1. A functional web application on with implemented security features

Our site aims to make it easier for the cybersecurity student to maintain his tools that he learns in an orderly and clear manner and the possibility of modification with deletion on these tools, in home Page we navigation menu section contains links for Home, Profile, My Blogs, and Logout (for logged-in users) or Login and Register (for guests).we displays a clear welcome message “Cyber Security Blog Manager” and provides a concise summary of the application's purpose “Store and manage your cybersecurity blogs and tools securely”



Security Considerations Implemented:

1. Content Security Policy (CSP) Implementation for prevents unauthorized script execution and mitigates cross-site scripting (XSS) attacks.

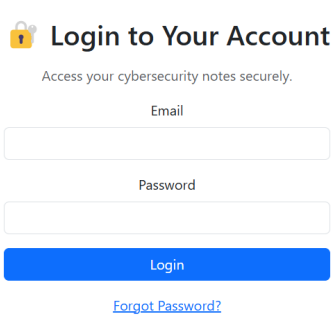
```
Send a raw HTTP header
header( string $header [, bool $replace [, int $http_response_code ]]): void
header("Content-Security-Policy: default-src 'self'; script-src 'self' https://cdn.jsdelivr.net; style-src 'self' https://cdn.jsdelivr.net; img-sr
```

2. Use of HTTPS we guarantees that all traffic is securely transmitted over HTTPS.

```php
if (!isset($_SERVER['HTTPS']) || $_SERVER['HTTPS'] !== 'on') {
    if (!headers_sent()) {
        header("Location: https://" . $_SERVER['HTTP_HOST'] . $_SERVER['REQUEST_URI']);
        exit();
    }
}
```

3. Minimal Information Exposure we avoids exposing sensitive information and ensures that session management is handled securely with session_start(), we don’t Exposure full email after user login .

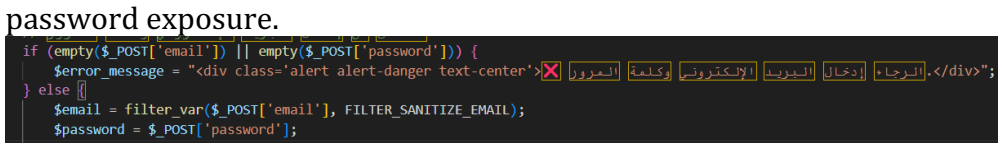
In Login Page we asked user to enter email and pass



Security Considerations Implemented:

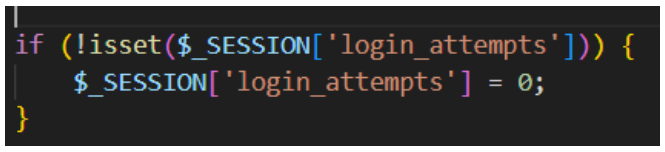
1. We implemented Secure Login Process have HTTPS Enforcement , CSRF Protection, Secure Session Handling we Regenerates session ID after login to prevent session fixation attacks
2. Input Validation

1. Email Validation & Sanitization to removes malicious characters from user input + Password Verification Using Secure Hashing Ensures passwords are stored and verified using secure password hashing (bcrypt), preventing password exposure.



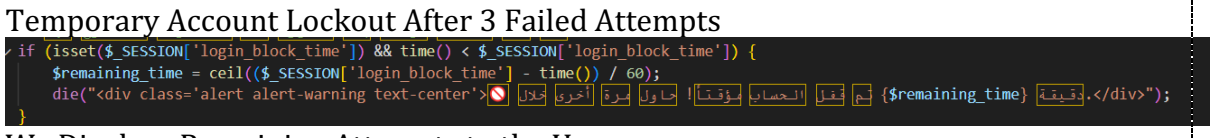
```php
if (empty($_POST['email']) || empty($_POST['password'])) {
    $error_message = "<div class='alert alert-danger text-center'>❌ المرور وكلمة الإلكتروني البريد إدخال الرجاء.</div>";
} else {
    $email = filter_var($_POST['email'], FILTER_SANITIZE_EMAIL);
    $password = $_POST['password'];
```

2. Prepared Statements (to Prevent SQL Injection) we Uses PDO prepared statements, preventing SQL Injection.
3. Login Attempt Logging & Account Lockout we Tracks failed login attempts in the session.



```php
if (!isset($_SESSION['login_attempts'])) {
    $_SESSION['login_attempts'] = 0;
}
```

