МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

ВИПУСКНА РОБОТА

на тему:

«Захист персональних даних при розробці веб-додатку»

Завідувач випускаючої кафедри — **Довбиш А.С.**

Керівник роботи — **Проценко О. Б.**

Студента групи КБ – 71 — **Годунов В. О.**

СУМИ 2021

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Кафедра комп'ютерних наук

Затверджую_____

Зав. кафедрою Довбиш А.С.

"_____" _____2021 р.

ЗАВДАННЯ
до випускної роботи

Студента четвертого курсу, групи КБ-71, спеціальності "Кібербезпека" денної форми навчання Годунова Владислава Олександровича.

Тема: "**Захист персональних даних при розробці веб-додатку**"

Затверджена наказом по СумДУ

№_____ від _____2021 р.

Зміст пояснювальної записки: 1) аналіз предметної області 2) огляд існуючих рішень в сфері захисту персональних даних 3) розробка комплексу заходів для забезпечення захищеності веб-додатку 4) розробка веб-додатку з використанням розробленого комплексу заходів для забезпечення захисту персональних даних.

Дата видачі завдання "_____"_____2021 р.

Керівник випускної роботи _____ Проценко О.Б.

Завдання прийняв до виконання _____ Годунов В.О.

РЕФЕРАТ

Записка: 75 сторінок, 25 рисунків, 3 таблиці, 1 додаток, 24 джерел.

Об'єкт дослідження – Захист персональних даних при розробці веб-додатку.

Мета роботи – Розробка комплексу заходів та використання базових методів захисту для забезпечення цілісності, доступності та конфіденційності персональних даних користувачів веб-додатку, що розроблюється.

Перший розділ – продемонстровано базові інструменти захисту веб-додатку, огляд існуючих рішень щодо захисту від витоків даних, типів загроз та методів захисту від них

Другий розділ – визначено комплекс заходів які необхідно виконувати при розробці веб-додатку, описано важливість забезпечення захищеної автентифікації користувачів, перевірки введених даних в форму та перевірки параметрів які надсилаються до бази даних, продемонстровано приклади, чому важливо захищати дані при передачі

Третій розділ – розроблено веб-додаток, в якому були використано базові інструменти захисту персональних даних користувачів та методи захисту від загроз.

Результати – розроблено комплекс заходів та надання рекомендацій й методів захисту персональних даних користувачів, для цього було розроблено та протестовано веб-додаток, який відповідає визначеним критеріям захищеності.

ЗМІСТ

ВСТУП 5

1 ІНСТРУМЕНТИ ЗАХИСТУ ВЕБ-ДОДАТКУ 7

1.1.Базові інструменти захисту 7

1.2.Захист від витоку даних 8

1.3.Методи захисту від загроз 12

1.4.Постановка задачі 16

2 ЗАХИСТ ПЕРСОНАЛЬНИХ ДАНИХ 18

2.1.Забезпечення захищенної автентифікації користувачів 18

2.2.Перевірка введених даних в форму 22

2.3.Кодування виводу HTML 25

2.4.Прив’язування параметрів для запитів до бази даних 26

2.5.Захист даних при передачі 27

3 РОЗРОБКА ВЕБ-ДОДАТКУ 34

3.1.Архітектура веб-додатку 34

3.2.Програмна реалізація 35

3.3.Тестування захищенності веб-додатку 48

ВИСНОВКИ 52

СПИСОК ЛІТЕРАТУРИ 53

ДОДАТОК А. ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБ-ДОДАТКУ 56

ВСТУП

З розвитком інформаційних технологій і доступних засобів масових комунікацій, діяльність людини все більше і більше пов'язується з глобальною мережею Інтернет. Працюючи в мережі людина має доступ до великої кількості корисної інформації, але часто не помічає, як добровільно надає доступ до своїх персональних даних, внаслідок чого, важлива інформація знаходиться під загрозою.

Питання захисту персональних даних є актуальним, як ніколи. Особливо важливо захищати персональні дані які попадають в мережу інтернет, їх інформаційна безпека. Персональними даними можна назвати будь-яку інформацію, що так чи інакше стосується прямо або побічно деякої фізичної особи (суб'єкта персональних даних) [1]. Очевидно, що при широкому використані комп'ютерів та мережі для обробки та передачі інформації, ці галузі повинні бути дуже надійно захищені від можливостей доступу до даних третіми особами, втрати або спотворення інформації.

Сучасний розробник програмного забезпечення повинен не тільки писати код, який відповідає вимогам замовника, а й такий, що є зрозумілим та розширюваним. Це необхідно для того щоб з легкістю добавляти новий функціонал задля еволюціонування програмного забезпечення, але при цьому залишатися стабільним та надійним. Звичайно програма повинна бути швидкою, мати дружелюбний інтерфейс, гарну оптимізацію та легкий процес дистрибуції, але перш за все програмне забезпечення повинно бути захищеним. [2] Згідно зі статистичними даними, більше ніж 80% компаній несуть фінансові збитки за причини, порушення цілісності та конфіденційності персональних даних користувачів.

Отже, компанії все частіше отримують доступ до конфіденційних даних користувачів, але при цьому ігнорують етап перевірки своїх продуктів на захищеність від атак, чи раптових витоків даних. Саме тому розробляючи веб-

додатки необхідно враховувати можливість атак з боку хакерів. Якщо декілька років тому було достатньо лише очистити жорсткий диск від зайвих чи застарілих файлів та кодів, то в міру посилення активності з боку зловмисників необхідні більш серйозні заходи безпеки.

1. ІНСТРУМЕНТИ ЗАХИСТУ ВЕБ-ДОДАТКУ

1.1.Базові інструменти захисту

Найбільш захищеною інформаційною системою можна назвати ту, в якій зловмиснику не надано єдиного шансу на втручання в роботу її базових елементів, таких як: мережеві пристрої, операційні системи, додатки та Система управління базами даних.

Антивіруси

Захист від найпоширеніших вірусів є одним із ефективних засобів запобігання витоку конфіденційної інформації, в тому числі й персональних даних користувачів веб-додатку. Саме віруси, черв'яки та шкідливі програми досить часто займаються крадіжкою чи пошкодженням інформації та організовують приховані канали витоку даних. Сучасні антивіруси надають надійні рішення, які включають в себе, на додаток до стандартного сигнатурного захисту, поведінковий аналіз програм, контроль цілісності деяких даних, які критичні для операційної системи, екрани рівня додатків та інші методи захисту серверів та робочих місць.

Міжмережеві екрани

Для забезпечення більш впевненого захисту веб-додатку, корпоративна мережа та кожне робоче місце необхідно захищати не тільки від масових атак за допомогою вірусів, а й від націлених атак на мережу. Як метод захисту від такого роду атак достатньо налаштувати систему блокування мережевих протоколів і сервісів, які не використовуються. Для цього й використовується міжмережевий екран. На додаток можна додати й засоби організації віртуальних приватних мереж - VPN.

Деякі антивірусні рішення класу Internet Security мають вбудовані мережеві екрани, які здатні зафіксувати атаки з мережних протоколів, а також спроби мережевих черв'яків проникнути на захищену машину. Окрім того, для мережевих екранів необхідно активувати на шлюзовому маршрутизаторі (як

правило, такий функціонал має знаходитися на мережевих пристроях), це дозволяє створити так звану демілітаризовану зону (DMZ), для зовнішніх веб-додатків та систем електронної пошти. Слід підкреслити, що в DMZ, діють більш жорстокі правила для доступу, розміщуються мережеві ресурси, доступ до яких доступний ззовні та захисні механізми для них [6].

Системи запобігання вторгнень

Для виявлення в поточному трафіку ознак нападу та як наслідок блокування виявленої найбільш популярної атаки, використовуються системи запобігання вторгнень (Intrusion Prevention System, IPS), які встановлюються в розрив мережі. IPS аналізують вміст IP-пакетів, використовуваних протоколів та коректність їх використання. Внаслідок чого, спектр атаки від яких захищають системи запобігання вторгнень ширше, ніж у шлюзових антивірусів.

1.2.Захист від витоку даних

Засоби для захисту конфіденційних даних від витоків знаходиться зараз в певній стадії формування. Можна виділити три класи таких продуктів: засоби шифрування, системи захисту від витоків (DLP - Data Leak Prevention) та системи контролю над периферійними пристроями [23]. Для більш надійної безпеки необхідно поєднувати всі вище перераховані типи продуктів. На сьогоднішній день комплексних рішень на ринку немає.

Контроль над пристроями. Часто витік даних відбувається через знімні носії інформації та несанкціоновані канали зв'язку: флеш-пам'ять, USB-диски, Bluetooth, або Wi-Fi, саме через це, контроль за використанням портів та іншого периферійного обладнання є одним із ефективних способів контролю витоків.

DLP. За допомогою використання спеціальних алгоритмів, системи захисту від витоків дозволяють виділити з потоку даних конфіденційну інформацію та заблокувати її передачу. В таких системах передбачені механізми контролю різноманітних каналів передачі інформації: миттєві повідомлення, пошти, друк на принтері та інші [6].

Шифрування

Так чи інакше захист даних від витоків використовує механізми шифрування. Шифрування потрібно використовувати не тільки для самих баз персональних даних, а також передачу по мережі та резервні копії баз даних. Деякі бази даних мають вбудовані механізми шифрування, їх також можна використовувати. Для більш надійного захисту персональних даних можна шифрувати цілі розділи файлової системи на яких зберігаються дані.

Шифрування також використовується й при передачі персональних даних в мережі розподіленої системи. Для досягнення цього можна застосовувати різні продукти класу VPN, які й базуються на шифруванні [7].

RMS (Right Management System)

Системи даного класу базуються на алгоритмах шифрування та керують ключами дешифрування. Дані ключі зберігаються на центральному сервері системи та доступ до них дозволяється лише за умови проходження процедури суворої аутентифікації користувача. Виходячи з цього, можливість розшифрувати документ може тільки користувач який володіє відповідними правами. Microsoft (Microsoft RMS), Oracle (Oracle IRM) одні з компаній які пропонують дані рішення, тенденція поширена перш за все для зарубіжних компаній. Також можна підмітити те, що в існуючих продуктах правами доступу до ключів розпоряджаються автори документів, як наслідок, не дозволяє захиститися від внутрішніх загроз, як це відбувається в DLP-системах. Саме за цієї причини ці системи не отримали належного розповсюдження в якості засобів захисту від витоків інформації [6].

Керування правами доступу

Одною з поширених та головних проблем адміністратора є правильна організація доступу співробітників до ресурсів великої інформаційної системи. Від коректного налаштування прав доступу залежить збереження конфіденційних даних, саме тому система управління правами доступу має бути включена в систему захисту інформаційної системи. Зазвичай, це дозволяє

ввести рольове управління правами доступі та саме контролює дотримання цих прав, вона також блокує спроби змінити права доступу без дозволу адміністратора безпеки, що забезпечує захист від локальних адміністраторів. Найбільш популярними рішеннями цього сімейства можна назвати Oracle IAM, IBM Tivoli Access Manager. Слід зазначити, що методика захисту персональних даних передбачає управління правами доступу в системах, що обробляють таку інформацію та які мають будь-яких розмір, однак у невеликих базах достатньо буде ручного управління правами доступу [6].

Кореляція подій

Система захисту може генерувати безліч повідомлень про потенційні напади, які лише з деякою ймовірністю здатні призвести до реалізації певної загрози. Досить часто такі повідомлення бувають тільки попередженнями, проте адміністратори безпеки повинні бути здатні розібратися в тому що відбувається. Одним з ймовірних інструментів може стати система кореляції подій, яка дозволяє зв'язати певну кількість повідомлень від пристроїв захисту в єдиний ланцюг подій та комплексно оцінити наскільки ланцюжок небезпечний. Це дозволяє виявити найбільш небезпечні події та полегшити роботу адміністраторів безпеки. Одним з прикладів такого роду систем можна назвати CIsco MARS, або netForensics.

Управління безпекою

Системи централізованого управління захисними механізмами дозволяють контролювати всі події, які так чи інакше пов'язані з безпекою інформаційної системи. Інструменти цього рівня дозволяють виявити захисні механізми, встановлені на підприємстві, здійснювати управління над ними та отримувати від них звіти про події, що відбуваються. Ці ж інструменти можуть автоматизувати рішення простих проблем, або допомагати адміністраторам безпеки за короткий час розібратися в більш складних атаках.

На рисунку 1.1 наведено методи захисту конфіденційної інформації, з яких п'ятірка перших забезпечують поодинокі сценарії захисту конфіденційних даних

від витоку, а за допомогою 6 підходу можна отримувати фінансово значущий результат в досить короткий час після впровадження системи. Підхід під номером 7 вимагає досить кропіткого впровадження та налагодження, в середньому від одного до трьох років. Підходи 8-10 доповнюють згадані вище, DLP-системи, за рахунок зниження ризиків витоку внаслідок втрати чи крадіжки техніки. Підходи 11-12, так чи інакше, вже використовуються у всіх компаніях та можуть побічно вплинути на зниження ризиків витоку інформації [6].

N	Подход к защите конфиденциальной информации	Примеры реализации
1	Фильтрация исходящей информации по ключевым словам, регулярным выражениям для идентификации конфиденциальных данных	Системы фильтрации трафика (MIMEsweeper, «Дозор Джет», «Контур безопасности», McAfee Network Data Loss Prevention, eSafe)
2	Установка грифов конфиденциальности на защищаемые документы и слежение за жизненным циклом помеченного документа	Системы мандатного доступа к документам и протоколирования обращений (SecretNet, SecrecyKeeper)
3	Слежение за манипуляциями с конфиденциальными данными и протоколирование действий пользователя на рабочем месте	Системы контроля действий пользователя (StaffCop и др.)
4	Управление доступом к устройствам ввода/вывода	Системы контроля сменных носителей (SmartLine DeviceLock, SecurIT ZLock, Device Control)
5	Сканирование данных, хранимых на рабочих местах, с целью обнаружения конфиденциальных данных	Системы сканирования хранилищ и рабочих станций (Symantec DLP, Websense Data Discover, Контур безопасности, McAfee Network DLP Discover)
6	Исполнение всего комплекса вышеперечисленных подходов и сведение управления политиками и событиями к единой консоли	Системы Data Loss Prevention (Websense DSS, Symantec DLP, InfoWatch Traffic Monitor, McAfee Network DLP Manager)
7	Анализ протоколов разнообразных систем безопасности в унифицированном виде и выявление аномальных активностей со стороны сотрудников и внешних злоумышленников	Системы Computer Forensics (netForensics и др.)
8	Контроль доступа пользователей к компьютерам и информационным системам с дополнительными элементами контроля	Системы двухфакторной аутентификации (Aladdin eToken, ruToken), системы с использованием биометрии
9	Разветвленная система разграничения прав доступа к конфиденциальным документам	Системы класса Enterprise Rights Management (Microsoft RMS, Oracle IRM) и защищенного документооборота (Perimetrix)
10	Шифрование хранимой конфиденциальной информации	Системы шифрования хранилищ, дисков, накопителей (SecurIT, Aladdin, «Физтех-софт»)
11	Контроль посещения сотрудниками Web-ресурсов	Системы Web-фильтрации (Websense Web Security или на основе MS ISA Server, SQUID, eSafe)
12	Создание почтового архива – доказательной базы для расследования инцидентов постфактум	Системы архивирования электронной почты (средствами почтовой системы, InfoWatch, Symantec Enterprise Vault)

* Классификация предложена компанией Leta IT-Company.

