Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

# КВАЛІФІКАЦІЙНА РОБОТА
# Пояснювальна записка

рівень вищої освіти другий (магістерський)

**ДОСЛІДЖЕННЯ МЕТОДІВ ПІДВИЩЕННЯ БЕЗПЕКИ ВЕБЗАСТОСУНКІВ ДЛЯ PHP-ФРЕЙМВОРКІВ**
(тема)


Виконав:
студент 2 курсу, групи ІНФм-23-2
Ходонович А.Б.
(прізвище, ініціали)


Спеціальності 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник доц. Тітова О.В.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Кобилін О.А.
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)

Кафедра Інформатики
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2025 р.

# ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Ходоновичу Андрію Борисовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження методів підвищення безпеки вебзастосунків для PHP-фреймворків

затверджена наказом по університету від 25 листопада 2024 року № 1246Ст

2. Термін подання студентом роботи до екзаменаційної комісії 27 грудня 2024 р.

3. Вихідні дані до роботи науково-технічна література з методиками виявлення вразливостей вебзастосунків, інтегроване середовище розробки, методи побудови комплексної системи безпеки для вебзастосунків.

4. Перелік питань, що потрібно опрацювати в роботі
1. Аналіз існуючих методів захисту.
2. Дослідження найчастіших загроз та методів їх усунення.
3. Побудова комплексної системи безпеки навколо вебзастосунку.
4. Аналіз ефективності впроваджених методів безпеки.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) актуальність дослідження, об'єкт та мета дослідження, методи та підходи, проєктування вебзастосунка, реалізовані методи захисту, тестування безпеки вебзастосунка.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

| Найменування розділу | Консультант (посада, прізвище, ім'я, по батькові) | Позначка консультанта про виконання розділу | |
|---|---|---|---|
| | | підпис | дата |
| | | | |

**КАЛЕНДАРНИЙ ПЛАН**

| № з/п | Назва етапів роботи | Терміни виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Отримання завдання на кваліфікаційну роботу | 25.11.2024 | |
| 2 | Аналіз завдання, підбір літератури | 25.11.24-28.11.24 | |
| 3 | Аналіз літератури з досліджуваної проблеми | 28.11.24-01.12.24 | |
| 4 | Аналіз методів захисту | 01.12.24-05.11.24 | |
| 5 | Дослідження методів захисту | 05.12.24-07.12.24 | |
| 6 | Програмна реалізація | 07.12.24-18.12.24 | |
| 7 | Оформлення пояснювальної записки | 18.12.24-20.12.24 | |
| 8 | Перевірка на плагіат | 21.12.2024 | |
| 9 | Рецензування | 23.12.2024 | |
| 10 | Підготовка презентації та доповіді | 28.12.2024 | |
| 11 | Занесення роботи в електронний архів | 31.01.2025 | |
| 12 | Попередній захист кваліфікаційної роботи | 07.01.2025 | |

Дата видачі завдання 25 листопада 2024 р.

Студент _____
(підпис)

Керівник роботи _____ доц. Тітова О.В.
(підпис) (посада, прізвище, ініціали)

# РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 69 с., 4 табл., 39 рис., 35 джерел.

БЕЗПЕКА ВЕБЗАСТОСУНКІВ, PHP ФРЕЙМВОРКИ, LARAVEL, SQL-ІН'ЄКЦІЇ, УПРАВЛІННЯ ДОСТУПОМ, КОНТРОЛЬ СЕСІЙ, БАГАТОФАКТОРНА АВТЕНТИФІКАЦІЯ.

Об'єктом дослідження є підвищення безпеки вебзастосунків, розроблених з використанням PHP-фреймворків.

Метою дослідження є аналіз методів підвищення безпеки вебзастосуноків на основі PHP-фреймворків, розробка та впровадження підходів, спрямованих на захист від найпоширеніших загроз.

У ході дослідження проведено аналіз уразливостей, таких як SQL-ін'єкції, вразливості контролю доступу, міжсайтовий скриптинг (XSS), неправильне налаштування безпеки, вразливості автентифікації тощо. Розроблено та протестовано методи захисту з використанням параметризованих запитів, управління сесіями, багатофакторною автентифікацією та суворої політики доступу. Практична частина включає реалізацію цих механізмів у вебзастосунках на Laravel та їх тестування щодо стійкості до основних видів атак.

SECURITY OF WEB APPLICATIONS, PHP FRAMEWORKS, LARAVEL, SQL INJECTION, ACCESS CONTROL, SESSION CONTROL, MULTIFACTOR AUTHENTICATION.

The object of the study is to improve the security of web applications developed using PHP frameworks.

The goal of the research is to study and analyze methods for enhancing the security of web applications based on PHP frameworks, as well as to develop and implement approaches aimed at protecting against the most common threats.

The research includes an analysis of vulnerabilities such as SQL injections, access control vulnerabilities, cross-site scripting (XSS), security misconfiguration, authentication vulnerabilities, etc. Protection methods have been developed and tested, including the use of parameterized queries, session management, multi-factor authentication, and strict access policies. The practical part involves the implementation of these mechanisms in a Laravel web application and testing them for resistance to major types of attacks.

# ЗМІСТ

# ВСТУП

Сучасні вебпрограми відіграють ключову роль у забезпеченні цифрової взаємодії між бізнесом та користувачами, надаючи доступ до різних послуг та сервісів. Однак зі зростанням їхньої популярності збільшується і кількість загроз, пов’язаних із безпекою. Кібератаки, спрямовані на порушення конфіденційності даних, компрометацію облікових записів та пошкодження застосунків, стають дедалі витонченішими. У зв’язку з цим забезпечення безпеки вебзастосунків стає одним із першочергових завдань для розробників.

PHP залишається однією з найпопулярніших мов для розробки вебзастосунків, а фреймворки значно спрощують створення складних систем. Однак поширеність PHP-програм також робить їх привабливою мішенню для зловмисників. Проблеми, пов’язані з ін’єкціями, несанкціонованим доступом, уразливістю в аутентифікації та авторизації, залишаються актуальними, попри розвиток технологій. Це обумовлює необхідність глибокого аналізу методів захисту, вбудованих у PHP-фреймворки, а також розробки нових підходів, спрямованих на підвищення безпеки.

В рамках дослідження будуть розглянуті як загальні теоретичні підходи до безпеки вебзастосунків, так і конкретні практики, що використовуються в популярних фреймворках. Крім того, робота включає аналіз безпеки на рівні проєктування та коду, а також методів моніторингу та виявлення вторгнень.

Таким чином, представлена робота актуальна для розробників, які використовують PHP-фреймворки та фахівців з інформаційної безпеки. Результати дослідження дозволять розробити більш захищені вебпрограми, мінімізувати ризики та підвищити стійкість систем до зовнішніх загроз.

# 1 ОГЛЯД МЕТОДІВ ПІДВИЩЕННЯ БЕЗПЕКИ ВЕБЗАСТОСУНКІВ

## 1.1 Введення в безпеку вебзастосунків

Вебпрограми є невід'ємною частиною сучасної інфраструктури, що робить їх привабливою метою для зловмисників. Регулярно ми чуємо про те, що вебсайти стають недоступними через атаки типу «відмова в обслуговуванні» або відображають змінену (і часто шкідливу) інформацію на своїх головних сторінках. В інших гучних випадках мільйони паролів, адрес електронної пошти та даних кредитних карток знаходяться у відкритому доступі, наражаючи користувачів вебсайтів як на особисті незручності, так і на фінансові ризики. Мета веббезпеки полягає в запобіганні цих (або інших) видів атак. Більш формальним визначенням веббезпеки є способи захисту вебсайтів від несанкціонованого доступу, використання, зміни, знищення або порушення роботи.

### 1.1.1 Основні принципи безпеки вебзастосунків

Безпека вебзастосунків базується на низці принципів, спрямованих на захист даних, користувачів та систем від несанкціонованого доступу та атак. Ці принципи забезпечують фундамент для створення застосунків, стійких до поширених загроз та вразливостей. Розглянемо ключові з них:

– принцип мінімізації прав (Least Privilege). Кожному компоненту системи або користувачу повинні надаватися лише ті права та доступи, які необхідні для виконання конкретних завдань. Це допомагає знизити ризик потенційних збитків, якщо одна з ланок системи буде скомпрометована;

– захист даних на всіх рівнях (Defense in Depth). Використання багаторівневих механізмів захисту даних як на стороні сервера, так і клієнта. Це включає шифрування даних при передачі та зберіганні, використання

захищених протоколів (наприклад, HTTPS), а також ізоляцію критичних компонентів системи;

– принцип стандартної відмови (Fail-Safe Defaults). У разі виникнення помилки або нештатної ситуації, система повинна повертатися до безпечного стану. За стандартом доступ до ресурсів повинен бути закритий, якщо не надано явних прав доступу;

– валідація всіх вхідних даних (Input Validation). Вхідні дані повинні проходити строгу перевірку на допустимість. Це важливо для запобігання ін’єкційним атакам, таким як SQL-ін’єкції або XSS. Валідація даних дозволяє контролювати їх формат та зміст, не допускаючи виконання шкідливого коду;

– принцип сегрегації обов’язків (Separation of Duties). Відповідальності у системі мають бути розподілені таким чином, щоб зловмисник не міг легко отримати повний контроль над усіма критичними функціями. Наприклад, адміністрування системи та управління безпекою повинні здійснюватися різними користувачами;

– аутентифікація та авторизація (Authentication and Authorization). Важливим аспектом безпеки є правильна реалізація механізмів автентифікації та авторизації. Аутентифікація підтверджує особистість користувача, а авторизація визначає, які ресурси можна використовувати;

– регулярні оновлення та патчі. Підтримка системи в актуальному стані шляхом своєчасної установки оновлень та виправлень уразливостей. Це допомагає знизити ризик атак, які використовують відомі вразливості програмного забезпечення;

– логування та моніторинг подій безпеки. Для виявлення та запобігання потенційним атакам важливо організувати систему логування та моніторингу активності в застосунку. Це дозволяє оперативно реагувати на підозрілі дії та аналізувати інциденти для покращення захисту.

Основні принципи безпеки вебзастосунків забезпечують фундамент захисту даних і користувачів від потенційних загроз. Ці принципи, такі як мінімізація прав, захист даних на всіх рівнях та сувора валідація вхідних

даних, допомагають знизити ризик поширених атак. Дотримання цих підходів дозволяє розробникам створювати більш стійкі до уразливостей застосунки та захищати їх від сучасних кіберзагроз.

### 1.1.2 Короткий огляд сучасних загроз та вразливостей вебзастосунків

У цьому підрозділі представлений огляд найпоширеніших вразливостей, заснований на рекомендаціях OWASP Top 10 – авторитетного джерела безпеки вебзастосунків, який регулярно оновлюється відповідно до поточних кіберзагроз (рис. 1.1).


Рисунок 1.1 – Топ 10 вразливостей на основі даних від OWASP

– порушення контролю доступу: неправильне налаштування контролю доступу дозволяє зловмисникам отримати несанкціонований доступ до даних або функціональності, які мають бути обмежені;

– криптографічні збої: уразливості пов’язані з неналежним використанням криптографічних алгоритмів або їх повною відсутністю.

Неправильне шифрування даних, незахищені ключі та слабкі алгоритми можуть призвести до витоків конфіденційної інформації, включаючи паролі, персональні дані та фінансову інформацію;

– ін’єкційні атаки: ін’єкції, такі як SQL-ін’єкції та ін’єкції LDAP, відбуваються, коли неперевірені дані впроваджуються в команди, що передаються серверу. Це дозволяє зловмиснику маніпулювати запитами та отримувати несанкціонований доступ до даних або функцій програми;

– небезпечний дизайн: уразливості в архітектурі та проєктуванні системи можуть спричинити серйозні проблеми безпеки. Небезпечний дизайн часто виникає на етапі розробки програми та включає недостатнє опрацювання заходів захисту, що може зробити застосунок вразливим до широкого спектра атак;

– помилки в конфігурації безпеки: помилки в налаштуваннях безпеки є однією з найчастіших вразливостей. Це може включати використання застарілого програмного забезпечення, незахищені служби, неправильні параметри конфігурації та недостатні заходи для захисту серверів та баз даних;

– вразливі та застарілі компоненти: використання застарілих чи вразливих бібліотек, плагінів, фреймворків чи інших компонентів може надати зловмисникам можливість експлуатації відомих уразливостей. Регулярне оновлення та патчінг всіх зовнішніх компонентів життєво важливе для підтримки безпеки програми;

– збої ідентифікації та аутентифікації: недоліки в механізмах ідентифікації та аутентифікації можуть дозволити зловмисникам обійти систему захисту та отримати доступ до ресурсів. Це може бути викликане слабким керуванням сесіями, відсутністю багатофакторної автентифікації або поганим керуванням паролями;

– помилки цілісності даних та програмного забезпечення: ця категорія вразливостей включає недоліки в процесі оновлення програмного забезпечення, а також маніпуляції з даними в реальному часі. Наприклад,

використання недовірених джерел для оновлення програмного забезпечення може призвести до компрометації всієї системи;

– недоліки журналювання та моніторингу безпеки: недостатнє журналювання та моніторинг роблять системи вразливими до тривалих атак, які можуть залишитися непоміченими. Адекватне ведення логів та відстеження активності користувачів дозволяє своєчасно виявляти загрози та реагувати на інциденти безпеки;

– атаки з підробкою серверних запитів (SSRF): SSRF дозволяє зловмиснику надсилати запити на сервер від імені програми. Ці атаки можуть використовуватися для отримання доступу до внутрішніх систем, даних або виконання шкідливих дій через довірені сервери.

