Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

РОЗРОБКА МЕНЕДЖЕРУ ПАРОЛІВ ДЛЯ БРАУЗЕРУ CHROME

Текстова частина до курсової роботи

за спеціальністю «Інженерія програмного забезпечення» - 121

Керівник курсової роботи

д.т.н., доц. Глибовець А.М.

_____

(підпис)

«____» _____ 2021 р.

Виконав студент БП ІПЗ-3

Кучменко Я.О.

_____

(підпис)

«____» _____ 2021 р.

Київ-2021

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

ЗАТВЕРДЖУЮ

_____ *(прізвище та ініціали)*

_____ *(підпис)*

«____» _____ 2020 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студену Кучменку Ярославу Олеговичу

факультету інформатики 3 р.н. бакалаврської програми

ТЕМА: Розробка менеджеру паролів для браузеру Chrome

Зміст ТЧ до курсової роботи:

Календарний план

Вступ

Розділ 1. Аналіз предметної області. Постановка завдання

Розділ 2. Теоретичні відомості

Розділ 3. Опис реалізації застосунку

Висновки

Список використаної літератури

Додатки (за необхідністю)

Дата видачі «____» _____ 2020 р. Керівник_____*(підпис)*

Завдання отримав _____*(підпис)*

Тема: Розробка менеджеру паролів для браузеру Chrome

Календарний план виконання роботи:

№	Назва етапу	Термін виконання	Примітка
1.	Отримання теми курсової роботи	23.09.2020	
2.	Пошук літератури за темою роботи	12.10.2020	
3.	Огляд літератури за темою роботи	08.11.2020	
4.	Огляд аналогічного програмного забезпечення	22.11.2020	
5.	Аналіз необхідних вимог щодо безпеки та дизайну взаємодії з користувачем (англ. – “UX”) клієнт-серверного застосунку	27.12.2020	
6.	Обрання програмного каркасу (англ. – “framework”) для написання серверної частини застосунку	27.12.2020	
7.	Налаштування середовища розробки. Організація роботи з системою керування версій. Налагодження роботи зі сховищами даних на віддаленому сервері.	16.02.2021	
8.	Написання серверної частини застосунку	15.04.2021	
9.	Написання клієнтської частини застосунку	02.05.2021	
10.	Написання текстової частини курсової роботи	16.05.2021	
11.	Перегляд вмісту курсової роботи керівником	17.05.2021	
12.	Внесення змін до текстової частин відповідно за зауважень наукового керівника	17.05.2021	

13.	Створення презентації	17.05.2021	
14.	Подання роботи для перевірки на плагіат	17.05.2021	
15.	Захист роботи		

Студент Кучменко Я.О.

Керівник Глибовець А.М.

«___» _____

ЗМІСТ

АНОТАЦІЯ

У роботі було зосереджено увагу на особливостях розробки клієнт-серверного застосунку менеджеру паролів. Клієнтською частиною слугуватиме розширення для вебпереглядача Google Chrome.

Основну увагу було приділено безпеці зберігання й передачі інформації, а також авторизованому доступу до ресурсів.

Розроблене програмне рішення демонструє тей мінімум вимог, які зобов’язані бути виконаними при розробці цього типу застосунків.

Ключові слова: клієнт-серверний застосунок, менеджер паролів, безпека застосунку, зберігання чутливої інформації, розширення для браузеру, Kotlin розробка

ПЕРЕЛІК ПРИЙНЯТИХ ТЕРМІНІВ І СКОРОЧЕНЬ

БД – база даних.

КСЗ – клієнт-серверний застосунок.

ПЗ – програмне забезпечення.

СКБД – система керування базами даних

API – англ. ”Application Programming Interface”.

CORS – англ. ”Cross-Origin Resource Sharing”.

CSRF – англ. ”Cross-Site Request Forgery”.

HTML – англ. “HyperText Markup Language”.

HTTP – англ. ”Hypertext Transfer Protocol”.

HTTPS – англ. ”Hypertext Transfer Protocol Secure”.

ID – англ. “Identifier”.

JSON – англ. ”JavaScript Object Notation”.

JWS – англ. “JSON Web Signature”.

JWT – англ. “JSON Web Tokens”.

NoSQL – англ. “Non relational”.

SQL – англ. ”Structured Query Language”.

TOTP – англ. ”Time-Based One-Time Password Algorithm”.

URL – англ. ”Uniform Resource Locator”.

URI – англ. ”Uniform Resource Identifier”.

XSS – англ. ”Cross-Site Scripting”.

ВСТУП

Актуальність та практичне значення обраної теми

У сучасному світі паролі відіграють дуже важливу роль і виступають у більшості випадків обов'язковою ланкою при надаванні доступу користувачеві до захищених даних [1]. Можливою альтернативою слугує використання біометричних даних, яке накладає свої обмеження на апаратну та програмні платформи, та проект FIDO2, проте підтримка його стандартів тільки нещодавно почала впроваджуватись. [2][3]

Для подальшого розгляду варто пояснити декілька фактів:

- Якщо один і той же пароль використовується користувачем на декількох ресурсах, то зловмисник, дізнавшись його значення, потенційно отримує доступ до всіх облікових записів жертви, які використовували дане гасло (англ. "password").
- Ентропія паролю - це математична міра складності та непередбачуваності паролю, а саме: його ефективність проти атак перебірного (англ. "brute-force"), словникового (англ. "dictionary") або іншого типу. [4] Вимірюється у бітах та обчислюється за формулою

$$H = \log_2 b^l$$

де b – кількість дозволених для використання унікальних символів;

l – довжина паролю.

У дослідженні Microsoft Research за 2007-й рік [5] було яскраво продемонстровано ставлення середньостатистичних користувачів Інтернету до паролів декаду тому:

- У середньому особа використовувала всього лише 7 різних паролів, мала 25 облікових записів, захищених паролями, щоденно вводила ~8.11 паролів й один і той же пароль використовувала на 6 різних сайтах.
- Чим частіше пароль повторно використовувався, то тим частіше він мав низьку ентропію (менше 30 біт).
- Дуже малу частину паролів можна було класифікувати як складні, навіть для таких сервісів як PayPal, хоча й ентропія банківських паролів була вищою.
- Значно поширеними були паролі, що складались тільки з літер нижнього регістру.

Якщо подивитись на дані сучасніших оглядів (2010-2020 роки) [6][7][8][9], то можемо побачити позитивну динаміку: користувачі почали серйозніше ставитись до питань їх безпеки в мережі Інтернет та з плином часу стали надавати перевагу використанню складних для запам’ятовування паролів, а не зручним легким, які часто опиняються серед тих, які повторно використовують на різних ресурсах. У середньому рівень виконання рекомендацій щодо забезпечення належного захисту персональних даних (у тому числі і звичок при використанні паролів) значно зріс, проте є великий прошарок людей (52% за дослідженням в 2018 році)[9], котрий продовжує практику частого використання одних і тих самих паролів на некритичних ресурсах.

За останню декаду здобули популярність різноманітні менеджери паролів, які існують як альтернатива записуванню на папір паролів, котрі стали довгими та почали містити різноманітні неалфавітні символи та літери верхнього регістру. Дані програми допомагають виживати у світі, де у людини може бути лише на одній пошті по 130 різних облікових записів [6].

Очевидна проблема, з якою борються будь-які розробники даних сервісів, – довіра користувача. Оскільки менеджери паролів зберігають дуже чутливу інформацію, то вони повинні мати міцну архітектуру, спрямовану на унеможливлення потрапляння даних користувачів до сторонніх осіб. Достатньо зробити одну помилку в певному компоненті архітектури, і наслідки можуть

завдати серйозного удару по репутації сервісу. [10] До того ж, для користувачів є велика небезпека втратити доступ до усіх своїх ресурсів, якщо значення їх зашифрованих даних можна буде з легкістю отримати зловмисникам, бо велика кількість паролів буде зберігатись в одному місці.

З огляду на те, що попит на дане програмне забезпечення є доволі високим і стрімко зростає [11], ця робота має на меті встановити, які основні моменти повинні бути враховані при розробці менеджера паролів, аби мінімізувати ризики використання даного застосунку для кінцевих користувачів.