Рисунок 1.1 – Методи захисту конфіденційної інформації

Термінальні рішення

Одну з серйозних проблем захисту персональних даних яку мають сучасні децентралізовані інформаційні системи, можна охарактеризувати наступним чином: виконання вимог захисту окремих серверів більш легше запровадити ніж

вимоги захисту для усіх персональних комп'ютерів. Тома одним за ймовірних варіантів зниження складності проекту по захисту персональних даних, є впровадження досить класичного термінального рішення, в якому дані знаходяться в межах сервера та не покидають їх, а на робочі місця співробітників встановлюються бездискові термінальні станції.

Одним з варіантів побудови подібної системи є рішення, яке було розроблене спільно компаніями "Свемед" та Sun Microsystems, що складається із захищеного сервера на якій встановлено операційну систему "Циркон-10", сервера термінального доступу "Циркон-Т", а також набору терміналів Sun Ray 2, Sun Ray 270, або Sun Ray 2FS, які були вироблені компанією Sun, за технічними вимогами "Свемел". Вихідні коди вбудованої системи захисту Trusted Extension в операційній системі "Циркон-10" сертифіковані ФСТЕК на відсутність програмних закладок і надійністю контролю доступу [6].

1.3.Методи захисту від загроз

Методами захисту інформації вид загроз можна назвати певні дії з боку розробників веб-додатків, які націлені на максимальному захисті інформації. Основні методи захисту інформації від загроз наведені в таблиці 1.

Програмні, апаратні та фізичні методи захисту найчастіше застосовуються для захисту від зовнішніх порушників, які не мають прямого доступу до конфіденційних даних.

Для захисту від внутрішніх порушників перш за все застосовуються організаційні, законодавчі та психологічні заходи, що залежать від типу співпраці між власником та особами, що мають доступ до конфіденційної інформації.

Для забезпечення максимального захисту розроблюваного веб-додатку потрібно постійно стежити за тим як з'являються нові загрози інформаційної безпеки та при необхідності максимально оперативно усувати порушення зазначених вимог безпеки веб-додатку.

Для більш надійного захисту в веб-додатку має бути реалізовано автоматичне резервне копіювання бази даних і програмних файлів.

Бекап баз даних повинен створюватися щодня, а бекап файлів, щонайменше, один раз на тиждень. Ці дії дозволяють досить швидко відновити роботу веб-додатку при збою в системі, або непередбачених втратах даних. Резервні копії мають зберігатися на резервному сервері.

Таблиця 1.1 – Основні методи захисту від загроз

Методи захисту	Опис
Програмні	Програми, за допомогою яких відображаються хакерські атаки, виконується відновлення втрачених даних, а також створювати резервні копії інформації.
Апаратні	Пристрої для обробки інформації
Фізичні	За допомогою фізичних засобів захисту запобігає доступ сторонніх осіб на територію, на якій зберігаються конфіденційні дані. Найбільш актуальним методом захисту є установка міцних дверей з надійними замками, а також решітки на вікнах.
Організаційні	Методи захисту, які мають на увазі регламентацію, управління і примус. До них відноситься розробка посадових інструкцій, різні бесіди зі співробітниками, заходи заохочення і покарання. Співробітники несуть відповідальність за зловживання посадовими повноваженнями, за витік або втрату даних.

Законодавчі	Нормативно-правові акти, які регулюють діяльність співробітників, що мають доступ до конфіденційних даних, і визначають міру відповідальності за втрату або крадіжку секретної інформації.
Психологічні	Заходи для створення особистої зацікавленості співробітників в збереженні цілісності і достовірності даних. Кожен співробітник повинен відчувати себе важливою частиною системи і повинен бути зацікавлений в успіхах компанії.

Загрози при передачі даних між користувачами та веб-додатком

При аутентифікації користувачів задля подальшої їх взаємодії з веб-додатком, передача даних здійснюється по протоколу HTTPS, за використанням SSL-сертифіката, що в свою чергу дозволяє шифрувати всі дані, що передаються між користувачами і сервером. Це призводить до того, що навіть за випадку крадіжки даних, що передаються, зловмисник отримає лише зашифровані дані, які не мають ніякої цінності.

Мінімізація загроз здійснення несанкціонованого доступу до розділів веб-додатку, що вимагає авторизацію користувачів, можлива лише за умови захисту від наступних атак:

- SQL-injection - захистом проти ін'єкцій є точна фільтрація і перевірка вхідних даних від користувачів;
- XSS - для захисту проти цієї атаки використовується кілька способів, до яких відносяться екранування даних, білі списки, вказівка кодування сторінки, установка прапора HttpOnly, а також впровадження Content Security Policy;

- Контроль доступу до функцій веб-додатку - реалізована система перевірки прав доступу користувачів до кожного функціонального компоненту;
- Підробка міжсайтових запитів (CSRF) - всі важливі дані передаються за допомогою POST запитів, а також перевіряється джерело даних;
- Прямі посилання на сторінки з конфіденційною інформацією - при запиті даних перевіряються права доступу користувача;

Крадіжка пароля адміністратора – при вході користувача з роллю «Адміністратор» відбувається двоетапна аутентифікація, з підтвердженням по СМС.

Також необхідно вести написання логів всіх дій не тільки користувачів, а також й адміністраторів веб-додатку та адміністраторів піддоменів. Логами називають інформацію про будь-яку скоєну дію будь-якого користувача.

Загрози, що призводять веб-додаток в непрацездатний стан

Існує два типи загроз, які можуть призвести до порушення доступності веб-додатку:

- Фізичний несанкціонований доступ до веб-сервера і виведення його з робочого стану
- Зовнішні атаки на веб-додаток, які так само можуть вивести веб-сервер з ладу.

Для мінімізації загрози фізичного доступу до веб-серверів, можна розташовувати їх в межах контрольованих зон, що виключають доступ сторонніх осіб.

Зовнішньою атакою можна вважати DDoS-атаку, під час якої користувачі не можуть отримати доступ до веб-додатку. Процес полягає в організації великої кількості помилкових запитів на веб-сервер жертви.

Метою такої атаки може бути отримання критичної інформації, такої як, деякі частини програмного коду, конкуренція, шахрайство, або навіть банальна розвага. У разі шахрайства, зловмисники, можуть забажати грошову винагороду для відновлення працездатності ресурсів. В деяких випадках для великих

компаній простіш згодитися на ці умови, щоб не зазнати великих фінансових збитків та відновити репутацію компанії.

Трирівнева архітектура передбачає собою:

- Керуючу консоль зловмисника, можливо, навіть декілька, за допомогою якої ініціюється атака;
- Проміжні комп'ютери - ті, які транслюють сигнали на «зомбовані» комп'ютери, а їх кількість залежить від масштабів DDoS-атаки;
- «Зомбовані» - це саме ті комп'ютери. що атакують веб-сервер, відправляючи запити на нього, ні про що не підозрюючи.

Виявити DDoS-атаку досить просто - це уповільнення або відмова роботи веб-додатку. При виявленні атаки, насамперед потрібно ідентифікувати тип трафіку, який перевантажує нашу мережу. Найчастіше DDoS завжди посилає велику кількість ICMP, UDP, або TCP пакетів, зазвичай з підробленими ip адресами. При виявленні таких пакетів необхідно додати їх в списки обмеження доступу.

1.4.Постановка задачі

Аналіз способів захисту персональних даних користувачів веб-додатків, показав як важливо створювати безпечне програмне забезпечення. Часто розробникам не вистачає знань, щодо існуючих вразливостей та методів захисту від них. Тому вони, навіть, не замислюються про безпеку користувачів та допускають елементарні помилки при розробці, наприклад, не екранують поля вводу, зберігають в базах даних інформацію в не зашифрованому вигляді, та що найімовірніше, не проводять тестування додатків на вразливості.

На основі цього було прийняте рішення, щодо розробки веб-додатку, в якому будуть використані сучасні методи захисту персональних даних. На прикладі якого, будуть продемонстровані способи захисту від можливих загроз втрати, пошкодження, або витоку даних.

Актуальним завданням є розроблення такого додатку, що забезпечить:

- Використання базових інструментів захисту
- Захист персональних даних користувачів
- Доступність додатку для користувачів
- Безпечність зберігання та оновлення програмного забезпечення
- Зручний та зрозумілий україномовний інтерфейс

2. ЗАХИСТ ПЕРСОНАЛЬНИХ ДАНИХ

2.1. Забезпечення захищеної автентифікації користувачів

Коли користувач створює обліковий запис на сайті, ймовірно, він довіряє даному ресурсе. Часто в цьому процесі користувач може погодитись із умовами надання послуг про те, як вона може взаємодіяти з веб-сайтом та послугами, та як власники сайтів використовуватимуть дані та інформацію, яку користувачі надають у програмі. Одне з найголовніших питань в цій угоді, є захист персональних даних користувачів.

Створення власної системи авторизації

Створюючи власну систему авторизації, надзвичайно важливо, щоб інформація надсилалася через з'єднання HTTPS та ефективно приховувалися паролі користувачів, коли вони зберігаються в базі даних. Щоб ефективно це зробити, необхідно використовувати комбінацію хешування та «засолювання».

Хешування - це акт приховування рядка тексту, перетворюючи його на, здавалося б, випадковий рядок. Хешування відбувається в «один кінець», тобто, після того, як текст був захешований, його не можна повернути назад до початкового рядка. При хешуванні пароля текстова версія пароля ніколи не повинна зберігатися в базі даних[3].

Соління - це акт генерації випадкового рядка даних, який буде використовуватися як додаток до хешованого пароля. Це гарантує те, що навіть якщо у двох користувачів паролі однакові, хешована та засолена версії будуть унікальними[4].

`bcrypt` - це популярна функція хешування, заснована на шифрі Blowfish, який зазвичай використовується в ряді веб-фреймворків. Node.js надає можливість використовувати модуль bcrypt для соління та хешування паролів користувачів[5].

У коді програми необхідно імпортувати потрібний модуль і написати функцію для обробки засолювання та хешування за допомогою bcrypt. На рисунку 2.1 зображено приклад коду:

```
const bcrypt = require('bcrypt');
const saltRounds = 10;
const passwordEncrypt = (username, password) => {
  bcrypt.genSalt(saltRounds, function(err, salt) {
    bcrypt.hash(password, salt, function(err, hash) {

// збереження імені користувача, хеша та солі в БД
    });
  });
}
```

Рисунок 2.1 – Приклад засолювання та хешування паролю

Наприклад, пароль PizzaP@rty99 генерує хешований і «солений» рядок $2a$10$HF2rs.iYSvX1l5FPrX697O9dYF/O2kwHuKdQTdy.7oaMwVga54bWG (що є сіллю плюс зашифрований рядок пароля). Тепер, перевіряючи пароль користувача щодо хешованого та засоленого пароля, можна використовувати метод порівняння bcrypt. На рисунку 2.2 зображено приклад коду для порівняння паролю:

```
// password - це значення яке передає користувач
// hash - це значення отримане з БД
bcrypt.compare(password, hash, function(err, res) {
  // res може бути true чи false
});
```

Рисунок 2.2 – Порівняння паролю

Надійність паролю

Надійні паролі користувачів корисні для безпеки, але надані користувачам вимоги щодо символів, які повинен містити пароль та необхідна довжина часто

їх засмучує. Деякі політики щодо паролів прості та інколи навіть абсурдні. Найгірше, що отримані паролі насправді можуть бути менш захищеними, оскільки політики надають потенційним хакерам рекомендації щодо форматів паролів при спробі перебору паролю. Крім того, користувачі, які не використовують програмне забезпечення для управління паролями, швидше за все шукатимуть обхідні шляхи або записуватимуть їх, або зберігатимуть паролі в менш захищених місцях.

Задля спрямування користувачів на створення більш безпечних паролів, можна використовувати оцінювачі надійності пароля. Візуальна індикація буде спонукати користувачів створювати довші паролі. Однак значне збільшення стійкості паролю, можливе тільки за допомогою вимірювачів, які б оцінювали пароль за багатьма факторами. Такі міри змушують користувачів включати більше цифр, символів та великих літер.

Багатофакторна автентифікація

Одним із способів надати більш безпечну систему автентифікації для користувачів є імплементації багатофакторної автентифікації. Багатофакторна автентифікація здійснюється шляхом поєднання двох або більше пунктів з наступного списку:

1. Секрет, відомий користувачеві, наприклад, пароль або PIN-код
2. Фізичний об'єкт, яким користувач володіє, наприклад, мобільний телефон або USB FIDO U2F.
3. Фізична характеристика користувача, така як відбиток пальця або швидкість друку

У веб-додатках найпоширенішим шаблоном є надання двофакторної автентифікації, забезпечуючи фізичну автентифікацію на додаток до стандартних імені користувача та пароля. Часто користувачі отримують текстове повідомлення на свій мобільний телефон або встановлюють багатофакторну програму автентифікації, яка надає відповідний код для додаткової перевірки. Додавання фізичного виміру зменшує можливість викрадення пароля,

забезпечуючи доступ до облікового запису користувача. Хоча багато користувачів можуть не вмикати двофакторну автентифікацію, надання цієї опції є хорошим кроком до кращої безпеки, ніж звичайна автентифікація по імені користувача та пароля.

Шифрування даних користувача

Залежно від типів програм, які розроблюються, вони можуть містити іншу конфіденційну інформацію про користувача, крім облікових даних користувача. Програми можуть зберігати місцеположення користувачів, записи в журналах, номери соціального страхування, медичні записи або будь-яку приватну інформацію, яку користувачі нам довірили. У цьому випадку стає важливим шифрування конфіденційної інформації користувача, а не тільки паролів.

Однак шифрування даних користувача може бути корисним не тільки для обміну повідомленнями та програмам електронної пошти. Наприклад, державним службовцям було б дуже корисно, якщо їх записи шифрувалися б, а не зберігалися як звичайний текст у базі даних.