Ці вразливості, описані в OWASP Top 10 2021, досі залишаються актуальними для вебзастосунків. Знання цих загроз та застосування заходів щодо їх усунення – необхідний крок для створення безпечних та стійких до атак застосунків.

## 1.2 Методи захисту вебзастосунків

Зі збільшенням кількості загроз, спрямованих на вебпрограми, розробка та впровадження надійних методів захисту стали обов’язковою частиною життєвого циклу розробки програмного забезпечення. Сучасні методи безпеки спрямовані на запобігання найбільш поширеним уразливості та забезпечення захисту даних, а також на мінімізацію ризиків компрометації систем.

Методи захисту вебзастосунків включають як проактивні заходи, такі як керування доступом та шифрування даних, так і реактивні – регулярне оновлення програмного забезпечення, моніторинг та аналіз подій безпеки. У рамках цього розділу будуть розглянуті ключові підходи та механізми, які використовуються для захисту вебзастосунків на різних рівнях їхньої архітектури.

### 1.2.1 Керування доступом, автентифікацією та авторизацією

Керування доступом, автентифікація та авторизація є основними механізмами захисту вебзастосунків. Вони спрямовані на те, щоб надати доступ до ресурсів та функціональності лише авторизованим користувачам з відповідними правами, мінімізуючи можливість зловживання системними функціями та доступом до даних.

Автентифікація – це процес підтвердження особи користувача. Це перший етап в управлінні доступом, який встановлює, що користувач є тим, за кого себе видає. Основні методи аутентифікації:

– паролі. Найпоширеніший метод, проте використання слабких чи загальнодоступних паролів робить систему вразливою. Найкращі практики включають вимоги до складних паролів, регулярне оновлення та використання хешування (наприклад, алгоритмів bcrypt або Argon2) для їх зберігання;

– багатофакторна автентифікація (MFA). Додатково до пароля користувачі підтверджують свою особу за допомогою другого фактору, наприклад, одноразового пароля, SMS-коду або біометричного сканування. Це значно підвищує рівень безпеки, особливо у випадках компрометації пароля;

– OAuth та OpenID Connect. Ці механізми дозволяють користувачам авторизуватися через інші сервіси (наприклад, Google, Facebook). При цьому знижується необхідність керування паролями в самому застосунку, що зменшує ризик витоку.

Авторизація – це процес перевірки прав доступу, що визначає, до яких ресурсів або операцій користувач має право. Це відбувається після автентифікації. Авторизація зазвичай здійснюється на основі різних підходів:

– рольове керування доступом (RBAC). Доступ до ресурсів визначається залежно від ролі користувача (наприклад, адміністратор або звичайний користувач). Кожна роль має наперед визначений набір дозволів. Переваги

RBAC полягають у простоті управління: адміністратори системи можуть легко призначати ролі та змінювати набір дозволів для конкретних груп користувачів. Крім того, цей підхід дозволяє дотримуватися принципу мінімальних привілеїв, що мінімізує ризики випадкового або навмисного несанкціонованого доступу до конфіденційних даних. Однак, однією з недоліків цієї моделі є її обмежена гнучкість – у великих організаціях, де користувачі можуть виконувати різні завдання залежно від контексту, RBAC може бути занадто жорстким;

– керування доступом на основі атрибутів (ABAC). Це складніша модель, в якій доступ залежить від різних атрибутів, наприклад, розташування, пристрої, час дня або посади, що дозволяє будувати більш гнучкі політики. Ця модель управління доступом дає можливість створити величезну кількість комбінацій умов для різних політик безпеки. Основна перевага ABAC - це його гнучкість і здатність врахувати безліч умов прийняття рішення про надання доступу. Однак цей підхід також складніший у реалізації та потребує великих зусиль на етапі проєктування та управління політиками;

– управління доступом на основі відносин (ReBAC). ReBAC – це відносно нова модель управління доступом, в якій рішення про надання доступу приймаються на основі відносин між суб’єктами (користувачами) та об’єктами (ресурсами). Ця модель є особливо актуальною для соціальних мереж та платформ спільної роботи, де доступ до даних часто залежить від динамічних зв’язків між користувачами. Наприклад, якщо один користувач є менеджером іншого, менеджер може отримати доступ до його звітів. ReBAC дозволяє гнучко керувати доступом в середовищах з високою динамікою взаємодій. Недоліками є складність моделювання та реалізації, а також модель вимагає постійного моніторингу та управління відносинами;

– керування доступом на основі списків контролю доступу (ACL). Це класичний механізм контролю доступу, який ґрунтується на списках дозволів для кожного ресурсу. В ACL кожен об’єкт має список користувачів або груп, які мають право виконувати певні дії, такі як читання, запис або виконання.

Кожен об’єкт системи має власний ACL, який перераховує, які користувачі можуть виконувати якісь дії. Наприклад, файл може мати права, що дозволяють одному користувачеві його читання, іншому – запис, а третьому – виконання. Основними недоліками є складність управління в масштабних системах з безліччю користувачів та ресурсів та погана масштабованість в умовах динамічних змін прав доступу.

Кожна з описаних моделей контролю доступу має свої переваги та недоліки. Вибір моделі залежить від потреб конкретної організації, рівня її безпеки та складності процесів.

Авторизація у вебзастосунках часто здійснюється через системи токенів, наприклад JSON Web Tokens, які перевіряють права користувача при кожному запиті.

Також одним із критично важливих аспектів захисту вебзастосунків та систем є безпечне керування сесіями. Сесії дозволяють користувачам взаємодіяти з застосунком, і їхня безпека безпосередньо впливає на захист даних та конфіденційність користувачів. Розглянемо основні методи забезпечення безпеки сесій:

– використання захищених файлів cookie. Для зберігання інформації про сесії зазвичай використовуються файли cookie. Важливо переконатися, що вони захищені від вразливостей, таких як атаки на міжсайтовий скриптинг (XSS). Для цього слід використовувати прапори HttpOnly та Secure. Також важливо використовувати сесійні ідентифікатори, які є унікальними для кожного сеансу і генеруються випадковим чином, щоб уникнути передбачуваності та спростити можливі атаки;

– автоматичне завершення сесій – важливий захід безпеки, який допомагає знизити ризик несанкціонованого доступу, особливо на загальнодоступних або спільно використовуваних пристроях. Сесії, в яких немає активних дій користувача протягом певного часу, повинні автоматично завершуватися;

– захист від перехоплення сесій. Перехоплення сесій – це одна з найпоширеніших атак, коли зловмисник отримує доступ до сесії користувача, щоб виконувати дії від його імені. Для захисту від таких атак використовують протоколи шифрування, такі як TLS або SSL, одноразові токени або часті зміни ідентифікаторів сесій. Також важливо відстежувати аномальну активність у сесіях користувачів, таку як спроби доступу з різних IP-адрес або пристроїв у короткі проміжки часу. У разі виявлення такої активності сесія може бути автоматично завершена, а користувач повідомлений про підозрілу активність.

### 1.2.2 Обробка вхідних даних та запобігання ін’єкціям

Обробка вхідних даних – важливий аспект безпеки вебзастосунків, оскільки саме через некоректну або неочищену інформацію користувача зловмисники можуть впровадити шкідливий код і отримати доступ до даних або системних ресурсів. Однією з найнебезпечніших атак, пов’язаних з некоректною обробкою вхідних даних, є ін’єкція, яка залишається серед найпоширеніших вразливостей вебзастосунків.

Будь-яка інформація, що отримується через форми введення, URL-адреси, заголовки HTTP-запитів та cookies, повинна розглядатися як потенційно шкідлива, оскільки зловмисники можуть спеціально модифікувати вхідні дані з метою обходу захисних механізмів та здійснення атак на застосунок.

У зв’язку з цим валідація та фільтрація даних є фундаментальними заходами для запобігання різним типам атак.

Валідація даних це процес перевірки відповідності введених даних очікуваним форматам, типам і діапазонам значень. Цей процес необхідний для того, щоб переконатися, що дані є коректними та безпечними для подальшої обробки у застосунку. Наприклад, якщо поле форми очікує числове значення,

то рядки, які містять текст або спеціальні символи, повинні бути автоматично відхилені. Також валідація може містити перевірку відповідності даних специфічним шаблонам, наприклад, у випадку з адресами електронної пошти або телефонними номерами.

Важливо відзначити, що валідація даних повинна здійснюватися як на стороні клієнта, так і на стороні сервера. Перевірка на стороні клієнта (наприклад, за допомогою JavaScript) може запобігти надсиланню неправильних даних користувачем. Однак, покладатися тільки на валідацію на стороні клієнта не можна, оскільки зловмисники можуть обійти її, змінюючи запити безпосередньо через інструменти розробки браузера або спеціальні програми. Тому основна й обов’язкова валідація повинна завжди відбуватися на стороні сервера.

Фільтрація даних – це процес видалення або екранування потенційно небезпечних символів, які можуть бути використані для проведення атак, таких як XSS або SQL-ін’єкції. Небезпечні символи, такі як кутові дужки "<" та ">", часто застосовуються для впровадження шкідливого коду в вебсторінки. Наприклад, в разі атаки XSS зловмисник може вставити JavaScript-код, який буде виконаний в браузері жертви.

Для ефективної фільтрації рекомендується використовувати білі списки припустимих значень. Білі списки є набором строго дозволених символів, та все, що не входить до цього списку, автоматично відхиляється. Це набагато безпечніший підхід, тому що в чорних списках можна не врахувати всі можливі шкідливі символи або їх комбінації.

Також для запобігання атак типу переповнення буфера або зловживання надмірними розмірами запитів, наприклад, DoS-атаки, необхідно встановлювати обмеження на довжину даних. Якщо довжина введених даних не контролюється, зловмисник може спробувати надіслати занадто довгі рядки або запити, що може призвести до збоїв у роботі програми або небажаного навантаження на сервер. Прикладом може бути обмеження довжини рядка під час введення пароля або адреси електронної пошти. Навіть якщо дані

проходять валідацію та фільтрацію, вони мають бути обмежені в розумних межах для захисту системи від навантаження.

Якщо ж розглядати ін’єкційні атаки, існує кілька типів ін’єкцій:

SQL-ін’єкція – це атака, яка може призвести до серйозних наслідків, таких як витік даних, їх зміна або навіть повне видалення. Зловмисник маніпулює SQL-запитом, вводячи спеціально відформатовані дані через форми введення або параметри запитів. Якщо вебзастосунок не застосовує строгу фільтрацію та екранування, зловмисник може змінювати SQL-запити таким чином, щоб отримати доступ до даних, на які він не має дозволу, обійти автентифікацію або виконати інші шкідливі дії.

Приклад класичної SQL-ін’єкції: зловмисник вводить у поле логіна рядок виду «OR 1 = 1», що може призвести до зміни логіки SQL-запиту таким чином, що він завжди повертає істину, надаючи доступ без введення правильних облікових даних.

Міжсайтовий скриптинг (XSS) – це тип ін’єкційної атаки, при якій зловмисник впроваджує шкідливий код (найчастіше JavaScript) у вебсторінку, яка згодом виконується у браузері жертви. Існує кілька типів XSS-атак:

– reflected XSS – атака, при якій шкідливий код впроваджується через запит користувача і негайно повертається сервером у відповіді. На сьогодні є найпоширенішою XSS-атакою;

– stored XSS – атака, коли шкідливий код зберігається на сервері, наприклад, у базі даних, і відправляється іншим користувачам під час завантаження вебсторінки. Є найбільш руйнівним типом атаки;

– DOM XSS – різновид XSS, коли атака відбувається на стороні клієнта, маніпулюючи об’єктною моделлю документа (DOM) через JavaScript.

Наслідки успішної XSS-атаки можуть бути різноманітними: крадіжка сесійних куки, заміна контенту сторінки, виконання дій від імені користувача або перенаправлення його на шкідливі сайти.

Командна ін’єкція – це атака, при якій зловмисник впроваджує свої команди в системні виклики, що виконуються на сервері. Це можливо, якщо

вебзастосунок використовує введення користувача для формування команд операційної системи, не забезпечуючи при цьому належної перевірки та екранування даних. Наприклад, у PHP функції exec(), shell_exec() можуть використовуватися для виконання команд на рівні системи, що при недостатньому захисті відкриває можливість для зловмисника виконати будь-які довільні команди.

Наслідки такої атаки можуть містити повний компрометуючий контроль над сервером, зміну файлів, отримання конфіденційної інформації та порушення роботи системи. Командна ін’єкція є однією з найнебезпечніших атак, оскільки дозволяє зловмиснику безпосередньо взаємодіяти з операційною системою.

### 1.2.3 Захист та шифрування даних

Шифрування даних – це важливий метод захисту, спрямований на забезпечення конфіденційності як при передачі, так і при зберіганні інформації. Шифрування дозволяє зробити дані недоступними для третіх осіб, навіть якщо вони отримають фізичний або логічний доступ до них.

Для захисту даних, що передаються між клієнтом та сервером, використовуються такі протоколи, як TLS або його попередник SSL. Ці протоколи забезпечують шифрування каналу зв’язку, запобігаючи перехопленню трафіку або атаки типу «Man-in-the-Middle». Це критично важливо для вебсайтів, що обробляють особисті дані чи фінансову інформацію

Усі конфіденційні дані, такі як паролі, номери кредитних карток та персональна інформація, повинні зберігатися у зашифрованому вигляді. Широко використовуються такі алгоритми як AES для симетричного шифрування і RSA для асиметричного шифрування. Для зберігання паролів рекомендується використовувати алгоритми хешування із сіллю, такі як

bcrypt, Argon2 або PBKDF2, які захищають дані від атак методом перебору (brute force) навіть у разі витоку бази даних.