Для клієнтської частини буде обрано веб-розширення для браузера Google Chrome через декілька причин:

- З даних компанії StatCounter станом на квітень 2021-го року [12] маємо, що Google Chrome займає 67.53 % ринку персональних комп'ютерів. До того ж, компанія Google має проєкт з відкритим кодом Chromium, на який спирається чимало інших браузерів: з 2019-го року вебпереглядач Microsoft Edge (7.96% ринку) повністю перейшов на дану кодову базу [13], а браузер Opera (2.65% ринку) ще аж в 2013-го перейшов на нього [14]. Для нас це означає, що дані браузери також можуть встановити застосунок розроблений для Google Chrome. [15] Сумарно маємо 75.08% потенційного охоплення ринку.
- У абсолютної більшості менеджерів паролів присутня реалізація розширення для браузерів [16].

Структура роботи

Робота складається з трьох розділів.

У першому розділі здійснюється огляд предметної області, аналіз аналогічних застосунків на ринку та різновиди існуючих менеджерів паролів в цілому. Також буде з'ясований перелік необхідних обов'язкових вимог для продукту разом із особливостями роботи даного ПЗ.

У другому здійснено теоретичний огляд реалізації системи авторизації та збереження користувацьких даних, а також особливостей розробки розширень та типових вразливостей даних застосунків.

У третьому описано деталі розробки й роботи отриманого КСЗ. Проблеми, що виникали під час розробки програмного застосунку, та прийняті дії щодо їх вирішення теж будуть оглянуті в даному розділі разом із обґрунтуванням архітектури та особливостей.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ. ПОСТАНОВКА ЗАВДАННЯ

1.1 Базовий огляд предметної області

Основне призначення менеджеру паролів – забезпечення гарного користувацького досвіду при взаємодії з даними для входу до облікових засобів, забезпечуючи при цьому надійне та безпечне зберігання паролів. [17]

Кореневий функціонал [18]:

- створення записів (тек), що містять: посилання або назву ресурсу, ім’я, що використовується для входу до облікового запису та відповідний пароль;
- записи з легкістю підлягають редагуванню та вилученню;
- автоматичне створення паролів за певними параметрами щодо їх складності;
- всі дані зберігаються в зашифрованому головним (англ. – “master”) паролем вигляді;
- наявна підтримка хоча б одного виду багатофакторної автентифікації.

Автентифікація – це процес ідентифікації особи за певним набором даних, що вона надає системі. За умови, що передані дані клієнтом збігаються із записом у системі, то даний процес вважають успішно виконаним. [19]

Багатофакторна автентифікація – автентифікація, що відбувається за поєднання декількох факторів ідентифікації [20]:

- Щось, що особа знає (пароль, короткий числовий код (англ – “PIN”))
- Щось, що особа має (TOTP [21], секрет прихований у файлі або девайсі, фізична картка, FIDO U2F ключ)
- Щось, чим особа є (біометричні дані: відбиток пальця, сітківка ока, фото обличчя)

Менеджери паролів є завжди вразливими до атак, що відбуваються внаслідок інфікування кінцевого пристрою шкідливим ПЗ, яке може зчитувати дані з будь-яких ділянок оперативної пам’яті [22] або відправляти інформацію отриману з буферу обміну чи натиснення клавіш. Звідси слідує, що користувачів потрібно попереджати про те, що, наприклад, найкритичніші паролі не варто зберігати в даному застосунку, бо повністю завадити цьому не можливо на рівні застосунку. [23]

Варто ще зазначити, що існує безліч різних векторів атак, тому застосунки даного типу необхідно постійно підтримувати й оперативно прибирати знайдені вразливості. Далі в роботі буде розглянуто, як найпопулярніші атаки діють.

1.2 Аналіз існуючих аналогів предметної області на ринку

Серед сучасних аналогів на ринку можна виділити дані продукти: 1Password, Bitwarden, NordPass, Zoho Vault, Lastpass, KeePass, Keeper, EnPass, iCloud Keychain, Dashlane, RoboForm та ще багато інших.

Їх функціонал є дуже схожим між з собою і він забезпечується на усіх платформах застосунків, підтримка яких дуже широка.

Розглянемо можливості популярного розширення менеджеру паролю на прикладі сервісу Bitwarden.

При реєстрації (рисунок 1.1) необхідно надати адресу електронної пошти та ввести пароль, який буде використовуватись для входу. Bitwarden вимагає мінімальну довжину паролю в 8 символів й повідомляє користувача якщо він використовує криптографічно прості паролі (використання лише символів з латиниці, поєднання слів, що містяться в словнику, тощо). Деякі аналоги, такі як LastPass, не дають можливості створити користувачеві простий для вгадування пароль, в той же час Bitwarden просто виводить повідомлення про те що відбувається спроба створити обліковий запис із ненадійним паролем (рисунок 1.2).

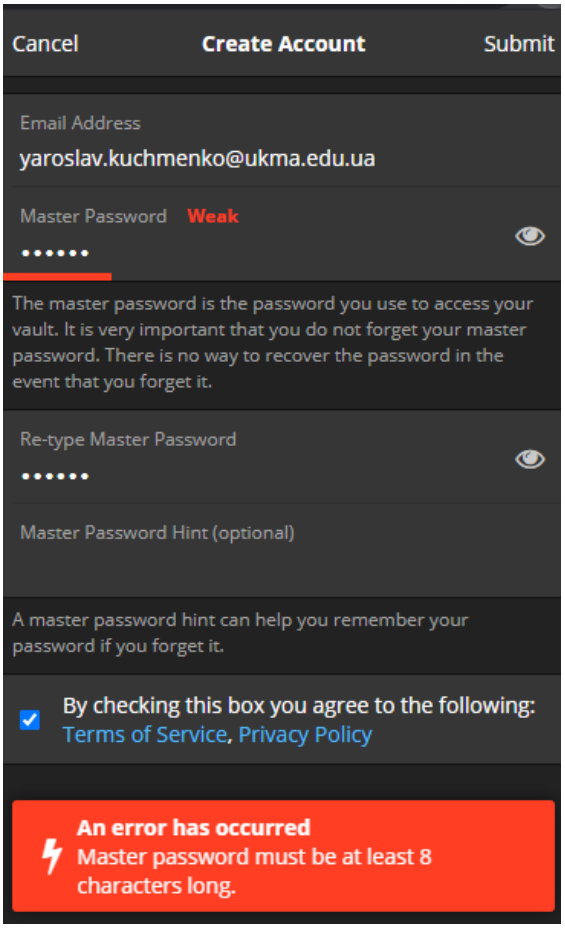

Рисунок 1.1 – Bitwarden. Екран реєстрації.. Пароль менше восьми символів

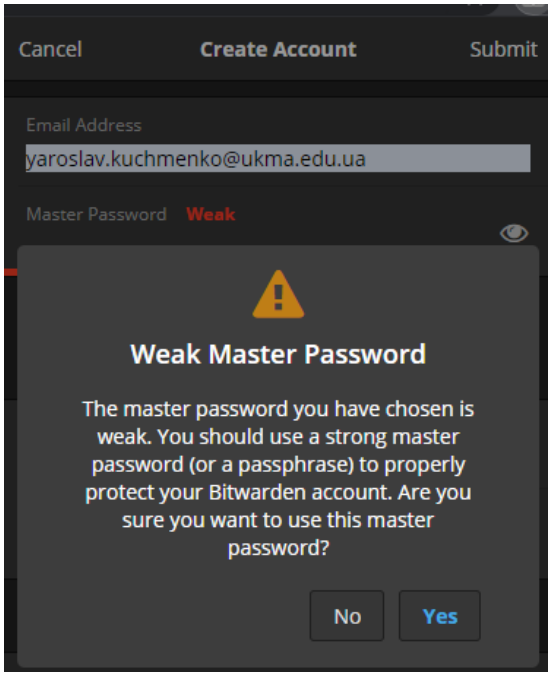

Рисунок 1.2 – Bitwarden. Процес реєстрації. Пароль криптографічно слабкий

При автентифікації (рисунок 1.3), за успішного введення коректної пари імені користувача, паролю та проходження додаткового етапу двоетапної перевірки (залежить від налаштувань облікового запису), користувач отримує від серверу своє захищене сховище. Компанія визначає, що жодна інформація пов'язана з паролями не в зашифрованому вигляді не покидає клієнт [24].

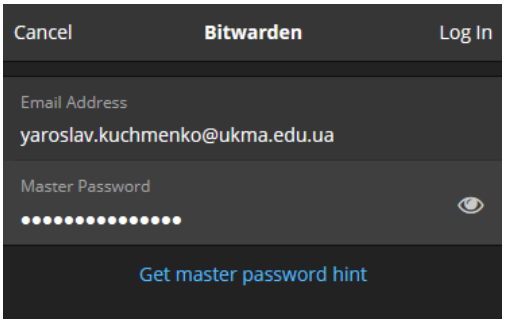

Рисунок 1.3 – Bitwarden. Процес автентифікації

Серед функціоналу, що відразу впадає в око для уже авторизованого користувача можна виділити зберігання даних входу по окремим текам (рисунок 1.4). Кожна з тек надає можливість до редагування паролю, перегляду асоціацій запису даного з URL сайтами (для коректної роботи пропонування заповнення даних на вебсторінках) (рисунок 1.5).