У Node.js можна використовувати вбудовану бібліотеку crypto для шифрування та дешифрування даних користувача. Зберігаючи дані користувачів у зашифрованому форматі, робиться додатковий крок до захисту даних користувачів. На рисунку 2.3 зображено дуже базовий приклад того, як це може виглядати з парою функцій, які шифрують і розшифровують деякий звичайний текст за допомогою наданого пароля:

```
const crypto = require('crypto');
const dataEncrypt = (password, text) => {
  const cipher = crypto.createCipher('aes192', password);
  let encrypted = cipher.update(text, 'utf8', 'hex');
  encrypted += cipher.final('hex');
  return encrypted;
};

const dataDecrypt = (password, encrypted) => {
  const decipher = crypto.createDecipher('aes192', password);
  let decrypted = decipher.update(encrypted, 'hex', 'utf8');
  decrypted += decipher.final('utf8');
  return decrypted;
}

// Шифруємо дані
const encrypt = dataEncrypt('Password', 'This is encrypted!');
// returns f53a6a423a11be8f27ff86effa5ace548995866009190a90...
const decrypt = dataDecrypt('Password', encrypt);
// returns This is encrypted!
```

Рисунок 2.3 – Приклад функцій для шифрування та дешифрування

2.2. Перевірка введених даних в форму

Форми HTML можуть створити ілюзію управління введенням. Автор розмітки форми може вважати, що оскільки вони обмежують типи значень, які користувач може ввести у форму, дані будуть відповідати цим обмеженням. Але можна бути впевненим, це не більше, ніж ілюзія. Навіть перевірка форми JavaScript на стороні клієнта абсолютно не надає жодної цінності з точки зору безпеки.

Ненадійне введення даних

Довіра до даних, що надходять із браузера користувача, незалежно від того, надається форма чи ні, і незалежно від того, захищено з'єднання HTTPS, фактично дорівнюють нулю.

Користувач міг дуже легко змінити розмітку перед її відправленням або використати програму командного рядка, як curl, для подання несподіваних даних. Або цілком невинний користувач може мимоволі подати змінену версію форми з ворожого веб-сайту. Same Origin Policy не заважає ворожому веб-сайту

відправляти дані до обробника форми на оригінальному сайті. Щоб забезпечити цілісність вхідних даних, перевірку потрібно обробляти на сервері.

Але чому неправильно сформовані дані є проблемою безпеки? Залежно від логіки програми та використання вихідного кодування, запрошується можливість несподіваної поведінки, витоку даних.

Наприклад, слід уявити, що існує форма з перемикачем, що дозволяє користувачеві вибрати тип комунікації. На рисунку 2.4 зображено код обробки форм, який має логіку програми з різною поведінкою залежно від цих значень.

```
const { communicationType } = req.params;
if ("email" === communicationType)) {
  sendByEmail();
} else if ("text" === communicationType) {
  sendByText();
} else {
  sendError(resp, format("Can't send by type %s", communicationType));
}
```

Рисунок 2.4 – Приклад обробки інпуту

Цей код може бути або не бути небезпечним, залежно від того, як реалізований метод `sendError`. Очікується, що логіка подальшого потоку правильно обробляє ненадійний вміст. Перевірка може бути, а може не бути. Буде набагато краще, якщо можливість непередбачуваного потоку управління буде повністю відключена.

Отже, що може зробити розробник, щоб мінімізувати небезпеку того, що ненадійний ввід матиме небажані наслідки в коді програми? Очевидно, перевірка введених даних.

Перевірка вводу

Перевірка вхідних даних - це процес забезпечення відповідності вхідних даних очікуванням програми. Дані, які не потрапляють у очікуваний набір значень, можуть призвести до того, що додаток дасть несподівані результати,

наприклад, порушення бізнес-логіки, активація несправностей і навіть можливість зловмиснику взяти під контроль ресурси або саму програму. Введення, яке оцінюється на сервері як виконуваний код, наприклад, запит до бази даних, або виконується на клієнті як HTML JavaScript є особливо небезпечним. Перевірка вхідних даних є важливою першою лінією захисту для захисту від цього ризику.

Розробники часто створюють додатки з принаймні деякою базовою валідацією, наприклад, щоб переконатися, що значення не є нульовим або ціле число є позитивним. Думка про те, як додатково обмежити введення лише логічно прийнятними значеннями, є наступним кроком до зменшення ризику атаки.

Перевірка вхідних даних є більш ефективною для вхідних даних, які можуть бути обмежені невеликим набором. Числові типи зазвичай можуть бути обмежені значеннями в межах певного діапазону. Наприклад, немає сенсу для користувача вимагати переказу від’ємної суми грошей або додавання кількох тисяч предметів до свого кошика для покупок. Ця стратегія обмеження введення відомими прийнятними типами відома як позитивна перевірка, або білий список. Білий список може обмежуватися рядком певної форми, наприклад URL-адресою або датою форми “pppp / мм / dd ”. Це може обмежити довжину введених даних, одне прийнятне кодування символів або, для прикладу, лише значення, доступні у вашій формі.

Відхилення вхідних даних, що містять відомі небезпечні значення, є стратегією, яку називають негативним підтвердженням або внесенням до чорного списку. Проблема такого підходу полягає в тому, що кількість можливих поганих входів надзвичайно велика. Ведення повного переліку потенційно небезпечних матеріалів було б дорогим та трудомістким заходом. Його також слід постійно підтримувати. Але іноді це єдиний варіант, наприклад у випадках вільного введення. Якщо потрібно щось додати до чорного списку, необхідно бути дуже обережним, щоб охопити всі випадки, потрібно писати хороші тести,

максимально обмежувати, і посилатися на OWASP XSS Filter Evasion Cheats Sheet, щоб вивчити загальні методи, які зловмисники використовуватимуть, щоб обійти захист веб-додатку.

Необхідно боротися зі спокусою, щоб відфільтрувати недійсний ввід. Це практика, яку зазвичай називають "санітарною обробкою". По суті, це чорний список, який вилучає небажані дані, а не відкидає їх. Як і в інших чорних списках, важко отримати правильний результат і надає нападнику більше можливостей ухилитися від нього. Наприклад, у наведеному вище випадку, було вирішено відфільтрувати теги `<script>`. Зловмисник може обійти його чимось таким простим, як: <scr<script>ipt>.

Функціональність перевірки вводу вбудована в більшість сучасних фреймворків, а за відсутності її також можна знайти у зовнішніх бібліотеках, які дозволяють розробнику застосовувати кілька обмежень, що застосовуються як правила для кожного поля. Вбудована перевірка загальних шаблонів, таких як адреси електронної пошти та номери кредитних карток, є корисним бонусом. Використання перевірки веб-платформи забезпечує додаткову перевагу, просуваючи логіку перевірки до самого краю веб-рівня, спричиняючи відхилення недійсних даних, перш ніж вони коли-небудь потраплять до складного коду програми, де критичні помилки легше зробити.

2.3. Кодування виводу HTML

Окрім обмеження та валідації даних, що надходять у додаток, розробники веб-додатків повинні пильно стежити за даними які вони віддають. Сучасна веб-програма зазвичай має базову розмітку HTML для структури документа, CSS для стилів, JavaScript для логіки додатків. Це все текст. І це часто все відображається в одному документі.

Вихідне кодування

Кодування вихідних даних - це перетворення вихідних даних у кінцевий вихідний формат. Складність кодування вихідних даних полягає в тому, що

потрібен інший кодек залежно від способу споживання вихідних даних. Без відповідного вихідного кодування програма може надати своєму клієнту неформатовані дані, що робить їх непридатними для використання або, що ще гірше, небезпечними. Зловмисник, який натрапляє на недостатнє або невідповідне кодування, знає, що у них є потенційна вразливість, яка може дозволити йому принципово змінити структуру результату від закладеної розробником.

2.4. Прив’язування параметрів для запитів до бази даних

Незалежно від того, чи використовується SQL для реляційної бази даних, чи об'єктно-реляційна структура відображення, або запитується база даних NoSQL, потрібно турбуватися про те, як вхідні дані використовуються у запитах.

База даних часто є найважливішою частиною будь-якої веб-програми, оскільки вона може містити дані, які неможливо легко відновити. Вона може містити важливу та конфіденційну інформацію про клієнта, яку потрібно захищати. Саме дані це основа додатку та бізнесу. Отже, розробникам необхідно проявляти найбільшу обережність при взаємодії зі своєю базою даних.

Чистий та безпечний код

Іноді, розробники стикаються із ситуаціями, коли існує напруга між хорошою безпекою та чистим кодом. Безпека іноді вимагає від програміста додавання певної складності для захисту програми. Однак, у цьому випадку отримується одна з тих випадкових ситуацій, коли хороша безпека та хороший дизайн повинні узгоджуватися. На додаток до захисту програми від ін'єкцій, введення пов'язаних параметрів покращує зрозумілість, забезпечуючи чіткі межі між кодом і вмістом, та спрощує створення валідного SQL.

Імплементація прив’язки параметрів для заміни форматування або об’єднання рядків, також надає знайти можливості для введення узагальнених функцій прив’язки до коду, додатково покращуючи чистоту та безпеку коду. Це

ще одне підтвердження того що хороший дизайн та хороша безпека перетинаються.

2.5. Захист даних при передачі

Окрім важливості захисту введення та виведення даних, є ще один важливий момент: конфіденційність та цілісність даних, що передаються. Використовуючи звичайне з'єднання HTTP, користувачі піддаються багатьом ризикам, що виникають внаслідок передачі даних у відкритому тексті. Зловмисник, здатний перехоплювати мережевий трафік в будь-якому місці між браузером користувача та сервером, може підслуховувати або навіть підробляти дані, та залишатися повністю не виявленим в атаці "людина посередині". Немає обмежень щодо того, що може зробити зловмисник, включаючи викрадення сеансу користувача або його особистої інформації, введення шкідливого коду, який буде виконаний браузером у контексті веб-сайту, або зміна даних, які користувач надсилає на сервер.

Зазвичай розробники не можуть контролювати мережу, яку обирають користувачі. Вони цілком можуть використовувати мережу, де кожен може легко спостерігати за їхнім трафіком, наприклад, відкриту бездротову мережу в кафе чи в літаку. Вони могли нічого не підозрюючи підключитися до ворожої бездротової мережі з назвою типу "Безкоштовний Wi-Fi", встановленою зловмисником у громадському місці. Можливо, вони використовують Інтернет-провайдер, який підмішує рекламу у їхній веб-трафік, або вони можуть навіть знаходитись у країні, де уряд регулярно проводить огляд своїх громадян.

Якщо зловмисник може підслуховувати користувача або підробляти веб-трафік, то обмінюваним даним не можуть довіряти жодна із сторін. Для забезпечення захисту від багатьох із цих ризиків використовується HTTPS.

HTTPS та TLS (Transport Layer Security)

Спочатку HTTPS використовувався в основному для захисту чутливого веб-трафіку, такого як фінансові операції, але зараз він використовується за

замовчуванням на багатьох веб-сайтах, які люди використовують в повсякденному житті, таких як соціальні мережі та пошукові системи. Протокол HTTPS використовує протокол Transport Layer Security (TLS), наступника протоколу Secure Sockets Layer (SSL), для захисту зв'язку. При правильній настройці та використанні він забезпечує захист від прослуховування та втручання, а також обґрунтовану гарантію того, що веб-сайт є саме тим, який очікується для користування. Або, більш технічно, це забезпечує конфіденційність та цілісність даних, а також автентифікацію ідентичності веб-сайту.

З огляду на безліч ризиків, з якими користувач стикаються, все частіше стає сенсом трактувати весь мережевий трафік як чутливий і шифрувати його. При роботі з веб-трафіком це робиться за допомогою HTTPS. Кілька виробників браузерів оголосили про намір припинити захист незахищеного HTTP і навіть відображати візуальні вказівки користувачам, щоб попередити їх, коли сайт не використовує HTTPS. Більшість реалізацій HTTP / 2 у браузерах підтримуватимуть спілкування лише через TLS.

Сертифікат сервера

Можливість автентифікації ідентичності веб-сайту підкріплює безпеку TLS. За відсутності можливості перевірити, що сайт є тим, ким він себе називає, зловмисник, здатний здійснити атаку "посередині", може видавати себе за сайт і підірвати будь-який інший захист, який надає протокол.

При використанні TLS сайт підтверджує свою особу за допомогою сертифіката відкритого ключа. Цей сертифікат містить інформацію про сайт разом із відкритим ключем, який використовується, щоб довести, що сайт є власником сертифіката, що він й робить, використовуючи відповідний закритий ключ, який відомий лише йому. У деяких системах від клієнта може також вимагатися використання сертифіката для підтвердження своєї ідентичності, хоча сьогодні це на практиці відносно рідко через складність управління сертифікатами для клієнтів.

Якщо сертифікат для сайту не відомий заздалегідь, клієнту потрібен певний спосіб перевірити, чи можна довіряти сертифікату. Це робиться на основі моделі довіри. У веб-браузерах та багатьох інших додатках довірена третя сторона, яка називається Центром сертифікації (ЦС), перевіряє ідентичність веб-сайту, а іноді й організації, яка ним володіє, а потім надає сайту підписаний сертифікат, щоб підтвердити, що він був перевірений.

Найпомітнішим показником безпеки, який відображається у багатьох веб-браузерах, є те, коли зв’язок із сайтом захищений за допомогою HTTPS, а сертифікат є надійним. Без нього браузер відображатиме попередження про сертифікат і заважатиме користувачеві переглядати ваш сайт, тому важливо отримати сертифікат від надійного ЦС.

Можна сформувати власний сертифікат для тестування конфігурації HTTPS, але перед тим, як надавати послугу користувачам, знадобиться сертифікат, підписаний надійним ЦС. Для багатьох випадків безкоштовний CA є гарною відправною точкою. При пошуку CA є можливість зіткнення з різним рівнем пропонованої сертифікації. Найпростіша, перевірка домену (DV), підтверджує, що власник сертифіката контролює домен. Дорожчими варіантами є перевірка організації (OV) та розширена перевірка (EV), які передбачають проведення ЦС додаткових перевірок для перевірки організації, що вимагає сертифікат.

Використовування HTTPS для всього

Нерідко трапляється веб-сайт, де HTTPS використовується для захисту лише деяких ресурсів, які він обслуговує. У деяких випадках захист може бути поширений лише на обробку поданих форм, які вважаються чутливими. В інших випадках він може використовуватися лише для ресурсів, які вважаються чутливими, наприклад, до того, до чого користувач може отримати доступ після входу на сайт.

Проблема цього непослідовного підходу полягає в тому, що все, що не обслуговується через HTTPS, залишається сприйнятливим до тих видів ризиків,

які були описані раніше. Наприклад, зловмисник, який робить атаку "людина посередині", може просто змінити форму, згадану вище, щоб натомість надіслати конфіденційні дані через відкритий текст HTTP. Якщо зловмисник вводить виконуваний код, який буде виконуватися в контексті нашого сайту, не буде мати великого значення те, що його частина захищена HTTPS. Єдиний спосіб запобігти цим ризикам - використовувати HTTPS для всього.

Рішення не настільки чітке, як перемикання комутатора та обслуговування всіх ресурсів через HTTPS. Веб-браузери за замовчуванням використовують HTTP, коли користувач вводить адресу у свій адресний рядок, не вводячи явно "https: //". Як результат, просто вимкнути мережевий порт HTTP не завжди можливо. Натомість веб-сайти зазвичай переспрямовують запити, отримані через HTTP, на використання HTTPS, що, можливо, не є ідеальним рішенням, але часто найкращим із доступних [8].

