and activity logging functionality with role-based redirection. The main code module is located at */login/login.py*.

This module ensures that only authenticated users gain access to the system and that their activities are logged for accountability. It also integrates error handling when access is denied or when log files are unavailable.

4.8.1 Dependencies

The module depends on:

- `django.shortcuts` for rendering templates and managing redirects.
- `django.http.HttpResponse` for returning responses such as log file contents.
- `login.models.Login` for accessing user credentials and roles.
- `login.forms.NewLoginForm` for validating login inputs.
- Utility function `simple_hash` from *login.views* for password hashing.
- Error handlers from *error_ views.py* for permission denial and missing resources.
- Python's `logging` module for recording login attempts.
- Python's `os` module for log file management.

4.8.2 Main Functions

View Logs

- Function: `view_logs`
- Allows administrators to access login activity logs.

- Validates that the requesting user has admin privileges before serving the log file.
- Unauthorized access attempts are blocked with a permission error.

Serve Log File

- Function: `serve_log_file`
- Reads and displays the contents of the `login_activity.log` file.
- If the log file is missing, raises a custom "Page Not Found" error.

Index (Login Page)

- Function: `index`
- Validates submitted credentials using `NewLoginForm`.
- Compares entered passwords with stored hashes to authenticate users.
- On success, assigns role-specific session variables and redirects to the appropriate dashboard (Admin, Teacher, or Student).
- Logs both successful and failed login attempts for accountability.

Logout

- Function: `logout_view`
- Clears the user session completely and redirects to the index page.

4.8.3 Security Considerations

This module strengthens authentication by logging all login attempts and enforcing role-based access. However, intentional weaknesses are present for demonstration purposes:

- Passwords are hashed with a custom `simple_hash` function, which is weaker than Django's built-in password hasher and could be vulnerable to cracking.
- Role-based session flags are used instead of Django's more secure built-in authentication system, reducing overall robustness.

Sensitive Data Exposure via Role Parameter Tampering

Sensitive Data Exposure in this context arises from improper enforcement of role-based access control. When the application relies on client-controllable values to determine privileges, an attacker can manipulate the request to gain unauthorized access to sensitive resources such as administrative logs.

Exploit The vulnerable log-viewing endpoint accepted a client-supplied flag to confirm the user role as Admin and used its presence to grant access to administrative logs. An unauthenticated or non-admin user can simply append the parameter to the URL:

```
http://127.0.0.1:8000/login/view-logs?admin=1
```

Because the server trusted this parameter for authorization, the attacker receives the same admin-level content as a legitimate administrator. This compromises confidentiality of operational data and may expose sensitive information about users, system events, or internal errors.

Possible Mitigations

- **Server-side RBAC enforcement:** All protected endpoints should derive user roles and permissions exclusively from authenticated session data or signed JWT claims. Any client-supplied `admin` parameter is ignored for authorization purposes.
- **Centralized authorization middleware:** A middleware layer performs role checks before controller logic executes. This ensures consistent enforcement of permissions across routes and prevents bypass from inconsistent access checks.
- **Audit logging and alerting:** All attempts to access admin resources are logged with username, session id, IP address, and result (granted/denied). Repeated unauthorized attempts trigger alerts.

4.9 Marks Management Module

The `marks.py` module provides functionality for teachers to add marks for students and for students to view their marks. It supports role-based workflows: teachers can enter marks for a class and subject, while students can view their own marks with optional filtering and sorting. The main code module is located at */login/marks.py*.

4.9.1 Dependencies

The module relies on several Django components and Python standard libraries:

- `django.shortcuts` for rendering templates and managing redirects.
- `django.contrib.messages` for user-facing notifications.
- `django.views.decorators.cache.never_cache` to prevent caching of sensitive pages.

- `django.http.JsonResponse` for AJAX responses.
- `django.shortcuts.get_object_or_404` for safe object retrieval.
- `datetime` and `json` from Python standard library.
- Custom models from *login.models*: `Student`, `Marks`, `ClassSection`, `Subject`, `TeacherAvailability`.
- Custom forms from *login.forms*: `EnterStudentMarksForm`, `SelectClassSubjectForm`.

4.9.2 Session Decorator

The decorator `session_required` enforces role-based access control by checking session keys. It ensures that only authenticated users with the correct role (teacher or student) can access marks functionality. Invalid sessions are flushed, and the user is redirected to the application index.

4.9.3 Main Functions

The module contains several core functions, which can be grouped by teacher and student responsibilities:

Teacher Workflows

- **Add Marks Step 1** (`add_marks_step1`): Allows teachers to select class, subject, exam type, exam date, and total marks for entering student results.
- **Add Marks Step 2** (`add_marks_step2`): Enables teachers to input marks for all students in the selected class and subject. Prevents duplicate entries and validates input.

Student Workflows

- **View Marks** (`view_marks`): Displays marks for the logged-in student. Provides filtering by subject and class section, and allows sorting by marks.
- **Set Student Session from Storage** (`set_sid_from_storage`): Updates the `student_username` session variable via AJAX to synchronize frontend local storage with server session.

4.9.4 Security Weakness

The module has potential security risks:

- Teachers might accidentally or maliciously enter marks for unauthorized classes or subjects.
- Input validation prevents duplicate entries but does not fully prevent session tampering.

Insecure Direct Object Reference (IDOR)

Insecure Direct Object Reference (IDOR) occurs when an application exposes internal object identifiers (for example database primary keys, usernames, or file IDs) to clients and relies on those client-supplied values to authorize access. If the server does not verify that the authenticated principal is permitted to access the referenced object, an attacker can tamper with the identifier to access or modify another user's resources.

Exploit In the vulnerable School Management System the client-side session storage contained a key (e.g., `username`) that the frontend used to request the marks page. An authenticated student can open browser developer tools, locate and edit this sessionStorage key, replace their username with an arbitrary valid username, and then refresh the marks page. Because the backend trusted the identifier provided by the client, the application returned the marks belonging to the substituted username. This attack directly compromises confidentiality of student's academic records and can be performed without knowing other user's passwords.

Possible Mitigations for the vulnerebility

- **Server-side authorization:** All endpoints that return user-specific resources now ignore client-supplied identifiers for authorization. The backend derives the target user from the authenticated session (session cookie or token) and validates ownership before returning any data.
- **Removed sensitive client storage:** Sensitive identifiers (usernames, raw primary keys) were removed from `sessionStorage/localStorage`. Any client-side state is treated only as UI hinting and never as an authorization source.
- **Indirect references and validation:** Where it is necessary to reference objects from the client, the system should use opaque, short-lived tokens or UUIDs that are mapped and validated server-side, reducing risk from predictable sequential IDs.

- **Encrypted server-side sessions:** Session data need to be stored and validated on the server, session identifiers are securely signed and transmitted via secure, HttpOnly cookies to reduce tampering risk.

4.10 Password Management Module

The `password.py` module provides functionality for users to change their password and set or update a security question. It supports role-based workflows: all users (Admin, Teacher, Student) can update their password after entering the current password correctly, and optionally configure their security question. The main code module is located at *IntegratedApplication/bobby/login/views/password.py*.

4.10.1 Dependencies

The module relies on several Django components and custom forms:

- `django.shortcuts` for rendering templates and managing redirects.
- `django.contrib.messages` for user-facing notifications.
- Custom models from *login.models*: `Login`, `Login2`.
- Custom forms from *login.forms*: `ChangePasswordForm`, `SecurityQuestionForm`.
- Custom views/utilities from *login.views*: `simple_hash` for password hashing.

4.10.2 Session Decorator

Session management is handled implicitly by checking the `login_username` and `current_role` session variables. If a session is invalid or missing, the user is redirected to the login index page.

4.10.3 Main Functions

The module contains two core functions, which apply to all roles (Admin, Teacher, Student):

- **Change Password** (`change_password`): Allows a logged-in user to update their password. Validates the current password, ensures the new passwords match, updates the hashed password in both `Login` and `Login2` models, and provides success/error messages. All users follow the same workflow.
- **Set/Update Security Question** (`security_question`): Enables a user to configure or update their security question and answer. Validates input and stores the data in the `Login` model.

4.10.4 Security Weakness

Potential security risks include:

- If session variables are manipulated, a user could potentially attempt unauthorized password changes.
- No rate-limiting is implemented; repeated attempts could facilitate brute-force attacks.
- Security question functionality is optional, and skipping it could leave accounts with weaker recovery options.

4.11 Student Question Posting Module

The `post_questions.py` module enables students to post questions to the learning platform. It ensures that only authenticated students can submit questions, and stores submissions in the database. The main code module is located at *integratedApplication\bobby\login\views\post_ questions.py*.

This functionality lays the foundation for an interactive environment, allowing students to engage actively and fostering communication and collaboration.

4.11.1 Dependencies

The module relies on the following Django components and custom forms/models:

- `django.shortcuts.render` for template rendering.
- `django.shortcuts.redirect` for page redirection.
- `django.shortcuts.get_object_or_404` for safe object retrieval.
- Custom models from *login.models*: `Student`, `Question`.
- Custom forms from *login.forms*: `QuestionForm`.
- Custom session management from *login.views*: `session_required` decorator.

4.11.2 Session Decorator

The decorator `session_required` enforces that only authenticated students can access this functionality. Invalid or missing sessions result in redirection to the student login page.

4.11.3 Main Functions

The module contains a single core function for student workflow:

- **Post Question** (`post_questions`): Validates the student's session, retrieves the logged-in student object, and handles form submission. On POST request, the question text is validated and saved to the `Question` model. On GET request, an empty form is rendered for the student. Successful submissions redirect back to the student dashboard.

4.11.4 Security Weakness

Potential security concerns include:

- Malicious input in question text could lead to stored XSS attacks if not properly sanitized.
- Session tampering could allow unauthorized users to attempt posting, though the decorator mitigates this risk.
- No rate-limiting or spam prevention, allowing repeated submissions.

4.12 Search Module

The `search.py` module provides search functionality for students and teachers. It allows users to search for other students or teachers based on name queries. The main code module is located at *integratedApplication/ bobby/ login/views/ search.py*.

4.12.1 Dependencies

The module relies on several Django components and custom forms/models:

- `django.shortcuts.render` for rendering templates.
- `django.db.connection` for executing raw SQL queries.
- Custom models from *login.models*: `Student`, `Teacher`.
- Custom forms from *login.forms*: `SearchFormStudent`, `SearchFormTeacher`.