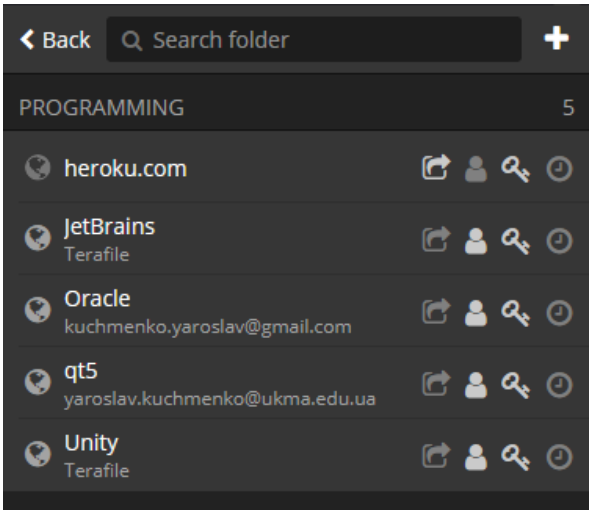

Рисунок 1.4 – Bitwarden. Теки даних для даних входу

Генерувати паролі з вказання великої кількості параметрів теж є можливість (рисунок 1.6).

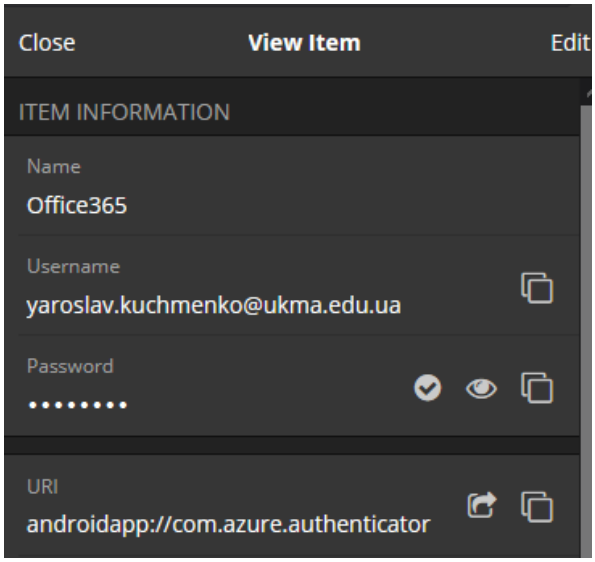

Рисунок 1.5 – Bitwarden. Вміст тек з даними для входу

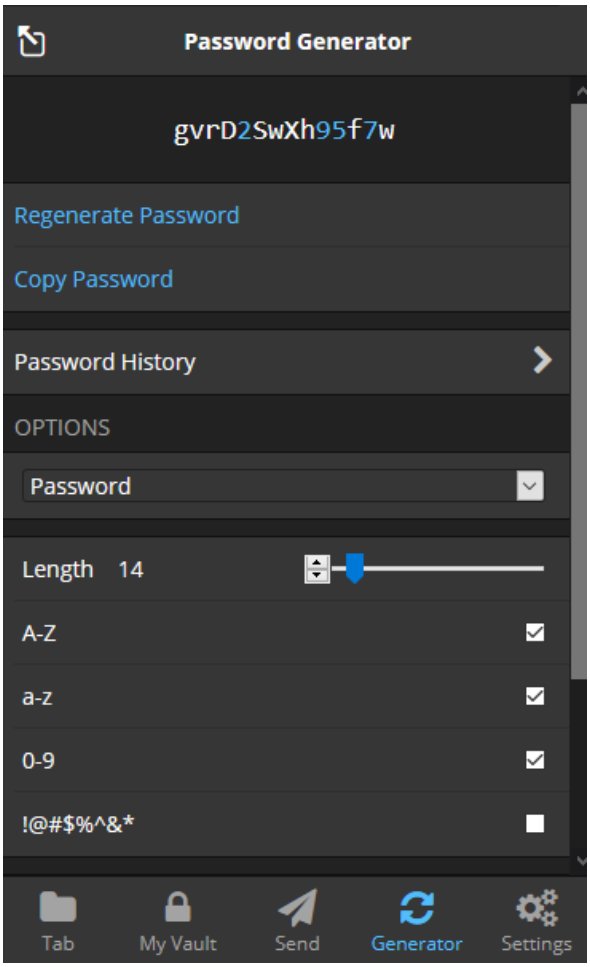

Рисунок 1.6 – Bitwarden. Генератор паролів

Варто зауважити, що користувач не зобов'язаний вводити пароль для кожної дії застосунку. Після закриття браузеру або зі спливанням часу,

визначеного в налаштуваннях автоблокування, необхідно буде заново вводити дані входу (рисунок 1.3).

1.3 Постановка завдання

Враховуючи все попередньо сказане, можемо виокремити такі завдання для даної курсової роботи:

- написати клієнт-серверний застосунок, який міститиме відповідний даному типові програмного забезпечення вище згаданий функціонал;
- клієнтською частиною зробити найпопулярніший формат менеджерів паролів, а саме: розширення для браузеру Google Chrome;
- приділити особливу увагу в проекті на захист від популярних та небезпечних вразливостей;
- обґрунтувати набір використаних технологій та архітектурних рішень;
- зробити висновки щодо отриманого застосунку.

РОЗДІЛ 2. ТЕОРЕТИЧНІ ВІДОМОСТІ

2.1 Розповсюджені типи атак для КСЗ даної предметної області

Поставленою метою щодо наступних розглянутих атак є знайомство з можливими вразливостями, на які потрібно зважати під час розробки менеджерів паролів. Саме такий перелік було обрано, оскільки вразливості до згаданих атак найбільш часто згадуються у оглядах аналітиків з кібербезпеки [25][26][27][28].

У зв’язку з тим, що способи захисту від цих атак залежать від конкретного застосування та багатьох інших факторів [28], вони не будуть розглянутими. Для найповнішої картини щодо усунення потенційних вразливостей програмного забезпечення варто звертатись до спеціалістів з даної сфери, наприклад OWASP [29].

2.1.1 XSS injection

Cross-Site Scripting (XSS) Injection – це атака, яка спрямована на вставку та виконання клієнтських скриптів з ворожими намірами у вебпереглядач користувача, коли він відвідує атакований вебсайт. [30]

Найчастіше веб-сервіси виявляються ураженими XSS атакою внаслідок того, що зберігають та відображають будь-яку інформацію, яку сервер отримує від користувача (відсутня фільтрація даних при їх обробці) [31].

Оскільки браузер не має можливості визначити, що певний скрипт несе небезпеку, то будь-який вставлений зловмисницький код (HTML, JavaScript, Flash, тощо) має тей самий доступ до браузерного середовища, як і будь-який інший “безпечний”. [31] Наприклад, JavaScript матиме доступ до вмісту cookies, local або session storage, значень полів форм, тощо.

Виділяють два основні види [31]:

- Stored Attacks - ворожий скрипт зберігається на серверах або базі даних і доступний усім відвідувачам уражених вебсторінок. Blind Scripting його підвид: ворожий скрипт передається на сервер, але отримати доступ до сторінки або даних вражених ним може тільки адміністрація сайту.
- Reflected Attacks – користувач переходить за посиланням, яке міститься на певній веб-сторінці або на пошті, й виконується ворожий скрипт. Можливе за умови, якщо вебсервер очікує за якимось шляхом передачу значення параметру в GET запиті, наприклад запит на пошук.

2.1.2 SQL та NoSQL injection

SQL та NoSQL Injection – дуже небезпечні атаки на бази даних, що полягають у неочікуваному виконані запитів, які було передано на сервер через клієнтський застосунок.

Шляхи отримання даної вразливості: дозвіл стороннім особам та застосункам надавати можливість доступу до бази даних; для побудови SQL запитів на стороні серверу використовується динамічне їх створення, наприклад конкатенація стрічок, а не раннє компілювання запитів; [32] відсутність фільтрації даних переданих на сервер користувачем; використання неактуальних версій NoSQL баз даних [33].