Використання HSTS

Перенаправлення користувачів з HTTP на HTTPS представляє ті самі ризики, що й будь-який інший запит, надісланий через звичайний HTTP. Щоб допомогти вирішити цю проблему, сучасні браузери підтримують потужну функцію захисту під назвою HSTS (HTTP Strict Transport Security), яка дозволяє веб-сайту вимагати, щоб браузер взаємодіяв з ним лише через HTTPS. Вперше він був запропонований у 2009 році у відповідь на відомі атаки видалення SSL від Moxie Marlinspike, які продемонстрували небезпеку обслуговування вмісту через HTTP. Увімкнути це так само просто, як надіслати заголовок у відповідь:

Strict-Transport-Security: max-age=15768000

Вказаний вище заголовок вказує браузеру взаємодіяти з веб-сайтом за допомогою HTTPS лише протягом шести місяців (вказано в секундах). HSTS - це важлива функція, яку слід увімкнути завдяки суворій політиці, яку вона застосовує. Після увімкнення браузер автоматично перетворює будь-які незахищені HTTP-запити на використання HTTPS, навіть якщо допущена помилка або користувач явно вводить "http://" у свій адресний рядок. Він також

вказує браузеру заборонити користувачеві обходити попередження, яке відображається, якщо під час завантаження сайту виявляється недійсний сертифікат [9].

Захист файлів cookie

Браузери мають вбудовану функцію захисту, яка допомагає уникнути розголошення файлів cookie, що містять конфіденційну інформацію. Встановлення прапорця "захищений" у файлі cookie вказує браузеру надсилати файли cookie лише під час використання HTTPS. Це важливий захист для використання навіть тоді, коли HSTS увімкнено.

Коли файли cookie використовуються для сеансів, необхідно вжити декілька простих запобіжних заходів, щоб переконатись, що вони не піддаються випадковим діям. Для цього важливо зрозуміти чотири атрибути: Domain, Path, HttpOnly та Secure.

Domain обмежує область застосування файлу cookie певним доменом та його субдоменами, а Path додатково обмежує область застосування шляхом та його підшляхами. Для обох атрибутів за замовчуванням встановлюються досить обмежувальні значення, якщо вони явно не встановлені. За замовчуванням для Domain дозволено надсилати файли cookie лише до вихідного домену та його субдоменів, а за замовчуванням для Path обмежить файл cookie шляхом до ресурсу, де було встановлено файл cookie, та його підшляхами.

Використання для Domain менш обмежувального значення може бути ризикованим. Якби було встановлено для Domain sumdu.edu.ua під час відвідування payments.sumdu.edu.ua для оплати навчання. Це призведе до того, що файл cookie буде відправлений на sumdu.edu.ua та будь-який з його субдоменів за подальшими запитами. Окрім того, що потенційно непотрібно надсилати файли cookie до всіх субдоменів, якщо не контролюється кожен субдомен та його безпеку (наприклад, чи використовують вони HTTPS?), Це може допомогти зловмиснику захопити файли cookie. Що станеться, якщо користувач відвідає scam.sumdu.edu.ua?

Атрибут Path також повинен бути якомога обмежувальним. Якщо ідентифікатор сеансу потрібен лише при доступі до /secret/ path та його підшляхів після входу в систему /login, бажано встановити для нього значення /secret/.

Два інших атрибути, Secure та HttpOnly, контролюють використання файлів cookie. Прапор захищеності вказує на те, що браузер повинен надсилати файли cookie лише під час використання HTTPS. Прапор HttpOnly вказує браузеру, що файл cookie не повинен бути доступним через JavaScript або інші сценарії на стороні клієнта, що допомагає запобігти його викраденню зловмисним кодом.

Склавши його разом, файл cookie може виглядати так:

Set-Cookie: sessionId=[top secret value]; path=/secret/; secure; HttpOnly;
domain=sumdu.edu.ua

Кінцевим ефектом вищезазначеного твердження буде файл cookie з вимкненим доступом до скрипту клієнта, який доступний лише для запитів до шляхів нижче https://payments.sumdu.edu.ua/secret/. Обмежуючи сферу використання файлу cookie, поверхня атаки стає значно меншою.

Інші ризики

Є кілька інших ризиків, про які слід пам’ятати, які можуть призвести до випадкового розголошення конфіденційної інформації, незважаючи на використання HTTPS.

Небезпечно розміщувати конфіденційні дані всередині URL-адреси. Це представляє ризик, якщо URL-адреса кешована в історії браузера, не кажучи вже про те, якщо вона записана в журналах на стороні сервера. Крім того, якщо ресурс за URL-адресою містить посилання на зовнішній сайт і користувач натискає, конфіденційні дані будуть розкриті в заголовку Referer.

Крім того, конфіденційні дані все ще можуть бути кешовані в клієнті або проміжними проксі-серверами, якщо браузер клієнта налаштований на їх використання та дозволяє їм перевіряти трафік HTTPS. Для звичайних користувачів вміст трафіку не буде видимим для проксі-сервера, але практика,

яка часто використовується для підприємств, полягає в тому, щоб встановлювати власну ЦС на системах своїх співробітників, щоб їх системи пом'якшення загрози та відповідності могли контролювати трафік. Необхідно подумати про використання заголовків для вимкнення кешування, щоб зменшити ризик витоку даних через кешування.

3. РОЗРОБКА ВЕБ-ДОДАТКУ

3.1. Архітектура веб-додатку

Архітектура веб-додатку була розроблена для забезпечення високої степені захищеності. Головні цілі, які переслідувалися при розробці:

- Написання чистого та безпечного коду
- Перевірка введених даних користувачем
- Безпечна автентифікація та авторизація
- Безпечне збереження важливих даних для доступу до бази даних
- Шифрування вразливих даних
- Обробка помилок

На рисунку 3.1 зображена архітектура взаємодії між браузером та сервером.

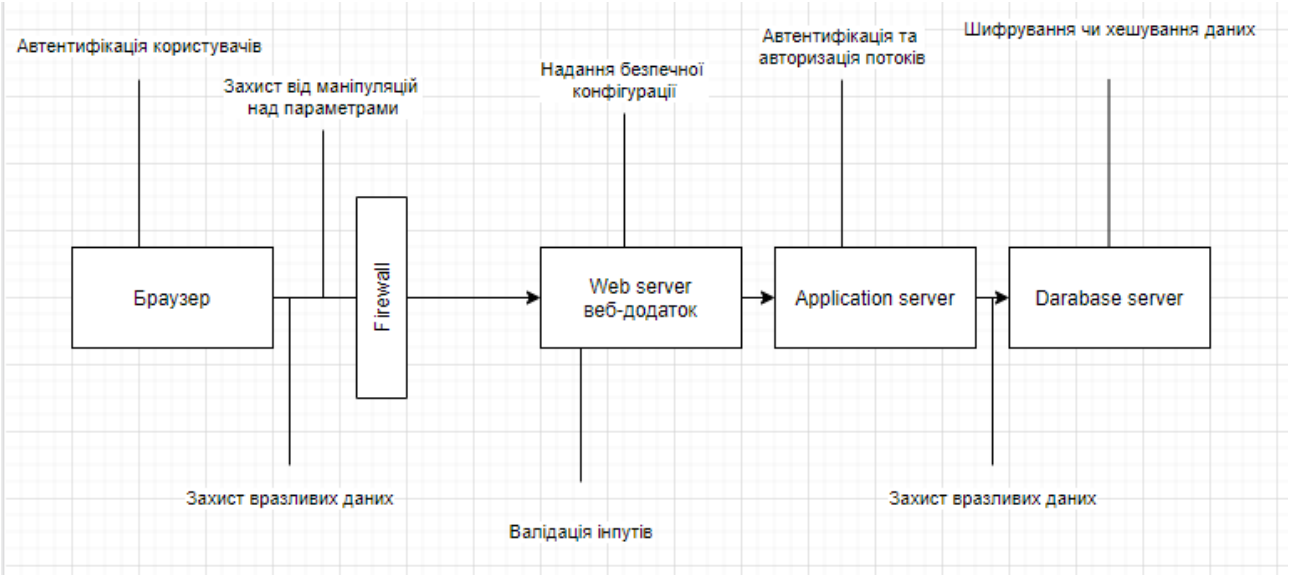

Рисунок 3.1 – Архітектура взаємодії між браузером та сервером

Опис подій. При автентифікації користувачів клієнтська частина додатку валідує дані перед тим як відправити запит до серверу. Відбувається захист від маніпуляцій над параметрами, зайві параметри видаляються, інші приводяться до очікуваного формату. Задля для того, щоб бути більш впевненим в безпечності даних які надійшли на сервер, на ньому відбувається більш жорстка

перевірка даних на валідність. Отримані дані порівнюються на відповідність з очікуваною схемою, перевіряється кількість символів для паролю та його відповідність з хешованою версією, формат електронної пошти та інше. В результаті вдалої автентифікації користувача, та валідації даних виконується запит до бази даних для отримання запрошуваного результату. На рисунку 3.2 можна спостерігати взаємодію користувача з веб-додатком.

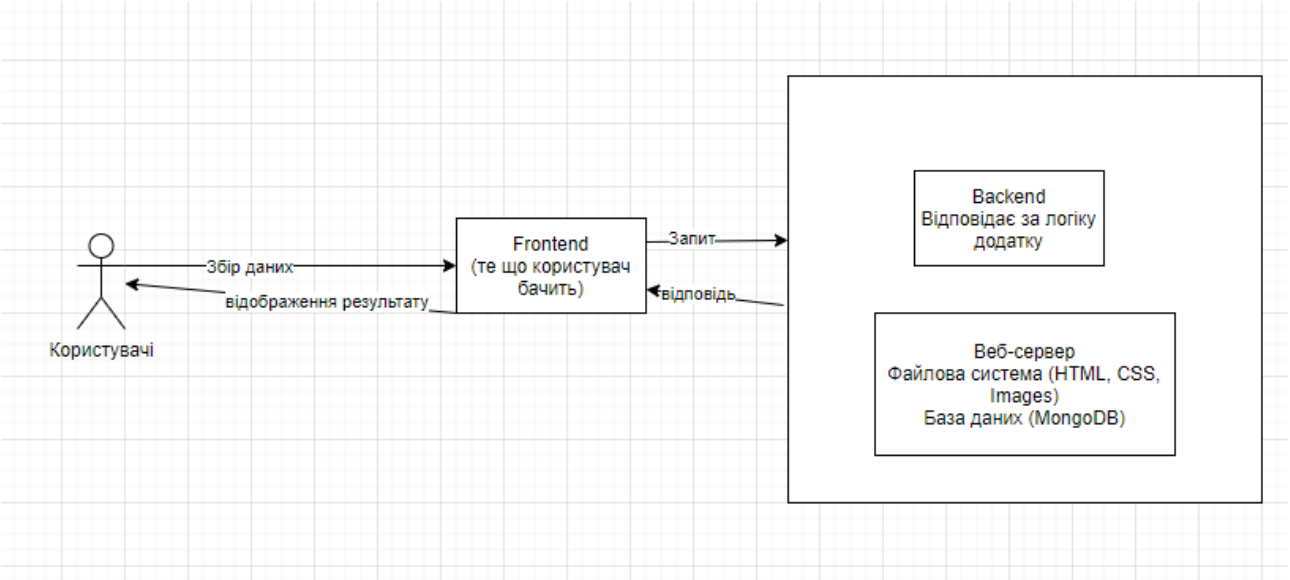

Рисунок 3.2 – Взаємодія користувача з веб-додатком

Для забезпечення надійної автентифікації користувача було прийняте рішення використовувати JWT автентифікацію. Додаток визначає користувача за спеціальним файлом cookie authToken, в якому знаходиться закодована деяка інформація про користувача у вигляді JSON [21].

3.2.Програмна реалізація

Для реалізації поставленої задачі було вирішено використовувати Node.js з Express.js для серверної частини, MongoDB як база даних, React.js (Next.js) для клієнтської частини [22]. Переваги такого стеку технологій:

- Стабільні оновлення та виправлення помилок
- Зручність у використання

- Швидкість розробки нових модулів
- Вбудований захист від великої кількості загроз

Перш за все необхідно визначитися з структурою проекту. Було вирішено розділити серверну та клієнтську частину по різним дерикторіям, оскільки для розробки серверної частини буде використовуватися архітектура REST API. Таке рішення дозволить розробляти бекенд та фронтенд ізольовано один від одного, що в свою чергу призводить до більш зручного та безпечного написання коду різними командами, оскільки найчастіше при розробці додатків фронтенд та бекенд розділяється між членами команди. Також, це може слугувати як додатковий рівень захисту, оскільки команда фронтенду не може отримати код серверної частини, це дозволяє компанії мати ще один рівень захисту від конкурентів, які забажають отримати доступ до реалізації бізнес логіки додатку. На рисунку 3.3 зображено структуру серверної частини веб-додатку:

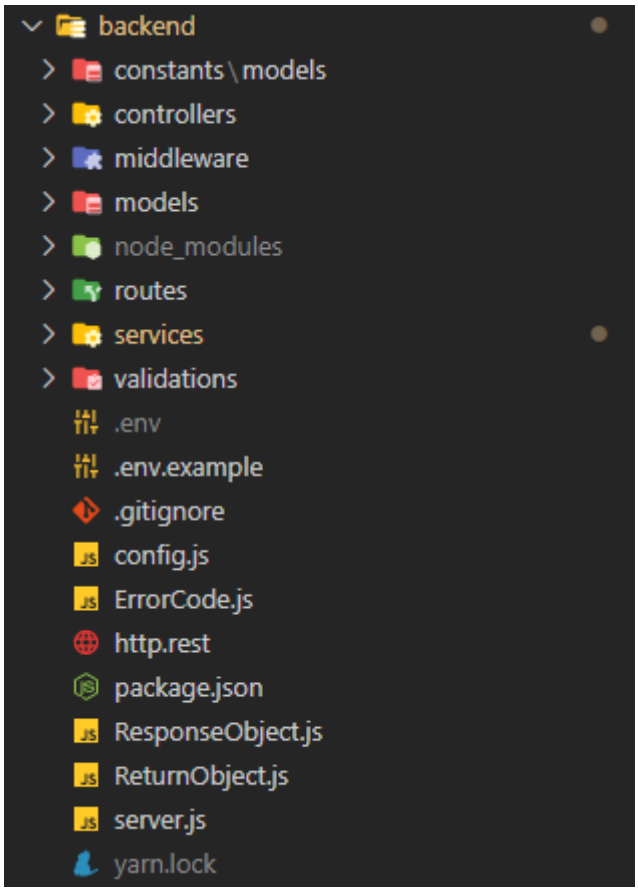

Рисунок 3.3 – Структура серверної частини веб-додатку

В таблиці наведений короткий опис директорій та файлів, те за що вони відповідають та для чого потрібні.