4.12.2 Session Decorator

No session decorator is applied in this module; it is assumed that access is controlled via higher-level authentication checks.

4.12.3 Main Functions

The module contains two primary functions, one for searching student and one for searching teacher. These functions handle the search workflow, form validation, query execution, and result rendering.

4.12.4 Main Functions

The module contains two primary functions, one for student searches and one for teacher searches. These functions handle the search workflow, form validation, query execution, and result rendering.

Student Search Workflow (`search_student`)

- Displays a search form to the user.
- Accepts a name query submitted via POST request.
- Constructs and executes a raw SQL query on the `login_student` table to find matches by `student_first_name`.
- Retrieves query results and formats them as dictionaries.
- Renders the `login/search.html` template with the form, search results, and query.

Teacher Search Workflow (`search_teacher`)

- Displays a search form to the user.
- Accepts a name query submitted via POST request.
- Constructs and executes a raw SQL query on the `login_teacher` table to find matches by `firstname`.
- Retrieves query results and formats them as dictionaries.
- Renders the `login/search_teacher.html` template with the form, search results, and query.

4.12.5 Security Weakness

The module contains a significant security vulnerability:

- The use of raw SQL queries with direct string interpolation exposes the application to SQL Injection attacks.
- User input is not sanitized or parameterized, allowing malicious users to manipulate queries and potentially access unauthorized data.

SQL Injection

SQL Injection occurs when user-supplied input is incorporated into SQL statements without proper parameterization or escaping, allowing an attacker to modify the intended query logic. Exploitation can lead to unauthorized data disclosure, modification, or destruction, and in severe cases complete database compromise.

Exploit The vulnerable search box in the Student/Teacher dashboard accepted raw input that was concatenated into SQL queries. An attacker can craft a `UNION SELECT` payload to merge arbitrary query results into the application's expected result set. Using the MySQL `information_schema` the attacker can enumerate table and column names and then extract sensitive columns (for example from the `login_login` table). A typical exploitation workflow observed was:

1. Submit progressively larger `UNION SELECT` payloads to discover the correct number of columns returned by the search query.
2. Use `UNION SELECT` against `information_schema.tables` to enumerate table names.
3. Use `UNION SELECT` against `information_schema.columns` to enumerate column names for a target table.
4. Finally, extract sensitive data with payloads such as:

   ```
   a' UNION SELECT username, password, 'x', 'x', ...  FROM login_login; -
   ```

   which injects usernames and password hashes into the visible search results.

Successful exploitation compromises confidentiality (user credentials, personal data) and may enable privilege escalation.

Possible Mitigation

- **ORM and parameterized queries:** Replaced dynamic SQL string construction with Django ORM calls (e.g., `Model.objects.filter(...)`) and, where raw SQL remained necessary, used parameterized queries to ensure user input is treated as data, not code.
- **Input validation and output encoding:** Applied strict server-side validation for search inputs (length limits, allowed character sets) and encoded outputs to avoid unintended rendering of injected content.

- **Least-privilege database account:** The application connects to the database using an account with minimal privileges (SELECT-only for search operations) to limit the impact if an injection were to succeed.
- **Query result shaping and pagination:** Limited query result size and enforced pagination to reduce the utility of injection for exfiltrating large datasets.
- **Web Application Firewall (WAF) and anomaly detection:** Deployed WAF rules to detect common SQLi patterns (e.g., `UNION SELECT`, SQL metacharacters) and log/deny suspicious requests. Added anomaly detection to flag unusual search queries.

4.13 Student Registration Module

The `student_registration.py` module provides functionality for new students to register via an online form. The registration workflow includes form submission, CAPTCHA validation, and database entry of student details. The CAPTCHA is intended to prevent automated submissions, but its implementation contains a deliberate vulnerability. The main code module is located at *integratedApplication\bobby\login\views\student_registration.py.*

4.13.1 Dependencies

The module relies on the following Django components, libraries, and custom models/forms:

- `django.shortcuts` for rendering templates and handling redirects.
- `django.contrib.messages` for user notifications.
- `django.core.cache` for storing and retrieving CAPTCHA reuse attempts.
- `django.http.JsonResponse` for returning JSON responses in replay attack scenarios.
- `time` from Python standard library for timestamp tracking.
- `captcha.models.CaptchaStore` and `captcha.helpers.captcha_image_url` for CAPTCHA generation and validation.
- Custom model from `login.models`: `StudentReg`.
- Custom form from `login.forms`: `StudentRegistration`.
- Custom helper: `validate_captcha_manual` for manual CAPTCHA verification.

4.13.2 Session Decorator

No session decorator is applied in this module, as registration is intended for new, unauthenticated users.

4.13.3 Main Functions

The module contains a single core function:

Student Workflow (`student_registration`)

- Displays the student registration form along with a CAPTCHA challenge.
- Accepts POST submissions and performs manual CAPTCHA validation.
- Tracks CAPTCHA reuse attempts within a defined time window (20 seconds).
- If CAPTCHA is reused more than the limit, responds with a JSON message confirming replay attack success.
- If the form is valid, stores student information (`firstname`, `lastname`, `dob`, `gender`, `email`, `classlevel`) in the `StudentReg` table.
- Returns success notifications and redirects the student upon successful registration.

4.13.4 Security Weakness

This module intentionally implements CAPTCHA in a vulnerable way:

- The same CAPTCHA response can be reused multiple times within a 20-second window.
- This design enables replay attacks, allowing attackers to bypass CAPTCHA protection.
- Exploiting this vulnerability, attackers can flood the system with fake registrations, leading to spam data, resource exhaustion, and database integrity issues.

Broken Anti-Automation - CAPTCHA Reuse

Broken anti-automation occurs when an application fails to reliably distinguish human users from automated scripts. Common causes include reusable or non-expiring CAPTCHA tokens, predictable challenges, missing rate-limiting, and endpoints that accept unlimited requests. When these protections are weak, attackers can script interactions to brute-force, scrape, or flood services.

Exploit In the vulnerable configuration the registration forms accepted CAPTCHA values that remained valid for multiple submissions. An attacker can:

1. Complete a single successful registration and capture the outgoing POST request via browser DevTools or a proxy like Burp Suite.
2. Extract the CAPTCHA parameters for instance `captcha_0` — challenge token, and `captcha_1` — response.
3. Replay the exact POST request (same CAPTCHA token and response) multiple times in rapid succession using an automated script or request-replay tool.
4. Observe that the server accepts repeated submissions with the same CAPTCHA proof, allowing mass automated registrations that consume database space, distort analytics, or create a denial-of-service scenario.

This attack directly undermines availability, integrity, and can facilitate further attacks.

Possible Mitigations for the vulnerebility

- **One-time, expiring CAPTCHA tokens:** Each CAPTCHA challenge should be bound to a unique server-side token that is marked consumed immediately after a successful submission. Tokens include a creation timestamp and a short TTL (e.g., 60 seconds).
- **Server-side token store and validation:** CAPTCHA tokens are stored server-side (in-memory or persistent store) and validated on submission. The server rejects submissions for expired, missing, or already-consumed tokens.
- **Integration of vetted CAPTCHA library:** Adopt a proven CAPTCHA library (`django-simple-captcha`) to avoid custom implementation errors and to leverage built-in nonce and validation features.

- **CAPTCHA binding to session / request fingerprint:** CAPTCHA tokens are tied to the originating session (or a short-lived fingerprint derived from client attributes) so that a token captured from one client cannot be used from another.
- **Rate limiting and progressive throttling:** Implement a per-IP and per-endpoint rate limits (requests per minute) and backoff for repeated failed attempts. High-volume or anomalous activity triggers temporary blocking or additional challenges (e.g., reCAPTCHA or email verification).
- **Logging and monitoring:** All registration zattempts, token issuance, and token consumption events are logged. Alerts are configured for abnormal rates of successful registrations or repeated token-reuse attempts.

4.14 Student Timetable Module

The `student_timetable.py` module provides functionality for students to view their academic timetable. It ensures that only authenticated students can access timetable data by validating sessions before fetching records. The function retrieves the logged-in student's class level and matches it with timetable entries, optionally filtering by day. The main code module is located at *integratedApplication\bobby\login\views\student_ timetable.py*.

4.14.1 Dependencies

The module depends on the following Django components and models:

- `django.shortcuts.render` for rendering templates.
- Custom models from `login.models`: `Student`, `TimetableEntry`.
- Custom decorator from `login.views`: `session_required`.

4.14.2 Session Decorator

The decorator `session_required('Student_login')` enforces that only authenticated students with valid sessions can access the timetable view. If the session is invalid or expired, the function returns an error message to the frontend.

4.14.3 Main Functions

The module contains a single core function responsible for rendering student timetables:

Student Workflow (`student_timetable_view`)

- Retrieves the logged-in student's username from the session.
- Validates that the username exists in the `Student` table; otherwise, returns an error message.
- Extracts the student's `classlevel` and queries the `TimetableEntry` model for matching timetable records.
- Optionally filters timetable entries by day if a day parameter is provided in the request.
- Optimizes query performance by using `select_related` to fetch related objects such as subject, teacher, room, timeslot, and class section in a single query.
- Renders the `login/student_timetable.html` template with timetable entries, available days, the selected day, and the student's class information.

4.14.4 Security Weakness

The module does not explicitly introduce intentional vulnerabilities. However, possible concerns include:

- Insufficient input validation on the optional `day` parameter could lead to unexpected behavior if improperly handled.
- Debugging statements (e.g., `print`) expose internal information, which should be removed in production environments to prevent information leakage.

4.15 Teacher Registration Module

The `teacher_registration.py` module provides functionality for registering new teachers via an online form. The workflow includes form submission, CAPTCHA validation, and file upload handling. The module demonstrates both CAPTCHA replay vulnerabilities and weak file validation practices, highlighting security pitfalls in registration systems. The main code module is located at *integratedApplication\ bobby\ login\ views\ teacher_ registration.py.*

4.15.1 Dependencies

The module relies on the following Django components, libraries, and custom models/forms:

- `django.shortcuts` for rendering templates and handling redirects.

- `django.contrib.messages` for user notifications.
- `django.core.cache` for tracking CAPTCHA reuse attempts.
- `django.http.JsonResponse` for returning JSON responses during replay attack confirmation.
- `time` from Python standard library for timestamp tracking.
- `captcha.models.CaptchaStore` and `captcha.helpers.captcha_image_url` for CAPTCHA generation and validation.
- Custom model from `login.models`: `TeacherReg`.
- Custom form from `login.forms`: `TeacherRegistration`.
- Custom helper: `validate_captcha_manual` for manual CAPTCHA verification.