Потенційно без впровадженого захисту за допомогою цього способу можна отримати повний доступ до бази даних, а саме: мати можливість змінити стан вмісту баз даних (CRUD операції), відновити дані, видалити вміст усієї БД, зупинити її роботу, тощо. [32][34]

Як зазначає OWASP [32] – найбільш частими жертвами SQL виявляються старі застосунки написані на PHP та ASP, в той же час більш сучасні ORM інтерфейси та бібліотеки мають вбудований захист.

Багато хто помилково вважає, що NoSQL бази даних не вразливі до даних атак, хоча вони можуть навіть змусити виконати певні маніпуляції на сервері, на відміну від SQL Injections, де може постраждати тільки база даних. [36]

2.1.3 CSRF

Cross-Site Request Forgery (CSRF) – це атака, яка спрямована на примус користувачів до виконання небажаних дій на вже автентифікованих вебсайтах шляхом соціальної інженерії. Чим вищий рівень авторизації жертви, тим серйознішими можуть стати результати атаки. [37]

Атака буде дієвою на всіх веб-сервісах, що використовують зберігання клієнтських даних авторизації, а також потенційно дозволяє зловмисникам здійснити зміну стану облікового запису особи, зманіпулювати токенами доступу, перевести кошти. Варто зазначити, що зловмисник не зможе, наприклад, змусити людину надати якусь конфіденційну інформацію сторонній особі, оскільки ця атака дозволяє лише виконати строго визначені розробниками вебзастосунку дії. [37]

CSRF може поєднуватись з XSS для того, аби залишити на вебсайті певний елемент, клік на який виконає зловмисницьку дію.

2.1.4 Brute-force

Brute Force Attack працює шляхом перебору можливих комбінацій даних, допоки не буде отримано успіх або зібрано їх множину. [38]

Може бути спрямованою для визначення значення зашифрованих даних, підбору даних входу, пошуку прихованих сторінок, портів тощо.

Зазвичай зловмисниками впроваджуються певні дії для підвищення ефективності підбору, оскільки повний перебір усіх можливих комбінацій в більшості випадків не можна буде фізично здійснити. Наприклад, для підбору паролів використовують Rainbow-таблиці.[39]

2.1.5 Denial-of-service

Denial-of-service (DOS) – тип атак, які спрямовані на припинення можливості застосунків або серверів до обслуговування будь-яких запитів. [40] Способи досягнення цієї мети можуть різнитись від використання вразливостей в програмного та апаратному забезпеченні до простого великого напливу нових модифікованих ICMP-пакетів (пінгування (англ. “ping”) кожного пристрою в системі) [41] , SYN - запитів (сервер не зможе дочекатись TCP рукостискання (англ. “handshake”) [42] або звичайних звернень до API за одиницю часу.

Distributed Denial-of-service (DDOS) – Denial-of-service атака, яка здійснюється з величезної кількості джерел, що перешкоджає оперативному блокуванню доступу до ресурсу для усіх учасників атаки. Велика кількість пристроїв, котрі атакують ресурс, забезпечується використанням інфікованих комп’ютерів певним шкідливим ПЗ. [43]

2.1.6 Man-in-the-middle

Manipulator-in-the-middle (MITM) – це атака, за якої третя особа втручається в спілкування будь-яких двох систем. Вектор атаки полягає в розриванні TCP-з’єднанні на два нових, між клієнтом-1 та зловмисником й між зловмисником та клієнтом-2. Таким чином третя особа виступає посередником, через якого будуть проходити пакети. [44]

До цієї атаки є дуже вразливим популярний HTTP-протокол, який здійснює все спілкування у відкритому до читання форматі (використовується кодування даних в ASCII). Зловмисник перехоплює пакет, прочитує його вміст. До справжнього отримувача пакет може дійти вже зі зміненим вмістом або взагалі не дійти. [44]

Використання TLS-протоколів у з’єднанні HTTPS не може повністю захистити від цієї атаки, оскільки за використання того ж процесу розривання

TCP-пакетів можна просто відкрити 2 захищені з’єднання між учасниками взаємодії, проте в такому випадку браузер користувача має можливість повідомити його про використання недійсного сертифікату захисту з’єднання. [45]

OWASP пропонує використання згаданої атаки під час розробки для вивчення рівня вразливостей, які має застосунок. [44]

2.1.7 Session Prediction та Session Fixation

Session Prediction – атака, спрямована на передбачення значення ідентифікатора сесії, яке надасть зловмисникові авторизований доступ до вебресурсу. Полягає в аналізі згенерованих токенів сесії, їх структури, методів їх компонування та захисту. Часто комбінується з Brute-force типом атаки. [46]

Session Fixation – атака, яка покладається на вразливість в застосунках, коли вони багаторазово використовують одне і те ж значення сесії для автентифікації користувача. [47]

Для успіху такої атаки зловмисник спочатку повинен отримати дійсний ідентифікатор користувацької сесії. Отримання його залежить від конкретної реалізації на сервері, проте найчастіше ідентифікатор сесії міститься або в частині URL сайту, або в cookie, або в прихованому полі HTML-форми. Отримавши потрібне значення, зловмисник доступними йому методами (наприклад, соціальна інженерія або XSS ін’єкції) змушує користувача автентифікуватися за цим ідентифікатором сесії. Далі сесії вважається авторизованою сервером. Таким чином, зловмисник отримує доступ до даних користувача. [47]

2.1.8 Autofill sweep атака

Sweep-атака є мало розповсюдженою проте дуже небезпечною. Непопулярність пов’язана з тим, що дану атаку можна здійснити тільки за умови

наявного функціоналу автозаповнення даних для входу, котрий завжди присутній у менеджерах паролів. [48]

Одним із поширених способів розгортання цього виду атаки полягає в тому, що зловмисникові потрібно налаштувати Man-in-the-Middle-атаку з можливістю модифікувати трафік, що надходить від користувача та Wi-Fi маршрутизатора. Далі користувач повинен перейти в браузер, де його буде перенаправлено на звичайнісіньку вітальну сторінку (англ. “landing page”) від Wi-Fi локальної мережі, проте вона міститиме невидимі елементи, які присутні там для здійснення крадіжки паролів. Отримавши необхідні дані, вони непомітно для користувача можуть бути відправленими на сторонній ресурс (невидимі iFrame, модифікації логін-форм). [48]

Самі паролі можу добуватись одним з трьох способів [48]:

1. iFrame sweep

   Вітальна сторінка повинна містити невидимі iFrame сторінки, дані входу для яких зловмисник хоче отримати. Після завантаження сторінки відбувається ін’єкція форми входу в кожен з iFrame-ів. Активний менеджер паролів з увімкненою функцією автозаповнення відреагує на дані форми, заповнивши їх даними входу до відповідних сайтів.
2. Window sweep

   Схожа на попередню, проте замість iFrame відкриваються нові вікна, які клієнтський ін’єктований JavaScript може спробувати приховати (розташувати у кутку екрана).
3. Redirect sweep

   Використовується серія з перенаправлень з ін’єктованими скриптами від зловмисника. Принцип дії той самий.

Варто зазначити, що переходу користувача з сайту, котрий було вражено XSS-атакою, на вебсторінку зловмисника достатньо для того, щоб можна було перехопити паролі за вищезазначених методик.

Таким чином, маємо справу з дуже небезпечною вразливістю, котра була виявлена в 2014 році, і досі не було створено дієвого захисту, окрім як прибирання автозаповнення даних без явної взаємодії користувачів [48].

2.2 Збереження та транспортування чутливої інформації

2.2.1 Хеш-функції та функції формування ключів

В програмуванні часто оперують хеш-функціями різного типу, користуючись її особливістю – неможливістю швидкого перетворення від отриманого хеш-значення до вихідного. В криптографії дана їх особливість користується особливим попитом у зберіганні паролів. [49]

Одною з характеристик хеш-функції є ймовірність колізій. Колізіями прийнято називати ситуації, якщо одному значення хеш-функції може відповідати дві або більше стрічки даних, подані на вхід. Для криптографії виникнення даної ситуації є надзвичайно небажаним. [50]

Криптографічними хеш-функції (CHF) називають, якщо вони мають надзвичайно малу ймовірність колізії та можуть перетворювати будь-який набір даних довільної довжини в результат фіксованого розміру, який вимірюються у бітах (рисунок 2.1). [50] Розмір отриманого хешу варіюється в межах від 128 до 512 біт. Чим більша його розмір, то тим очікується менша вірогідність колізії.