При захисті даних так само дуже важливо правильне використання ключів шифрування. Без надійного управління ключами ефективність самої криптографічної системи може бути зведена нанівець.

Приклади надійного керування ключами включають:

– поділ ключів та даних. Шифрувальні ключі не повинні зберігатися разом із зашифрованими даними, оскільки це може призвести до їх компрометації у разі витоку;

– регулярна ротація ключів. Постійне використання одного і того ж ключа збільшує ризик його злому або витоку. Регулярна зміна ключів мінімізує цей ризик. Наприклад, для ротації ключів можна використовувати автоматизовані системи керування ключами;

– використання апаратних модулів безпеки (HSM). HSM – це спеціалізовані пристрої, розроблені для безпечного зберігання та керування криптографічними ключами. Ці пристрої забезпечують захист від несанкціонованого доступу і можуть використовуватися для генерації, зберігання та керування ключами, що особливо важливо для захисту даних, які потребують високого ступеня конфіденційності.

Для запобігання несанкціонованому поширенню конфіденційної інформації застосовуються системи запобігання витоку даних DLP.

DLP-системи аналізують вихідний трафік і можуть блокувати або попереджати про передачу конфіденційної інформації за межі корпоративної мережі.

## 1.3 Аналіз підходів до безпеки в різних PHP-фреймворках

Вебфреймворки відіграють ключову роль у розробці сучасних програм та надають вбудовані механізми для підвищення безпеки. Завдяки вбудованим функціям та бібліотекам вони можуть значно спростити впровадження

передових заходів захисту та забезпечити безпеку на рівні коду. Основна перевага безпеки на рівні фреймворків полягає в тому, що багато з поширених уразливостей можуть бути усунені або знижені шляхом правильного налаштування та використання фреймворків.

У світі PHP існує кілька популярних фреймворків, які пропонують свої вбудовані рішення для підвищення безпеки вебзастосунків. Серед найбільш популярних фреймворків PHP можна виділити Laravel, Symfony та CodeIgniter. Кожен з них включає різні інструменти та практики безпеки, які допомагають розробникам захищати свої програми. У цьому розділі ми розглянемо основні особливості цих фреймворків у контексті забезпечення безпеки, їх сильні та слабкі сторони.

Laravel – один з найпопулярніших PHP-фреймворків, що широко використовується для розробки вебзастосунків завдяки своїй простоті, елегантній синтаксичній структурі та великій кількості вбудованих функцій. Він пропонує кілька корисних інструментів для забезпечення безпеки:

– захист від CSRF. Laravel пропонує вбудований захист від міжсайтової підробки запитів за допомогою токенів CSRF;

– захист від SQL ін’єкцій. Laravel використовує Eloquent ORM для взаємодії з базою даних, що робить запити безпечнішими шляхом екранування та параметризації;

– хешування паролів. Для безпеки зберігання паролів Laravel використовує алгоритм bcrypt, який автоматично додає сіль до пароля перед його хешуванням;

– шифрування даних. Laravel надає вбудовані механізми шифрування та дешифрування даних із використанням безпечних ключів.

Основні недоліки та переваги показано у таблиці 1.1.

Таблиця 1.1 – Переваги та недоліки Laravel

| **Переваги** | **Недоліки** |
|---|---|
| Простота та зручність впровадження заходів безпеки завдяки широким вбудованим засобам. | Автоматизація багатьох аспектів безпеки може приховувати потенційні проблеми для тих, хто недостатньо добре розуміє внутрішні механізми. |
| Активна спільнота та регулярні оновлення, що сприяє швидкому виправленню вразливостей. | Залежність від вбудованих рішень безпеки може обмежити гнучкість за необхідності застосування кастомних чи нестандартних підходів. |
| Висока інтеграція із сучасними технологіями, такими як OAuth, JWT, та API, що розширює можливості з налаштування безпеки. | |

Symfony – це потужний та гнучкий PHP-фреймворк, який часто використовується для створення складних корпоративних програм. Він славиться своєю модульною структурою та гнучкими інструментами для реалізації безпеки.

– модуль безпеки. Symfony надає сучасну систему безпеки, яка дозволяє гнучко налаштовувати аутентифікацію та авторизацію користувачів з використанням ролей, політик доступу та додаткових рівнів перевірки прав;

– захист від XSS та SQL-ін’єкцій. Symfony пропонує строгі механізми для фільтрації та екранування даних користувача;

– HTTP Security Headers. Symfony надає вбудовані засоби для налаштування політики безпеки HTTP, такі як Content Security Policy (CSP), Strict-Transport-Security (HSTS) та інші заголовки, які захищають від атак на рівні браузера, таких як XSS та clickjacking.

Основні недоліки та переваги показано у таблиці 1.2.

Таблиця 1.2 – Переваги та недоліки Symfony

| Переваги | Недоліки |
|---|---|
| Висока гнучкість у налаштуванні безпеки, що дозволяє детально контролювати всі аспекти захисту програми. | Symfony складніше в освоєнні в порівнянні з Laravel, що може збільшити час на навчання та впровадження. |
| Підтримка корпоративних стандартів безпеки та комплексних рішень, що робить Symfony привабливим вибором для великих проектів. | Точне налаштування системи безпеки потребує уважності, оскільки некоректна конфігурація може призвести до вразливостей. |

CodeIgniter – це легкий і швидкий PHP-фреймворк, який пропонує мінімальний набір функцій для створення вебзастосунків, включаючи базові засоби безпеки.

– захист від XSS. Включає вбудований фільтр для захисту від XSS-атак;

– захист від SQL ін’єкцій. Фреймворк забезпечує екранування та фільтрацію запитів до бази даних, що допомагає запобігти SQL-ін’єкції;

– сесії та cookies. CodeIgniter надає підтримку безпечних сесій, у тому числі із шифруванням cookies, що захищає дані користувача від перехоплення.

Таблиця 1.3 – Переваги та недоліки CodeIgniter

| Переваги | Недоліки |
|---|---|
| Простота та висока продуктивність, що робить CodeIgniter відповідним вибором для невеликих проєктів та застосунків з обмеженими вимогами до безпеки. | Обмежені вбудовані механізми безпеки. Багато функцій потребують додаткового налаштування або використання сторонніх бібліотек. |
| Легкість в освоєнні та мінімальні вимоги до налаштування роблять його ідеальним для розробників-новачків. | Найменша гнучкість у питаннях безпеки порівняно з більш комплексними фреймворками, такими як Laravel та Symfony, що може бути критичним недоліком для великих проєктів із високими вимогами до захисту даних. |

Вибір фреймворку для розробки залежить від специфіки проєкт та вимог до безпеки. Важливо враховувати як вбудовані механізми захисту, так і можливості кастомізації, щоб забезпечити надійний захист від сучасних загроз.

## 1.4 Постановка задачі дослідження

Вебпрограми на PHP-фреймворках продовжують займати значну частку у світі розробки, але при цьому залишаються вразливими перед безліччю загроз. Сучасні кібератаки стають все більш витонченими, що вимагає застосування ефективніших методів захисту на рівні коду та архітектури застосунків. Це робить дослідження в галузі підвищення безпеки вебзастосунків на основі PHP-фреймворків особливо актуальним.

Об’єктом дослідження є методи підвищення безпеки вебзастосунків, розроблених з використанням PHP-фреймворків.

Метою дослідження є вивчення та аналіз методів підвищення безпеки вебзастосунок на основі PHP-фреймворків, розробка та впровадження підходів, спрямованих на захист від найпоширеніших загроз.

Для досягнення поставленої мети необхідно вирішити такі завдання:

– провести огляд сучасних методів та інструментів для підвищення безпеки вебзастосунків, розроблених на PHP-фреймворках;

– вивчити найпоширеніші загрози для вебзастосунків, розроблених на PHP, та методи їх запобігання;

– розробити та запропонувати підходи та практичні рекомендації щодо покращення безпеки PHP-застосунків з урахуванням поточних загроз;

– впровадити запропоновані методи на практиці та оцінити їх ефективність на основі створених тестових застосунків.

# 2 ОПИС ТА МОДЕЛЮВАННЯ МЕТОДІВ БЕЗПЕКИ У ФРЕЙМВОРКУ LARAVEL

## 2.1 Управління доступом, автентифікацією та авторизацією

Керування доступом, автентифікація та авторизація – це ключові аспекти безпеки вебзастосунків, спрямовані на захист ресурсів та даних від несанкціонованого використання. Основна мета цих процесів – забезпечити доступ тільки для тих користувачів та систем, яким він дозволений, та обмежити всі інші спроби взаємодії. Цей розділ описує модель, що поєднує три важливі компоненти: автентифікацію, авторизацію та контроль доступу.

Автентифікація – це процес перевірки автентичності користувача або системи для підтвердження їх прав доступу. Найбільш поширений спосіб – введення логіна та пароля, але сучасні методи включають:

– двофакторну (2FA) та багатофакторну автентифікацію (MFA), наприклад, з використанням OTP;

– OAuth та OpenID – протоколи для аутентифікації через сторонні послуги, наприклад, Google та Facebook;

– JWT (JSON Web Token) для зберігання та передачі даних про аутентифікацію у зашифрованому вигляді.

Приклад реалізації JWT-автентифікації з використанням Laravel показаний на рисунку 2.1.

```php
use Firebase\JWT\JWT;
use Firebase\JWT\Key;

/**
 * Генерация JWT для пользователя.
 *
 */
function generateJWT($user) {
    $key = "secret_key";  // Секретный ключ для подписи токена
    $payload = [
        'iss' => "your-app-name",  // Источник токена (Issuer)
        'sub' => $user->id,        // Идентификатор пользователя (Subject)
        'iat' => time(),           // Время создания токена (Issued At)
        'exp' => time() + 3600     // Время истечения срока действия (1 час)
    ];

    return JWT::encode($payload, $key, 'HS256');
}

/**
 * Верификация и декодирование JWT токена.
 */
function verifyJWT($jwt) {
    $key = "secret_key";  // Секретный ключ для проверки подписи токена
    try {
        return JWT::decode($jwt, new Key($key, 'HS256'));
    } catch (Exception $e) {
        return false;  // В случае ошибки вернуть false
    }
}
```

Рисунок 2.1 – Реалізація JWT-автентифікації

Авторизація визначає, які дії автентифікований користувач має право виконувати у системі. У вебзастосунках часто використовується рольова модель доступу (RBAC) або доступ на основі атрибутів (ABAC).

Рольова модель доступу формалізується через кілька множин:

– $U$ – множина користувачів;

– $R$ – множина ролей;

– $P$ – множина дозволів;

– $UA \subseteq U \times R$ – призначення ролей користувачам;

– $PA \subseteq R \times P$ – призначення дозволів ролям.

Математичне правило RBAC виглядає як:

$$(u, p) = \exists r \in R \text{ таке, що } (u, r) \in UA \text{ та } (r, p) \in PA, \qquad (2.1)$$

Це означає, що користувач $u$ має доступ до дії $p$, якщо йому призначена роль $r$, яка має дозвіл на виконання дії $p$.

Реалізація RBAC показана на рисунку 2.2.

```php
// Пользователь назначен определенной роли
$user_role = $user->role;

// Проверяем, соответствует ли роль требуемым правам для действия
function hasAccess($role, $permission) {
    $permissions = [
        'admin' => ['edit', 'delete', 'create'],
        'user' => ['view', 'comment'],
    ];
    return in_array($permission, $permissions[$role]);
}

// Пример использования
if (hasAccess($user_role, 'edit')) {
    // Действие разрешено
} else {
    // Доступ запрещен
}
```

Рисунок 2.2 – Алгоритм керування доступом через RBAC

Контроль доступу на основі атрибутів розширює RBAC, додаючи залежність від властивостей користувача або ресурсу.

$$(u, p, \text{c}) = \exists r \in R, a \in A \text{ таке, що } (u, r) \in UA, (r, p) \in PA \text{ та } (u, a) \in C, \quad (2.2)$$

де $C$ – множина умов, що залежать від атрибутів;

$a$ – атрибут користувача чи ресурсу.

Це означає, що запит доступу від користувача $u$ до дії $p$ у контексті $c$ буде схвалений, якщо знайдуться відповідні ролі та атрибути, які виконують наступні умови:

– у користувача $u$ є роль $r$;

– роль $r$ дозволяє виконання дії $p$;

– умова $c$ дозволяє цю дію з урахуванням атрибутів $a$.

Приклад реалізації ABAC у Laravel показаний на рисунку 2.3.

```php
public function handle(Request $request, Closure $next)
{
    $currentTime = date('H:i');
    $userRole = $request->user()->role; // Получение роли пользователя
    $userIp = $request->ip(); // Получение IP-адреса пользователя

    // Условие доступа на основе времени и роли
    if ($currentTime >= '08:00' && $currentTime <= '18:00' && $userRole === 'admin') {
        return $next($request);
    }

    // Блокировка доступа для определённых IP-адресов
    if ($userIp === '192.168.0.1') {
        return response('Access denied from this IP.', 403);
    }

    return response('Access denied, outside business hours or insufficient privileges.', 403);
}
```

Рисунок 2.3 – Алгоритм керування доступом через ABAC

## 2.2 Шифрування даних та управління ключами

Шифрування даних відіграє ключову роль захисту інформації як у стані спокою, так і при передачі. У цьому розділі розглядаються алгоритми симетричного та асиметричного шифрування, а також методи безпечного керування ключами.