4.15.2 Session Decorator

No session decorator is applied in this module, as teacher registration is intended for new, unauthenticated users.

4.15.3 Main Functions

The module contains a single core function:

Teacher Workflow (`teacher_registration`)

- Displays the teacher registration form along with a CAPTCHA challenge.
- Accepts POST submissions and performs manual CAPTCHA validation.
- Tracks CAPTCHA reuse attempts within a defined time window (20 seconds).
- If CAPTCHA is reused more than the limit, responds with a JSON message confirming replay attack success.
- Validates submitted teacher details and saves them to the `TeacherReg` table.
- Handles optional file uploads:
  - Checks if the uploaded file extension matches allowed formats (`.jpg`, `.png`, `.pdf`, `.doc`, `.docx`, `.jpeg`).

  - Ensures that file size does not exceed 7 MB.
  - Associates the valid uploaded document with the teacher's record.

- Returns success notifications and redirects to the homepage upon successful registration.

4.15.4 Security Weakness

This module contains two intentional vulnerabilities:

- **CAPTCHA Replay Attack:** The CAPTCHA validation allows the same token to be reused multiple times within a short time window, enabling attackers to bypass CAPTCHA protection and flood the system with fake registrations.

Unrestricted File Upload

Unrestricted file upload vulnerabilities occur when an application fails to validate uploaded files correctly (type, size, content), allowing attackers to upload oversized files, unexpected file types, or files that can be executed on the server (for example `.php`). Such flaws can lead to denial of service (disk exhaustion), data corruption, information disclosure, or remote code execution.

Exploit The Teacher Profile / Announcement upload functionality accepts file attachments. In the vulnerable configuration an attacker can:

1. Upload a file larger than the intended maximum (for example >1 MB) to exhaust disk quota or cause application errors.

2. Upload an unauthorized file type (e.g., `.png` or `.php`) by changing filename and Content-Type headers in a proxy (Burp Suite) or by directly posting arbitrary content.

3. Upload a PHP script (or another executable payload) and then request the file URL. If the upload directory is served by the web server with script execution enabled, the attacker may achieve remote code execution.

4. Combine large uploads and many concurrent uploads to cause storage exhaustion or DoS.

These attacks threaten availability (disk/IO exhaustion), confidentiality (exposed uploads), and in the worst case integrity/remote code execution (if uploaded files are executed by the server).

Possible Mitigations for the vulnerebility

- **Server-side file size enforcement:** The system will enforce a strict maximum upload size (configurable, e.g., 1 MB) at both the web server and application level. Requests exceeding the limit will be rejected with an explanatory error.
- **Whitelist of allowed file types and magic-byte verification:** Only specific MIME types (e.g., `application/pdf`) will be accepted. The server will verify uploaded file signatures (magic bytes) in addition to the Content-Type and filename extension to prevent spoofing.
- **Filename sanitization and randomization:** Original filenames will be sanitized to remove path characters and will be replaced with randomized, non-guessable filenames (for example UUIDs). This will prevent directory traversal and make direct guessing of file URLs impractical.
- **Store uploads outside web root and disable execution:** Uploaded files will be stored in a directory outside the web root or served via a secure file-serving endpoint. Web server configuration for the upload directory will explicitly disable script execution (e.g., `php_admin_flag engine off` for PHP, or equivalent), ensuring uploaded code cannot be executed.
- **Quarantine and asynchronous processing:** Files will be accepted into a quarantine area and scanned (using an antivirus/malware scanner such as ClamAV or a cloud scanning API) before they are moved to permanent storage. The user will receive a pending/processing state until scanning and validation complete.
- **Least-privilege storage and restrictive permissions:** The service account that writes uploads will have minimal filesystem privileges, and upload directories will have restrictive permissions (no execute bit, limited read/write for the service only).
- **Deliver via authenticated signed URLs:** When serving uploaded files, the application will issue short-lived signed URLs or will stream file content through authenticated endpoints that enforce authorization checks instead of exposing static public paths.
- **Logging, monitoring and alerting:** Uploads will be logged with metadata (uploader id, original filename, stored name, size, MIME, IP). Suspicious events (large files, disallowed types, failed scans) will trigger alerts for manual review.

4.16 Timetable Management Module

The `timetable_management.py` module provides functionality for administrators and teachers to manage and view timetables. Administrators can add requirements (subjects, teachers, class sections, rooms, and timeslots), generate timetables based on these inputs, and view class schedules. Teachers can log in to view their assigned timetables. The main code module is located at *integratedApplication\ bobby\ login\ views\ timetable_ management.py*.

4.16.1 Dependencies

The module relies on the following Django components, models, and forms:

- `django.shortcuts` for rendering templates and handling redirects.
- `django.db.models` for ORM-based queries and filtering.
- `random` from Python standard library for randomized timetable slot assignments.
- Custom models from `login.models`: `TimetableEntry`, `ClassSection`, `Subject`, `TeacherAvailability`, `Room`, `TimeSlot`.
- Custom forms from `login.forms`: `SubjectForm`, `TeacherForm`, `Class SectionForm`, `RoomForm`, `TimeSlotForm`.
- Custom session management decorator from `login.views`: `session_ required`.

4.16.2 Session Decorator

All views in this module are protected by the `session_required` decorator:

- Administrator views require the session type `'Admin_login'`.
- Teacher views require the session type `'Teacher_login'`.

4.16.3 Main Functions

Administrator Workflows

- **Create Timetable** (`create_timetable`): Displays the timetable creation homepage.
- **Add Subject** (`add_subject`): Provides a form to add new subjects to the system.
- **Add Teacher** (`add_teacher`): Allows adding teacher details.

- **Add Class Section** (`add_classsection`): Facilitates creation of new class sections.
- **Add Room** (`add_room`): Allows the administrator to add rooms with specified types.
- **Add Time Slot** (`add_timeslot`): Adds available time slots to the schedule pool.
- **Generate Timetable View** (`generate_timetable_view`): Invokes the timetable generation process and displays any unmet requirements (failures).
- **Admin Timetable View** (`admin_timetable_view`): Displays timetables with filtering options by class, subject, teacher, and day.

Teacher Workflows

- **Teacher Timetable View** (`teacher_timetable_view`): Retrieves the logged-in teacher's assigned timetable, with an option to filter by day.

Timetable Generation Logic

- **Generate Timetable** (`generate_timetable`):
  - Clears existing timetable entries.
  - Randomly assigns subjects, teachers, rooms, and timeslots to meet weekly requirements.
  - Enforces teacher constraints such as maximum daily and weekly load.
  - Ensures no conflicts in room, teacher, or class section assignments.
  - Returns a list of failures for unassigned periods (if requirements cannot be fully met).

4.16.4 Security Weakness

This module does not contain any deliberate or intentional vulnerabilities. However, certain operational limitations may pose risks if not managed carefully:

- The random assignment of teachers, rooms, and timeslots may lead to inefficiencies or unfair workloads without additional scheduling constraints.
- The module relies on session-based access control; secure session management at the framework level is essential to prevent unauthorized access.

4.17 Manual CAPTCHA Validation Module

The `validate_captcha_manual.py` module provides a helper function for validating CAPTCHA responses submitted by users. It checks the provided CAPTCHA response against entries stored in Django's `CaptchaStore`. This function is used by other modules such as student and teacher registration to verify CAPTCHA inputs. The main code module is located at `view/validate_captcha_manual.py`.

4.17.1 Dependencies

The module relies on the following components:

- `captcha.models.CaptchaStore` for retrieving stored CAPTCHA entries.
- `captcha.helpers.captcha_image_url` for generating the image URL (though not directly used in validation).

4.17.2 Session Decorator

No session decorator is applied in this module, as it serves purely as a helper function for CAPTCHA validation and is not directly exposed as a user-facing view.

4.17.3 Main Functions

The module contains a single core function:

Function Workflow (`validate_captcha_manual`)

- Accepts a CAPTCHA ID (`captcha_id`) and user-provided response (`captcha_response`).
- Retrieves the corresponding CAPTCHA entry from the `CaptchaStore`.
- Validates the response against the stored CAPTCHA value.
- Returns `True` if the response matches, otherwise returns `False`.
- Handles cases where the CAPTCHA entry does not exist by returning `False`.

4.17.4 Security Considerations

This helper module does not introduce any direct security vulnerabilities on its own. It is a utility function designed for input validation and relies on Django's `CaptchaStore` for secure storage and retrieval of CAPTCHA entries.

4.18 Announcements Module

The `view_announcement.py` module provides functionality for students to view teacher announcements and interact with them through an upvote/ downvote feature. It serves as the foundation of an interactive student announcement system, enabling transparent communication and feedback between teachers and students. The main code module is located at `views/view_announcement.py`.

4.18.1 Dependencies

The module relies on the following Django components and models:

- `django.shortcuts` for rendering templates and handling redirects.
- `django.db.models.Count, Q` for database aggregation and filtering.
- `login.models.TeacherAnnouncement` for storing announcements.
- `login.models.AnnouncementVote` for tracking student votes on announcements.
- `login.models.Student` for identifying the logged-in student.
- Custom decorator: `session_required` for enforcing session-based access control.

4.18.2 Session Decorator

The `session_required('Student_login')` decorator ensures that only authenticated students can access the announcement view. If the session is invalid or missing, the user is redirected to the login page.

4.18.3 Main Functions

The module contains one primary view function:

Student Workflow (`view_announcement`)

- Retrieves the current student's username from the session.
- Ensures that the username exists, otherwise redirects the student to the login page.
- Fetches the `Student` object corresponding to the username.
- Queries all `TeacherAnnouncement` objects, annotating each with the total number of upvotes and downvotes.

- Retrieves the student's existing votes (`AnnouncementVote`) to indicate their voting status.
- Organizes the student's votes into a dictionary for efficient template rendering.
- Prepares a dictionary mapping announcement IDs to vote counts.
- Renders the `view_announcement.html` template with announcements, vote counts, and student vote information.

4.18.4 Security Considerations

This module does not introduce direct security vulnerabilities. The use of session-based access control ensures that only logged-in students can view and vote on announcements. However, input validation and vote submission logic must be carefully handled in complementary modules to prevent issues such as vote tampering or duplicate voting.