Ідеальною криптографічною хеш-функцією прийнято вважати, ту [49]:

- для якої не існує ситуації, що отриманий результат після застосування даної функції можна буде надати знову в неї ж і досягти початкового значення;
- колізії практично неможливо отримати;

- одні і ті ж дані на вхід завжди приводять до збігу отриманих значень;
- зміна одного символу призводить до радикальної зміни отриманого хеша;
- значення повинно швидко обраховуватись.

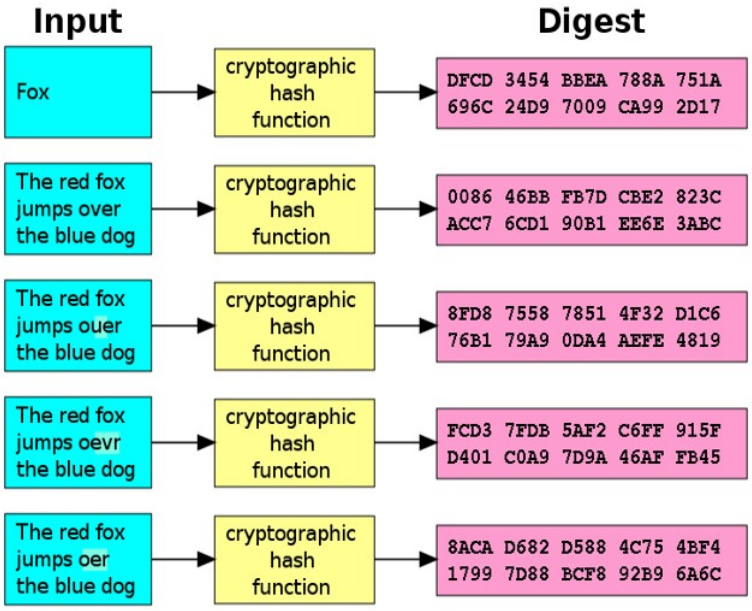

Рисунок 2.1 – Демонстрація роботи CHF [49]

CHF використовують для криптовалют, які базуються на Proof-of-work криптовалютних алгоритмах [51], для зберігання паролів, генерування унікальних ідентифікаторів для документів, пошти або програм [52].

До хеш-функцій, які визнано такими, що містять достатньо несприйнятливу кількість колізій та не можуть вважатись безпечними відносять MD5, SHA-0 та SHA-1.[49]

Серед яскравих проблем використання хеш-функцій для зберігання таких даних як паролі в чистому вигляді було проведено дослідження швидкості підбору стрічкового значення звичайною Brute-Force атакою для популярних раніше і нині хеш-функцій: MD5 та SHA-256.

Для демонстрації швидкості підбору значенням атакою Brute Force Було проведено наступний дослід.

Умови проведення:

- ОС: Windows 10
- Процесор: Intel i5-10400
- Відеокарта: AMD RX 580 4GB (20.4.1 драйвер)
- Програма: hashcat 6.1.1 версії

Таблиця 2.1 – Результати досліду

Стрічка на вході \ витрачений час	MD5	SHA-256
“1pp21c” (шість символів)	3 с	7 с
“aadddaa” (сім символів)	19 с	205 с

```
2c61509ef9b9aed1f11480893e10c616:aadddaa

Session..........: hashcat
Status...........: Cracked
Hash.Name........: MD5
Hash.Target......: 2c61509ef9b9aed1f11480893e10c616
Time.Started.....: Sun May 09 17:05:45 2021 (15 secs)
Time.Estimated...: Sun May 09 17:06:00 2021 (0 secs)
Guess.Mask.......: ?1?2?2?2?2?2?2 [7]
Guess.Charset....: -1 ?l?d?u, -2 ?l?d, -3 ?l?d*!$@_, -4 Undefined
Guess.Queue......: 7/15 (46.67%)
Speed.#2.........:  5316.6 MH/s (6.94ms) @ Accel:128 Loops:128 Thr:64 Vec:1
Recovered........: 1/1 (100.00%) Digests
Progress.........: 76563873792/134960504832 (56.73%)
Rejected.........: 0/76563873792 (0.00%)
Restore.Point....: 884736/1679616 (52.67%)
Restore.Sub.#2...: Salt:0 Amplifier:18432-18560 Iteration:0-128
Candidates.#2....: nclehx8 -> wtsqtuo
Hardware.Mon.#2..: Util:  1% Core:1411MHz Mem:1750MHz Bus:16

Started: Sun May 09 17:05:42 2021
Stopped: Sun May 09 17:06:01 2021
```

Рисунок 2.2 – Hashcat 6.1.1 - знаходження переданого значення (стрічка “aadddaa”) в MD5 функцію

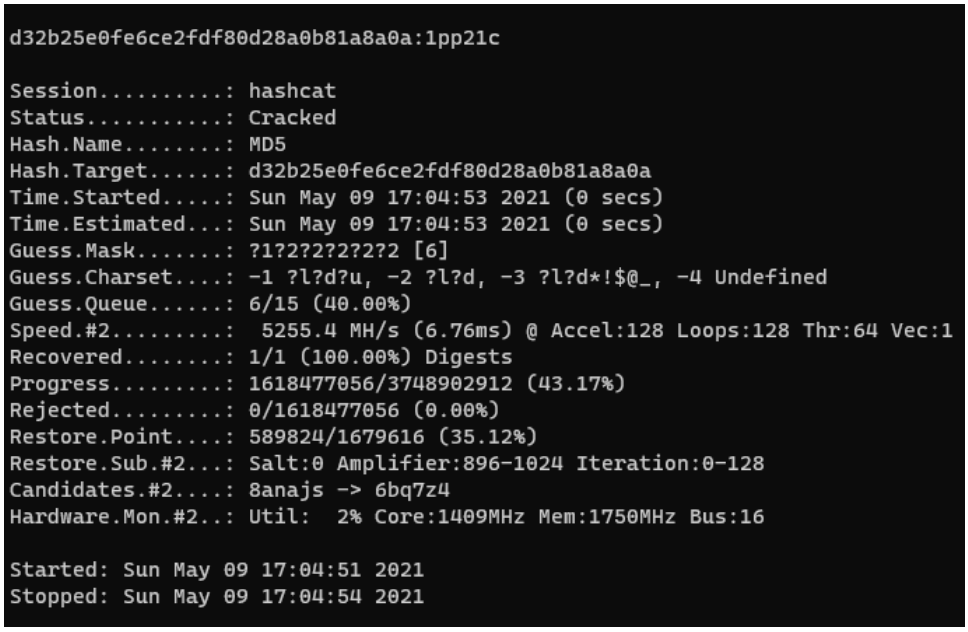

Рисунок 2.3 – Hashcat 6.1.1 - знаходження переданого значення (стрічка "1pp21c") в MD5 функцію

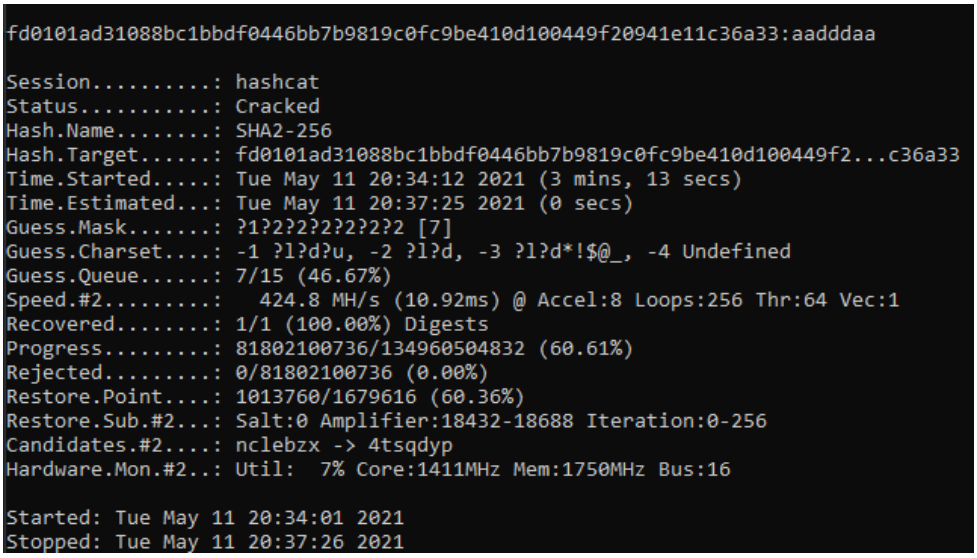

Рисунок 2.4 – Hashcat 6.1.1 - знаходження переданого значення (стрічка "aadddaa") в SHA-256 функцію