Симетричне шифрування засноване на використанні одного й того самого ключа для шифрування та дешифрування даних. Нехай:

– $M$ – вихідне повідомлення;

– $C$ – зашифроване повідомлення;

– $K$ – секретний ключ.

Шифрування являє собою перетворення $E_K(M) = C$, а дешифрування – зворотне перетворення $D_K(C) = M$. Використовується один ключ $K$. Прикладом симетричного шифрування є алгоритм AES.

Laravel для криптографії використовується вбудована підтримка бібліотеки OpenSSL через компонент «Laravel Encryption». Для симетричного шифрування використовується AES-256 або AES-128 у режимі CBC (рис 2.4).

```php
use Illuminate\Support\Facades\Crypt;

$data = 'Sensitive data';

// Шифрование данных
$encrypted = Crypt::encrypt($data);

// Дешифрование данных
$decrypted = Crypt::decrypt($encrypted);
```

Рисунок 2.4 – Симетричне шифрування та розшифровка

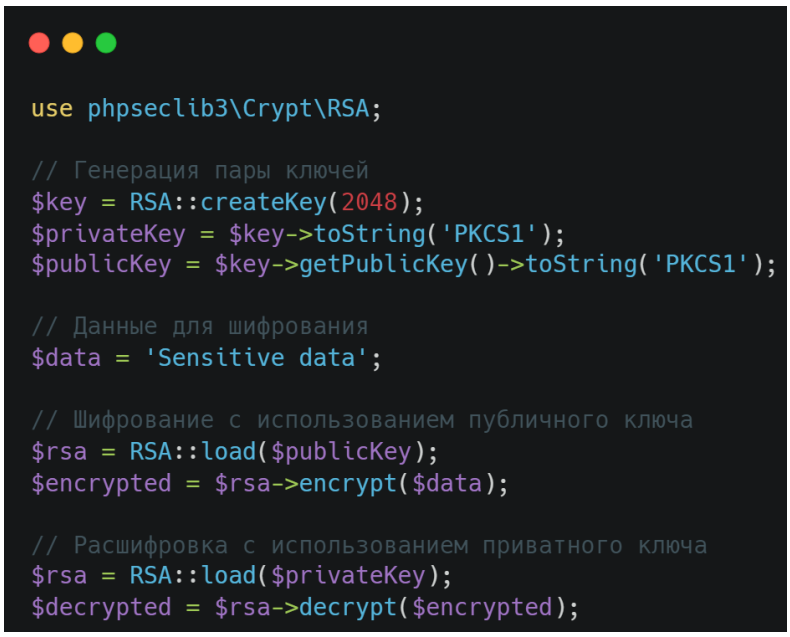
Асиметричне шифрування використовує два ключі – відкритий та закритий. Нехай:

– $K_{pub}$ – відкритий ключ;

– $K_{priv}$ – закритий ключ.

Шифрування виконується з використанням відкритого ключа $E_{K_{pub}}(M) = C$, а дешифрування – з використанням закритого ключа $D_{K_{priv}}(C) = M$. Приклад асиметричного шифрування є алгоритм RSA.

Laravel за умовчанням не підтримує асиметричне шифрування RSA безпосередньо через свої фасади. Для реалізації зазвичай використовують бібліотеку phpseclib (рис 2.5).



```php
use phpseclib3\Crypt\RSA;

// Генерация пары ключей
$key = RSA::createKey(2048);
$privateKey = $key->toString('PKCS1');
$publicKey = $key->getPublicKey()->toString('PKCS1');

// Данные для шифрования
$data = 'Sensitive data';

// Шифрование с использованием публичного ключа
$rsa = RSA::load($publicKey);
$encrypted = $rsa->encrypt($data);

// Расшифровка с использованием приватного ключа
$rsa = RSA::load($privateKey);
$decrypted = $rsa->decrypt($encrypted);
```

Рисунок 2.5 – Асиметричне шифрування

Керування ключами – це важливий процес забезпечення безпеки шифрування, який включає генерацію, зберігання, ротацію та знищення ключів.

У Laravel ключ шифрування генеруються за допомогою команди *«php artisan key: generate»*. Після виконання команда додає ключ до .env виду APP_KEY=base64:xxxxxxxxxxxxxxxxxxxxxxxx.

Процес ротації ключів можна формалізувати так:

Нехай $K_1$ – старий ключ, який використовується для шифрування даних до поточного часу. $K_2$ – новий ключ, згенерований для подальшого використання. При ротації ключів усі дані, зашифровані $K_1$, повинні бути перезашифровані з використанням $K_2$ або $K_1$ має бути доступним для розшифрування старих даних.

## 2.3 Запобігання ін’єкціям

Ін’єкції є одним з найбільш поширених видів атак на вебзастосунки. Вони відбуваються, коли зловмисник надсилає шкідливі дані до системи, щоб

змусити її виконувати несанкціоновані команди. Основні типи ін’єкцій включають ін’єкції SQL і XSS.

SQL-ін’єкції зазвичай поділяються на три категорії: In-band SQLi (класична), Inferential SQLi (сліпа) та Out-of-band SQLi.

In-band SQLi: зловмисник використовує один і той же канал зв’язку для здійснення атаки та збору результатів. Простота та ефективність роблять цей вид SQL-ін’єкцій одним із найпоширеніших.

Існує два підтипи цих атак:

– error-based SQLi - зловмисник викликає помилки бази даних, які можуть розкрити структуру бази. Ці помилки можуть бути використані для збору інформації про базу даних;

– union-based SQLi — цей метод використовує оператор UNION у SQL, який поєднує декілька запитів SELECT для отримання одного HTTP-відповіді. Відповідь може містити дані, які можна використовувати зловмисник.

Inferential SQLi: при таких атаках зловмисники вивчають відповіді та поведінку сервера після надсилання наборів даних, щоб дізнатися більше про структуру бази даних. При цьому жодні записи з бази даних вебсайту не передаються зловмиснику, і він не бачить їх у тому ж каналі зв’язку, як у випадку внутрішньосмугової атаки. Сліпі SQL-ін’єкції ґрунтуються на реакції та поведінці сервера, що робить їх повільнішими, але не менш небезпечними.

Такі атаки поділяють на два підвиди:

– boolean SQLi - зловмисник відправляє запит, який повертає результат, що залежить від істинності чи хибності запиту. Залежно від цього результату змінюється або залишається незмінною інформація у відповіді HTTP;

– time-based SQLi - зловмисники направляють SQL-запит до бази даних, змушуючи її зробити затримку кілька секунд перед відповіддю. Час відгуку дозволяє зрозуміти, чи правильний запит. HTTP-відповідь генерується одразу або після затримки, що допомагає зловмиснику визначити істинність запиту.

Out-of-band SQLi: використовується, коли неможливо використовувати той же канал для атаки та збору інформації, або якщо сервер занадто повільний

чи нестабільний. Цей метод залежить від можливості сервера відправляти DNS- або HTTP-запити для передачі даних зловмиснику. Досить рідкісний вид атаки.

Приклад SQL-ін’єкції:

Розглянемо ситуацію, коли у вебзастосунку є форма для введення ідентифікатора користувача (ID). Припустимо що стандартний SQL-запит для отримання даних про користувача з певним ID виглядає так:

$$SELECT * FROM\ users\ WHERE\ id = 1$$

Цей запит вибирає всі стовпці (*) з таблиці users для користувача з id = 1. Цей запит є безпечним, тому що він повертає дані тільки для користувача з конкретним ідентифікатором.

Зловмисник може спробувати маніпулювати введенням у формі, не просто вводячи значення 1, а 1 OR 1=1. Таким чином, згенерований SQL-запит виглядатиме так:

$$SELECT * FROM\ users\ WHERE\ id = 1\ OR\ 1 = 1$$

Тут важливо розуміти: вираз 1 = 1 завжди істинно. Таким чином, умова «id = 1 OR 1=1» означає, що запит виконається навіть якщо id не дорівнює 1, тому що друга частина умови (1=1) завжди повертає «істину». Це призведе до того, що запит стане еквівалентним наступному:

$$SELECT * FROM\ users$$

Цей запит є класичним прикладом In-band SQL-ін’єкції.

Для захисту від SQL-ін’єкцій у Laravel використовується Eloquent ORM або Query Builder. Вони автоматично налаштовують запити, що запобігає

більшості SQL-ін’єкцій. Однак при використанні динамічних параметрів необхідно бути уважним, щоб уникнути виникнення вразливостей.

Приклад небезпечного використання показано на рисунку 2.6.

```
$categoryId = $request->get('categoryId');
$orderBy    = $request->get('orderBy');

Product::query()
  ->where('category_id', $categoryId)
  ->orderBy($orderBy)
  ->get();
  return go(f, seed, [])
}
```

Рисунок 2.6 – Небезпечне використання Query Builder

У цьому випадку зловмисник може передати в параметр orderBy SQL-запит, що призведе до ін’єкції.

Для виправлення вразливості використовують підготовлені запити (рис 2.7) та валідацію даних (рис 2.8).

```
$categoryId = (int) $request->get('categoryId'); // Приведение к числу

$products = Product::query()
    ->where('category_id', $categoryId)
    ->orderBy('name')  // Безопасное поле для сортировки
    ->get();
```

Рисунок 2.7 – Підготовлені запити

```
$request->validate([
    'categoryId' => 'required|integer',
    'orderBy' => 'in:name,price,created_at',
]);
```

Рис 2.8 – Валідація даних

При XSS-ін’єкціях шкідливий код впроваджується у браузер користувача. XSS-атаки можливі за допомогою VBScript, ActiveX, Flash і навіть CSS. Проте вони найпоширеніші у JavaScript, оскільки JavaScript є основою більшості сучасних вебзастосунків.

Уразливості XSS сприймаються як менш небезпечні порівняно, наприклад, з уразливістю SQL Injection. На перший погляд, наслідки можливості виконання JavaScript на вебсторінці можуть здаватися не дуже серйозними, оскільки більшість веббраузерів запускають JavaScript у строго контрольованому середовищі. Однак JavaScript все ж таки може становити небезпеку, і використовуватися для таких дій як:

– крадіжка cookie-файлів сесії;

– виконання дій від імені користувача;

– фішинг;

– завантаження та виконання шкідливого ПЗ.

Розглянемо на прикладі, як можна за допомогою XSS-ін’єкції вкрасти cookie користувача. Припустим, що є XSS скрипт, який показаний на рисунку 2.9.

```
<script>
  document.location = 'http://attacker.com/?cookie=' + document.cookie;
</script
```

Рисунок 2.9 – Приклад XSS скрипту

Зловмисник впроваджує цей скрипт у вразливе місце на веб-сторінці. Наприклад, він може вставити його у форму з коментарями. Якщо веб-сайт ніяк не захищений, коментар зберігається в базі даних без фільтрації.

Коли браузер користувача завантажує сторінку із впровадженим скриптом, код автоматично виконується у контексті цього сайту. Це означає,

що скрипт виконується від імені жертви та має доступ до її даних, доступних у браузері, включаючи файли cookie сесії.

Команда document.cookie повертає всі файли cookie, доступні для поточного домену, після чого cookie відправляються на сервер зловмисника.

Для захисту від XSS у Laravel використовується шаблонизатор Blade, який автоматично екранує виведення змінних, і перетворює небезпечні дужки виду "<>" на безпечні HTML символи виду «&lt», «&gt».

Також для захисту від крадіжки cookie файлів за допомогою XSS-ін’єкції використовується політика CSP і прапор HttpOnly, який забороняє доступ до cookie користувачів через JavaScript.

Варто зауважити, що використання CSP як «панацею від усього» є поганою практикою, тому що не всі браузери можуть підтримувати усі конструкції CSP, які використовуються у спільній політиці CSP. Наприклад, багато застарілих браузерів не підтримують конструкції CSP рівня 2 і 3, від яких залежать запобігання XSS-ін’єкцій.

## 2.4 Запобігання атакам CSRF

У разі успішної атаки CSRF зловмисник спонукає жертву виконати певні дії без її відома. Наприклад, це може бути зміна адреси електронної пошти, пароля облікового запису або переказ коштів. Залежно від типу дії, зловмисник може отримати повний доступ до облікового запису користувача.

Якщо скомпрометований користувач має привілейовані права, зловмисник може заволодіти керуванням усіма даними та функціями програми.

Умови для успішної атаки:

– користувач авторизован в цільовому застосунку (наприклад, в інтернет-банку);

– сесія користувача активна, і браузер відправляє відповідні cookie при кожному запиті;

– у застосунку відсутні захисні механізми проти CSRF.

Механізм реалізації атаки CSRF:

Крок 1. Зловмисник створює запит, який браузер жертви виконає під час відвідування шкідливого сайту. Цей запит може бути прихований у посиланні, зображенні, формі або навіть усередині скрипту (рис. 2.10).


Рисунок 2.10 – Приклад шкідливого запиту

Код на рисунку 2.10 буде автоматично надіслано на сервер вебзастосунку як запит GET, як тільки користувач завантажить сторінку з цим зображенням.

Крок 2. Зловмисник відправляє посилання на шкідливу сторінку (або вбудовує її в картинку, електронний лист, повідомлення). Користувач переходить за посиланням або відкриває сторінку.