Таблиця 3.1. – Опис директорій та файлів

Назва	Опис
constants	всі константи додатку
controllers	контролери роутів Express.js для всіх ендпоінтів додатку
middleware	проміжні обробники, наприклад, валідація інпуту, перевірка авторизації користувача.
models	моделі MongoDB
routes	роути express.js
services	бізнес-логіка додатку
validations	схеми валідації даних
.env	змінні середовища
config.js	все що стосується конфігурації додатку
server.js	вхідна точка застосунку

Файл server.js представляє собою вхідну точку серверної частини додатку. В даному файлі відбувається підключення до бази даних, налаштування роутів Express.js, налаштування конфігураційних файлів, підключення різних модулів, наприклад, для більш зручного доступу до файлів cookie. Також, відбувається налаштування серверу для використання HTTPS, це є важливим кроком для захисту персональних даних користувачів. На рисунку 3.4 зображено код, на якому зображено налаштування для вище перерахованих пунктів.

```
const config = require('./config');
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
const https = require('https');
const cookieParser = require('cookie-parser');
const routes = require('./routes');

const app = express();
mongoose.connect(
    config.databaseURL,
    { useNewUrlParser: true, useUnifiedTopology: true },
    () => console.log('connected to db')
);
const port = config.port || 4000;

app.use(cors({ origin: ['http://localhost:5001'], credentials: true }));
app.use(cookieParser());
app.use(express.urlencoded({ extended: true }))
app.use(express.json());

app.use(config.api.prefix, routes);

app.get('/', (req, res) => {
    res.send('Welcome on board');
});

app.server = app.listen(port, () => {
    console.log(`Listening on port ${port}`);
});

https.createServer({
    key: fs.readFileSync('server.key'),
    cert: fs.readFileSync('server.cert')
}, app)
    .listen(port, function () {
        console.log(`Listening on port ${port}`)
    })

module.exports = app;
```

Рисунок 3.4 – Вхідна точка серверної частини веб-додатку

Dotenv - один з найкращих способів зберігання API-ключів та налаштувань підключень до бази даних. Необхідно створити .env-файл, який в жодному разі не повинен потрапити до git репозиторію, даний файл прихований в файловій системі. На рисунку 3.5 зображено змісту файлу config.js, в якому повертається об’єкт з усіма підключеними файлами змінних середовища.

```
const dotenv = require('dotenv');
const envFound = dotenv.config();

if (!envFound) {
    throw new Error(`Couldn't find .env file`);
}

module.exports = {
    port: process.env.PORT,
    databaseURL: process.env.MONGODB_URI,
    secret: process.env.SECRET,
    api: {
        prefix: '/api'
    }
}
```

Рисунок 3.5 – Зміст файлу config.js

З таким підходом код залишається структурованим та не забрудненим зайвими змінними, на зразок, process.env.EXAMPLE_VAR, до того ж в IDE, з'являється можливість використовувати автодоповнення.

Наступним кроком, необхідно описати список роутів Express.js, які будуть використані для додавання, редагування та отримання даних з клієнтської частини веб-додатку. Також, необхідно описати роути авторизації та реєстрації. Для цього, необхідно поділити логіку застосунку на окремі модулі, це необхідно задля забезпечення вимог принципу єдиної відповідальності. Кожен окремий модуль має відповідати лише за свою частину реалізації логіки, він повинен бути максимально відокремлений від інших модулів. На рисунку 3.6 зображено приклад налаштування роутів Express.js.

```
const authRouter = require('express').Router();
const authController = require('../controllers/auth.controller');
const authMiddleware = require('../middleware/auth');
const { registerStudentValidation, loginUserValidation } = require('../middleware/validation');

authRouter.route('/registration')
    .post(registerStudentValidation, authController.studentRegistration);

authRouter.route('/login')
    .post(loginUserValidation, authController.login);

authRouter.route('/reauthenticate')
    .post(authController.reauthenticate);

authRouter.route('/logout')
    .post(authMiddleware, authController.logout);

module.exports = authRouter;
```

Рисунок 3.6 – Роути Express.js

В таблиці 3.2 зображено короткий опис роутів Express.js та межі їх відповідальності.

Таблиця 3.2. – Опис роутів Express.js

Роут	Опис
/registration	Відповідає за реєстрацію користувача, валідацію інпуту, обробку помилок, хешування чутливих даних та повернення результату про вдалу, чи не вдалу реєстрацію
/login	Відповідає за валідацію інпуту, пошук користувача у системі, перевірка паролю на відповідність, обробку помилок та генерацію JWT токену, який в свою чергу записується в відповідний файл cookie, з позначкою httpOnly.
/reauthenticate	Використовується для генерації «свіжого» JWT токену, оскільки в нього може бути обмеження в часі життя. Відбувається пошук користувача, перевірка даних, в разі вдалого результату генерується «свіжий» токен.
/logout	Використовується для знищення авторизаційного файлу cookie, при виході з аккаунту.

В файл routes/index.js відбувається підключення налаштувань роутів Express.js всіх модулів, це необхідно для забезпечення більш чистої та зручної структури коду, адже в файлі server.js залишиться тільки підключити лише один файл. Також, така структура, в свою чергу, призводить до написання безпечного коду, оскільки це допомагає написанню автоматизованих тестів. На рисунку 3.7 можна спостерігати приклад підключення роутів модулюй додатку в єдиному файлі.

```
const routes = require('express').Router();
const authRouter = require('./auth.router');
const tutorsRouter = require('./tutors.router');
const profileRouter = require('./profile.router');

routes.use('/', authRouter);
routes.use('/', tutorsRouter);
routes.use('/', profileRouter);

module.exports = routes;
```

Рисунок 3.7 – Підключення модулюй веб-додатку

На рисунку 3.6 імпортуються проміжні обробники, такі як authMiddleware, registerStudentValidation та loginUserValidation. На рисунку 3.8 зображено приклад використання одного з проміжних обробників.

```
exports.registerStudentValidation = async (req, res, next) => {
    try {
        // validate data before user registration
        const userData = {
            firstName: req.body.firstName,
            lastName: req.body.lastName,
            email: req.body.email,
            password: req.body.password
        }

        const { error } = studentRegistrationSchema.validate(userData);
        if (error) return res.json(ReturnObject.fail(error.details[0].message));

        const ro = await accountService.findByEmail(userData.email);
        if (!ro.errors) return res.json(ReturnObject.fail('Email already exists'));

        next();
    } catch (error) {
        console.log('register error', error);
        res.json(ReturnObject.fail(error))
    }
}
```

Рисунок 3.8 – Реалізація методу проміжного обробника

В наведеному коді перевіряються дані на відповідність зі схемою даних, яка очікується для реєстрації користувача. В разі невідповідності одному з полів,

повертається помилка та подальше виконання коду припиняється. На рисунку 3.9 зображено приклад схеми валідації даних.

```
const studentRegistrationSchema = Joi.object({
    firstName: Joi.string()
        .required(),
    lastName: Joi.string()
        .required(),
    email: Joi.string()
        .min(6)
        .required()
        .email(),
    password: Joi.string()
        .min(6)
        .required()
});
```

Рисунок 3.9 – Схема валідації даних

Для опису схеми даних використовується бібліотека Joi, яка дозволяє описати необхідні поля, типи даних, мінімальну та максимальну кількість символів, в даній бібліотеці реалізовано механізм визначення шаблону електронної пошти та інше. Список правил для наведеної схеми можна прочитати як: ім’я та прізвище користувача повинно бути обов’язковим, з типом даних – рядок, електронна пошта – це обов’язковий рядок, в якому мінімальна кількість символів повинна дорівнювати шести та відповідати всім відомій структурі електронної пошти, пароль – це обов’язковий рядок, мінімальна кількість символів дорівнює шести. Також можна додати більш жорсткі правила для пароля, наприклад, наявність в рядку спеціальних символів, великих літер та інше.

При вдалій валідації інпуту продовжується виконання коду та відбувається реєстрація користувача. На рисунку 3.10 зображено код контролера який відповідає за реєстрацію.

```
exports.studentRegistration = async (req, res) => {
    try {
        const registrationData = {
            firstName: req.body.firstName,
            lastName: req.body.lastName,
            email: req.body.email,
            password: req.body.password
        }

        const ro = await registrationService.registerStudent(registrationData);
        if (ro.errors) return res.json(ro);

        res.status(200);
        res.json(ro.data);
    } catch (error) {
        res.json(ReturnObject.fail(error));
    }
};
```

Рисунок 3.10 – Код контролера реєстрації

Короткий опис подій. Після вдалої валідації інпуту, можна довіряти структурі та змісту даних, тому далі слід переходити до додавання нового користувача в базу даних. Логіка додавання користувача знаходиться у відповідному сервісі registrationService. На рисунку 3.11 знаходиться реалізація методу registerStudent.

```
exports.createUserAccount = async (profileId, profileType = AccountEnum.STUDENT, registrationData) => {
    try {
        const { email, password } = registrationData;

        const userAccount = new UserAccount({ email, password });
        const salt = await bcrypt.genSalt(10);

        userAccount.password = await bcrypt.hash(userAccount.password, salt);
        userAccount.profile = profileId;
        userAccount.profileType = profileType;

        const savedUserAccount = await userAccount.save();
        if (!savedUserAccount) return ReturnObject.fail(`Can't save user account`);

        return ReturnObject.success({ userAccount: savedUserAccount });
    } catch (error) {
        console.log('Registration Service Error: ', error);
        return ReturnObject.fail(error);
    }
};
```

Рисунок 3.11 – Код сервісу реєстрації

При розробці даного методу було виконано обробку помилок, наприклад, якщо виникла помилка при створенні нового запису, то зупиняється виконання коду, користувачу відображається відповідне повідомлення. При додаванні користувача, йому автоматично заповнюється поле з датою створення, та назначається унікальний ідентифікатор. При вдалій реєстрації повертається відповідне повідомлення. Тепер користувач може спробувати увійти до свого аккаунту.

Для авторизації користувача було використано JWT авторизацію. При спробі залогінитися, логіка приблизно схожа з реєстрацією. Спочатку валідуються вхідні дані, вони порівнюються з визначеною схемою, потім виконання коду переходить до контролеру, який в свою чергу форматує дані в очікуваний формат та передає їх до сервісу який відповідає за безпеку. На рисунку 3.12 зображено реалізацію контролеру авторизації.

```
exports.login = async (req, res) => {
    try {
        const userData = {
            email: req.body.email,
            password: req.body.password
        }

        const ro = await securityService.login(userData);
        if (ro.errors) return res.json(ro);

        // set auth token
        await securityService.setAuthToken(res, ro.data.token);

        res.status(200);
        res.json(ro.data)
    } catch (error) {
        console.log(error);
        await securityService.removeAuthToken(res);
        res.json(ReturnObject.fail(error));
    }
};
```

Рисунок 3.12 – Код контролера авторизації

Опис подій. Форматуються вхідні дані щоб привести їх до необхідного об’єкту, після чого віддаються їх до сервісу security, в разі вдалої авторизації, повертається JWT токен, який потім зберігається в файлах cookie в браузері користувача. Також, слід підмітити те, що в разі невдалого результату цей самий токен видаляється за браузеру, так як при іншій спробі може знову повторитися помилка. Це може відбутися, наприклад, якщо строк «життя» токену вичерпався та не був оновлений раніше. На рисунку 3.13 зображено код реалізації методу login.

```
exports.login = async (credentials) => {
    try {
        const ro = await accountService.findByEmail(credentials.email);
        if (ro.errors) return ReturnObject.fail('User not found!');

        const { user } = ro.data;

        const validPass = await bcrypt.compare(credentials.password, user.password);
        if (!validPass) return ReturnObject.fail('Invalid Password');

        const token = jwt.sign(
            {
                _id: user._id,
                profile: user.profile,
                profileType: user.profileType,
                dateRegistered: user.dateRegistered
            }, config.secret, { expiresIn: 60 * 1000 * 15 } //60 * 60 * 24
        );

        user.lastEntrance = new Date();
        await user.save();

        return ReturnObject.success({ token });

    } catch (error) {
        console.log('Error Log in: ', error);
        return ReturnObject.fail(error);
    }
};
```

Рисунок 3.13 – Код реалізації методу авторизації

Опис подій. Відбувається пошук користувача в базі даних за адресом електронної пошти, у випадку, якщо не було знайдено запису, повертається

відповідна помилка та припиняється подальше виконання коду, користувачу відображається помилка, в якій йому буде запропоновано повторити спробу ще раз. Якщо користувач, був знайдений то наступним кроком буде перевірка паролю на співпадіння. Також, слід згадати, що пароль в базі даних зберігається в зашифрованому вигляді, тому для перевірки необхідно скористатися бібліотекою bcrypt, для порівняння. Всередині вона приводить пароль який був введений користувачем при авторизації до зашифрованого вигляд та порівнює з паролем з бази даних. При вдалій перевірці, необхідно перейти до створення JWT токену. Для кодування використовуються наступні дані: ідентифікатор користувача, ідентифікатор аккаунту, тип його аккаунту, дата реєстрації. Для більш генерації більш безпечного токену, необхідно до цих даних «підмішати» секретний токен, який зберігається в змінних середовища та виставити строк життя токену. Також, користувачу оновлюється дата останнього входу, яку також можна записувати до журналу, щоб в разі необхідності скористатися файлом логів.

Після вдалої авторизації користувач може отримати доступ до своїх даних на сторінці профілю. При запиті до серверу на отримання даних, спочатку треба переконатися що токен була переданий, якщо ні, то необхідно повернути відповідну помилку. Наступним кроком, буде отримання даних користувача, які були закодовані при авторизації, з отриманого токену. Для розшифрування, необхідно передати таємний ключ, після чого бібліотека використовуючи вбудовані алгоритми дешифрації поверне дані користувача, або нічого. У випадку, якщо дані не були отриманні, то припиняється виконання коду, користувачу повертається повідомлення про те, що токен не був верифікований. На рисунку 3.14 продемонстровано код верифікації токену.

```
async function auth(req, res, next) {
    try {
        const authToken = req.cookies.authToken;
        if (!authToken) return res.json(
            ReturnObject.fail('Token is not specified')
        );

        // retrive user from auth token
        const user = jwt.verify(token, process.env.API_SECRET);
        if (user) return res.json(ReturnObject.fail('User not verified'));

        req.user = user;
        req.token = authToken;
        next();
    } catch (error) {
        res.status(401).json(ReturnObject.fail(error));
    }
```

Рисунок 3.14 – Код верифікації токену

Далі виконання коду переходить до контролеру, який в свою чергу використовує відповідний сервіс, який робить запит до бази даних на отримання інформації про користувача, використовуючи реалізацію логіки з відповідного сервісу аккаунта. Пошук відбувається за унікальним ідентифікатором, також в разі невдалого пошуку користувачу необхідно повернути відповідну помилку. Для забезпечення захисту від атаки «людина посередині», користувачу не повертається зашифрований пароль, щоб його не можна було використати для підміни запиту. На рисунку 3.15 показано приклад формування об’єкту, який буде повертатися користувачу.

```
exports.buildProfile = async (Profile, options, fields = []) => {
    try {
        const requiredProfileFields = [
            'firstName',
            'lastName',
            'skype',
            'profileImage',
            'dateRegistered'
        ];

        fields = [...fields, ...requiredProfileFields];

        const profile = await Profile.findOne(options)
            .select(fields.join(' '))

        return ReturnObject.success({ profile });
    } catch (error) {
        console.log('Build Profile Error: ', error);
        return ReturnObject.fail(error);
    }
};
```

Рисунок 3.15. – Формування об’єкту профіля користувача

3.3. Тестування захищеності веб-додатку

Зберігання даних про користувача можливе лише після проходження ним реєстрації в системі. Тому, необхідно впевнитися в тому, що цей процес працює коректно та всі дані форми проходять валідацію. На рисунку 3.16 продемонстровано спробу реєстрації з паролем, у якому кількість символів менше шести.