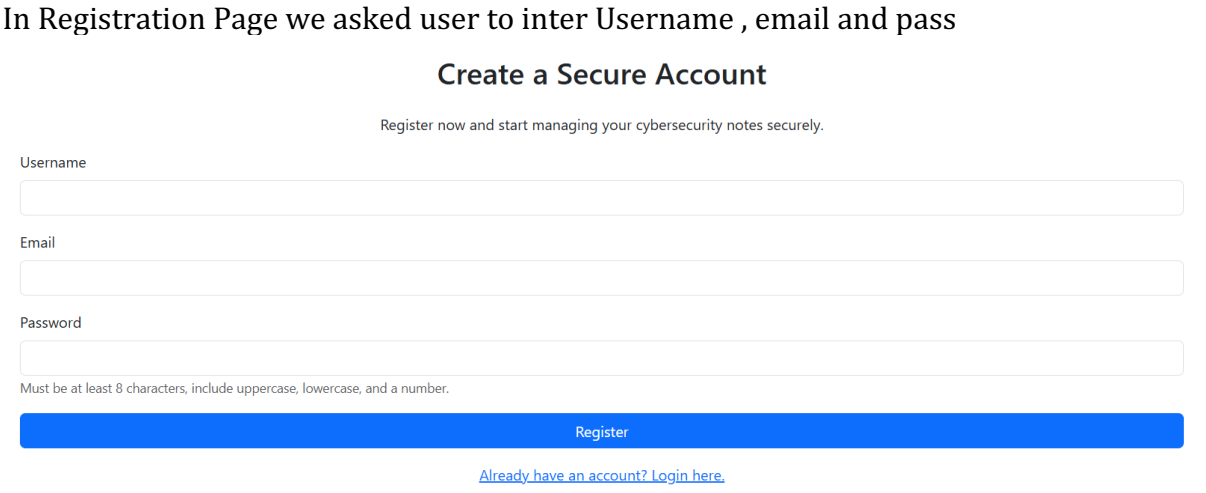
4. Temporary Account Lockout After 3 Failed Attempts



```php
if (isset($_SESSION['login_block_time']) && time() < $_SESSION['login_block_time']) {
    $remaining_time = ceil(($_SESSION['login_block_time'] - time()) / 60);
    die("<div class='alert alert-warning text-center'>🚫 خلال أخرى مرة حاول !مؤقتًا الحساب قفل تم {$remaining_time} دقيقة.</div>");
}
```

5. We Displays Remaining Attempts to the User

In Registration Page we asked user to inter Username , email and pass



Security Considerations Implemented:

1. Input Sanitization we Sanitizing Username Input we Removes HTML tags and trims spaces to prevent XSS attacks and Sanitizing Email Input Strips unnecessary characters to prevent email injection attacks. We Validating Email Format Ensures the email is in a valid format, we imputation Password Requirements, Hashing, and Checking and Secure Password Hashing



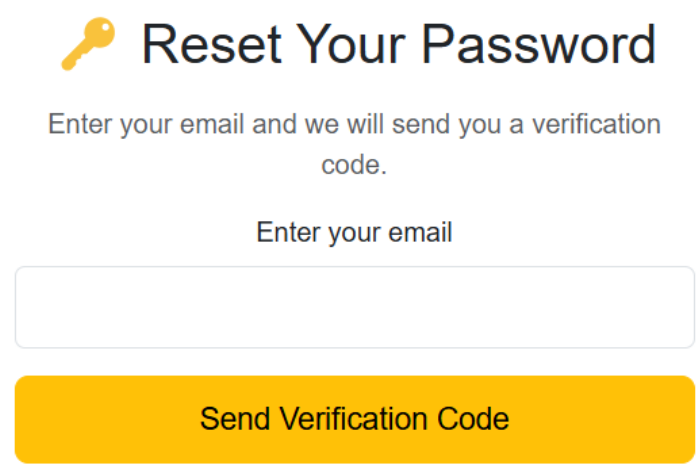
```php
// المدخلات تعقيم
$username = htmlspecialchars(trim($_POST['username']));
$email = filter_var($_POST['email'], FILTER_SANITIZE_EMAIL);
$password = $_POST['password'];

// الإلكتروني البريد لتنسيق مز التحقق
if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
    die("<div class='alert alert-danger'>❌ Invalid email format.</div>");
}


if (!preg_match('/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d).{8,}$/', $password)) {
    die("<div class='alert alert-danger'>❌ Password must be at least 8 chara
}

// المرور كلمة تجزئة
$password_hash = password_hash($password, PASSWORD_BCRYPT);
```

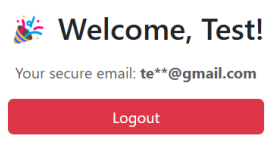
In Password Reset Page the user shuod inter email to send you a verification code



Security Considerations Implemented:

1. Secure Password Reset Mechanism (Partial) we Ensures email exists before proceeding for Prevents attackers from knowing if an email is registered.
2. Emails are sent over SMTP with TLS encryption
3. We Use a cryptographically secure token to Harder to guess and more resistant to attacks
4. Token Expiration Store an expiration timestamp in the database to prevents users from using expired reset codes.