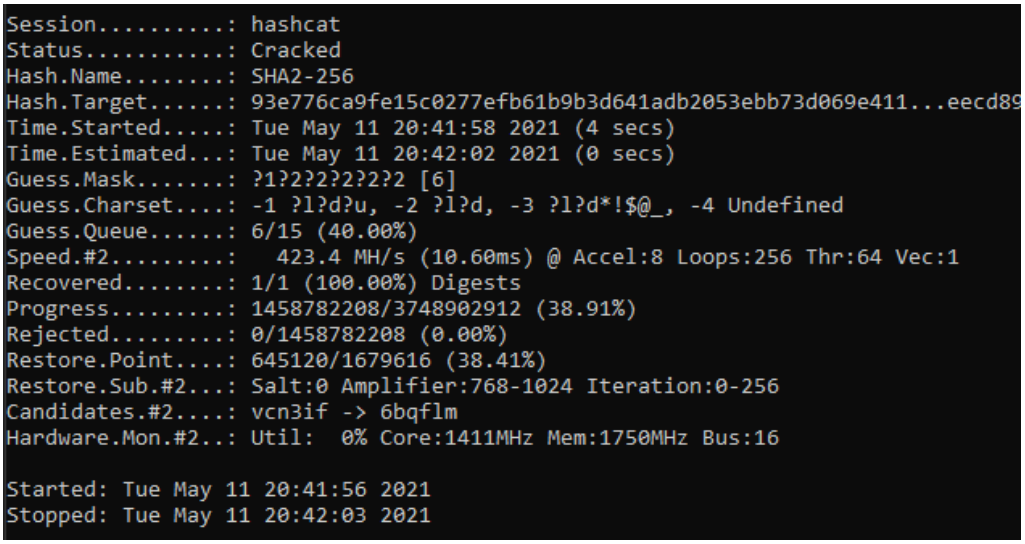

Рисунок 2.5 – Hashcat 6.1.1 - знаходження переданого значення (стрічка "1pp21c") в SHA-256 функцію

Отже, з вище наданих результатів досліду (рисунки 2.2 – 2.5) можна зробити висновок, що навіть без використання спеціалізованого обладнання та підходів, можна було отримати значення, яке було вкладене в хеш за лічені секунди. Також можна побачити, що швидкість знаходження значення напряму залежить від довжини стрічки, яка була передана в хеш-функцію (особливості виконання методу Brute-Force).

Паролі користувачів у своїй більшості – це значення з низькою ентропією, як було показано у вступі, то потрібно це враховувати в провадженні системи збережень бази паролів користувачів. Перший метод, який пропонується для вирішення цієї проблеми – це використання випадково згенерованої стрічки - "солі" , яка буде доданою до того значення паролю, що передається у функцію утворення хешу, проте містить низку недоліків, що були вказані в багатьох статтях та дослідженнях. [51][52] Натомість пропонується використання функцій спеціально створених для використання у криптографічно безпечних хешів для даних з низькою ентропією – функцій утворення ключа (англ. "key-derivation function") або KDF. [55]

KDF використовують випадково згенеровану сіль в утворення хешу та техніки Key Stretching, які надають даним функціями властивості які найбільш

потрібні в захисті від Brute-force атак, а саме: в рази збільшують вимогу до ресурсів обчислювальної машини для вирахування кожного значення. Параметри складності потрібно регулювати в залежності від сприйнятливої швидкості створення (наприклад 200 мс) та необхідних виділених ресурсів від системи на один отриманий хеш (наприклад необхідність задіяти 30 МБ оперативної пам’яті). [49][55]

Серед популярних KDF виділяють: PBKDF2, BCrypt, Scrypt та Argon2(i,d,id).

Останній, Argon2, де-факто став стандартом в індустрії для збереження критичного важливих паролів. [56][49] Відрізняється від своїх конкурентів добрим захистом від швидкого здобуття на графічних адаптерах та на спеціалізованому залізі по типу ASIC-ів (англ. “Application-specific integrated circuit”*)*. Містить варіації з різним рівнем захисту від side-channel атак, які задіють вразливості апаратного забезпечення. [56] Організація OWASP рекомендує використовувати нижче наведені в таблиці параметри для даного алгоритму як базовий мінімум [57]:

Таблиця 2.2 – Рекомендовані параметри для Argon2 функції

Навантаження на \ Параметр	Об’єм ОП	Кількість потоків	Паралелізм
ЦП	15 МБ	2	1
ОП	37 МБ	1	1

Отже, використання KDF є ключовим в безпечному транспортуванні та зберіганні ідентифікуючих даних таких як пароль. Розглянутий фаворит Argon2 у варіанті id (баланс між захистом від side-channel атак та brute-force атак за використання відеокарт) [56] [57] варто обирати для написання нових застосунків, які вимагають високі параметри безпеки даних, що були пропущені через хеш-функцію, до яких менеджери паролів і відносять.

2.2.2 Методи шифрування користувацьких даних

Дані користувачів менеджерів паролів не можуть зберігатись у відкритій формі, оскільки складаються із чутливої інформації (дані входу до облікових записів). Шифрування користувацьких даних є рішенням для даної проблеми [49].

В криптографії є два типи методів шифрування: симетричне та асиметричне.

Симетричне шифрування від асиметричного відрізняється тим, що для перетворення із зашифрованого в розшифрований стан здійснюється одним і тим же ключ, на відміну від іншого – використовуються два ключі: один відкритий – для шифрування, а другий приватний – для розшифрування інформації. Використання асиметричної криптографії набуло популярність в технологіях TLS, Blockchain, цифрових сертифікатах та системах [58][59]. В той же час симетричні алгоритми використовуються в системах банкінгу та для зберігання та захисту даних. [49]

Рекомендацією NIST [60], щодо алгоритмів шифрування використовуваних для зберігання будь-якої критично-важливої інформації, є використання алгоритму AES в режимі Cipher block chaining (CBC) та довжиною ключа в 256 біт [62].

AES – це алгоритм симетричного шифрування, котрий вирізняється с поміж інших своїм розповсюдженням, швидкістю та рівнем безпеки, який він пропонує. На вхід приймає блок з даними, ключ фіксованої довжини, отриманий від KDF функції, за допомогою якого буде виконано шифрування або дешифрування. Для AES-256 – цей ключ буде мати довжину в 256 біт. На виході – зашифровані блоки даних по 128 біт кожний (в не залежності від розміру ключа). [61]

Потенційною альтернативою може слугувати Cha-Cha алгоритм, який є показує більшу продуктивність на не вимагає спеціальних апаратних прискорювачів та за попередніми дослідженням [63], показує схожий рівень захисту. Проте він не здобув ще таке розповсюдження та всесвітнє визнання як AES, котрий вважається алгоритмом шифрування військового рівня (англ. “military-grade”) [64].

Отже, при реалізації сховища даних користувача для менеджеру паролів алгоритм AES-256 (CBC варіант) є одним з найкращих варіантів.

2.2.3 Методи безпечного транспортування користувацьких даних

Якщо представити роботу типового КСЗ, що містить механізм синхронізації та автентифікації, то можемо зрозуміти, що засоби підтвердження особи (токени доступу, cookie з даними сесії або інші дані для автентифікації) разом із синхронізованими даними будуть займати переважну частину вхідного та вихідного трафіку клієнта. Звідси маємо, що атака типу Man-in-the-Middle потенційно несе серйозну загрозу потрапляння конфіденційної інформації сторонній особі.

Основним шляхів для вирішення даної проблеми слугує використання TLS при спілкуванні між клієнтом та сервером. [65] TLS – це протокол шифрування трафіку. Для сучасних застосунків рекомендовано використання версії 1.2 та 1.3. Підтримку попередніх версій сервером необхідно заборонити через низку серйозних вразливостей, що були знайдені в них. [66]

Варто ще також зазначити, що TLS необхідно вмикати для абсолютно усіх шляхів веб-серверу або API в незалежності належності від типу інформації який на даному сервері міститься через можливості великого спектру загроз, а для уникнення атаки на вразливість CRIME[67] організація OWASP рекомендує відімкнути TLS компресію. [65]

З огляду на вищеописану інформацію, можна зрозуміти, що будь-який клієнт-серверний застосунок повинен використовувати HTTPS з’єднання з використанням TLS 1.2 або 1.3 протоколів.

2.3 Автентифікація та авторизація користувачів

Протоколом спілкування веб-браузерів із веб-серверами є HTTP[68]. Клієнти спілкуються із заздалегідь визначеними кінцевими точками й таким чином здійснюють свою взаємодію. Вона майже завжди потребує авторизації користувача, який здійснює запит.