Крок 3. Коли жертва відкриває сторінку, браузер надсилає автоматичний запит на цільовий сайт. Якщо сесія користувача активна, браузер автоматично прикріпить cookie автентифікації до запиту.

Крок 4. Сервер бачить валідні cookie та сприймає запит як законний. Якщо програма не перевіряє справжність запиту, сервер виконає дії, описані у запиті.

Для захисту від CSRF необхідно ввести унікальний токен, який передається разом із кожним запитом. Токен генерується сервером та перевіряється на валідність при кожному запиті. Нехай:

$T$ – унікальний токен для сесії користувача, згенерований сервером.

Запит користувача $R$ вважається коректним, якщо:

$$Valid(R) = \begin{cases} 1, \text{якщо токен } T \text{ збігається} \\ 0, \text{якщо токен } T \text{ відсутній або невірний} \end{cases}$$

У Laravel вбудована підтримка CSRF-захисту з використанням токенів. Кожен запит типу POST, PUT, PATCH та DELETE має містити CSRF-токен.

```
<form method="POST" action="/transfer">
    @csrf<!-- Вставка CSRF-токена -->
    <input type="text" name="to" placeholder="Получатель">
    <input type="number" name="amount" placeholder="Сумма">
    <button type="submit">Перевести</button>
</form
```

Рис 2.11 – Приклад використання CSRF-токену у формі

Якщо токен відсутній або невірний, запит буде відхилено з помилкою 419 (Authentication Timeout).

## 2.5 Наслідки недостатнього логування та моніторингу подій

Безпечне логування та моніторинг подій є ключовими компонентами безпеки вебзастосунків. Ці механізми дозволяють своєчасно виявляти атаки, аналізувати інциденти та забезпечувати дотримання стандартів безпеки. У разі відсутності ретельно спланованих механізмів логування, організація упускає можливості для аналізу безпеки, тим самим дозволяючи атакам мати досить часу для подальшого проникнення в екосистему застосунку.

Основні показники недостатнього логування системи:

– відсутність чи обмежене логування подій;

– неправильна структура логів;

– відсутність моніторингу у реальному часі;

– проблеми зі зберіганням та доступом до логів;

– недолік автоматизованого аналізу та оповіщень;

– відсутність механізмів аудиту;

– вразливість до заміни логів;

– невідповідність нормативним вимогам.

Розглянемо деякі наслідки недостатнього, або неправильного логування та моніторингу:

– зловмисник використовує вразливість ін’єкції SQL для отримання доступу до бази даних. Якщо система не веде логування невдалих SQL-запитів або спроб несанкціонованого доступу, адміністратори можуть не помітити проблему доти, доки дані не будуть скомпрометовані або вкрадені;

– відсутність моніторингу та логування в реальному часі може призвести до того, що підозріла активність (наприклад, повторні спроби входу з різних IP-адрес) не буде помічена вчасно. Це може ускладнити вживання оперативних заходів для запобігання атаці;

– якщо співробітники компанії, які мають доступ до чутливих даних, здійснюють несанкціоновані дії, такі як завантаження чи модифікація даних, але ці дії не логуються, компанія може не виявити внутрішню загрозу.

Для ефективного моніторингу та логування у вебзастосунках необхідно дотримуватися низки вимог, які допоможуть своєчасно виявляти та запобігати загрозам. Ось основні вимоги:

– логування всіх критичних подій. Успішні та неуспішні спроби авторизації, зміни в правах доступу та налаштуваннях безпеки, SQL-запити, спроби виконання підозрілих команд;

– збір метаданих. Логи повинні містити: час події, IP-адресу та геолокацію клієнта, унікальні ідентифікатори користувача чи сесії;

– резервне копіювання логів. Регулярне створення резервних копій для запобігання втраті даних;

– шифрування. Логи, що містять чутливі дані, мають бути зашифровані;

– налаштування оповіщень. Автоматичні повідомлення про підозрілі події;

– єдиний формат. Логи слід структурувати (наприклад, JSON, CSV) для автоматизованого аналізу;

– розподіл прав доступу. Доступ до логів має бути обмежений певним колом осіб;

– зберігання протягом встановлений термін. Логи повинні зберігатися відповідно до нормативних вимог та політик компанії;

– захист від змін. Логи мають бути захищені від несанкціонованого редагування;

– повідомлення про зміни. Система повинна повідомляти адміністраторів про спроби зміни логів.

Laravel підтримує потужну систему логування на базі пакета Monolog. Усі параметри логування задаються у файлі «config/logging.php». Laravel підтримує кілька каналів для запису логів, таких як stack, single, daily, syslog, та slack (рис. 2.12).

```php
'channels' => [
    'stack' => [
        'driver' => 'stack',
        'channels' => ['daily', 'slack'],
        'ignore_exceptions' => false,
    ],
    'daily' => [
        'driver' => 'daily',
        'path' => storage_path('logs/laravel.log'),
        'level' => 'debug',
        'days' => 14,
    ],
],
```

Рисунок 2.12 – Приклад налаштування каналу

Laravel використовує PSR-3 рівні для логів. Вони допомагають визначати важливість записаних повідомлень:

– emergency – критична помилка, що робить систему недоступною;

– alert – негайне втручання (наприклад компрометація даних);

– critical - збій системи (наприклад, помилка бази даних);

– error - помилка, що заважає виконанню операції;

– warning – потенційна проблема;

– notice – важлива подія, яка не є помилкою;

– info – інформація про нормальну роботу програми;

– debug – докладні повідомлення для налагодження.

Приклад використання показаний на рисунку 2.13

```
Log::warning('Неудачная попытка входа', ['user_id' => $user->id]);
Log::error('Ошибка базы данных', ['query' => $query]);
```

Рисунок 2.13 – Використання рівнів логування

## 2.6 Аналіз існуючих продуктів для пошуку вразливостей

Для забезпечення безпеки вебзастосунків недостатньо реалізувати лише внутрішні механізми захисту. Важливо також регулярно проводити перевірку застосунків на вразливість. Існує безліч інструментів, які автоматизують цей процес, допомагаючи виявити та усунути загрози.

### 2.6.1 OWASP ZAP

OWASP ZAP – це популярний та безкоштовний інструмент для тестування безпеки вебзастосунків, розроблений OWASP. ZAP виконує

динамічне тестування, імітуючи дії потенційного зловмисника, щоб виявити вразливість. Його особливості роблять його універсальним рішенням як для автоматизованого сканування, так й для ручного тестування (рис. 2.13).

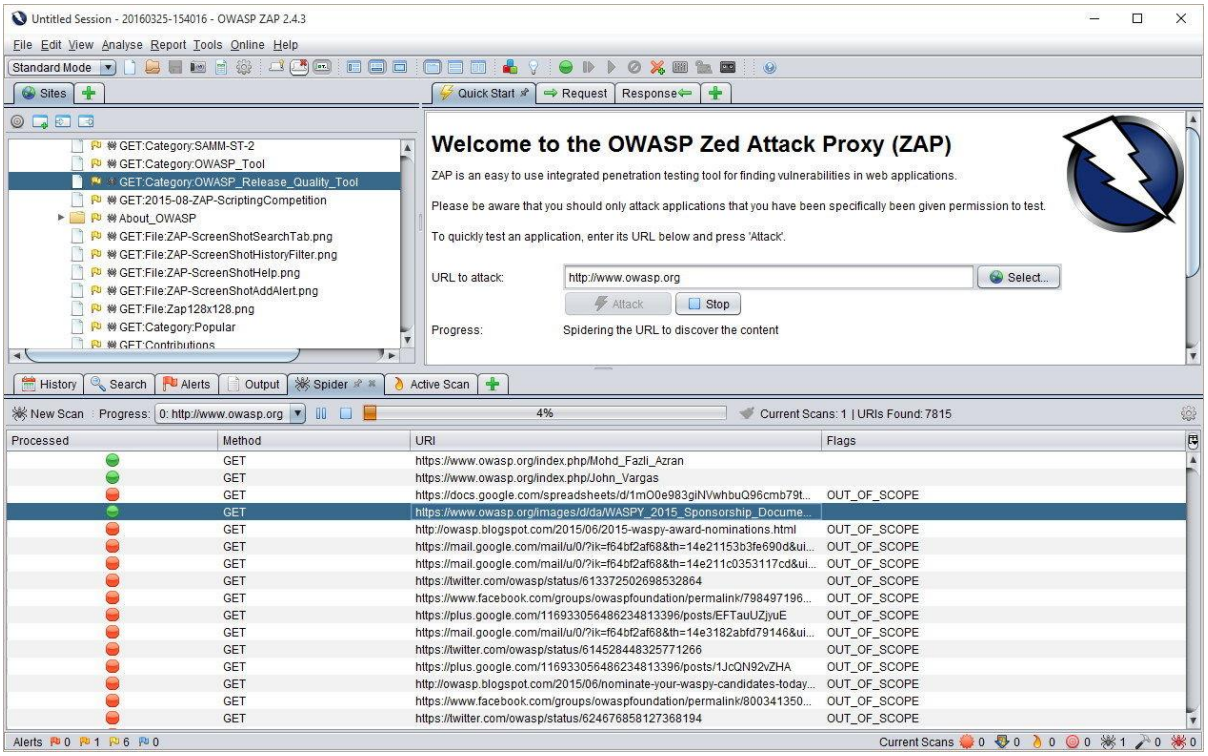



Рисунок 2.13 –Інтерфейс користувача ZAP

Основні можливості OWASP ZAP:

– ZAP може автоматично просканувати вебзастосунок на вразливості, такі як ін’єкції SQL, XSS, неправильні конфігурації безпеки та інші. Цей режим особливо корисний для швидкої перевірки відомих уразливостей;

– у режимі проксі програма може перехоплювати та змінювати запити HTTP між клієнтом і сервером. Це дозволяє більш детально аналізувати та тестувати безпеку на рівні HTTP-трафіку, виявляючи помилки у передачі даних;

– інструмент підтримує ручний режим тестування, дозволяючи тестувальникам вручну перевіряти складні вразливості, які потребують індивідуального підходу. Функція фаззингу також допомагає тестувати програму на стійкість до незвичайних вхідних даних;

– OWASP ZAP підтримує інтеграцію із системами CI/CD, такими як Jenkins, GitLab CI/CD та інші. Це дозволяє запускати сканування автоматично у процесі розробки, допомагаючи знаходити та виправляти вразливості ще на ранніх етапах;

– OWASP ZAP надає REST API, що дозволяє автоматизувати тестування та інтегрувати інструмент у власні системи моніторингу та захисту;

– через систему плагінів можна додавати функції ZAP, наприклад, для розширеного аналізу конкретних типів уразливостей.

### 2.6.2 Burp Suite

Burp Suite – це один з найпотужніших і найпопулярніших інструментів для тестування безпеки вебзастосунків. Розроблений компанією PortSwigger, Burp Suite використовується професійними пентестерами та дослідниками для виявлення вразливостей на різних рівнях застосунку. Цей інструмент є особливо ефективним завдяки широкому набору інструментів і плагінів, які дозволяють проводити як автоматичне, так і ручне тестування безпеки.

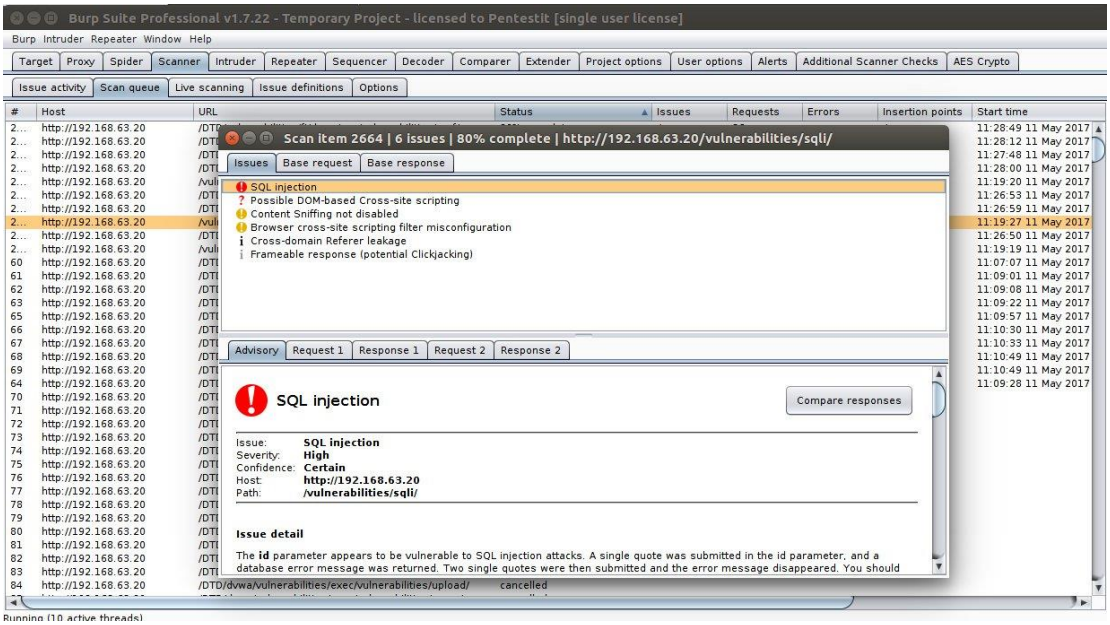



Рисунок 2.14 –Інтерфейс користувача Burp Suite

Основні можливості Burp Suite:

– інтерцептор та проксі-сервер. Burp Suite може перехоплювати запити HTTP і відповіді, дозволяючи користувачеві змінювати їх перед відправкою на сервер або в браузер. Це допомагає тестувальнику перевірити застосунок на стійкість до модифікацій запитів, виявити недоліки обробки даних;

– автоматичний сканер уразливостей. Програма включає потужний сканер, який автоматично виявляє такі вразливості, як SQL-ін’єкції, XSS, CSRF та інші;

– модуль для фаззингу та ручного тестування. Intruder - це один з найкорисніших модулів Burp Suite, який дозволяє проводити фазинг на різних полях введення. Користувач може надіслати на сервер безліч різних комбінацій даних для виявлення прихованих уразливостей;

– repeater та sequencer. Repeater дозволяє надсилати однотипні запити та аналізувати, як програма реагує на різні параметри, що корисно для тестування автентифікації та авторизації. Sequencer аналізує криптографічну стійкість сесійних ідентифікаторів та інших випадкових значень, що застосовуються для захисту даних;

– модуль для роботи з API та вебсервісами. Підтримує автоматичне та ручне тестування безпеки REST API та SOAP. Це дозволяє тестувати не тільки інтерфейси користувача, але й серверну частину програми;

– розширюваність та плагін-система. Програма надає вбудовані плагіни та підтримує Burp Extender API, що дозволяє додавати кастомні функції та інтегрувати сторонні інструменти. У доступній бібліотеці BApp Store можна знайти безліч розширень для розширення функціоналу, включаючи плагіни для специфічних уразливостей PHP-фреймворків.