4.19 Vote Announcement Module

The `vote_announcement.py` module provides functionality for students to submit votes (upvote or downvote) on teacher announcements. The module processes POST requests containing JSON data, validates the vote, records it in the database, and returns updated vote counts as a JSON response. This function is part of the interactive student announcement and voting system. The main code module is located at `integratedApplication/bobby/login/views/vote_announcement.py`.

4.19.1 Dependencies

The module relies on the following Django components and models:

- `django.http.JsonResponse` for returning JSON responses.
- `json` from Python standard library for parsing request payloads.
- `login.models.AnnouncementVote` for storing votes.
- `login.models.TeacherAnnouncement` for identifying announcements.
- `login.models.Student` for identifying the logged-in student.
- Custom decorator: `session_required` for enforcing session-based access control.

4.19.2 Session Decorator

The `session_required('Student_login')` decorator ensures that only authenticated students can submit votes. If the session is invalid or missing, the user cannot perform voting actions.

4.19.3 Main Functions

The module contains a single primary view function:

Student Workflow (`vote_announcement`)

- Accepts a POST request with JSON data specifying the `announcement_id` and `vote_type` ('upvote' or 'downvote').
- Validates the vote type to ensure it is either 'upvote' or 'downvote'.
- Retrieves the currently logged-in student from the session.
- Fetches the corresponding `Student` and `TeacherAnnouncement` objects.
- Records the vote in the `AnnouncementVote` table.
- Computes the updated counts of upvotes and downvotes for the announcement.
- Returns a JSON response containing the status, message, and updated vote counts.

4.19.4 Security Weakness

This module contains a notable security vulnerability:

- It does not prevent a student from submitting multiple votes on the same announcement.
- As a result, a user can cast repeated votes, potentially skewing the vote counts.
- Exploiting this vulnerability can manipulate perceived popularity or importance of announcements.
- Proper mitigation would include checking for an existing vote by the student before creating a new entry or allowing vote updates instead of duplicates.

4.20 Admin Pinboard Module

The `admin_pinboard.py` module allows administrators to create and view pinboard announcements. The main code is located at: `/login/view/admin_pinboard.py` functions: `create_pinboard`, `pinboard_list_admin`.

4.20.1 Dependencies

The module depends on several Django components and custom models/forms:

- **Django components**
  - `django.shortcuts.render`, `redirect` for template rendering and redirection.
  - `django.contrib.messages` for user notifications for successful or failed actions.
  - `django.core.paginator.Paginator` for handling paginated display of announcements.
  - `django.views.decorators.cache.never_cache` to prevent caching of pinboard pages.
- **Project models**
  - `Pinboard` tablestores announcements and metadata.
  - `Teacher` and `Student` tables are used to display creator names.
- **Project forms**
  - `PinboardForm` validates admin input for announcements.
- **Project decorators**
  - `session_required('Admin_login')` — ensures only admins can create/view announcements.

4.20.2 Session Decorator

The `session_required('Admin_login')` decorator enforces role-based access control. Only authenticated admins can create or view pinboard announcements. Invalid sessions are flushed, and the user is redirected to the admin login page.

The `@never_cache` decorator ensures that paginated announcements are always fetched fresh, preventing sensitive data from being stored in the cache.

4.20.3 Main Functions

`create_pinboard(request)`

- **Purpose:** Allow admins to post new announcements to the pinboard.
- **Logic:**
  - Handles POST requests containing `PinboardForm` data.
  - Valid form data is saved, with the `created_by` field set from the current session username.
  - Provides success feedback and redirects to the pinboard list.
  - For GET requests, renders an empty form for input.

`pinboard_list_admin(request)`

- **Purpose:** Display a paginated list of pinboard announcements for the admin.
- **Logic:**
  - Calls the common renderer `pinboard_list_common` with role 'Admin'.
  - Announcements are sorted in reverse chronological order.
  - Display names are resolved using `get_display_name`, combining first/last names of students or teachers, or a default name for admins.
  - Pagination is set to 10 announcements per page.
  - The back link is dynamically set to the admin dashboard URL.

`pinboard_list_common(request, role)`

- **Purpose:** Shared renderer for pinboard listings (Admin, Teacher, Student).
- **Logic:**
  - Annotates announcements with `display_name` based on creator.
  - Handles pagination.
  - Determines dashboard URL for navigation based on role.
  - Renders the common template `pinboard_list.html`.

`get_display_name(username)`

- **Purpose:** Helper function to display a readable name for the announcement creator.
- **Logic:**
  - Checks if username is an admin.
  - Queries the `Student` or `Teacher` model for matching names.
  - Defaults to the raw username if no match found.

4.20.4 Security Considerations

Intentional vulnerabilities are not present in this module.

- **Template escaping:** Ensure announcement text is escaped to prevent stored Cross-Site Scripting (XSS).
- **Session enforcement:** Only authenticated admins can access creation and listing views.
- **Pagination:** Prevents overloading the interface with large datasets.
- **No explicit file uploads:** Not applicable here, so no file validation needed.

4.21 Student Pinboard Module

The `student_pinboard.py` module allows students to view the pinboard announcements posted by admins or other authorized users. The main code is located at: `/login/view/student_pinboard.py` function: `pinboard_list_student`.

4.21.1 Dependencies

The module depends on several Django components and custom models/forms:

- **Django components**
  - `django.shortcuts.render` for template rendering.
  - `django.core.paginator.Paginator` for handling paginated display of announcements.
  - `django.views.decorators.cache.never_cache` to ensure fresh data for each page load.
- **Project models**
  - `Pinboard` table which stores announcements.

- `Teacher` and `Student` tables are used to display creator names.

- **Project decorators**
  - `session_required('Student_login')` ensures only logged-in students can view announcements.

4.21.2 Session Decorator

The `session_required('Student_login')` decorator enforces role-based access control. Only authenticated students can view the pinboard list. Invalid sessions are flushed, and the user is redirected to the student login page.

The `@never_cache` decorator ensures that paginated announcements are always fetched fresh, preventing caching of old announcements.

4.21.3 Main Functions

`pinboard_list_student(request)`

- **Purpose:** Display a paginated list of pinboard announcements for students.
- **Logic:**
  - Calls the common renderer `pinboard_list_common` with role 'Student'.
  - Announcements are sorted in reverse chronological order.
  - Display names are resolved using `get_display_name`, combining first/last names of students or teachers, or a default name for admins.
  - Pagination is set to 10 announcements per page.
  - The back link is dynamically set to the student dashboard URL.

`pinboard_list_common(request, role)`

- **Purpose:** Shared renderer for pinboard listings (Admin, Teacher, Student).
- **Logic:**
  - Annotates announcements with `display_name` based on the creator.
  - Handles pagination.
  - Determines dashboard URL for navigation based on role.
  - Renders the common template `pinboard_list.html`.

`get_display_name(username)`

- **Purpose:** Helper function to display a readable name for the announcement creator.
- **Logic:**
  - Checks if username is an admin (`ADM*`).
  - Queries the `Student` or `Teacher` model for matching names.
  - Defaults to the raw username if no match found.

4.21.4 Security Considerations

Intentional vulnerabilities are not present in this module.

- **Read-only access:** Students cannot create, edit, or delete announcements.
- **Template escaping:** Ensure announcement text is escaped to prevent stored XSS.
- **Session enforcement:** Only authenticated students can access this view.
- **Pagination:** Prevents overloading the interface with large datasets.

4.22 Teacher Pinboard Module

The `teacher_pinboard.py` module allows teachers to view the pinboard announcements posted by admins or other authorized users. The main code is located at: `/login/view/teacher_pinboard.py` function: `pinboard_list_teacher`.

4.22.1 Dependencies

The module depends on several Django components and custom models/forms:

- **Django components**
  - `django.shortcuts.render` — template rendering.
  - `django.core.paginator.Paginator` — handles paginated display of announcements.
  - `django.views.decorators.cache.never_cache` — ensures fresh data for each page load.
- **Project models**
  - `Pinboard` — stores announcements.

  - `Teacher` and `Student` — used to display creator names.

- **Project decorators**
  - `session_required('Teacher_login')` — ensures only logged-in teachers can view announcements.

4.22.2 Session Decorator

The `session_required('Teacher_login')` decorator enforces role-based access control. Only authenticated teachers can view the pinboard list. Invalid sessions are flushed, and the user is redirected to the teacher login page.

The `@never_cache` decorator ensures that paginated announcements are always fetched fresh, preventing caching of old announcements.

4.22.3 Main Functions

`pinboard_list_teacher(request)`

- **Purpose:** Display a paginated list of pinboard announcements for teachers.
- **Logic:**
  - Calls the common renderer `pinboard_list_common` with role Teacher.
  - Announcements are sorted in reverse chronological order.
  - Display names are resolved using `get_display_name`, combining first/last names of students or teachers, or a default name for admins.
  - Pagination is set to 10 announcements per page.
  - The back link is dynamically set to the teacher dashboard URL.

`pinboard_list_common(request, role)`

- **Purpose:** Shared renderer for pinboard listings (Admin, Teacher, Student).
- **Logic:**
  - Annotates announcements with `display_name` based on the creator.
  - Handles pagination.
  - Determines dashboard URL for navigation based on role.
  - Renders the common template `pinboard_list.html`.

`get_display_name(username)`

- **Purpose:** Helper function to display a readable name for the announcement creator.
- **Logic:**
  - Checks if username is an admin (`ADM*`).
  - Queries the `Student` or `Teacher` model for matching names.
  - Defaults to the raw username if no match found.

4.22.4 Security Considerations

- **Read-only access:** Teachers cannot create, edit, or delete announcements.
- **Template escaping:** Ensure announcement text is escaped to prevent stored XSS.
- **Session enforcement:** Only authenticated teachers can access this view.
- **Pagination:** Prevents overloading the interface with large datasets.

4.23 Pinboard Detail And Comments Module

The `pinboard_detail.py` module allows users to view a single pinboard announcement and its associated comments. Logged-in users (Admin, Teacher, Student) can post comments. The main code is located at: `/login/view/pinboard_detail.py`.