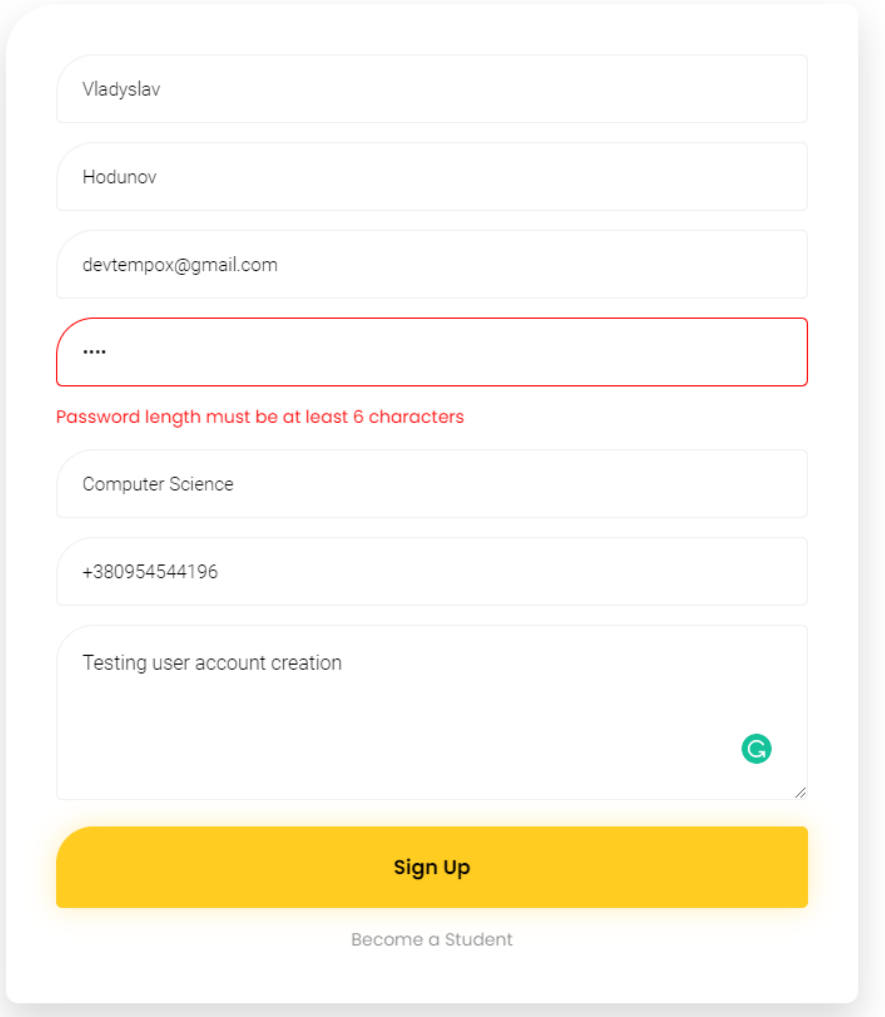

Рисунок 3.16 – Помилка при спробі ввести некоректний пароль

Наступним кроком, необхідно зробити перевірку на можливість реєстрації аккаунту з електронною поштою, яка належить іншому користувачу. Для цього, було створено аккаунт з електронною поштою devtempox@gmail.com . Як можна

спостерігати на рисунку 3.17, веб-додаток вдало пройшов дану перевірку. Користувачу було відображено відповідне повідомлення та підсвічено поле, в якому виникла помилка.

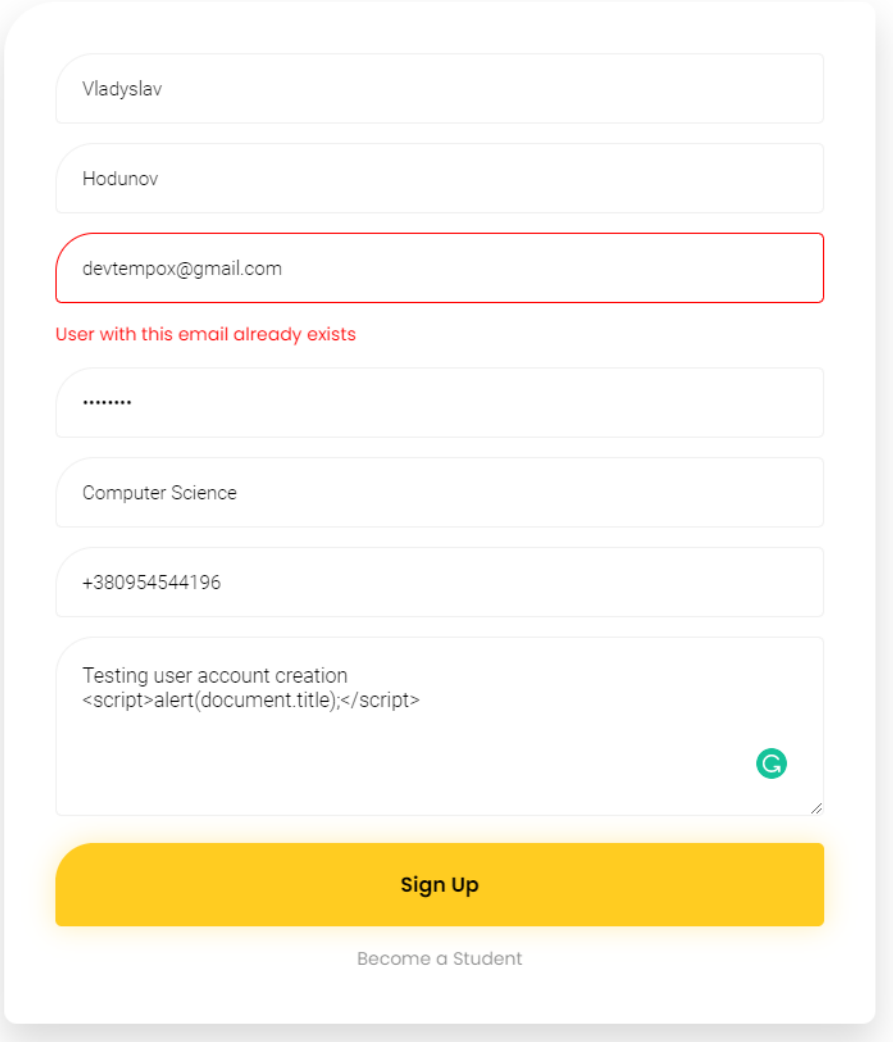

Рисунок 3.17 – Помилка при спробі створення аккаунту з існуючою електронною поштою

Далі було вирішено провести перевірку на захист від XSS-атак, для цього було додано до поля опису аккаунту користувача рядок: <script>alert(document.title);</script>. Тепер необхідно перейти на сторінку профіля та впевнитися в тому, що не з’являється жодне вікно попередження, оскільки дані в полі опису було екрановано. (рисунок 3.18)

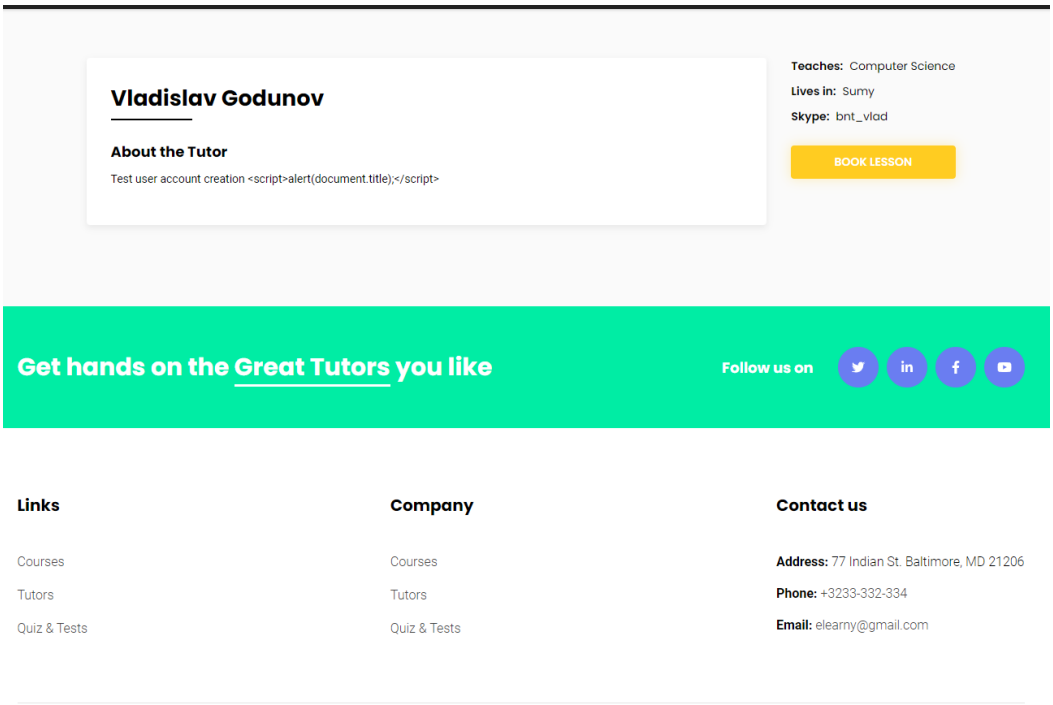

Рисунок 3.18 – Сторінка профіля користувача

Також, слід перевірити захист від SQL-ін’єкцій. Оскільки, при розробці веб-додатку для бази даних було вирішено використовувати MongoDB, то це дозволяє автоматично отримати захист від такого роду атак. При спробі в поле паролю вказати рядок «is null or 1=1;», користувачу було відображено відповідне повідомлення про помилку (рисунок 3.19).

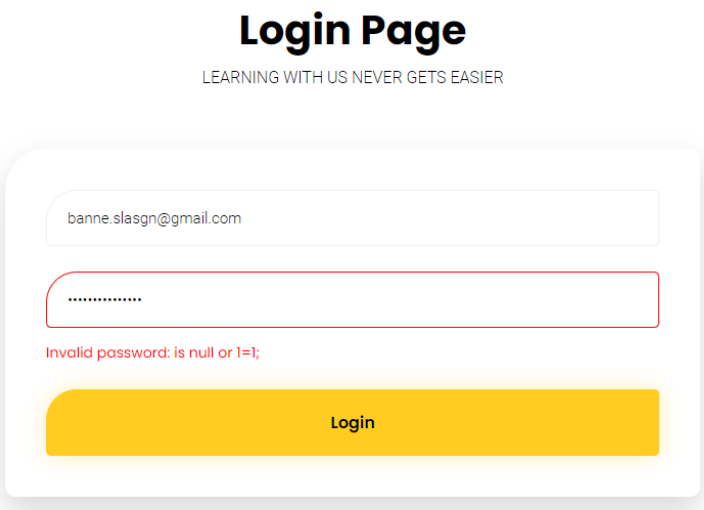

Рисунок 3.19 – Помилка при спробі здійснення SQL-ін’єкції

Також, необхідно впевнитися в тому, що при вдалій спробі авторизації користувачу зберігається токен у файл cookie, з позначкою httpOnly. Для цього необхідно відкрити інструменти розробника Google Chrome, перейти до вкладки Application та вибрати відповідний розділ. Як можна спостерігати на рисунку 3.20, користувачу було коректно збережено токен JWT авторизації.

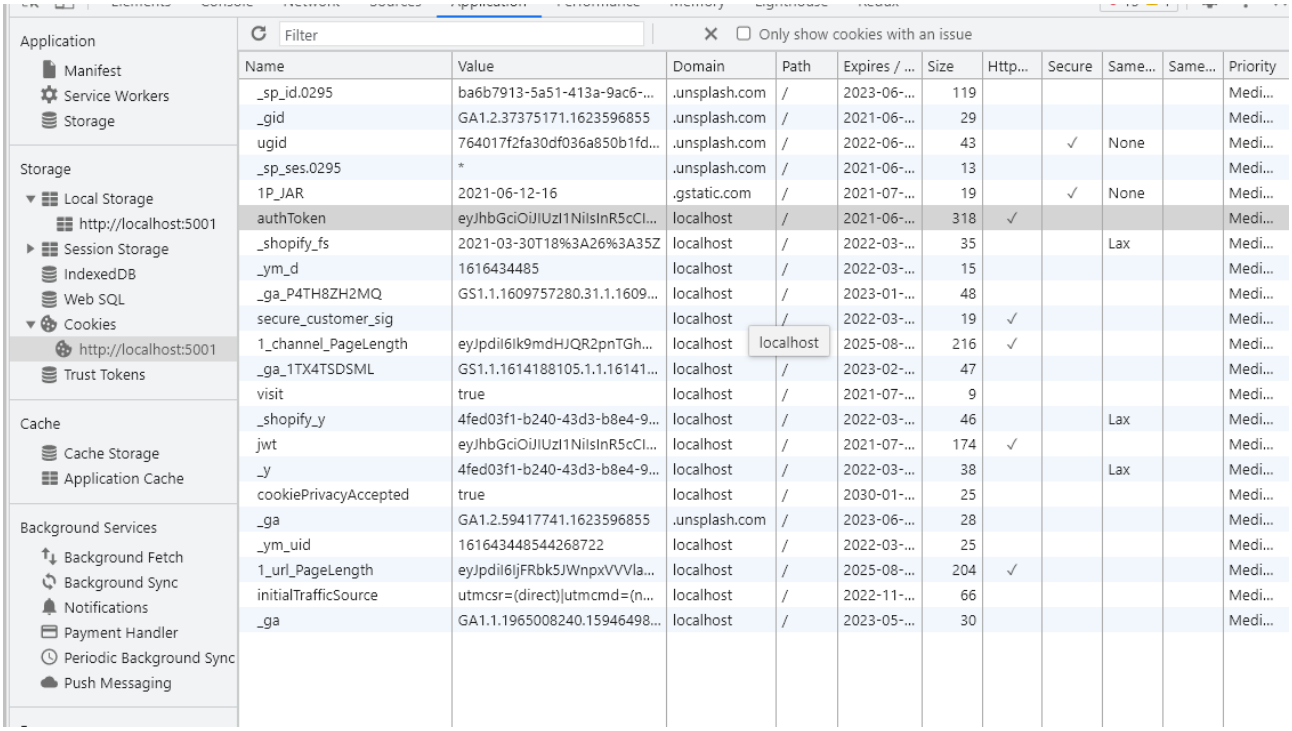

Рисунок 3.20 – Файл cookie authToken

ВИСНОВКИ

У кваліфікаційній роботі бакалавра було детально розглянуто питання захисту персональних даних користувачів веб-додатків, було практично продемонстровано методи захисту від загроз.