In User Profile page we display Welcome, user!Your secure email: te**@gmail.com
We hide email to prevent information exposer .



Security Considerations Implemented:

1. Authentication Check : ensures user is logged in before accessing the profile page to Prevents unauthenticated users from accessing the profile



```php
// بدء الجلسة والتحقق من وجود user_id
session_start();
if (!isset($_SESSION['user_id'])) {
    header("Location: login.php");
    exit();
}
```

2. User-Specific Data Retrieval : we Uses prepared statements to prevent SQL injection attacks , Filters user input (PDO::PARAM_INT) for extra security.

3. We have Secure Navigation because we Forces HTTPS , Encodes User Input to Prevent XSS for Prevents XSS attacks by escaping special characters.

In Data Management Page



Security Considerations Implemented:

1. The user can insert A new blog is entered into the database using a query and read by select Only user-specific blogs are fetched using a query , An existing blog is updated using a query The blog is deleted through a link to a page delete_data.php.
2. The function htmlspecialchars() is used Sterilizes entries before they are displayed on the page, preventing XSS (Cross-Site Scripting) attacks.
3. Function trim() is used to remove whitespace from the beginning and end of input texts.
4. Function intval() is used to convert values to integers when needed, preventing SQL injection.
5. Validation
    1. It is verified that the title of the blog must be between 3 and 100 characters.
    2. It is verified that the content must be at least 10 characters long.
    3. It is verified that tags should not exceed 255 characters.
6. Verify user permissions
    1. The user is verified as signed in by checking for user_id in the session.
    2. It is ensured that the user can only update or delete their blogs by including user_id in the UPDATE and DELETE queries.
7. View error messages Error messages are displayed in a secure manner using htmlspecialchars() to prevent XSS attacks.

Website link:
https://secureapp.free.nf/login.php?i=1

2. Threat Modeling and Risk Assessment

In this phase, we will identify potential threats and vulnerabilities in the proposed web application using the STRIDE methodology. We will also perform a risk assessment using the DREAD model to prioritize risks based on their severity.

1. Threat Modeling Using STRIDE

STRIDE is a threat modeling framework that categorizes threats into six categories:

- Spoofing: Impersonating a user or system.
- Tampering: Unauthorized modification of data.
- Repudiation: Denial of actions performed.

- Information Disclosure: Unauthorized access to sensitive data.
- Denial of Service (DoS): Disrupting service availability.
- Elevation of Privilege: Gaining unauthorized access to higher privileges.

Threat Model for the Web Application

Attack Surface Analysis

The attack surface of the web application includes the following components:

- User Authentication: Login, registration, and password reset.
- Session Management: User sessions and cookies.
- Database: Storing user credentials and sensitive data.
- Input Forms: User inputs in registration, login, and profile updates.
- APIs: Endpoints for user interactions.

Potential Threats and Vulnerabilities

STRIDE Category	Threat	Vulnerability
Spoofing	An attacker impersonates a legitimate user.	Weak password policies or lack of multi-factor authentication.
Tampering	An attacker modifies user data or application logic.	Lack of input validation or sanitization.
Repudiation	A user denies performing an action (e.g., posting content).	Lack of logging or auditing mechanisms.
Information Disclosure	Sensitive data (e.g., passwords) is exposed.	Insecure storage of passwords or lack of encryption.
Denial of Service	The application becomes unavailable due to excessive requests.	Lack of rate limiting or request throttling.
Elevation of Privilege	An attacker gains admin-level access.	Insecure session management or lack of role-based access control.

2.Risk Assessment Using DREAD

DREAD is a risk assessment model that evaluates threats based on five factors:

- Damage Potential: How severe is the damage if the threat is exploited?
- Reproducibility: How easy is it to reproduce the attack?
- Exploitability: How easy is it to exploit the vulnerability?
- Affected Users: How many users are affected?
- Discoverability: How easy is it to discover the vulnerability?

Each factor is scored on a scale of 0 to 3, and the total score determines the risk level.
Risk Assessment Table

Threat	Damage	Reproducibility	Exploitability	Affected Users	Discoverability	Total	Risk Level
Spoofing	3	2	2	3	2	12	High
Tampering	3	2	2	2	2	11	High
Repudiation	2	1	1	1	1	6	Medium
Information Disclosure	3	2	2	3	2	12	High
Denial of Service	3	3	3	3	2	14	Critical
Elevation of Privilege	3	2	2	2	2	11	High

Risk Prioritization

- Critical: Denial of Service (DoS).
- High: Spoofing, Tampering, Information Disclosure, Elevation of Privilege.
- Medium: Repudiation.