4.23.1 Dependencies

The module depends on several Django components and custom models/forms:

- **Django components**
  - `django.shortcuts.render, redirect, get_object_or_404` for template rendering and record lookup.
  - `django.views.decorators.cache.never_cache` ensures the page always shows fresh comments.
- **Project models**
  - `Pinboard` announcement model, with related comments.
  - `Teacher` and `Student` used to display the commenter's name.
- **Project forms**
  - `PinboardCommentForm` handles comment submission.

4.23.2 Session Decorator

- `@never_cache` ensures users see the latest comments without relying on browser cache.
- Comment posting relies on session keys (`Admin_login`, `Teacher_login`, `Student_login`) to associate the comment with the correct user.
- If no valid session exists, comments are marked as Anonymous.

4.23.3 Main Functions

`pinboard_detail(request, pk)`

- **Purpose:** Display a single pinboard announcement and all its comments.
- **Logic:**
  - Retrieves the announcement using its primary key (`pk`).
  - Populates `display_name` for the creator and all commenters using `get_display_name`.
  - Orders comments chronologically.
  - Handles POST requests to add a new comment:
    - Determines the current user based on session.
    - **VULNERABLE:** Raw user input is saved directly, allowing stored XSS if templates do not escape HTML.
  - Provides `dashboard_url` for a back link, depending on user role.
  - Renders `pinboard_detail.html`.

`get_display_name(username)`

- **Purpose:** Resolve a readable display name for the announcement/ comment creator.
- **Logic:**
  - Returns "Administration Office" for admin usernames.
  - Fetches the first and last name from `Student` or `Teacher` models.
  - Defaults to the raw username if no match is found.

4.23.4 Security Considerations

- **Stored XSS risk:** Comments are saved directly without sanitization. Any HTML or `<script>` tags will execute when viewed.
- **Template escaping required:** Always escape announcement and comment fields in templates to prevent XSS.
- **Session-based user tracking:** Ensures comments are attributed to the logged-in user; falls back to "Anonymous" otherwise.
- **Read/write access:** Only logged-in users can comment; all others have read-only access.

Cross-Site Scripting (XSS) in Pinboard Feature

Cross-Site Scripting (XSS) is a class of web vulnerability where untrusted input is included in web pages without proper validation or encoding, allowing an attacker to inject client-side scripts that execute in the browsers of other users. XSS can be used to steal session tokens, perform actions on behalf of users, or harvest sensitive information visible in the victim's browser context.

Exploit The Pinboard feature permits authenticated users (students, teachers, admins) to post messages which are then displayed to other users. If the application fails to sanitize or encode user-provided message content, an attacker can submit a message containing script content that will be interpreted by other users' browsers when they view the Pinboard. Possible consequences in our system include:

- Exfiltration of session cookies or authentication tokens leading to account takeover.
- Unauthorized actions performed using the victim's session (for example, submitting forms, changing preferences).
- Displaying deceptive content or capturing keystrokes and form data submitted by other users.

A simple test payload (for example, a script that triggers a visible browser alert) demonstrates that injected script content executes in viewers' browsers, confirming the presence of XSS.

Possible Mitigations for the vulnerebility

- **Output encoding / automatic template escaping:** Ensure that all user-supplied content is HTML-encoded before insertion into pages.

Templating engines should be configured to use auto-escaping by default, and raw HTML rendering should be permitted only in strictly controlled, reviewed components.

- **Input sanitization for rich content:** Where rich text or limited HTML is required, use a well-reviewed sanitizer (for example, server-side libraries that implement a safe whitelist policy) to remove scripts, event handlers, and dangerous attributes. Client-side rich-text editors should be configured to submit only a safe subset of markup.
- **Context-aware encoding:** Apply encoding appropriate to the output context (HTML body, attribute, JavaScript, URL) to prevent injection in multiple contexts.
- **Content Security Policy (CSP):** Introduce a restrictive CSP header (Report-Only during rollout) to prevent inline-script execution and to disallow arbitrary external script sources. Use CSP nonces or hashes for approved inline scripts when necessary.
- **Cookie hardening and session protection:** Set session cookies with `HttpOnly`, `Secure`, and `SameSite` attributes to reduce theft risk from client-side script access.
- **Input validation and length limits:** Implement server-side validation to restrict message length and remove suspicious control characters; disallow common vectors such as embedded `<script>` tags and dangerous attributes.

4.24 Student Application Module

The `student_application.py` module implements the core functionality for submitting a student application. It integrates with existing student registration records to prefill form data where available and stores validated applications in the StudentApplication model. This feature provides the entry point for new students to apply to the school system.The main code module is located at `/login/view/student_application.py`. Although no deliberate vulnerability is introduced here, insecure coding practices may lead to issues such as unvalidated input, stored XSS, or weak data handling. The feature therefore requires careful form validation and safe template rendering.

4.24.1 Dependencies

The module depends on multiple Django components and third-party libraries:

- `django.shortcuts` for rendering templates and managing redirects.

- `django.contrib.messages` for user-facing notifications.
- Custom models from *login.models*: `StudentApplication` for storing final application data and `StudentReg` for storing student registration data; the latest record is used for pre-filling.
- Custom forms from *login.forms*: `StudentApplicationForm` for validating and cleaning submitted application data.

4.24.2 Prefill Logic

The view fetches the most recent `StudentReg` record using:

```
latest_reg = StudentReg.objects.last()
```

If present, relevant fields (name, Date of birth, gender, email, class level) are pre-populated into the form. Gender codes (M, F, O) are mapped into human-readable strings. This feature improves user experience by reducing redundant typing.

4.24.3 Main Function: `student_application_view`

The function provides two request flows:

- **GET request**
  - Initializes a blank `StudentApplicationForm`.
  - If a `StudentReg` record exists, fields are prefilled with existing values.
  - Renders the `studentapplication.html` template with the form.
- **POST request**
  - Processes form submission via `StudentApplicationForm`.
  - On successful validation:
    - Extracts cleaned data.
    - Creates a new `StudentApplication` record with the supplied details (student information, parent info, previous school information).
    - Adds a success message and redirects to the index page.
  - On invalid submission, the form is redisplayed with error messages.

POST Request

- Processes submitted form data using `StudentApplicationForm(request.POST)`.
- If the form is valid:
  - Extracts `form.cleaned_data`.
  - Creates a `StudentApplication` record with explicit field mapping:
    - **Student information:** first name, last name, Date of birth, gender, email, class level, mobile, nationality, blood group.
    - **Student address:** street, house, city, state (optional), postal.
    - **Parent information:** first name, last name, email, mobile, emergency contact.
    - **Parent address:** street, house, city, state (optional), postal.
    - **Previous school info:** school name, class/grade, TC number, street, house, city, state, postal.
  - Adds a success message and redirects to the index page.
- If the form is invalid, the form with errors is re-rendered for user correction.

4.24.4 Security Considerations

- Only saves `form.cleaned_data`, ensuring that only validated input is persisted.
- Assumes templates correctly escape user-provided content to prevent XSS.
- Prefill uses the latest `StudentReg` record; ensure this behavior is acceptable and does not expose unintended data.
- No deliberate vulnerabilities are present in this module.

4.25 Student Approval Module

The `student_approval.py` module enables administrators to manage pending student applications. Admins can review submitted applications, approve them (creating both a student record and a corresponding Login account), or reject them (deleting the application record).The main code module is located at:`/login/view/student_approval.py`

This module demonstrates secure session handling via a decorator and enforces role-based access control. While the code itself is functional and secure, passwords are generated as plain tokens and must be hashed before production deployment.

4.25.1 Dependencies

The module depends on several Django components and custom models/forms:

- **Python standard libraries**
  - `secrets`, `string`, `random`, `hashlib` — for generating usernames, passwords, and hashing (simple MD5 in current implementation).
- **Django libraries**
  - `django.shortcuts.render`, `redirect`, `get_object_or_404` — template rendering, redirection, and object fetching.
  - `django.contrib.messages` — for displaying flash messages to the admin.
- **Project models**
  - `StudentApplication` model for pending student applications.
  - `Student` model for approved student records.
  - `Login` model for user authentication table, storing username, password, role, and email.
- **Project decorators**
  - `session_required` (from `login.views`) which enforces that only authenticated admins can access these views.

4.25.2 Session Decorator

All views in this module are decorated with: `@session_required('Admin_login')`. This ensures that only authenticated administrators can access student approval or rejection functionalities. If a session is invalid or missing, the user is redirected to the application index.

4.25.3 Main Functions

Admin Workflows

List Pending Students (`student_approval`)

- Retrieves all `StudentApplication` records.
- Filters out applications already approved (by checking existing emails in `Login` with role `'Student'`).
- Renders `login/student_approval.html` template with the list of pending students.

Approve Student (`approve_student`)

- Accepts a `student_id`.
- Fetches the corresponding `StudentApplication` record.
- Checks whether the student email is already approved; if so, issues a warning message.
- Generates a random username (3 letters + 3 digits) and a random password token (8-character URL-safe string).
- Creates a new `Student` record using all application details (personal, address, parent info).
- Creates a new `Login` record for authentication purposes, with plain token as password (note: should be hashed in production).
- Displays a success message and redirects back to pending applications.

Reject Student (`reject_student`)

- Accepts a `student_id`.
- Fetches the corresponding `StudentApplication` record.
- Deletes the record.
- Shows an informational message and redirects to the pending applications page.

4.25.4 Security Considerations

No intentional vulnerabilities exist in this module, though production deployments must ensure proper password hashing and possibly audit logs for administrative actions.

- **Session enforcement:** All views require `Admin_login` session, preventing unauthorized access.
- **Data integrity:** Explicit field mapping ensures all relevant student and parent information is transferred correctly from `StudentApplication` to `Student`.
- **Duplicate check:** Approval function ensures the same email cannot be approved multiple times.
- **Deletion safety:** Rejection permanently deletes the student application record.

4.26 Teacher Approval Module

The `teacher_approval.py` module enables administrators to manage pending teacher registrations. Admins can review submitted registrations (`TeacherReg`), approve them (creating both a `Teacher` record and a corresponding `Login` account), or reject them (deleting the registration record). The main code module is located at: `/login/view/teacher_approval.py`

This module uses secure session enforcement and explicit approval logic. While it is secure in its basic operation, care must be taken to handle file uploads (teacher documents) safely.