В роботі було визначено актуальність розробки безпечних веб-застосунків, оскільки кількість загроз збільшується з кожним днем, тому розробник програмного забезпечення повинен швидко приймати рішення для запобігання витоків конфіденційних даних користувачів. Також, було розглянуто важливість написання зрозумілого та розширюваного коду, для забезпечення стабільності та надійності розроблюваного веб-застосунку.

Було проведено аналіз загроз та визначено список методів захисту по відношенню до персональних даних користувачів, які потім були застосовані під час розробки програмного забезпечення. Внаслідок аналізу, було вирішено використати наступний список технологій: Node.js та Express.js, MongoDb – для бази даних та React.js для розробки клієнтської частини веб-додатку.

В процесі розробки було розроблена архітектура додатку, яка дозволяє з легкістю розширювати функціональні можливості застосунку, що дозволяє швидко добавляти нові шари захисту даних. Дана структура надає можливість зручно та швидко тестувати веб-додаток, що в свою чергу призводить до меншої кількості помилок та більш безпечної роботи додатку.

Результатом кваліфікаційної роботи є розроблений комплекс заходів та надання рекомендацій й методів захисту персональних даних користувачів, для цього було розроблено та протестовано веб-додаток, який відповідає визначеним критеріям захищеності.

СПИСОК ЛІТЕРАТУРИ

1. КОНФІДЕНЦІЙНА ІНФОРМАЦІЯ, ІНФОРМАЦІЯ ПРО ОСОБУ ТА ПЕРСОНАЛЬНІ ДАНІ: СПІВВІДНОШЕННЯ І РЕГУЛЮВАННЯ [Електронний ресурс]. – URL: https://cedem.org.ua/analytics/konfidentsijna-informatsiya-informatsiya-pro-osobu-ta-personalni-dani-spivvidnoshennya-i-regulyuvannya/

2. IBM Report: Compromised Employee Accounts Led to Most Expensive Data Breaches Over Past Year [Електронний ресурс]. – URL: https://newsroom.ibm.com/2020-07-29-IBM-Report-Compromised-Employee-Accounts-Led-to-Most-Expensive-Data-Breaches-Over-Past-Year

3. Хешування — Wiki ТНТУ [Електронний ресурс]. – URL: https://wiki.tntu.edu.ua/%D0%A5%D0%B5%D1%88%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F

4. Сіль (криптографія) [Електронний ресурс]. – URL: https://uk.wikipedia.org/wiki/Сіль_(криптографія)

5. bcrypt - npm [Електронний ресурс]. – URL: https://www.npmjs.com/package/bcrypt

6. Захист персональних даних: проблеми і рішення [Електронний ресурс]. – URL: https://dtpa.dn.ua/qa/zahist/uk/baza-zahist-personalnih-danih-problemi-i-risenna.html

7. Що таке шифрування та як воно працює? [Електронний ресурс]. – URL: [Електронний ресурс]. – URL: https://experience.dropbox.com/uk-ua/resources/what-is-encryption

8. What is HTTPS? [Електронний ресурс]. – URL: https://www.cloudflare.com/learning/ssl/what-is-https/

9. What Is HSTS and How Do I Implement It? [Електронний ресурс]. – URL: https://www.globalsign.com/en/blog/what-is-hsts-and-how-do-i-use-it

10. УГРОЗЫ ИНФОРМАЦИОННОЙ БЕЗОПАСНОСТИ [Електронний ресурс]. – URL: https://searchinform.ru/informatsionnaya-bezopasnost/osnovy-ib/ugrozy-informatsionnoj-bezopasnosti/.

11. 11 Ways to Improve Your Web Application Security [Електронний ресурс]. – URL: https://patchstack.com/web-application-security/.

12. О. Бондаренко, І. Ушкаленко БЕЗПЕКА WEB-ДОДАТКІВ: АКТУАЛЬНІ ПРОБЛЕМИ ТА ЇХ АНАЛІЗ, 2017 – 28-36 с.

13. WEB APPLICATION SECURITY: 12 BEST PRACTICES YOU SHOULD KNOW [Електронний ресурс]. – URL: https://www.imaginovation.net/blog/web-app-security-best-practices/.

14. Paco Hope, Paco, Walter, Ben. Web Security Testing Cookbook. –Sebastopol (USA): O'Reilly Media, 2008. – 314 p.

15. Same-origin policy [Електронний ресурс]. – URL: https://en.wikipedia.org/wiki/Same-origin_policy

16. Як захистити веб-додатки: основні поради, інструменти, корисні посилання [Електронний ресурс]. – URL: https://echo.lviv.ua/dev/6231.

17. OWASP Top Ten [Електронний ресурс]. – URL: https://owasp.org/www-project-top-ten/

18. Web application Security: Vulnerabilities and major security practices [Електронний ресурс]. – URL: https://parkardigital.com/web-application-security/

19. Leon Shklar, Richard Rosen. Web application architecture. Principles, protocols, and practices. - John Wiley & Sons Ltd, 2003. – 374 p.

20. Hari Simhadri. Application Security Architecture. - GSEC Practical Requirements (v1.4b) 2003. – 15p.

21. JWT authentication: When and how to use it [Електронний ресурс]. – URL: https://blog.logrocket.com/jwt-authentication-best-practices/

22. Express 4.x - API Reference [Електронний ресурс]. – URL: https://expressjs.com/en/4x/api.html

23. What is Data Loss Prevention (DLP)? A Definition of Data Loss Prevention [Електронний ресурс]. – URL: https://digitalguardian.com/blog/what-data-loss-prevention-dlp-definition-data-loss-prevention

24. EDRi’s members and observers. An introduction to DATA PROTECTION. - The EDRi papers. – 22 p.

ДОДАТОК А. ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБ-ДОДАТКУ

Файл server.js

```
const config = require('./config');
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
const https = require('https');
const cookieParser = require('cookie-parser');
const routes = require('./routes');

const app = express();
mongoose.connect(
    config.databaseURL,
    { useNewUrlParser: true, useUnifiedTopology: true },
    () => console.log('connected to db')
);
const port = config.port || 4000;

app.use(cors({ origin: ['http://localhost:5001'], credentials:
true }));
app.use(cookieParser());
app.use(express.urlencoded({ extended: true }))
app.use(express.json());

app.use(config.api.prefix, routes);

app.get('/', (req, res) => {
    res.send('Welcome on board');
});

app.server = app.listen(port, () => {
    console.log(`Listening on port ${port}`);
});

https.createServer({
    key: fs.readFileSync('server.key'),
    cert: fs.readFileSync('server.cert')
}, app)
    .listen(port, function () {
        console.log(`Listening on port ${port}`)
    })

module.exports = app;
```

Файл .env

```
PORT=5000
MONGODB_URI=mongodb+srv://test:pass@cluster0-
3c2ev.mongodb.net/test?retryWrites=true&w=majority
```

```
API_SECRET=kdfe!@#34asdfggerqtqCXvsad
```

Файл config.js

```
const dotenv = require('dotenv');
const envFound = dotenv.config();

if (!envFound) {
    throw new Error(`Couldn't find .env file`);
}

module.exports = {
    port: process.env.PORT,
    databaseURL: process.env.MONGODB_URI,
    secret: process.env.SECRET,
    api: {
        prefix: '/api'
    }
}
```

Файл routes/index.js

```
const routes = require('express').Router();
const authRouter = require('./auth.router');
const profileRouter = require('./profile.router');

routes.use('/', authRouter);
routes.use('/', profileRouter);

module.exports = routes;
```

Файл routes/auth.router.js

```
const authRouter = require('express').Router();
const authController = require('../controllers/auth.controller');
const authMiddleware = require('../middleware/auth');
const { registerStudentValidation, registerTutorValidation,
loginUserValidation } = require('../middleware/validation');

authRouter.route('/registration')
    .post(registerStudentValidation,
authController.studentRegistration);

authRouter.route('/registration/tutor')
    .post(registerTutorValidation,
authController.tutorRegistration);

authRouter.route('/login')
    .post(loginUserValidation, authController.login);

authRouter.route('/reauthenticate')
    .post(authController.reauthenticate);
```

```
authRouter.route('/logout')
    .post(authMiddleware, authController.logout);

module.exports = authRouter;
```

Файл routes/profile.router.js

```
const profileRouter = require('express').Router();
const profileController =
require('../controllers/profile.controller');
const authMiddleware = require('../middleware/auth');

profileRouter.route('/profile')
    .post(authMiddleware, profileController.profile);

module.exports = profileRouter;
```

Файл middleware/auth.js

```
const ReturnObject = require('../ReturnObject');
const jwtAuthenticatorService =
require('../services/jwtAuthenticator.service');

async function auth(req, res, next) {
    try {
        const authToken = req.cookies.authToken;
        if (!authToken) return res.json(ReturnObject.fail('Token
is not specified'));

        // retrive user from auth token
        const ro = await
jwtAuthenticatorService.retriveUser(authToken);

        if (ro.errors) return res.json(ro);
        const { user } = ro.data;

        console.log('Auth middleware: ', authToken);

        req.user = user;
        req.token = authToken;
        next();
    } catch (error) {
        res.status(401).json(ReturnObject.fail(error));
    }
}

module.exports = auth;
```

Файл middleware/validation.js

```
const ReturnObject = require('../ReturnObject');
const { userLoginSchema } = require('../validations/login');
const { studentRegistrationSchema, tutorRegistrationSchema } =
require('../validations/registration');

const accountService =
require('../services/account/account.service');

exports.registerStudentValidation = async (req, res, next) => {
    try {
        // validate data before user registration
        console.log('validate');

        const userData = {
            firstName: req.body.firstName,
            lastName: req.body.lastName,
            email: req.body.email,
            password: req.body.password
        }

        const { error } =
studentRegistrationSchema.validate(userData);
        if (error) return
res.json(ReturnObject.fail(error.details[0].message));

        // Check if the user already registered
        const ro = await
accountService.findByEmail(userData.email);

        if (!ro.errors) return res.json(ReturnObject.fail('Email
already exists'));

        next();
    } catch (error) {
        console.log('register error', error);
        res.json(ReturnObject.fail(error))
    }
}

exports.registerTutorValidation = async (req, res, next) => {
    try {
        // validate data before user registration
        console.log('validate');

        const userData = {
            firstName: req.body.firstName,
            lastName: req.body.lastName,
            email: req.body.email,
            password: req.body.password,
```

```
            city: req.body.city,
            subject: req.body.subject,
            hourlyRate: req.body.hourlyRate,
            phone: req.body.phone,
            skype: req.body.skype,
            description: req.body.description,
            profileImage: req.body.profileImage
        }

        const { error } =
tutorRegistrationSchema.validate(userData);
        if (error) return
res.json(ReturnObject.fail(error.details[0].message));

        // Check if the user already registered
        const ro = await
accountService.findByEmail(userData.email);

        if (!ro.errors) return res.json(ReturnObject.fail('Email
already exists'));

        next();
    } catch (error) {
        console.log('register error', error);
        res.json(ReturnObject.fail(error))
    }
};

exports.loginUserValidation = async (req, res, next) => {
    try {
        // validate data before user registration
        const userData = {
            email: req.body.email,
            password: req.body.password
        }

        const { error } = userLoginSchema.validate(userData);
        if (error) return
res.json(ReturnObject.fail(error.details[0].message));

        next();
    } catch (error) {
        console.log(error);
        res.json(ReturnObject.fail(`Bad credentionals`));
    }
}
```

Файл controllers/auth.controller.js

```
const registrationService =
require('../services/registration.service');
```

```
const securityService = require('../services/security.service');
const ReturnObject = require('../ReturnObject');
const ResponseObject = require('../ResponseObject');

exports.studentRegistration = async (req, res) => {
    try {
        const registrationData = {
            firstName: req.body.firstName,
            lastName: req.body.lastName,
            email: req.body.email,
            password: req.body.password
        }

        const ro = await
registrationService.registerStudent(registrationData);

        console.log('Student Registration ro: ', ro);

        if (ro.errors) return res.json(ro);

        res.status(200);
        res.json(ro.data);
    } catch (error) {
        res.json(ReturnObject.fail(error));
    }
};

exports.tutorRegistration = async (req, res) => {
    try {
        const registrationData = {
            firstName: req.body.firstName,
            lastName: req.body.lastName,
            email: req.body.email,
            password: req.body.password,
            city: req.body.city,
            subject: req.body.subject,
            hourlyRate: req.body.hourlyRate,
            phone: req.body.phone,
            skype: req.body.skype,
            description: req.body.description,
            profileImage: req.body.profileImage
        }

        const ro = await
registrationService.registerTutor(registrationData);

        console.log('Tutor Registration ro: ', ro);

        if (ro.errors) return res.json(ro);

        res.status(200);
```

```
        res.json(ro.data);

    } catch (error) {
        res.json(ReturnObject.fail(error));
    }
};

exports.login = async (req, res) => {
    try {
        const userData = {
            email: req.body.email,
            password: req.body.password
        }

        const ro = await securityService.login(userData);
        if (ro.errors) return res.json(ro);

        // set auth token
        await securityService.setAuthToken(res, ro.data.token);

        res.status(200);
        res.json(ro.data)
    } catch (error) {
        console.log(error);
        await securityService.removeAuthToken(res);
        res.json(ReturnObject.fail(error));
    }
};

// TODO: generate fresh token
exports.reauthenticate = async (req, res) => {
    try {
        const authToken = req.cookies.authToken;

        if (!authToken) {
            res.status(401).json(ReturnObject.fail('Access
Denied!'));
        }

        const ro = await
securityService.authenticateByRequestHeaders(authToken);
        if (ro.errors) {
            // remove auth token
            await securityService.removeAuthToken(res);
            return res.json(ro);
        };

        await securityService.setAuthToken(res, ro.data.token);
        return res.json(ro.data);
    } catch (error) {
        console.log(error);
```

```
        await securityService.removeAuthToken(res);
        res.json(ReturnObject.fail(error));
    }
};

exports.logout = async (req, res) => {
    try {
        await securityService.removeAuthToken(res);
        res.json(ResponseObject.success({ message: 'Logged out'
}));
    }
    catch (error) {
        console.log(error);
        res.json(ReturnObject.fail(error));
    }
}
```