Авторизація – процес керування доступу до певного захищеного ресурсу. Відбувається після ідентифікації та автентифікації особи [69].

Серед основних та поширених видів HTTP автентифікації можна виділити дані схеми [70]:

- Basic;
- Digest;
- Bearer.

Іншими особливими способами автентифікації є:

- API ключі;
- OpenID Connect за використання OAuth 2.0.

Розглянемо більш детально вищезгадані способи.

2.3.1 Схеми HTTP автентифікації та авторизації

2.3.1.1 Basic схема автентифікації та авторизації

Basic-схема є найпростішим методом реалізації автентифікації користувача і полягає в тому, що клієнт передає стрічку-ідентифікатор входу

(англ. “username”), пароль користувача та вказує, до якого захищеного простору (англ. “realm”) збирається звернутись. Ці дані будуть запаковані Base64-кодуванням (рисунок 2.6). Сервер повинен пропустити виконання цього запиту тільки за умови, що передані дані для входу збігаються із записом на сервері (додатково можливе попереднє використання хеш-функції або функції знаходження ключа до вказаного паролю, якщо зберігається хеш). За успішної авторизації користувачеві надається доступ до вказаного захищеного ресурсу. [71]

```
// username:password in Base64 == dXNlcm5hbWU6cGFzc3dvcmQ=

Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=
```

Рисунок 2.6 вигляд заголовку Authorization клієнтської відповіді

Відправці даних користувача передує запит на доступ до ресурсу. Далі надходить відповідь-відмова з боку сервера, у якій вказано необхідний тип автентифікації та назва захищеного простору (рисунок 2.7). Тобто у заголовках HTTP-відповіді з кодом 401 матимемо назву метода автентифікації - Basic та захищеного простору - realm. Без цієї інформації клієнт-браузер не міг би знати, як потрібно відреагувати на такий запит. [71]

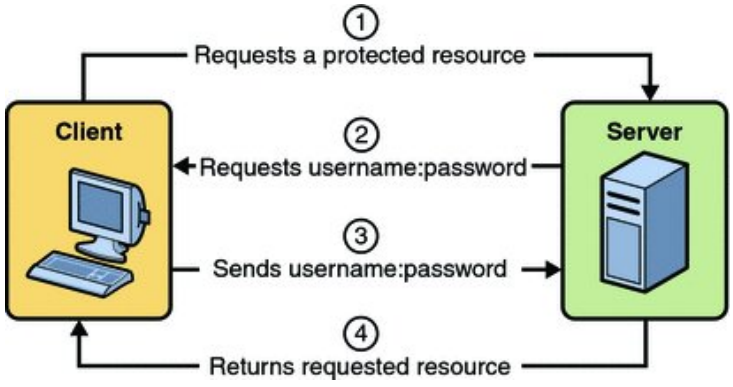

Рисунок 2.7 Візуалізації Basic схеми [72]

Цей спосіб автентифікації дає можливість прибрати необхідність постійно змушувати користувача вводити дані для входу на кожному запиті за рахунок механізмів кешування в браузері (за значенням “realm” визначається належність даних до певного запиту). У той же час унаслідок такої поведінки виникає

проблема із прибиранням даних входу для реалізації функціоналу виходу з системи (англ. “logout”), оскільки стандартного механізму очищення даних з кешу немає. [73]

До суттєвих недоліків слід додати і те, що користувацькі паролі жодним чином не шифруються або перетворюються в хеш. До того ж, значення заголовку Authenticate присутні на кожному зверненні до захищеного ресурсу. Звідси маємо велике вікно для атак, а без використання захищеного каналу передачі даних HTTPS чутлива інформація користувача в незахищеному вигляді може легко опинитись у руках зловмисника, наприклад, при атаці Man-in-the-Middle.

2.3.1.2 Digest схема автентифікації та авторизації

Digest-автентифікація є складнішою схемою порівняно з Basic. Дані до серверу передаються за використання SHA2-256, SHA2-512/256 або старого MD5 алгоритмів хешування даних входу користувача. Вибір застосованого алгоритму залежить від конфігурування сервера та клієнта. Порядок подій при використанні даного типу автентифікації [74]:

1. Користувач відправляє GET запит на ресурс з відсутнім Authorization заголовком.
2. Сервер формує HTTP-відповідь з кодом 401 та переліком доступних способів автентифікації (першим йде найбажаніший), містить перелік заголовків WWW-Authenticate (рисунок 2.8). Це робиться задля зворотної сумісності.

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Digest
    realm="http-auth@example.org",
    qop="auth, auth-int",
    algorithm=SHA-256,
    nonce="7ypf/xlj9XXwfDPEoM4URrv/xwf94BcCAzFZH4GiTo0v",
    opaque="FQhe/qaU925kfnzjCev0ciny7QMkPqMAFRtzCUYo5tdS"
WWW-Authenticate: Digest
    realm="http-auth@example.org",
    qop="auth, auth-int",
    algorithm=MD5,
    nonce="7ypf/xlj9XXwfDPEoM4URrv/xwf94BcCAzFZH4GiTo0v",
    opaque="FQhe/qaU925kfnzjCev0ciny7QMkPqMAFRtzCUYo5tdS"
```

Рисунок 2.8 - Приклад відповіді сервера на запит клієнта без вказання Authorization заголовку[74]

3. Клієнт за потреби запитує у користувача його дані для входу і далі відповідає на перший отриманий від сервера виклик для автентифікації, обробка якого можлива (залежить від підтримки зазначеного алгоритму хешування) (рисунок 2.9).