4.26.1 Dependencies

The module depends on several Django components and custom models/forms:

- **Python standard libraries**
  - `secrets`, `string`, `random`, `hashlib` for generating usernames, passwords, and hashing (simple MD5 in current implementation).
- **Django libraries**
  - `django.shortcuts.render`, `redirect`, `get_object_or_404` template for rendering, redirection, and object fetching.
  - `django.contrib.messages` for displaying flash messages to the admin.
- **Project models**
  - `TeacherReg` model for pending teacher registrations.
  - `Teacher` model for approved teacher records.
  - `Login` model for user authentication table, storing username, password, role, and email.
- **Project decorators**
  - `session_required` (from `login.views`) which enforces that only authenticated admins can access these views.

4.26.2 Session Decorator

All views in this module are decorated with: `@session_required('Admin_login')`. This enforces that only authenticated administrators can access teacher approval or rejection views. Invalid sessions are redirected to the application index.

4.26.3 Main Functions

Admin Workflows

List Pending Teachers (`teacher_approval`)

- Retrieves all `TeacherReg` records.
- Filters out registrations already approved (by checking existing emails in `Login` with role `'Teacher'`).
- Renders `login/teacher_approval.html` template with the list of pending teachers.

Approve Teacher (`approve_teacher`)

- Accepts a `teacher_id`.
- Fetches the corresponding `TeacherReg` record.
- Checks whether the teacher email is already approved; if so, issues a warning message.
- Generates a random username (3 letters + 3 digits) and a random password token (8-character URL-safe string).
- Creates a new `Teacher` record using registration details: first name, last name, DOB, gender, email, and uploaded document.
- Creates a new `Login` record for authentication purposes, with the generated plain password (**should be hashed in production**).
- Displays a success message and redirects back to the pending registrations page.

Reject Teacher (`reject_teacher`)

- Accepts a `teacher_id`.
- Fetches the corresponding `TeacherReg` record.
- Deletes the record.
- Shows an informational message and redirects to the pending registrations page.

4.26.4 Security Considerations

No intentional vulnerabilities exist in this module, though production deployments must ensure proper password hashing and possibly audit logs for administrative actions.

- **Session enforcement:** All views require an admin session, preventing unauthorized access.
- **File safety:** Teacher documents (`document` field) must be handled carefully when serving to prevent unauthorized access.
- **Duplicate check:** Approval function ensures the same email cannot be approved multiple times.
- **Deletion safety:** Rejection permanently deletes the teacher registration record.

4.27 Student Profile Module

The `student_profile.py` module enables students to view and update their personal profile. It integrates data from both the `Student` model and `StudentApplication` model, allowing students to update fields selectively. Profile photo upload is supported, with changes applied only to modified fields. The main code module is located at: `/login/view/student_profile.py`

This module demonstrates secure session handling, selective field updates, and file upload support. No deliberate vulnerabilities are present, assuming proper template escaping and safe file handling.

4.27.1 Dependencies

The module depends on several Django components and custom models/forms:

- **Django libraries**
  - `django.shortcuts.render`, `redirect`, `get_object_or_404` are used for template rendering, redirection, and safe object fetching.
  - `django.contrib.messages` is used for displaying flash messages to the user.
- **Project models**
  - `Login` is a authentication table to retrieve the current user session.
  - `Student` is a main student profile model.
  - `StudentApplication` is a source for prefilled student application data.

- **Project forms**
  - `StudentProfileForm` is responsible for validating user-submitted profile updates.
- **Project decorators**
  - `session_required` ensures only authenticated students can access the profile.

4.27.2 Session Decorator

The view is decorated with: `@session_required('Student_login')`. This ensures that only logged-in students can access the profile page. Invalid or missing sessions result in a redirect to the student index page.

4.27.3 Main Function: `student_profile`

GET Request

- Retrieves the logged-in student's `Login` object via username stored in session.
- Fetches corresponding `Student` and `StudentApplication` objects.
- Prepares initial form data using `StudentApplication` fields.
- Renders `login/student_profile.html` with:
  - `form` prefilled `StudentProfileForm`.
  - `profile_photo` current profile photo from `Student`.

POST Request

- Instantiates `StudentProfileForm(request.POST, request.FILES)`.
- On valid form submission:
  - Extracts `cleaned_data`.
  - Compares each field against `Student` and `StudentApplication` objects to track changes.
  - Updates only fields that have changed:
    - **Student fields** (including profile photo if uploaded).
    - **StudentApplication fields** (if applicable).
  - Saves changes selectively using `update_fields` for efficiency.
  - Displays success message and redirects to the profile page.
- On invalid submission:
  - Form is redisplayed with errors for correction.

4.27.4 Data Mapping

- **Student model updates**
  - Any field present in `StudentProfileForm` that exists in `Student` and differs from the current value.
  - `profile_photo` handled separately, saved if a new file is uploaded.
- **StudentApplication updates**
  - The fields that exist in `StudentApplication` and differ from the values submitted are updated via `.update()`.
- **Read-only fields**
  - Fields marked as read-only in the form are never modified.

4.27.5 Security Considerations

Intentional vulnerabilities are not present in this module.

- **Session enforcement:** Only authenticated students can access this view.
- **File uploads:** Profile photo files must be stored securely, and the files that are served should prevent unauthorized access.
- **Template safety:** Text fields rendered in templates must be escaped to prevent XSS.
- **Selective updates:** Only modified fields are saved, preventing unintended overwrites.
- **Data consistency:** Synchronizes changes across both `Student` and `StudentApplication` tables.

4.28 Teacher Profile Module

The `teacher_profile.py` module allows teachers to view and update their profile information, including uploading profile photos and documents. It also provides a document download endpoint.

This module is intentionally designed with insecure coding practices for lab/demo purposes. Vulnerabilities such as CSRF bypass and Insecure Direct Object Reference (IDOR) are introduced to demonstrate real-world attack scenarios.The main code is located at: `/login/view/teacher_profile.py`

4.28.1 Dependencies

The module depends on several Django components and custom models/forms:

- **Django libraries**
  - `django.shortcuts.render`, `redirect` for template rendering and redirection.
  - `django.contrib.messages` for flash messages for user feedback.
  - `django.http.FileResponse`, `Http404` for serving file downloads and handling errors.
  - `django.views.decorators.csrf.csrf_exempt` for disables CSRF protection.
- **Python standard libraries**
  - `os` for file handling (checking existence, deleting old files).
- **Project models**
  - `Login` table for teacher authentication and linking accounts.
  - `TeacherReg` table for original teacher registration data (with uploaded documents).
  - `Teacher` table for permanent teacher profile data.
- **Project decorators**
  - `session_required('Teacher_login')` is ensures only authenticated teachers access these views.
- **Project forms**
  - `TeacherProfileForm` for updating teacher profile fields.

4.28.2 Session Decorator

The view is decorated with: `@session_required('Teacher_login')`. This ensures that only logged-in teachers can access the profile page. Invalid or missing sessions result in a redirect to the teacher index page.

4.28.3 Main Function: `teacher_profile`

`teacher_profile(request)`

- **Purpose:** View and update teacher profile data.
- **Features:**

- Loads teacher information from both `TeacherReg` and `Teacher`.
- Allows updating first name, last name, document, and profile photo.
- Deletes old files (document/photo) when replaced with new ones.
- Displays feedback messages on successful or failed updates.

- **Vulnerability:**
  - IDOR — accepts `?username=KP0001` to target another teacher's data instead of using the session user.
  - CSRF disabled with `@csrf_exempt`, allowing forged requests.

`download_teacher_document(request)`

- **Purpose:** Allows teachers to download their uploaded documents.
- **Features:**
  - Retrieves the document from either `TeacherReg` or `Teacher`.
  - Returns the file as an attachment (`FileResponse`).
  - Handles missing files gracefully with error messages.
- **Vulnerability:**
  - IDOR — accepts `?username=KP0001` to download another teacher's document.
  - CSRF disabled, allowing attackers to trick teachers into unintended downloads.

4.28.4 Security Weakness

This module deliberately demonstrates insecure practices:

1. **CSRF bypass:** Both profile update and document download are exempt from CSRF checks.
2. **IDOR:** Query parameter `?username=` allows attackers to read or modify other teachers' data.
3. **File upload restrictions bypass:**
   - No server-side validation exists for file type or size.
   - Users can upload files larger than intended or with different types (e.g., PHP, JS), even if the frontend restricts input to PDFs under 1MB.

- This could lead to malicious uploads, file execution, or server compromise.

4. **File handling risks:** Uploaded files are deleted/replaced directly on the filesystem, which may lead to race conditions or unintended deletions.

5. **Authorization gap:** Authorization is not strictly tied to the session user.

Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is an attack that causes a victim's browser to submit state-changing requests to a web application in which the victim is authenticated. Because browsers automatically include session cookies and certain authentication information, the forged request appears legitimate to the server unless explicit anti-CSRF defenses are in place.

Exploit The teacher profile update endpoint was marked with `@csrf_exempt`, and it accepts POST requests to change profile fields (e.g., `firstname`, `lastname`). An attacker can host a malicious page that auto-submits a form targeting that endpoint. If a teacher is already authenticated in the victim browser and visits the malicious page, the browser will send the teacher's session cookie along with the forged POST request. Since the endpoint does not verify a CSRF token or origin, the server will accept the request and update the teacher's profile without the teacher's consent. This allows unauthorized profile manipulation and can be extended to other sensitive actions.

Possible Mitigations for the vulnerebility

- **Restore framework CSRF protection:** The project will remove any `@csrf_exempt` decorators from sensitive views and ensure Django's `CsrfViewMiddleware` is enabled globally.

- **Template-level tokens:** All HTML forms (including profile update forms) will include `{% csrf_token %}` so that each submission carries a server-generated token that will be validated on receipt.

- **AJAX support:** For XHR/fetch-based requests, the client will read the CSRF token from the cookie or DOM and send it via the `X-CSRFToken` header; the server will verify this header token on incoming requests.

- **SameSite and secure cookies:** Session cookies will be configured with `SameSite=Lax` (or `Strict` where appropriate) and the `Secure` flag; cookies will remain `HttpOnly` to prevent client-side script access.

- **Origin/Referer checks for sensitive endpoints:** The server will perform Origin/Referer validation for high-value actions to complement token checks, rejecting requests whose origin does not match the application's host.
- **Enforce POST and idempotency rules:** State-changing actions will accept only POST (or other non-safe) methods, and safe GET endpoints will never perform state changes.
- **Re-authentication for sensitive operations:** For very sensitive profile changes, the system will require recent re-authentication (password prompt) or a second factor before applying changes.