### 2.6.3 Порівняльна таблиця застосунків

Таблиця 2.1 – Порівняльна таблиця для OWASP ZAP та Burp Suite за ключовими параметрами

| Параметр | OWASP ZAP | Burp Suite |
|---|---|---|
| Призначення | Динамічне тестування безпеки веб-застосунків, фокус на автоматизованому скануванні | Комплексне тестування безпеки, включаючи автоматизоване та ручне тестування |
| Тип ліцензії | Безкоштовна, з відкритим кодом | Платна та безкоштовна версія з обмеженнями |
| Сканування вразливостей | Автоматичне сканування для основних уразливостей | Більше просунутий сканер, який покриває широкий спектр вразливостей |
| Підтримка проксі | Підтримує перехоплення запитів та їх зміну | Потужний проксі з можливістю перехоплення та зміни запитів |
| Фазинг | Обмежені можливості | Intruder – потужний інструмент для фазингу |
| Модуль для ручного тестування | Є, але з обмеженими можливостями | Повний набір інструментів для ручного тестування (Repeater, Decoder, Comparer та ін.) |
| Аналіз випадковості | Немає | Sequencer для перевірки криптографічної стійкості токенів |
| Розширюваність | Доступні плагіни, можна додавати кастомні функції. | Підтримка Burp Extender API, доступ до BApp Store для додаткових модулів |
| Інтеграція із CI/CD | Можлива інтеграція із CI/CD системами | Інтеграція з CI/CD, підтримка REST API для автоматизації |
| Документація та підтримка | Активна спільнота OWASP, хороша документація | Обширна документація, спільнота, підтримка PortSwigger |
| Переваги | Безкоштовність, простота у використанні | Широкий функціонал для глибокого аналізу, просунута кастомізація |
| Обмеження | Менший функціонал для просунутого тестування. | Висока вартість професійної версії, складна крива навчання |

# 3 РЕАЛІЗАЦІЯ МЕТОДІВ ЗАХИСТУ ВЕБЗАСТОСУНКІВ

## 3.1 Підготовка вебзастосунку до впровадження захисних механізмів

На етапі підготовки вебзастосунка до впровадження заходів безпеки важливо створити надійну інфраструктуру та спроектувати структуру бази даних, інтерфейси та ключові функціональні елементи системи. Ця підготовка дозволить провести повноцінне тестування захисних механізмів, оптимізувати процеси розробки та впровадження безпеки та виявити можливі вразливості на початкових етапах.

### 3.1.1 Структура бази даних

Для зберігання даних вебзастосунку спроектована база даних, що включає таблиці, необхідні для роботи соціальної мережі. База складається з п’яти сутностей «users», «posts», «comments», «friendships» та «likes». Для створення бази даних були використані міграції, які надає Laravel. В якості СУБД використана MySQL (рис. 3.1).

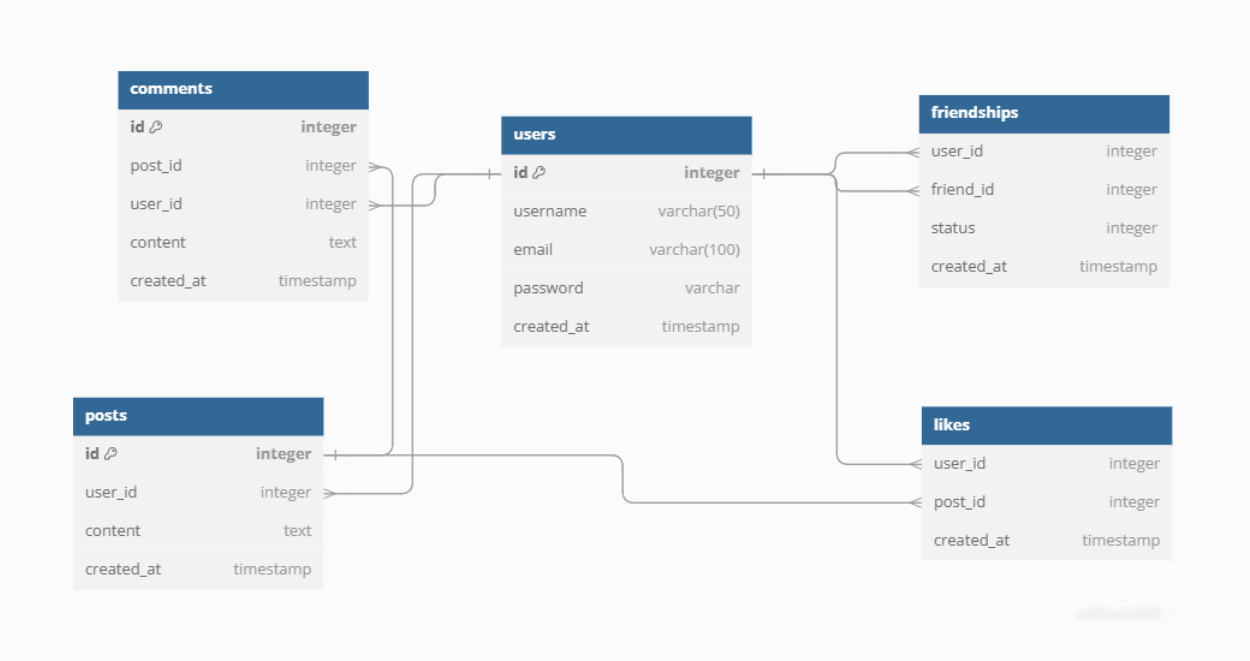



Рисунок 3.1 – Фізична модель бази даних

Таблиця «users» зберігає інформацію про користувачів застосунку. Структура:

– id – унікальний ідентифікатор користувача;

– username – ім’я користувача, унікальне поле;

– email – адреса електронної пошти, також унікальна;

– password – пароль, що зберігається у зашифрованому вигляді.

Створення таблиці за допомогою міграції показано на рисунку 3.2.

```php
Schema::create('users', function (Blueprint $table) {
    $table->id();
    $table->string('username', 50)->unique();
    $table->string('email', 100)->unique();
    $table->string('password');
    $table->timestamps();
});
```

Рисунок 3.2 – Створення таблиці «users» за допомогою міграції

Таблиця «posts» містить публікації користувачів. Структура:

– id – унікальний ідентифікатор;

– user_id – зовнішній ключ, який зв’язує публікацію з конкретним користувачем;

– content – вміст публікації;

– created_at – часова мітка, що фіксує час створення публікації.

Створення таблиці за допомогою міграції показано на рисунку 3.3.

```php
Schema::create('posts', function (Blueprint $table) {
    $table->id();
    $table->foreignId('user_id')->constrained('users')->onDelete('cascade');
    $table->text('content');
    $table->timestamps();
});
```

Рисунок 3.3 – Створення таблиці «posts» за допомогою міграції

Таблиця «comments» призначена для зберігання коментарів до публікацій. Структура:

– id – унікальний ідентифікатор;

– post_id – зовнішній ключ, що вказує, до якої публікації належить коментар;

– user_id – зовнішній ключ для відстеження автора коментаря;

– content – вміст коментаря;

– created_at – часова мітка створення коментаря.

Створення таблиці за допомогою міграції показано на рисунку 3.4.

```php
Schema::create('comments', function (Blueprint $table) {
    $table->id();
    $table->foreignId('post_id')->constrained('posts')->onDelete('cascade');
    $table->foreignId('user_id')->constrained('users')->onDelete('cascade');
    $table->text('content');
    $table->timestamps();
});
```

Рисунок 3.4 – Створення таблиці «comments» за допомогою міграції

Таблиця «friendships» моделює дружні зв’язки між користувачами. Структура:

– user_id – зовнішній ключ для відстеження користувача, який надіслав запит;

– friend_id – зовнішній ключ для відстеження користувача, який отримав запит;

– status – поле з можливими значеннями pending, accepted, rejected, яке визначає статус дружнього запиту;

– created_at – часова мітка створення запиту.

Створення таблиці за допомогою міграції показано на рисунку 3.5.

```php
Schema::create('friendships', function (Blueprint $table) {
    $table->foreignId('user_id')->constrained('users')->onDelete('cascade');
    $table->foreignId('friend_id')->constrained('users')->onDelete('cascade');
    $table->enum('status', ['pending', 'accepted', 'rejected'])->default('pending');
    $table->timestamps();

    $table->unique(['user_id', 'friend_id']);
});
```

Рисунок 3.5 – Створення таблиці «friendships» за допомогою міграції

Таблиця «likes» фіксує лайки, котрі користувачі ставлять на публікації. Структура:

– user_id – зовнішній ключ для відстеження користувача, який поставив лайк;

– post_id – зовнішній ключ для відстеження публікації, на яку поставлено лайк;

– created_at – час, коли лайк було поставлено.

Створення таблиці за допомогою міграції показано на рисунку 3.6.

```php
Schema::create('likes', function (Blueprint $table) {
    $table->foreignId('user_id')->constrained('users')->onDelete('cascade');
    $table->foreignId('post_id')->constrained('posts')->onDelete('cascade');
    $table->timestamps();

    $table->unique(['user_id', 'post_id']);
});
```

Рисунок 3.6 – Створення таблиці «likes» за допомогою міграції

## 3.2 Реалізація методів захисту

### 3.2.1 Захист від ін’єкцій

Почнемо із захисту від SQL-ін’єкцій. Атаки типу SQL-ін’єкцій досить поширені в сучасних вебзастосунках. Вони припускають, що зловмисники надають шкідливі дані у запитах з метою втручання у виконання SQL-запитів.

Першим та головним кроком до запобігання SQL-ін’єкцій є використання Eloquent ORM. Eloquent пропонує зручний інтерфейс для взаємодії з базою даних, використовуючи об’єктно-орієнтований підхід замість традиційного написання SQL-запитів. Eloquent використовує підготовлені вирази взаємодії з базою даних. Підготовлені вирази поділяють SQL-код та дані, що запобігає виконання шкідливого коду. Замість прямої вставки даних на запит, Eloquent передає дані як параметри.

У наступних пунктах потрібно буде реалізувати реєстрацію користувача, а для цього потрібно додати його до бази даних. Використання для цього Eloquent ORM показано на рисунку 3.7

```php
User::create([
    "username" => $data["login"],
    "email" => $data["email"],
    "password" => Hash::make($data["password"])
]);
```

Рисунок 3.7 – Створення запису до бази даних за допомогою Eloquent

На малюнку 3.7 можна побачити створення нового запису таблицю з ім'ям «Users». Цей код буде перетворено на SQL-запит з використанням плейсхолдерів, виду:

$$INSERT\ INTO\ users\ (username, email, password)$$
$$VALUES\ (?, ?, ?)$$

Laravel передасть реальні значення «login», «email» та «password» як параметри, окремо від самого SQL-запиту. Ці параметри будуть оброблені базою даних як дані, а не як частина SQL-команди, що запобігатиме впровадженню шкідливого коду.

Окрім Eloquent ORM, Laravel також надає можливість використати сирі вирази та запити для створення складних запитів. Хоча це забезпечує більшу гнучкість, необхідно бути обережним та завжди використовувати прив'язки даних для таких запитів.

Розглянемо такий запит: потрібно здійснити пошук користувача в таблиці users, у яких значення в полі email точно збігається з тим, що користувач передав у запиті (рис 3.8).

```php
User::whereRaw('email = "'.$request->input('email').'"')->get();
```

Рисунок 3.8 – Пошук користувача по email

Запит, показаний на рисунку 3.8 не є захищеним від SQL-ін’єкцій, оскільки введення даних не екранується і можуть бути інтерпретовані як частина SQL-запиту. Для того, щоб вирішити цю проблему, використовується прив’язка параметрів (?), що запобігає виконання шкідливого коду (рис 3.9).


Рисунок 3.9 – Виправлений запит до бази даних

Тепер перейдемо до ін’єкції типу XSS. XSS — це тип вразливості, коли зловмисник може вставити шкідливий код у сторінки, які відображаються іншим користувачам.