Файл controllers/profile.controller.js

```
const AccountEnum = require('../constants/models/AccountEnum');
const ResponseObject = require('../ResponseObject');
const profileService =
require('../services/profile/profile.service');

exports.profile = async (req, res) => {
    try {
        const userAccount = req.user;

        if (!userAccount.profileType) return
res.json(ResponseObject.fail('Profile type is not specified'));

        let ro;

        if (userAccount.profileType === AccountEnum.STUDENT)
            ro = await
profileService.getStudentProfile(userAccount.profile);

        if (userAccount.profileType === AccountEnum.TUTOR) {
            const profileFields = ['description', 'hourlyRate',
'subject', 'city', 'phone'];
            ro = await
profileService.getTutorProfile(userAccount.profile,
profileFields);
        }

        res.json(ResponseObject.success({ profile: ro.data,
profileType: userAccount.profileType }));
    } catch (error) {
        console.log(error);
        res.json(ResponseObject.fail(`Can't load profile`));
    }
```

```
}
```

Файл validations/login.js

```
const Joi = require('@hapi/joi');

const userLoginSchema = Joi.object({
    email: Joi.string()
        .min(6)
        .required()
        .email()
        .error(() => { return { message: 'Email is required' } }),
    password: Joi.string()
        .min(6)
        .required()
        .error(() => { return { message: 'Password is required' }
})
});

module.exports.userLoginSchema = userLoginSchema;
```

Файл validations/registration.js

```
const Joi = require('@hapi/joi');

const userAccountSchema = Joi.object({
    email: Joi.string()
        .min(6)
        .required()
        .email(),
    password: Joi.string()
        .min(6)
        .required()
});

const studentRegistrationSchema = Joi.object({
    firstName: Joi.string()
        .required(),
    lastName: Joi.string()
        .required(),
    email: Joi.string()
        .min(6)
        .required()
        .email(),
    password: Joi.string()
        .min(6)
        .required()
});

const tutorRegistrationSchema = Joi.object({
```

```
    firstName: Joi.string()
        .required(),
    lastName: Joi.string()
        .required(),
    email: Joi.string()
        .min(6)
        .required()
        .email(),
    password: Joi.string()
        .min(6)
        .required(),
    city: Joi.string()
        .required(),
    subject: Joi.string()
        .required(),
    hourlyRate: Joi.number()
        .greater(5)
        .required(),
    phone: Joi.string()
        .required(),
    skype: Joi.string()
        .required(),
    profileImage: Joi.string()
        .min(30)
        .required(),
    description: Joi.string()
        .min(30)
        .required()
});

module.exports.userAccountSchema = userAccountSchema;
module.exports.studentRegistrationSchema =
studentRegistrationSchema;
module.exports.tutorRegistrationSchema = tutorRegistrationSchema;
```

Файл services/account/account.service.js

```
const bcrypt = require('bcryptjs');
const ReturnObject = require('../../ReturnObject');
const { UserAccount } = require('../../models/UserAccount');

exports.get = async (_id) => {
    try {
        const account = await UserAccount.findById(_id);

        if (!account) return ReturnObject.fail(`User does't
exist`)

        return ReturnObject.success({ user: account });
    } catch (error) {
        return ReturnObject.fail(error);
```

```
    }
};

exports.createUserAccount = async (profileId, profileType =
AccountEnum.STUDENT, registrationData) => {
    try {
        const { email, password } = registrationData;

        const userAccount = new UserAccount({ email, password });
        const salt = await bcrypt.genSalt(10);

        userAccount.password = await
bcrypt.hash(userAccount.password, salt);
        userAccount.profile = profileId;
        userAccount.profileType = profileType;

        const savedUserAccount = await userAccount.save();
        if (!savedUserAccount) return ReturnObject.fail(`Can't
save user account`);

        return ReturnObject.success({ userAccount:
savedUserAccount });
    } catch (error) {
        console.log('Registration Service Error: ', error);
        return ReturnObject.fail(error);
    }
};

exports.findByProfileId = async (_id) => {
    try {
        const account = await UserAccount.findOne({ profile: _id
});

        if (!account) return ReturnObject.fail(`User does't
exist`)

        return ReturnObject.success({ user: account });
    } catch (error) {
        return ReturnObject.fail(error);
    }
};

exports.findByEmail = async (email) => {
    try {
        const account = await UserAccount.findOne({ email: email
});

        if (!account) return ReturnObject.fail(`User does't
exist`)

        return ReturnObject.success({ user: account });
```

```
    } catch (error) {
        return ReturnObject.fail(error);
    }
};
```

Файл services/account/accountBuilder.service.js

```
const ReturnObject = require('../../ReturnObject');
const { UserAccount } = require('../../models/UserAccount');

exports.buildItem = async (options, accountFields) => {
    try {
        const requiredAccountFields = ['profile', 'profileType',
'-_id'];
        accountFields = [...accountFields,
...requiredAccountFields];

        const account = await UserAccount.findOne(options)
            .select(accountFields.join(' '));

        return ReturnObject.success({ account });
    } catch (error) {
        console.log(error);
        return ReturnObject.fail(error);
    }
};

exports.buildItemWithProfile = async (options, accountFields,
profileFields) => {
    try {
        const requiredAccountFields = ['profile', 'profileType',
'-_id'];
        const requiredProfileFields = ['firstName', 'lastName'];

        accountFields = [...accountFields,
...requiredAccountFields];
        profileFields = [...profileFields,
...requiredProfileFields];

        const account = await UserAccount.findOne(options)
            .select(accountFields.join(' '))
            .populate({ path: 'profile', select:
profileFields.join(' ') })

        return ReturnObject.success({ account });
    } catch (error) {
        console.log(error);
        return ReturnObject.fail(error);
    }
};
```

```
exports.buildListWithProfiles = async (options, accountFields,
profileFields) => {
    try {
        const requiredAccountFields = ['profile', 'profileType',
'-_id'];
        const requiredProfileFields = ['firstName', 'lastName'];

        accountFields = [...accountFields,
...requiredAccountFields];
        profileFields = [...profileFields,
...requiredProfileFields];

        const accounts = await UserAccount.find(options)
            .select(accountFields.join(' '))
            .populate({ path: 'profile', select:
profileFields.join(' ') })

        return ReturnObject.success({ accounts });
    } catch (error) {
        console.log(error);
        return ReturnObject.fail(error);
    }
};
```

Файл services/profile/profile.service.js

```
const ReturnObject = require('../../ReturnObject');
const profileBuilderService = require('./profileBuilder.service');
const accountService = require('../account/account.service');
const { Student } = require('../../models/Student');
const { Tutor } = require('../../models/Tutor');

// TODO
exports.get = async (_id) => {
    try {
        const ro = await accountService.findByProfileId(_id);
        const { user } = ro.data;

        const profile = user.populate('profile');

        console.log(profile);

        return ReturnObject.success({ profile });
    } catch (error) {
        return ReturnObject.fail(error);
    }
};

exports.getStudentProfile = async (_id) => {
    try {
```

```
        const ro = await
profileBuilderService.buildProfile(Student, { _id });

        if (ro.errors) return ReturnObject.fail(`Error building
profile`);

        const { profile } = ro.data;

        return ReturnObject.success(profile);
    } catch (error) {
        console.log('Error: ', error);
        return ReturnObject.fail(`Can't get Student Profile`);
    }
};

exports.getTutorProfile = async (_id, fields = []) => {
    try {
        const ro = await profileBuilderService.buildProfile(Tutor,
{ _id }, fields);

        if (ro.errors) return ReturnObject.fail(`Error building
profile`);

        const { profile } = ro.data;

        return ReturnObject.success(profile);
    } catch (error) {
        return ReturnObject.fail(`Can't get Tutor Profile`);
    }
};
```

Файл services/profileBuilder.service.js

```
const ReturnObject = require('../../ReturnObject');

exports.buildProfile = async (Profile, options, fields = []) => {
    try {
        const requiredProfileFields = [
            'firstName',
            'lastName',
            'skype',
            'profileImage',
            'dateRegistered'
        ];

        fields = [...fields, ...requiredProfileFields];

        const profile = await Profile.findOne(options)
            .select(fields.join(' '))

        return ReturnObject.success({ profile });
```

```
    } catch (error) {
        console.log('Build Profile Error: ', error);
        return ReturnObject.fail(error);
    }
};

exports.buildList = async (Profile, options, fields = []) => {
    try {
        const requiredProfileFields = ['firstName', 'lastName',
'profileImage', 'dateRegistered', 'skype'];
        fields = [...fields, ...requiredProfileFields];

        const profiles = await Profile.find(options)
            .select(fields.join(' '))

        return ReturnObject.success({ profiles });
    } catch (error) {
        console.log(error);
        return ReturnObject.fail(error);
    }
};
```

Файл services/jwtAuthenticator.service.js

```
const jwt = require('jsonwebtoken');
const ReturnObject = require('../ReturnObject');

exports.retriveUser = async (token) => {
    try {
        const verified = jwt.verify(token,
process.env.API_SECRET);

        return ReturnObject.success({ user: verified });
    } catch (error) {
        console.log('JWT authenticator service error: ',
error.message ? error.message : error);
        return ReturnObject.fail(error.message ? error.message :
error);
    }
};
```

Файл services/registration.service.js

```
const AccountEnum = require('../constants/models/AccountEnum');
const ReturnObject = require('../ReturnObject');
const { Student } = require('../models/Student');
const { Tutor } = require('../models/Tutor');
const accountService = require('./account/account.service');

exports.registerStudent = async (registrationData) => {
```

```
    try {
        const {
            firstName,
            lastName
        } = registrationData;

        const student = new Student({ firstName, lastName });
        if (!student) return ReturnObject.fail('Error creating
Student');

        const ro = await
accountService.createUserAccount(student._id, AccountEnum.STUDENT,
registrationData);
        if (ro.errors) return ro;

        const savedStudent = await student.save();
        if (!savedStudent) return ReturnObject.fail(`Can't save
student`);

        return ReturnObject.success({ message: 'Account
successfully created!' });
    } catch (error) {
        console.log('Registration Service Error: ', error);
        return ReturnObject.fail(error);
    }
};

exports.registerTutor = async (registrationData) => {
    try {
        const tutorData = {
            firstName: registrationData.firstName,
            lastName: registrationData.lastName,
            description: registrationData.description,
            city: registrationData.city,
            subject: registrationData.subject,
            hourlyRate: registrationData.hourlyRate,
            phone: registrationData.phone,
            skype: registrationData.skype,
            profileImage: registrationData.profileImage
        }

        const tutor = new Tutor(tutorData);
        if (!tutor) return ReturnObject.fail('Error creating
Tutor');

        const ro = await
accountService.createUserAccount(tutor._id, AccountEnum.TUTOR, {
email: registrationData.email, password: registrationData.password
});
        if (ro.errors) return ro;
```

```
        const savedTutor = await tutor.save();
        if (!savedTutor) return ReturnObject.fail(`Can't save
tutor`);

        return ReturnObject.success({ message: 'Account
successfully created!' });
    } catch (error) {
        console.log('Registration Service Error: ', error);
        return ReturnObject.fail(error);
    }
};
```

Файл services/security.service.js

```
const ReturnObject = require('../ReturnObject');
const jwt = require('jsonwebtoken');
const bcrypt = require('bcryptjs');

const jwtAuthenticatorService =
require('../services/jwtAuthenticator.service');
const accountService = require('./account/account.service');

exports.login = async (credentials) => {
    try {
        const ro = await
accountService.findByEmail(credentials.email);

        if (ro.errors) return ReturnObject.fail('User not
found!');

        const { user } = ro.data;

        const validPass = await
bcrypt.compare(credentials.password, user.password);
        if (!validPass) return ReturnObject.fail('Invalid
Password');

        const token = jwt.sign(
            {
                _id: user._id,
                profile: user.profile,
                profileType: user.profileType,
                dateRegistered: user.dateRegistered
            }, process.env.API_SECRET, { expiresIn: 60 * 1000 * 15
} //60 * 60 * 24
        );

        user.lastEntrance = new Date();
        await user.save();

        return ReturnObject.success({ token });
```

```
    } catch (error) {
        console.log('Error Log in: ', error);
        return ReturnObject.fail(error);
    }
};

exports.authenticateByRequestHeaders = async (token) => {
    try {
        const ro = await
jwtAuthenticatorService.retriveUser(token);

        if (ro.errors) return ro;

        const { user } = ro.data;

        user.lastEntrance = new Date();
        await user.save();

        console.log('Authenticated by request header');
        return ReturnObject.success({ token });
    } catch (error) {
        console.log('Authenticate by request header error: ',
error);
        return ReturnObject.fail(error);
    }
};

exports.setAuthToken = async (res, token) => {
    try {
        res.cookie('authToken', token, {
            expires: new Date(Date.now() + 60 * 1000 * 15),
            httpOnly: true
        });
    } catch (error) {
        console.log('Set auth token error: ', error);
    }
};

exports.removeAuthToken = async (res) => {
    try {
        res.cookie('authToken', '', { expires: new
Date(Date.now()) });
    } catch (error) {
        console.log('Remove auth token error: ', error);
    }
};
```

Файл models/UserAccount.js

```
const mongoose = require('mongoose');
```

```
const AccountEnum = require('../constants/models/AccountEnum');

const UserAccountSchema = new mongoose.Schema({
    email: {
        type: String,
        required: true,
        unique: true,
        lowercase: true
    },
    emailVerified: {
        type: Boolean,
        default: false
    },
    password: {
        type: String,
        required: true,
        minLength: 7
    },
    profile: {
        type: mongoose.Schema.Types.ObjectId,
        required: true,
        refPath: 'profileType'
    },
    profileType: {
        type: String,
        required: true,
        enum: [AccountEnum.STUDENT, AccountEnum.TUTOR]
    },
    lastEntrance: { type: Date, default: Date.now },
    dateRegistered: { type: Date, default: Date.now }
});

const UserAccount = mongoose.model(AccountEnum.USER_ACCOUNT,
UserAccountSchema);

module.exports = { UserAccount, UserAccountSchema };
```

Файл models/Student.js

```
const mongoose = require('mongoose');
const AccountEnum = require('../constants/models/AccountEnum');

const Schema = mongoose.Schema;

const StudentSchema = new Schema({
    firstName: {
        type: String,
        default: '',
        trim: true
    },
    lastName: {
```

```
        type: String,
        default: '',
        trim: true
    },
    skype: {
        type: String,
        default: '',
        trim: true
    },
    phone: {
        type: String,
        default: '',
    },
    balance: {
        type: Number,
        default: 0
    },
    profileImage: {
        type: String,
        default: ''
    },
    lessons: [{
        tutorId: { type: mongoose.Schema.Types.ObjectId, ref:
AccountEnum.TUTOR, required: true },
        subjectName: { type: String, required: true },
        date: { type: Date, required: true }
    }],
    isActivated: { type: Boolean, default: false },
    dateRegistered: { type: Date, default: Date.now }
});

// - country of origin
//  - phone
//  - skype id
//  - image

const Student = mongoose.model(AccountEnum.STUDENT, StudentSchema)

module.exports = { Student };
```