1. Защита от XSS

Проблема состоит в том, что в исходной версии не все пользовательские данные экранировались перед выводом в HTML, что могло привести к выполнению вредоносного JavaScript-кода.

Для решения проблемы были внесены **исправления**:

1. Добавлено экранирование всех выводов пользовательских данных с помощью htmlspecialchars()
2. В шаблонизаторе реализовано автоматическое экранирование данных

Примеры изменений в коде:

```
// form.php - было
<div class="message"><?=
$message['html'] ?></div>
// form.php - стало
<div class="message"><?=
htmlspecialchars($message['html']) ?></div>

// template_helpers.php - добавлено автоматическое
экранирование
$value = htmlspecialchars($value);
```

2. Защита от Information Disclosure

Проблема заключается в том, что в исходной версии включен вывод всех ошибок, что могло привести к утечке чувствительной информации.

Исправления, необходимые для решения проблемы:

1. Удалены строки, включающие вывод ошибок в index.php
2. Учетные данные БД вынесены в отдельный конфигурационный файл (config.php)
3. Пароли хранятся в хешированном виде

Примеры изменений в коде:

```
// index.php - удалены строки
ini_set('display_errors' , 1);
ini_set('display_startup_errors' , 1);
error_reporting(E_ALL);
```

```
// config.php - создан
<?php
return [
    'db' => [
    'host' => 'localhost',
    'dbname' => 'u68596',
    'user' => 'u68596',
    'pass' => '2859691'
    ]
];
?>
```

3. Защита от SQL Injection

Проблема возникает вследствие того, что в исходной версии использовались прямые SQL – запросы без параметризации, что делало возможным внедрение
SQL-кода.
Исправления, внесенные для решения проблемы:
1. Все SQL-запросы переписаны с использованием подготовленных выражений
2. Внедрена параметризация всех запросов
Примеры изменений в коде:

```
// DatabaseRepository.php - было
$stmt = $this->db->query("SELECT * FROM application WHERE id = $id");
// DatabaseRepository.php - стало
$stmt = $this->db->prepare("SELECT * FROM application WHERE id = ?");
$stmt->execute([$id]);
```

4. Защита от CSRF

Проблема состоит в том, что в исходной версии отсутствовала защита от CSRF – атак, что позволяло выполнять действия от имени пользователя без его ведома.

Для решения проблемы были внесены следующие **исправления:**
1. Добавлена генерация CSRF-токена для каждой сессии
2. Реализована проверка токена при обработке POST-запросов
3. Токен добавлен во все формы
Примеры изменений в коде:

```
// admin.php - добавлены строки
if (empty($_SESSION['csrf_token'])) {
        $_SESSION['csrf_token'] =
        bin2hex(random_bytes(32));
}

// проверка токена
if (!isset($_POST['csrf_token']) || $_POST['csrf_token']
!== $_SESSION['csrf_token']) {
        die('Неверный CSRF-токен');
}

// добавление токена в формы
<input type="hidden" name="csrf_token" value="<?=
$_SESSION['csrf_token'] ?>">
```

5. Защита от Include и Upload уязвимостей

Проблема заключается в:
1. Возможности включения произвольных файлов через уязвимости в include
2. Отсутствии проверки загружаемых файлов
Исправления:
1. Для include уязвимостей:
   - Использование жестко заданных путей для включения файлов
   - Проверка существования файлов перед включением
2. Для upload уязвимостей:
   - Удален атрибут enctype="multipart/form-data" из форм, где загрузка файлов не требуется
   - Добавлена валидация типов файлов (если бы функционал загрузки был реализован)

Примеры изменений в коде:

```
// form.php - было
<form action="index.php" method="POST"
enctype="multipart/form-data">

// form.php - стало
<form action="index.php" method="POST" novalidate>
```

Заключение

В результате проведенной работы были устранены все основные уязвимости:

- Полностью защищено от XSS
- Реализованы базовые меры защиты от Information Disclosure
- Обеспечена полная защита от SQL Injection
- Реализована надежная защита от CSRF
- Приняты меры против Include и Upload уязвимостей