Шаблонізатор Blade в Laravel використовує оператори виведення {{}} для автоматичного екранування змінних за допомогою функції htmlspecialchars PHP, щоб захистити від атак XSS.

Виведення аватарки користувача з використанням екранування показано на рисунку 3.10.


Рисунок 3.10 – Екранування запиту

Laravel також надає можливість відображати неекрановані дані за допомогою синтаксису {!! !!}, але це потрібно робити дуже обережно, і тільки з довіреними даними.

Також одним з основних видів захисту від XSS є CSP. У Laravel він реалізується за допомогою middleware, яке додає необхідні заголовки в кожну відповідь (рис 3.11).

```php
public function handle(Request $request, Closure $next)
{
    $response = $next($request);
    $response->headers->set('Content-Security-Policy',
        "default-src 'self';" .
        "script-src 'self';" .
        "style-src 'self';" .
        "object-src 'none';" .
        "form-action 'self';" .
        "frame-ancestors 'none';"
    );
    return $response;
}
```

Рисунок 3.11 – Політика CSP

Як можна побачити на рисунку 3.11, цей CSP встановлює суворі обмеження завантаження та виконання ресурсів. Оскільки на сайті не використовуються ресурси з інших доменів, було встановлено обмеження на завантаження скриптів та стилів лише з поточного домену.

Також CSP забороняє завантаження об’єктів, таких як плагіни або вбудовані елементи, а також обмежує можливість відправлення даних з форми тільки на той же домен. Крім того, політика блокує вставку сторінки в інші сайти, запобігаючи атакам, таким як clickjacking.

Важливо пам’ятати, що використовувати CSP як єдину міру захисту є дуже поганою та практикою. OWASP рекомендує використовувати CSP тільки разом з іншими заходами безпеки у вигляді стратегії багатошарового захисту.

### 3.2.2 Реалізація аутентифікації та авторизації

У Laravel механізм автентифікації будується на використанні двох ключових компонентів: охоронців та провайдерів.

Охоронці (guards) відповідають за спосіб автентифікації користувачів при кожному запиті. Laravel за стандартом надає охоронець session, який підтримує стан користувача, використовуючи сесії та файли cookies для збереження інформації про його вхід до системи.

Провайдери (providers) визначають, як витягуються дані про користувачів з бази даних. Laravel підтримує два основних способи роботи з даними: через ORM Eloquent або за допомогою постачальника запитів query builder.

Почнемо з реалізації реєстрації користувача у системі. Створимо метод register, який оброблятиме процес реєстрації нового користувача. Використовуючи вбудований метод validate(), він валідує дані, отримані з запиту, включаючи логін, email і пароль. Якщо валідація проходить успішно, створюється новий користувач із зазначеними даними, де пароль хешується за допомогою методу Hash::make() для забезпечення безпеки. Після успішного створення користувача відбувається його автоматична автентифікація за допомогою охоронця web, що дозволяє користувачеві відразу увійти до системи (рис 3.12).

```php
public function register(Request $request) {
    $data = $request->validate([
        "login" => ["required", "string", "unique:users,username"],
        "email" => ["required", "email", "unique:users,email"],
        "password" => ["required", "string", "confirmed"],
    ]);

    $user = User::create([
        "username" => $data["login"],
        "email" => $data["email"],
        "password" => Hash::make($data["password"])
    ]);

    if($user) {
        auth("web")->login($user);
    }

    return redirect(route('home'));
}
```

Рисунок 3.12 – Метод реєстрації користувача

Це рішення є базовою реалізацією методу реєстрації та подальшої автентифікації, але не є оптимальним у міру захисту. Наприклад, хоч і проводитися валідація та хешування пароля, користувач все ще може створити пароль виду, «12345678» або «qwerty». Для запобігання цьому потрібно створити політику паролів. Щоб визначити ефективний пароль потрібно розрахувати його ентропію.

Для розрахунку ентропії паролів буде використана така формула:

$$e = \log_2 N^L \tag{3.1}$$

де $N$ – кількість можливих символів;

$L$ – кількість символів у паролі.

Розрахуємо ентропію для пароля, що складається з 8 символів, який може містити:

– малі літери (26 символів);

– великі літери (26 символів);

– цифри (10 символів);

– спеціальні символи ASCII (32 символи).

Таким чином, загальна кількість можливих символів у наборі N буде 94.

$$e = \log_2 94^8 \approx 52.44 \text{ біта} \quad (3.2)$$

Чим більше виходить значення ентропії, тим складніше буде зловмиснику підібрати пароль. Національний інститут стандартів та технологій (США) рекомендує використовувати пароль із 80-бітною ентропією для найкращого захисту. Але використання пароля з 80-бітною ентропією змушує використовувати пароль щонайменше з 12 символами. Складні паролі можна легко забути, та їх з більшою ймовірністю записуватимуть на папері, що передбачає деякий ризик. У нашому випадку компромісом стане використання мінімум 8-значного пароля.

Для створення політики паролів використаємо кастомне правило валідації, яке надає Laravel. Щоб його створити скористаємося командою «*php artisan make:rule PasswordStrength*» (рис 3.12).

```php
class PasswordStrength implements Rule
{
    public function passes($attribute, $value)
    {

        return preg_match('/[A-Z]/', $value) && // At least one capital letter
            preg_match('/[a-z]/', $value) && // At least one lowercase letter
            preg_match('/\d/', $value) && // At least one digit
            preg_match('/[\W_]/', $value); // At least one special character
    }

    public function message()
    {
        return 'The password must contain uppercase and lowercase letters, numbers and special symbols';
    }
}
```

Рисунок 3.12 – Політика валідації паролів

Тепер, у контролері, де здійснюється реєстрація, вказуються додаткові методи валідації (рис 3.14).

```php
$request->validate([
    "password" => ["required", "string", "confirmed", "min:8", new PasswordStrength()],
]);
```

Рисунок 3.14 – Оновлена політика валідації пароля

Щоб ще більше убезпечити користувачів від злому методом brute force, використаємо middleware «throttle», який обмежує кількість запитів, які користувач може надіслати на сервер протягом певного часу. Midleware відстежує кількість запитів від кожного користувача й якщо користувач перевищує встановлений ліміт запитів, Laravel відхиляє подальші запити та видає помилку «429 Too Many Requests». За стандартом throttle використовує IP-адресу користувача для відстеження кількості запитів (рис 3.15).

```php
Route::middleware('throttle:10,1')->group(function () {
    Route::post('/login_process',[AuthController::class,'login'])->name('login_process');
    Route::post('/register_process',[AuthController::class,'register'])->name('register_process');
});
```

Рисунок 3.15 – Використання middleware throttle

Як можна побачити, на рисунку 3.14 були введені обмеження, які не дозволяють користувачеві з однієї IP-адреси спробувати увійти або зареєструватися на сайті більше 10 разів за одну хвилину. Якщо кількість запитів перевищить це число, користувач отримає 429 помилку.

Останнє, що потрібно зробити – це налаштувати правильне управління, зберігання та знищення сесій. Неправильна робота з сесіями може призвести до вразливостей, таких як викрадення та фіксування сесій.

Насамперед потрібно налаштувати, де зберігатимуться сесії. Laravel включає безліч драйверів, які вказують, де зберігати сесії – це може бути реляційна база даних, cookies, кеш і т.д. У нашому випадку для зберігання сесій будуть використовуватися cookies.

Оскільки за стандартом в Laravel сесії зберігатися у вигляді звичайних файлів, які підходять тільки для локальної розробки, потрібно змінити файл config/session.php, а саме змінити параметр «driver» з «file» на «cookie». Далі потрібно убезпечити самі cookie:

– для того, щоб обмежити час життя cookie до 7 днів, потрібно встановити ‘lifetime’ => 10080;

– для того, щоб зашифрувати cookie перед тим, як вони будуть відправлені користувачеві, потрібно встановити ‘encrypt’ => true;

– щоб зробити недоступними cookie через JavaScript потрібно встановити ‘http_only’ => true;

– оскільки cookie не будуть передаватися через крос-сайтові запити, потрібно встановити ‘same_site’ => ‘strict’.

Повне налаштування файлу config/session.php можна побачити на рисунку 3.16


Рисунок 3.16 – Налаштування роботи сесії

Завершальною частиною роботи із сесіями є знищення сесії та генерація нового CSRF-токену при виході користувача із системи.

Знищення сесії дає захист від фіксації сесії, а також гарантує, що жодна інформація від попередньої сесії не втече або не буде повторно використана.

Генерація нового CSRF-токена запобігає повторному використанню старого токена для атаки. У більшості випадків Laravel автоматично регенерує CSRF-токени, проте виклик regenerateToken() вручну дає додатковий захист та 100% гарантію, що старий токен буде знищений.

Реалізацію функції виходу користувача та знищення сесії та токену показано на рисунку 3.17.

```php
public function logout(Request $request)
{
    auth("web")->logout();
    $request->session()->invalidate();
    $request->session()->regenerateToken();
    return redirect()->route('login');
}
```

Рисунок 3.17 – Функція виходу користувача з системи

### 3.2.3 Шифрування даних та управління ключами

Laravel надає вбудовані інструменти для шифрування через OpenSSL із використанням шифрування AES-256 та AES-128. Це дозволяє захистити конфіденційні дані, такі як персональна інформація користувачів, платіжні дані або секретні токени від несанкціонованого доступу.

Першим кроком для налаштування шифрування є створення унікального ключа шифрування. Для цього необхідно виконати команду «*php artisan key: generate*». Згенерований ключ зберігається у файлі «.env».

Варто враховувати, що зміна ключа шифрування призведе до завершення всіх поточних сеансів автентифікованих користувачів. Це відбувається тому, що Laravel шифрує всі cookie-файли, включаючи сеансові. Також розшифрувати дані, зашифровані з використанням попереднього ключа, буде неможливо. Щоб уникнути втрати доступу до даних, можна вказати старі ключі шифрування в змінній «APP_PREVIOUS_KEYS» у файлі «.env». У цьому випадку Laravel спробує використовувати поточний ключ

шифрування, і якщо розшифровка не вдасться, фреймворк спробує всі попередні ключі.

Laravel надає фасад Crypt для шифрування та розшифрування даних. Шифрування реалізується за допомогою методу «Crypt::encrypt()», а розшифрування – за допомогою «Crypt::decrypt()».

Реалізацію шифрування та розшифрування даних можна побачити на рисунку 3.18.

```php
public function createPosts(Request $request, $username)
{
    $request->validate([
        'content' => 'required|string|max:500',
    ]);
    $user = auth()->user();
    // Encrypt the content before saving it to the database
    $encryptedContent = Crypt::encrypt($request->content);

    $user->posts()->create([
        'content' => $encryptedContent,
    ]);
    return back();
}

public function showPosts()
{
    $user = auth()->user();

    $posts = $user->posts->map(function ($post) {
        // Decrypt the 'content' field of the post
        $post->content = Crypt::decrypt($post->content);
        return $post;
    });

    return view('profile', compact('posts'));
}
```

Рисунок 3.18 – Реалізація шифрування та дешифрування

Як можна побачити на рисунку 3.13 було реалізована шифрування та дешифрування особистих повідомлень користувачів. При додаванні нового запису до бази даних – повідомлення шифрується. При відображенні – повідомлення дешифрується. У цьому випадку навіть якщо станеться витік з бази даних, зловмисники не зможуть використовувати інформацію, яка була

написана в повідомленнях, щоб скомпрометувати користувачів, або вкрасти їх важливі дані.

Шифрування також має інші варіанти використання, наприклад:

– захист запитів API;

– захист інтелектуальної власності;

– запобігання SQL-ін’єкцій;

– захист електронних листів;

– захист файлів.

Важливим аспектом, який необхідно враховувати, це різниця між шифруванням і хешуванням. Хешування є одностороннім процесом перетворення даних на так зване «хеш-значення». Особливістю цього процесу є незворотність: після перетворення неможливо відновити вихідні дані з хешу. На відміну від шифрування, де дані можна закодувати й потім декодувати назад за наявності ключа, хешування не має на увазі зворотного перетворення, що робить його придатним для зберігання паролів та інших даних, де важлива перевірка відповідності, а не відновлення вихідної інформації.

### 3.2.4 Запобігання CSRF

CSRF-атака дозволяє зловмиснику змусити користувача виконати небажану дію на довіреному вебсайті, на якому той автентифікований.

Оскільки в Laravel захист від CSRF включений за замовчуванням, то всі POST, PUT, PATCH і DELETE-запити перевіряються на наявність дійсного CSRF-токена. У поточному проєкті використовується Blade шаблонизатор, так що для того, щоб додати CSRF-токен використовується директива @csrf, яка повинна додаватися до початку будь-якої форми.

На рисунку 3.19 наведений код форми авторизації у застосунок з додаванням CSRF-токену.

```
<form action="{{route("login_process")}}" method="post" class="form-container">
    @csrf <!-- Add CSRF-token -->
    <input name="login" type="text" placeholder="Login" class="input-field">
    <input name="password" type="password" placeholder="Password" class="input-field">
    <button type="submit" class="btn btn-primary">Login</button>
    <a href="{{ route("register") }}" class="btn-secondary">Sign Up</a>
</form>
```

Рисунок 3.19 – Форма авторизації з CSRF-токеном

Якщо подивитися на HTML-код форми, можна побачити приховане поле для токена. Приховане поле містить токен, який використовується посередником для перевірки запиту.