Insecure Direct Object Reference (IDOR) — Teacher Profile

Exploit In the vulnerable teacher profile endpoint the application accepted a `username` query parameter and returned the corresponding profile page without confirming that the requested username matched the logged-in user. An attacker who is authenticated as a teacher can:

1. Navigate to the profile URL used by the application, e.g. `http://127.0.0.1:8000/login/teacher/profile/?username=TPX627`
2. Modify the `username` query parameter in the address bar to another valid teacher username, for example `?username=TPX999`.
3. Reload the page. If the server returns the profile of `TPX999` without verifying ownership, the attacker has successfully accessed another teacher's private profile information.

This attack compromises confidentiality (exposes personal information) and may enable further attacks (social engineering, impersonation).

Possible Mitigations for the vulnerebility

- **Server-side ownership enforcement:** The profile endpoint will ignore any client-supplied username parameters for sensitive self-profile pages. Instead, it will derive the target user from the authenticated session (using the server-side session ID or token) and fetch that user's record directly from the database.
- **Explicit access-control checks for cross-user views:** For endpoints where viewing another user's public profile is legitimate, the server will perform explicit authorization checks (such as role verification or visibility flags) before returning data and will expose only permitted fields.

- **Use of indirect/opaque references:** When object references need to be exposed to the client (for example, in API responses), opaque identifiers (such as UUIDs or HMAC-signed tokens) will be used instead of predictable usernames or incremental IDs. These tokens will be validated server-side.
- **Removal of sensitive identifiers from URLs:** Sensitive identifiers will not be included as query parameters for authenticated endpoints. Routes will be refactored to use session-driven endpoints (e.g., `/profile/me`) for self-service actions.
- **Session hardening:** Session handling will be strengthened by enforcing shorter lifetimes, secure cookie flags, and session rotation on privilege changes to minimize the risk of session theft being used to exploit IDOR vulnerabilities.

4.29 Admin Profile Module

The `admin_profile.py` module allows administrators to view their profile information. Access is controlled via session-based authentication. The main code is located at: `/login/view/admin_profile.py`

4.29.1 Dependencies

The module depends on several Django components and custom models/forms:

- **Django components**
  - `django.shortcuts.render`, `redirect` for template rendering and redirection.
  - `django.contrib.messages` for feedback messages for errors or notifications.
- **Project models**
  - `Login` table stores authentication credentials and role information.
- **Project decorators**
  - `session_required('Admin_login')` ensures only authenticated admins can access the profile page.

4.29.2 Session Decorator

The `session_required('Admin_login')` decorator enforces role-based access control. Only users with an active admin session can view the profile. Invalid sessions are flushed, and the user is redirected to the admin login page.

4.29.3 Main Function

admin_profile(request)

- **Purpose:** Display the admin's profile information.
- **Logic:**
  - Retrieves the current admin username from the session (`login_username`).
  - Queries the `Login` table for a user with role 'Admin'.
  - Renders `admin_profile.html` with the retrieved data.
  - If no record is found, displays an error message and redirects to the admin login page.

4.29.4 Security Considerations

- **Session enforcement:** Only authenticated admins can access this page.
- **Data exposure:** Only minimal profile information is shown.
- **No deliberate vulnerabilities:** This module is considered secure; improper session handling could expose admin data.

4.30 Non-Vulnerable Configuration I

The first non-vulnerable system was implemented to address and mitigate the vulnerabilities identified in the baseline configuration. The objective was to demonstrate how secure configuration practices and defensive coding can prevent exploitation of common weaknesses. This configuration focused on three key vulnerabilities: CAPTCHA bypass, announcement vote tampering, and PDF metadata exposure. Each mitigation was designed with a defense-in-depth approach, combining both preventive and detective controls.

4.30.1 Mitigation of Broken Anti-Authentication - CAPTCHA Bypass Vulnerability

In the vulnerable system, the CAPTCHA token could be reused within a short time window, allowing automated scripts to bypass human verification checks. To address this weakness, the non-vulnerable system implemented the following countermeasures:

- Each CAPTCHA challenge is now associated with a unique, single-use token generated on the server. Once a CAPTCHA is solved, the token is invalidated immediately after submission, preventing reuse within any time frame.

- Tokens are stored temporarily in the server-side session with a strict expiration period. Expired or reused tokens result in immediate request rejection, effectively blocking automated replay attempts.
- The system integrates the `django-simple-captcha` library, which provides a robust, well-maintained solution that minimizes implementation flaws. It supports dynamic challenge generation and integrates smoothly with Django's form validation mechanisms.
- In addition to CAPTCHA hardening, a basic rate-limiting mechanism was introduced to restrict the number of registration attempts within a fixed time window. This provides further resilience against brute-force automation attempts.

Through these measures, the CAPTCHA mechanism became resistant to token reuse, ensuring that only genuine human interactions can complete the registration process. Automated attack simulations confirmed that repeated requests using the same CAPTCHA token were consistently rejected.

4.30.2 Mitigation of Cross Site Scripting - Announcement Vote Tampering

The second vulnerability allowed users to submit multiple votes for the same announcement, due to the lack of server-side enforcement of uniqueness. In the non-vulnerable configuration, a series of database and application-level improvements were implemented to ensure vote integrity:

- Before inserting a new vote record, the server checks whether a vote from the same user already exists for the targeted announcement. If a record is found, the system either updates the existing vote or rejects the new submission based on the intended voting policy.
- The previous system relied solely on front-end controls, which could easily be bypassed using browser developer tools. In the new design, all enforcement is handled server-side, ensuring that even forged HTTP requests cannot manipulate the vote count.
- The system maintains a vote history log. This enables auditing of vote activities and detection of abnormal voting patterns for future security analysis.

After implementing these measures, the system reliably enforced one-vote-per-user behavior. Attempts to resubmit votes using automated tools or direct API calls resulted in rejection responses, thereby maintaining fair and transparent voting outcomes.

4.30.3 Mitigation of PDF Metadata Exposure

The third vulnerability stemmed from insufficient handling of uploaded PDF documents. The previous implementation stored and served PDFs without sanitization, exposing metadata such as author details, software versions, timestamps, and reviewer annotations. To mitigate this issue, the following improvements were made:

- A metadata sanitization step was added during the upload process. Using libraries all metadata fields and annotations are stripped from the PDF before storage. Only essential content layers are preserved.
- Each uploaded file undergoes a multi-stage validation process, which checks for file type integrity, size constraints, and malicious embedded content. Files that do not meet validation criteria are rejected before being stored.
- Sanitized files are stored under randomized filenames in a secure directory outside the web root. This ensures that direct URL guessing or parameter tampering cannot expose internal file structures.
- The application enforces appropriate response headers when serving PDFs to prevent unintended content interpretation by the browser.

After these mitigations, all uploaded and served PDF documents were verified to be free of residual metadata and hidden annotations. This significantly reduced the risk of privacy breaches and data disclosure through document files.

Overall, the non-vulnerable configuration successfully neutralized the previously identified weaknesses. The application demonstrated improved security posture by integrating server-side validation, input sanitization, and data integrity mechanisms across multiple layers of its architecture.

4.31 Non-Vulnerable Configuration II

The second non-vulnerable system was implemented as a hardened configuration to eliminate the residual vulnerabilities found in earlier versions. This version introduced stricter server-side validation, improved authorization mechanisms, and secure input and file-handling strategies. The objective was to achieve a production-level secure configuration through layered defense techniques, covering vulnerabilities such as SQL Injection, Insecure Direct Object Reference (IDOR), Sensitive Data Exposure, XML External Entity (XXE) processing, and Security Misconfiguration in the password reset process.

4.31.1 Mitigation of SQL Injection Vulnerability

The original search functionality constructed database queries by directly concatenating user input into SQL strings. This insecure practice enabled attackers to craft injection payloads and manipulate the underlying query execution. Raw SQL statements were constructed using unsanitized user input from the search form, allowing special characters such as quotation marks and SQL keywords to alter query behavior.

- Replaced all raw SQL queries with Django's Object-Relational Mapper methods, which internally use parameterized queries to prevent injection.
- Implemented strict server-side input validation, ensuring that search queries match predefined patterns before any database interaction occurs.
- Applied the principle of least privilege to the database user used by the application, ensuring it has only SELECT permissions for the search feature.

Any injected SQL syntax is now treated as plain text rather than executable code. Search operations are performed safely through ORM abstractions, completely preventing SQL injection.

4.31.2 Mitigation of Insecure Direct Object Reference

In the vulnerable configuration, the marks viewing module allowed manipulation of the username stored in the client's session storage. By altering this value, a user could access another student's marks. The system relied on client-side identifiers to determine the resource owner, without validating ownership on the server.

- Introduced robust server-side authorization logic that verifies ownership of each resource request. Every marks retrieval now checks the authenticated session user against the database entry.
- Modified backend endpoints to ignore any client-provided identifiers. The logged-in session user is now the sole source for determining which records are accessible.
- Implemented unique session tokens for each login session and encrypted all user session data on the server to prevent tampering.
- Conducted unit and penetration testing to ensure unauthorized access attempts are met with proper responses.

The marks viewing system is now fully session-driven. Even if a user alters local session values or request parameters, the backend enforces strict ownership verification before any data is returned.

4.31.3 Mitigation of Sensitive Data Exposure - View Logs Vulnerability

In the earlier system, administrative logs could be accessed by appending a URL parameter such as `?admin=1`. This insecure design exposed sensitive data due to the lack of real role validation. Role verification was handled through a client-supplied flag instead of using session-based authentication or server-side role mapping.

- Implemented a centralized Role-Based Access Control system where privileges are derived directly from the authenticated user's session.
- The backend now validates the role of each request against the stored session credentials. The mapping between usernames and roles is maintained securely in the database.
- Introduced middleware-level authorization that automatically rejects any request from users without sufficient privileges, ensuring consistent enforcement across endpoints.
- Implemented detailed access logs to record every admin-page access attempt, enabling detection of unauthorized or failed role checks.

Privilege escalation via URL parameter manipulation is no longer possible. Only users with the "admin" role, verified server-side, can access system logs.

4.31.4 Mitigation of XML External Entity (XXE) Vulnerability

The assignment upload feature previously accepted XML files and parsed them using the `lxml` library with external entity resolution enabled. Malicious XML files containing recursive entities (Billion Laughs payloads) could cause the system to consume excessive resources, resulting in denial-of-service. Unrestricted XML parsing with DTD and entity resolution allowed the application to process untrusted XML content directly.