3. Mitigation Strategies

Based on the identified threats and risks, here are the recommended mitigation strategies:

1. Spoofing
   - Implement strong password policies (e.g., minimum length, complexity).
   - Use multi-factor authentication (MFA) for added security.
   - Ensure secure session management (e.g., use secure cookies, regenerate session IDs after login).

2. Tampering
   - Validate and sanitize all user inputs to prevent SQL injection or XSS attacks.
   - Use HTTPS to encrypt data in transit.
   - Implement CSRF tokens to prevent cross-site request forgery.

3. Repudiation
   - Implement logging and auditing mechanisms to track user actions.
   - Store logs securely and ensure they cannot be tampered with.

4. Information Disclosure
   - Use strong encryption (e.g., bcrypt) for storing passwords.
   - Avoid exposing sensitive data in URLs or error messages.
   - Implement role-based access control (RBAC) to restrict access to sensitive data.

5. Denial of Service (DoS)
   - Implement rate limiting to prevent brute-force attacks.

- Use a Web Application Firewall (WAF) to filter malicious traffic.
- Monitor and block suspicious IP addresses.

6. Elevation of Privilege
   - Implement role-based access control (RBAC) to restrict access to admin functions.
   - Regularly review and update user permissions.
   - Use secure session management to prevent session hijacking.

4. Example: Applying Mitigation to the Web Application

Login Page
- Threat: Spoofing (brute-force attacks).
- Mitigation:
  - Implement rate limiting (e.g., 5 failed attempts locks the account temporarily).
  - Use CAPTCHA to prevent automated attacks.

Registration Page
- Threat: Tampering (SQL injection or XSS).
- Mitigation:
  - Sanitize and validate all user inputs.
  - Use prepared statements to prevent SQL injection.

Profile Page
- Threat: Information Disclosure (exposing sensitive data).
- Mitigation:
  - Encrypt sensitive data (e.g., email addresses) in the database.
  - Restrict access to the profile page to authenticated users only.

Database
- Threat: Information Disclosure (data breaches).
- Mitigation:
  - Use strong encryption for sensitive data.
  - Regularly update and patch the database software.

5. Conclusion

By using the STRIDE methodology, we identified potential threats and vulnerabilities in the web application. Using the DREAD model, we prioritized these risks and proposed mitigation strategies to address them. Implementing these measures will significantly enhance the security of the application and protect it from common attacks.

3. Security Analysis and Testing

For HTML/CSS:

stylelint:

- . stylelintrc.json file is a configuration file where you define the rules that you want the **Stylelint** tool to check. Which we need to write first before running this tool.

```
  "rules": {
    "color-hex-length": "short",
    "declaration-no-important": null,
    "selector-max-id": 0,
    "selector-max-universal": 0,
    "selector-max-specificity": "0,3,0",
    "max-nesting-depth": 3,
    "no-duplicate-selectors": true,
    "block-no-empty": true,
    "length-zero-no-unit": true,
    "shorthand-property-no-redundant-values": true,
    "rule-empty-line-before": [
     "always",
     {
      "except": ["first-nested"],
      "ignore": ["after-comment"]
     }
    ],
    "at-rule-no-unknown": [
     true,
     {
      "ignoreAtRules": ["extend", "tailwind", "apply"]
     }
    ],
    "unit-disallowed-list": ["cm", "mm", "in"],
    "font-family-no-duplicate-names": true,
    "property-no-unknown": true
  }
}
```

This Stylelint setup enforces clean and readable CSS by shortening color codes (e.g., `#fff` instead of `#ffffff`), limiting complex selectors, and preventing empty blocks or duplicate selectors. It blocks units like cm and mm, avoids extra values in shorthand properties, and adds a blank line before most rules. This setup helps maintain tidy and well-structured CSS.

The first test:

```
ju@Mac secure_web_app % npx stylelint "**/*.css"


css/styles.css
  45:1  ✖  Unexpected duplicate selector "body", first used at line 1  no-duplicate-selectors

✖ 1 problem (1 error, 0 warnings)

ju@Mac secure_web_app %
```

Indicated us there was a duplicate. The body section in the css code we written twice.

The second test:

```
ju@Mac secure_web_app % npx stylelint "**/*.css"

ju@Mac secure_web_app % npx stylelint "**/*.css"

ju@Mac secure_web_app %
```

After we removed the duplicate and combined the body sections into one body section the test give us no errors.

For PHP:

PHPStan:

PHPStan scans your whole codebase and looks for both obvious & tricky bugs.

```
ju@Mac secure_web_app % vendor/bin/phpstan analyse .

 40/40 [▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓] 100%


 [OK] No errors

 💡 Tip of the Day:
PHPStan is performing only the most basic checks.
You can pass a higher rule level through the --level option
(the default and current level is 0) to analyse code more thoroughly.

ju@Mac secure_web_app %
```

As shown no errors/bugs where found.