Щоразу при генерації сторінки Laravel створює одноразовий токен з обмеженням часу і додає їх у форму. CSRF-посередник перевіряє цей токен під час відправки форми та засвідчується, що токен був створений застосунком, перш ніж пропустити запит. Якщо час токену закінчився або не відповідає очікуваному значенню, Laravel відобразить помилку з кодом HTTP 419.

### 3.2.5 Логування та моніторинг подій

Оскільки жодна система не може забезпечити абсолютний захист від усіх можливих загроз, важливо постійно моніторити її стан, щоб оперативно реагувати на атаки чи спроби їх реалізації.

Для організації ефективного логування в Laravel використовується вбудована бібліотека Monolog, яка дозволяє записувати події системи та спрощує їх аналіз.

На початку налаштування потрібно вибрати драйвер каналу. У Laravel підтримується 6 драйверів для логування, які дозволяють настроювати різні способи зберігання та обробки логів:

– single – підходить для невеликих застосунків, де немає потреби у складній організації логів;

– daily – підходить для систем, де логи важливі протягом тривалого часу і повинні бути поділені по днях;

– syslog та errorlog зручні, якщо логи повинні бути інтегровані із системними інструментами моніторингу;

– stack зручний, якщо необхідно комбінувати кілька методів логування;

– slack – вибір для моніторингу в реальному часі.

Для проєкту був обраний драйвер daily, який дозволяє керувати логами по днях, через зручність у відстежуванні та налагодженні, а також тому, що він автоматично видаляє старі логи.

Також потрібно налаштувати рівень логування, запропонований у PSR-3. Рівні за зменшенням критичності виглядають наступним чином: emergency, alert, critical, error, warning, notice, info, та debug. У нашому випадку використовуватиметься рівень info. Це означає, що будь-які логи, які будуть записані через рівень debug, не будуть відображатися в логах.

Кінцева настройка конфігурації логування показана на рисунку 3.20.

```
'default' => env('LOG_CHANNEL', 'daily'),

'channels' => [
    'daily' => [
        'driver' => 'daily',
        'path' => storage_path('logs/laravel.log'),
        'level' => 'info',
        'days' => 14,
    ],
],
```

Рисунок 3.20 – Налаштування каналу

Наступним кроком буде відстеження подій, які мають бути записані до логів. Реалізуємо запис у логи, коли користувачі авторизуються на сайті.

Спочатку створимо слухача для події. Так як подія Login вже вбудована у фреймворк, то створювати нову подію не потрібно.

Створюємо слухача за допомогою команди *php artisan make:listener LogSuccessfulLogin*. Далі використовуючи фасад Log та рівень notice занесемо інформацію до журналу. Важливо пам’ятати, що в логи категорично не рекомендується заносити конфіденційні дані, паролі, кредитні картки тощо (рис 3.21).

```php
use Illuminate\Auth\Events\Login;
use Illuminate\Support\Facades\Log;

public function handle(Login $event)
{
    Log::info('User logged in', [
        'user_id' => $event->user->id,
        'time' => now(),
        'ip' => request()->ip(),
    ]);
}
```

Рисунок 3.21 – Логування авторизації користувачів

Залишилося зареєструвати слухача в EventServiceProvider. Після цього при кожній авторизації на сайт, в журнал буде потрапляти запис рівня notice, де буде інформація, хто авторизувався на сайті, з якого IP-адреси та час авторизації (рис 3.22).

```
[2024-12-01 17:03:45] local.INFO: User logged in {"user_id":81,"time":"2024-12-01 17:03:45","ip":"127.0.0.1"}
[2024-12-01 17:04:55] local.INFO: User logged in {"user_id":82,"time":"2024-12-01 17:04:55","ip":"127.0.0.1"}
[2024-12-01 17:05:25] local.INFO: User logged in {"user_id":83,"time":"2024-12-01 17:05:25","ip":"127.0.0.1"}
[2024-12-01 17:05:58] local.INFO: User logged in {"user_id":84,"time":"2024-12-01 17:05:58","ip":"127.0.0.1"}
```

Рисунок 3.22 – Журнал логування

## 3.3 Тестування та аналіз безпеки вебзастосунку

У цьому розділі описане тестування розробленого застосуноку за допомогою програми OWASP ZAP, на різні вразливості. Для виконання тестів був використаний автоматичний режим сканування, а також проводилось ручне налаштування та аналіз уразливостей. Застосунок був протестованний на захист від:

– SQL-ін’єкції;

– XSS;

– CSRF;

– уразливості конфігурації сервера;

– порушення контролю доступу;

– ін’єкції LDAP;

– помилки в обробці вхідних даних;

– вразливості у cookies;

– директорії та файли.

Звіт сканування програмою OWASP ZAP можна побачити на рисунку 3.23.

**Summary of Alerts**

| Risk Level | Number of Alerts |
|---|---|
| High | 0 |
| Medium | 0 |
| Low | 2 |
| Informational | 4 |

**Alerts**

| Name | Risk Level |
|---|---|
| Big Redirect Detected (Potential Sensitive Information Leak) | Low |
| X-Content-Type-Options Header Missing | Low |
| Authentication Request Identified | Informational |
| GET for POST | Informational |
| Session Management Response Identified | Informational |
| User Agent Fuzzer | Informational |

Рисунок 3.23 – Звіт сканування

На рисунку 3.23 можна побачити звіт, в якому зазначено, що після сканування було знайдено 2 потенційні загрози низького рівня, а також 4 інформаційні попередження. Розберемо докладніше:

– big redirect detected – це попередження вказує на наявність перенаправлення з великою кількістю параметрів URL, що може бути ознакою витоку чутливої інформації. Наприклад, якщо в URL-адресу передаються параметри з даними про сесію або особисту інформацію, це може призвести до витоку даних при перехопленні запиту. Не було виправлено;

– X-content-type-options header missing – заголовок X-content-type-options інформує браузер про те, що він повинен суворо дотримуватись типу контенту, зазначеного в заголовку content-type. Відсутність цього заголовка може призвести до уразливостей, пов’язаних з інтерпретацією даних браузером (наприклад, XSS-атаки), якщо контент може бути неправильно інтерпретований. Виправлено додаванням «X-Content-Type-Options: nosniff»;

– authentication request identified – це попередження вказує, що система виявила запити, пов’язані з автентифікацією (наприклад, спроби входу). Не є вразливістю;

– GET for POST – це попередження пов’язане з тим, що було зроблено HTTP-запит типу GET, який зазвичай використовується для отримання даних, але з параметрами, які мають бути передані через POST. Після аналізу застосунку та її маршрутів не було виявленого подібної поведінки. Можливе хибне спрацьовування;

– session management response identified – це попередження означає, що система ідентифікувала відгук, пов’язаний з керуванням сесіями (наприклад, кукі або токени). Не є вразливістю;

– user agent fuzzer – це попередження пов’язане зі спробами фуззингу значення заголовка user-agent. Не є вразливістю.

# ВИСНОВКИ

В рамках кваліфікаційної роботи було проведено комплексну розробку та дослідження методів підвищення безпеки вебзастосунків для PHP-фреймворків на прикладі створення простої соціальної мережі у фреймворку Laravel. Робота включала проєктування структури бази даних, реалізацію функціональності програми та послідовне впровадження методів захисту від поширених загроз, таких як SQL-ін’єкції, XSS, CSRF, уразливості в управлінні доступом, шифруванні даних, а також у логуванні та моніторингу подій.

Для досягнення поставлених цілей використовувалися сучасні інструменти та підходи, включаючи вбудовані механізми безпеки фреймворку Laravel, такі як Eloquent ORM для запобігання SQL-ін’єкцій, вбудовані фільтри для захисту від XSS, токени CSRF для запобігання міжсайтовим запитам, шифрування даних з використанням алгоритмів AES сесіями. Проведено тестування з використанням інструменту OWASP ZAP, який дозволив виявити потенційні вразливості, оцінити рівень їх критичності та підтвердити ефективність реалізованих заходів безпеки.

Наукова новизна роботи полягає в адаптації та застосуванні сучасних методів безпеки в контексті PHP-фреймворків, що дозволило розробити та запропонувати практичні рекомендації щодо побудови захищених вебзастосунків. Особлива увага приділялася демонстрації правильних способів реалізації захисту, що дозволило глибше зрозуміти природу вразливостей та їх вплив на безпеку застосування. Наведені методи та рекомендації можуть бути корисними розробникам для підвищення стійкості вебзастосунків до загроз.

Результати дослідження апробовано у вигляді тез доповідей під час VII Міжнародної студентської наукової конференції «Цифровізація науки та сучасні тренди її розвитку» [35]

# ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. McDonald, M. (2020). Web security for developers: real threats, practical defense. No Starch Press.

2. Stuttard, D. (2011). The Web Application Hacker’s Handbook: Finding and Exploiting Security Flaws. Wade AIcorn.

3. Stauffer, M. (2019). Laravel: Up & running: A framework for building modern php apps. " O’Reilly Media, Inc.".

4. Welling, L., & Thomson, L. (2003). PHP and MySQL Web development. Sams publishing.

5. Shiflett, C. (2005). Essential PHP security: A Guide to building secure web applications. " O’Reilly Media, Inc.".

6. Lockhart, J. (2015). Modern PHP: New features and good practices. " O’Reilly Media, Inc.".

7. Introduction - OWASP cheat sheet series. (б. д.). Introduction - OWASP Cheat Sheet Series. https://cheatsheetseries.owasp.org/index.html

8. Zandstra, M. (2010). PHP Objects, Patterns and Practice. Apress.

9. Hoffman, A. (2024). Web application security. " O’Reilly Media, Inc.".

10. Butler, T. (2022). PHP & MySQL: Novice to Ninja. SitePoint Pty Ltd.

11. Subedi, H. (2017). Mathematical Modelling of Delegation in Role Based Access Control.

12. Zalewski, M. (2011). The tangled Web: A guide to securing modern web applications. No Starch Press.

13. Zaninotto, F., & Potencier, F. (2007). The definitive guide to Symfony. Apress.

14. Pitt, C. (2012). Pro PHP 8 MVC.

15. Laravel - the PHP framework for web artisans. (б. д.). Laravel - The PHP Framework For Web Artisans. https://laravel.com/docs/11.x/readme

16. StackHawk application security blog. (б. д.). StackHawk. https://www.stackhawk.com/blog/

17. Li, X., & Xue, Y. (2011). A survey on web application security. Nashville, TN USA, 25(5), 1-14.

18. Porebski, B., Przystalski, K., & Nowak, L. (2011). Building PHP Applications with Symfony, CakePHP, and Zend Framework. John Wiley and Sons.

19. PHP: PHP manual - manual. (б. д.). PHP: Hypertext Preprocessor. https://www.php.net/manual/en/index.php

20. Hope, P., & Walther, B. (2008). Web security testing cookbook: systematic techniques to find problems fast. "O'Reilly Media, Inc.".

21. Seitz, J., & Arnold, T. (2021). Black Hat Python: Python Programming for Hackers and Pentesters. No Starch Press.

22. Bean, M. (2015). Laravel 5 essentials. Packt Publishing Ltd.

23. Gupta, C., Singh, R. K., & Mohapatra, A. K. (2020). Securing web applications using security patterns. In ICT for Competitive Strategies (pp. 485-494). CRC Press.

24. Krunalsinh, R. (2024). Laravel security: Hack-proof tips & tricks.

25. The web application security consortium / threat classification. (б. д.). http://projects.webappsec.org/w/page/13246978/Threat%20

26. Тітов, С. В., & Тітова, О. В. (2015). Оцінка юзабіліті освітніх сайтів: методи і технології.

27. Sahin, C. S., Lychev, R., & Wagner, N. (2015). General framework for evaluating password complexity and strength.

28. Sitnikov, D., Titova, O., Minukhin, S., Kovalenko, A., & Titov, S. (2018, October). Informativity of association rules from the viewpoint of information theory. In 2018 International Scientific-Practical Conference Problems of Infocommunications. Science and Technology (PIC S&T) (pp. 595-598). IEEE.

29. Ситніков, Д. Е., & Тітова, О. В. (2013). Аналіз веб-сайтів органів місцевої влади як механізму забезпечення права доступу до публічної інформації. Вісник Харківської державної академії культури, (41), 134-142.

30. Sen, J. (2011). A robust mechanism for defending distributed denial of service attacks on web servers.

31. Sunardi, A. (2019). MVC architecture: A comparative study between laravel framework and slim framework in freelancer project monitoring system web based. Procedia Computer Science, 157, 134-141.

32. Vanderlei, I., Araujo, J., Rocha, R., Silva, G., Pacheco, F., & Dantas, J. (2021, June). Analysis of Laravel Framework Security Techniques Against Web Application Attacks. In 2021 16th Iberian Conference on Information Systems and Technologies (CISTI) (pp. 1-7). IEEE.

33. Замула, О. А., & Черниш, В. І. (2011). Аналіз міжнародних стандартів в галузі оцінювання ризиків інформаційної безпеки. Системи обробки інформації, (2), 53-56.

34. Яремчук, К., Воскобойников, Д., & Мелкозьорова, О. (2022). Сучасні загрози та способи забезпечення безпеки веб-застосунків. Комп'ютерні науки та кібербезпека, (2), 28-34.

35. Ходонович, А. Б. (2024). Методи покращення безпеки веб-додатків, розроблених на PHP та Laravel. У *Цифровізація науки та сучасні тренди її розвитку* (с. 106–107).