- Replaced insecure XML parsing and securely configured `lxml` parsers by disabling DTD loading and entity resolution.
- Implemented a file-vetting mechanism that validates uploaded files before parsing.
- Added a layered file handling policy where uploads are processed under a non-administrative service account with minimal privileges, ensuring containment if a vulnerability is exploited.

- Introduced file content scanning to detect known malicious payload patterns and reject suspicious uploads early.

The upload system now safely handles XML files, preventing external entity resolution and recursive expansion attacks. Any Billion Laughs payload is detected and discarded before processing.

4.31.5 Mitigation of Security Misconfiguration - Insecure Password Reset

In the vulnerable setup, attackers could bypass the password reset process by directly navigating to restricted URLs, resetting another user's password without completing previous steps. Lack of server-side session validation between sequential password reset steps and absence of token-based state management.

- Implemented a multi-step verification process controlled entirely on the server. Each stage of the password reset requires a valid, one-time token generated during the preceding step.
- Added CSRF protection and session-based state tracking to ensure that the reset flow cannot be skipped or replayed.
- Validated security question answers and timestamps at each stage; tokens are time-bound and automatically invalidated upon expiration.
- Introduced timed session flushing to clear temporary reset sessions after a predefined interval or upon successful password change.

The password reset process now strictly enforces step-by-step validation and token verification, preventing attackers from bypassing any intermediate verification stage.

In summary, this second non-vulnerable configuration comprehensively resolved five high-impact vulnerabilities through a combination of secure coding, proper access control, secure file parsing, and improved session management. The implementation of these measures significantly enhanced the system's security posture and aligned it with best practices for secure web application deployment.

4.32 Third-Party Libraries

The School Management System leverages several third-party libraries and Python packages to implement its web-based functionalities efficiently and securely. This section lists all major external dependencies, their purpose, and the modules that utilize them.

- **Django** [**djangoproject**]: Django is the core web framework used throughout the application. It handles routing, request/response cycles, session management, ORM-based database interactions, authentication, and template rendering. All modules, including student registration, timetable management, and announcements, rely on Django for backend operations.
- **django-simple-captcha** [**djangocaptcha**]: This library is used for CAPTCHA generation and validation. Modules such as `teacher_registration` and `validate_captcha_manual` utilize it to prevent automated bot submissions. Manual validation is implemented in `validate_captcha_manual.py` with customized logic for response checking.
- **jQuery / AJAX**: Used primarily in the `vote_announcement` module to submit votes asynchronously via JSON without reloading the page. This allows students to upvote or downvote announcements interactively.
- **Bootstrap**: Provides responsive styling and UI components for HTML templates across the system. Although optional, it ensures consistent and professional front-end design for all web pages.
- **Pillow**: Used in modules that handle file uploads, such as `teacher_registration`. It provides image processing capabilities, including resizing and saving uploaded files.
- **Python Standard Libraries**:
  - `os` – file system and path handling.
  - `time` – timestamps for CAPTCHA validation and form submissions.
  - `json` – parsing and formatting JSON payloads (e.g., in `vote_announcement`).
  - `random` – selecting teachers, rooms, and timeslots in the timetable module.
  - `logging` – centralized logging and error tracking across modules.
  - `functools` – used for decorators like `session_required`.
  - `datetime` – date validation and formatting in forms and timetable entries.
- **Database Drivers**:
  - SQLite / PostgreSQL – Persistent storage of all entities such as `Student`, `TeacherReg`, `TimetableEntry`, and `AnnouncementVote`.

Managed through Django's ORM to ensure data integrity and security.

Notes on Custom Modifications:

- The manual CAPTCHA validation module includes a lowercasing feature for the user response to match stored values.
- The voting module currently allows duplicate votes by the same student on a single announcement; this vulnerability is included intentionally for educational purposes.

4.33 Challenges Faced During Docker Implementation

During the implementation phase of Docker for the Django and MySQL-based web application, several technical and configuration-related challenges were encountered that impacted the successful completion of the containerization process. These challenges primarily involved database connectivity, port conflicts, service initialization, and dependency management in hybrid Windows–Linux environments.

1. **Plugin Installation and Repository Configuration in WSL**
   Installing Docker components in the WSL environment presented additional challenges. The official Docker repository had not been added to the package manager's sources, preventing proper installation. This was resolved by manually adding the repository and reinstalling the components, highlighting the sensitivity of dependency management in hybrid Windows–Linux setups.

2. **MySQL Service Initialization and Port Conflicts**
   A major challenge arose when starting the MySQL service within the Windows Subsystem for Linux (WSL) environment. Executing `sudo service mysql start` resulted in a control process failure, as the MySQL data directory could not be initialized correctly. Further investigation revealed that port 3306, the default MySQL port, was already in use by a native Windows MySQL instance. Modifying the MySQL configuration to use an alternate port (3307) temporarily resolved the issue, but inconsistencies persisted when connecting from Docker containers due to mismatched environment configurations.

3. **Database Connectivity Between Django and MySQL**
   The Django application initially failed to connect to the MySQL container. This was due to the database host settings in `settings.py` pointing to `localhost`, which refers to the Django container itself rather than the MySQL container. Updating the host to the Docker service name partially resolved the issue; however, intermittent failures continued because the MySQL service was not fully initialized when Django attempted to connect.

4. **Service Initialization Dependencies**
   Even after correcting configuration issues, the order of service initialization remained problematic. The Django container often started before the MySQL service was fully operational, leading to repeated *connection refused* errors. Adjusting the container orchestration sequence and adding health checks was required to ensure proper startup order.

5. **Volume Mounting and Data Persistence Issues**
   Attempts to mount the MySQL data directory from the host system

using relative paths caused permission errors and risked data loss due to overwriting on container restarts. Configuring Docker volumes correctly and ensuring proper file permissions were necessary to maintain persistent and consistent data across container lifecycles.

These challenges underscored the complexities of containerizing multi-service applications, particularly in hybrid Windows–Linux environments. Although the Docker setup is still incomplete, working through these issues has provided valuable insights into dependency management, service orchestration, container configuration, and the importance of careful planning when deploying multi-service systems.

5 Evaluation

The evaluation of the Bobby School of Cyber Security focused on verifying that the project successfully demonstrated the intended security vulnerabilities and their secure mitigations. The primary objective was not performance testing or user studies, but rather ensuring that the insecure system reliably exhibited exploitable flaws and that the secure implementation effectively addressed them.

Each deliberately insecure feature was tested against its intended vulnerability scenario. For example, the login module was checked to confirm that it was vulnerable to SQL injection, while the announcement voting module was verified to allow duplicate submissions. Similarly, insecure direct object references (IDOR) and weak CAPTCHA handling were implemented and tested to demonstrate their real-world consequences. In all cases, the insecure implementations behaved as expected, confirming that they provided a meaningful basis for demonstrating common weaknesses.

The evaluation also included a review of logging and error-handling mechanisms. In the insecure version, failures and exceptions were often exposed directly to the user, revealing sensitive system details. In contrast, the secure version handled exceptions gracefully while logging the errors to the server side, thereby preserving functionality without disclosing critical information.

Overall, the evaluation demonstrated that the project met its core objectives: to highlight common vulnerabilities, to show their impact when left unaddressed, and to present the benefits of secure development practices. While no external participants were involved, the systematic testing of insecure and secure implementations confirms that the Bobby School of Cyber Security provides a practical and effective educational resource for illustrating the importance of secure software engineering.

6 Conclusion

The cyber security project aimed to design and implement a School Management System capable of demonstrating both the existence and mitigation of software vulnerabilities within a realistic web application environment. The system was developed with three primary user roles - administrator, teacher, and student and incorporated essential academic management functionalities such as registration and authentication, announcement publishing, timetable generation, assignment upload and evaluation, and marks management. These features collectively represented a functional, database-driven application suitable for security testing and configuration analysis.

In the initial phase, a deliberately vulnerable version of the system was developed to illustrate common security weaknesses found in web applications, including improper authentication, weak session handling, insecure database queries, and input validation flaws. These vulnerabilities were intentionally introduced to study their impact and to understand how attackers could potentially exploit them. In the subsequent development phases, two hardened versions Non-Vulnerable Version I and Non-Vulnerable Version II were implemented. These versions incorporated appropriate security mechanisms such as secure authentication practices, parameterized queries, encryption of sensitive data, and improved access control policies. The comparative evaluation of the vulnerable and secured versions enabled a deeper understanding of secure coding standards and defensive programming techniques.

The project successfully met its core objectives by demonstrating the entire security cycle from vulnerability identification to mitigation and verification. It also emphasized the importance of incorporating cybersecurity principles during the early stages of software development rather than treating security as an afterthought. Although the team encountered technical challenges during the Docker-based containerization phase, particularly related to configuration conflicts and dependency and unable to complete it, these difficulties provided valuable insights into secure deployment environments and system isolation techniques.

Overall, the project achieved both its functional and educational goals by integrating theoretical cybersecurity concepts with practical implementation. It strengthened the understanding of web application vulnerabilities, secure software engineering practices, and system hardening methods. The outcomes of this work contribute meaningfully to the learning objectives of applied cybersecurity and underscore the critical role of security awareness in modern software development.

7 Future Work

While the **Bobby School of Cyber Security** successfully demonstrates core vulnerabilities and their secure mitigations, there are several avenues for further development and refinement:

1. **Expanded Vulnerability Coverage**
   Additional categories of vulnerabilities (e.g., Cross-Site Scripting, Cross-Site Request Forgery, insecure file handling, broken access control) could be implemented to provide a more comprehensive educational experience.

2. **Gamification and Challenges**
   A structured challenge system could be added, where learners are tasked with finding and exploiting specific vulnerabilities, similar to Capture-The-Flag (CTF) platforms. Points, badges, or leaderboards could further encourage engagement.

3. **Integrated Learning Resources**
   Explanatory hints, guided tutorials, or links to OWASP documentation could be embedded within the platform, helping learners not only exploit but also understand vulnerabilities.

4. **Secure Development Best Practices**
   The secure version of the system could be expanded to demonstrate advanced techniques, such as rate-limiting, multi-factor authentication, and secure logging/auditing mechanisms.

5. **Scalability and Deployment**
   Deployment instructions could be enhanced to support containerization (e.g., Docker) and cloud deployment, making the system easier to distribute and run in classroom or training environments.