```
Authorization: Digest
    username="488869477bf257147b804c45308cd62ac4e25eb717
        b12b298c79e62dcea254ec",
    realm="api@example.org",
    uri="/doe.json",
    algorithm=SHA-512-256,
    nonce="5TsQWLVdgBdmrQ0XsxbDODV+57QdFR34I9HAbC/RVvkK",
    nc=00000001,
    cnonce="NTg6RKcb9boFIAS3KrFK9BGeh+iDa/sm6jUMp2wds69v",
    qop=auth,
    response="ae66e67d6b427bd3f120414a82e4acff38e8ecd9101d
        6c861229025f607a79dd",
    opaque="HRPCssKJSGjCrkzDg8OhwpzCiGPChXYjwrI2QmXDnsOS",
    userhash=true
```

Рисунок 2.9 - Приклад типового значення в Authorization заголовку за використання Digest автентифікації [74]

4. Сервер знаходить пароль, що відповідає вказаному логінові, застосовує хеш-функцію та параметри для функції хешування, зазначені у

клієнтському запиті до нього, і порівнює з даними, що отримав. За збігу відбувається успішна авторизація, клієнт отримує доступ до захищеного ресурсу.

Вказаний метод автентифікації має механізми керування доступом до різних частин захищеного простору ресурсів (“realm”, “domain” параметри у серверній відповіді), містить захист від replay-атак, котрі спрямовані на перехоплення запиту та використання даних з нього для несанкціонованого доступу, у вигляді унікального “nonce” у серверній відповіді та “cnonce” й “nc” у клієнтському запиті значення, які додаються до хеш-функції до даних входу. Також містить механізми перевірки того, що користувач знає пароль (клієнтський “response” параметр, способи генерування якого залежить від отриманого “qop” параметра та наявності частинки “-sess” у назві параметру “algorithm”) та надає клієнтові можливі варіанти поведінки за відправки хибного значення параметру “nonce” та хешування перед відправкою ім’я користувача (параметр “userhash”) [74].

Отже, така схема автентифікації, якщо порівнювати з Basic-схемою, має більш безпечний транспорт даних за рахунок розумного підходу до їх хешування (додавання випадкового згенерованих значень або інших фіксованих, що значно зменшує шанси на успіх зловмисницьких атак)[53] і гарний захист від фішингових атак, які спрямовані на заманювання користувача на несправжні сторінку входу в обліковий запис за рахунок передачі багатоетапного хешу.

Серед яскравих недоліків можна виділити зворотну сумісність автентифікації, яка може змусити клієнта перейти на більш вразливі алгоритми на кшталт MD5. Однак і SHA2-256 та його варіації не надають сильного захисту проти великого спектру атак, як призначені для паролів функції формування ключів (bcrypt, argon2), які було розглянуто в попередніх розділах роботи. Man-in-the-Middle атака несе доволі велику небезпеку для поданого типу автентифікації, оскільки відсутній спосіб верифікації успішності з’єднання із сервером, на відміну від наступних способів, що будуть розглянуті.

Для менеджерів паролів даний тип автентифікації не може підходити через необхідність використовувати слабкі алгоритми для перетворення паролю у хеш.

2.3.1.3 Bearer схема автентифікації та авторизації

Ця HTTP схема було впроваджена в стандарті делегування доступу до даних – OAuth 2.0, проте вона не обмежується тільки застосуванням в такій специфікації.

Клієнт, що використовує її, аби отримати доступ до захищеного ресурсу, у заголовку Authorization вказує текстове значення, що складається з надпису ”Bearer” та так званого токена-доступу (англ. “access-token”). [75] Даний токен є стрічкою з певним значенням, яка може не мати чіткої структури і складатись з набору букв та цифр або бути структурованою у JWT форматі. Для серверу він містить інформацію, за якою можна буде встановити до яких ресурсів клієнт, який надав цей токен, має доступ. Після відправки запиту, на сервері встановлюється дійсність токену і права доступні за ним.

JWT – спеціальний структурований JSON об’єкт, який використовує цифровий підпис. Складається із трьох частин: голови, тіла та підпису. Кожна частина закодована у Base64 кодування (рисунок 2.10) та відділена крапкою. RFC 7519 [76] визначає великий перелік параметрів, що повинні або можуть бути присутніми в даному об’єкті.

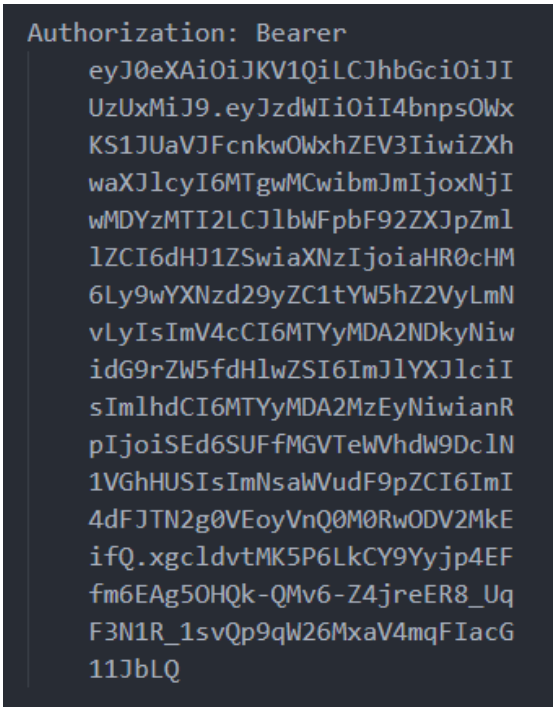

Рисунок 2.10 – Типовий вигляд заголовку в клієнтських запитах в Bearer схемі, токеном-доступу виступає значення JWT

Поза стандартами OpenID Connect 1.0 та OAuth 2.0 чітких вимог щодо реалізації процесу отримування токена-доступу клієнтом немає, проте зазвичай здійснюють POST-запит із даними входу. [75] За успішної автентифікації за переданими даними сервер відповідає згенерованим токеном-доступу та токеном-оновлення (спеціальний унікальний ідентифікатор, за яким за передачі до сервісу авторизації можна буде обміняти його на новий дійсний токен-доступу). [77]

Серед очевидних переваг даного способу автентифікації – зменшення відкритого вікна до атак на дані входу користувачів. Проте HTTPS- підключення є обов’язковим, бо немає належного механізму захисту даних, що будуть надіслані до сервера.

Про Bearer схему не можна говорити без розбору OAuth 2.0 та OpenID Connect 1.0, тому розглянемо ці протоколи.

2.3.2 OAuth 2.0 та OpenID Connect

OAuth 2.0 протокол базується на взаємодії чотирьох сутностей [77]:

- сервер з ресурсами (сервіс або сервер, що містить захищені ресурси, взаємодія з яким відбувається через токени-доступу);
- сервер авторизації (сервіс або сервер, що видає токени-доступу);
- клієнт (застосунок за допомогою якого здійснюється користувацька взаємодія з сервером ресурсів);
- власник ресурсів (користувач або сутність, яка має можливість надати доступ до захищеного ресурсу).

OAuth (Open Auth) 2.0 – це гнучкий стандарт делегації доступу до даних, який специфікує процес авторизації користувачем доступу сторонніх застосунків до ресурсів, що належать цій особі, без необхідності введення даних для входу.

Опис взаємодії, що відбувається за дотримання набору протоколів OAuth 2.0 [77]:

1. У клієнта виникає потреба отримати доступ до певного набору захищених даних користувача. Відправляється звернення до користувача, аби він надав доступ до ресурсів.
2. Клієнт отримує дозвіл від користувача на доступ до ресурсів, а також інформацію, необхідну для отримання токену-доступу.
3. Клієнт надає серверу авторизації дані про тип та значення авторизації, які отримав від користувача. Якщо автентифікація та підтвердження коректності отриманих даних пройшла успішно, то сервер видає токен-доступу.
4. Клієнт звертається до захищених даних користувача, проте цього разу він вже надає токен-доступу. Сервер з ресурсами здійснює обробку запиту від клієнта за умови коректності отриманого токену-доступу.

У практичній частині було вирішено використовувати токени-оновлення, тож розглянемо більш детально їх роль. У специфікації визначають наступну схема взаємодії клієнтів (рисунок 2.11) [77]:

1. Клієнт звертається до серверу авторизації із визначеним в ньому способом надання авторизації.
2. За успішної автентифікації користувача та підтвердження надання ресурсів до клієнта повертаються токен-доступу та токен-оновлення.
3. Клієнт звертається до серверу з ресурсами і надає токен доступу, який отримав в попередньому кроці.
4. Якщо отриманий сервером з ресурсами токен є дійсним та з користувачем було узгоджено доступ до нього (крок перший), то захищений ресурс відправляється до клієнта.
5. Кроки третій та четвертий повторюються допоки не станеться умова шостого пункту.
6. Отриманий токен недійсний. Повертається відповідь з помилкою до клієнта.
7. Клієнт звертається до серверу авторизації і надає йому токен-оновлення.
8. Сервер перевіряє актуальність токену-оновлення і якщо він дійсний, то відправляє у відповідь, або нову пару токена-доступу і токена-оновлення, або тільки токен-доступу (залежить від конкретної імплементації та вимог предметної області).

Рисунок 2.11 – Схема авторизації, що містить токени-оновлення (переклад RFC6749 з англ. [77])

Безпека даних під час використання OAuth 2 дуже сильно варіюється від обраної реалізації, тому варто звертатись до специфікацій даного протоколу, якщо є потреба дізнатись про весь перелік потенційних вразливостей конкретного програмного рішення.

Отже, враховуючи все згадане вище, можемо дійти висновку, що протокол OAuth 2.0 надає можливість стандартизувати спілкування між застосунками за рахунок визначених в його специфікації механізмів авторизації доступу. Для впровадження систем з Single-Sign On (одної автентифікації з передаванням даних для входу достатньо, аби вільно отримувати доступ до будь-якого сервісу, що налаштував дану схему автентифікації) та делегування автентифікації на рішення інших компаній існує протокол OpenID Connect 1.0.

OpenID Connect 1.0 – це стандарт, який є додатковим шаром над специфікацією OAuth 2.0. Він надає можливість встановити ідентичність особи

під час її автентифікації на сервері авторизації й дістати дані облікового запису користувача (пошта, ім’я користувача, аватар тощо). [78]

Протокол оперує двома сутностями: Identity provider (сервіс зберігання облікових даних користувачів, автентифікації, авторизації) та OpenID API (надає можливість за допомогою REST API клієнтові дізнатись атрибути ідентичності, що належать автентифікованому користувачеві). [78]

OpenID Connect 1.0 дозволяє уникнути впровадження інфраструктури, пов’язаної з автентифікацією та надійним зберіганням облікових даних користувачів, шляхом надання можливості використання сервісів від інших компаній (Google, Facebook та інших систем, що вважаються OpenID Provider-ми).

Абстрактний опис взаємодії, що відбувається за дотримання протоколу OpenID Connect 1.0 (рисунок 2.12):

1. Клієнт відправляє запит до OpenID Provider, наприклад до точки від Facebook або Google.
2. Користувач автентифікується на веб-сторінці провайдера й надає доступ до своїх ресурсів.
3. OpenID Provider надає клієнтові ID-токен (англ. “identity-token”) (рисунок 2.13) та токен-доступу.
4. Клієнт робить запит до точки в REST API, котре відповідає отриманню певних даних про користувача.
5. Клієнт отримує дані про які запитав, за умови що наданий токен-доступу виявився дійсним.

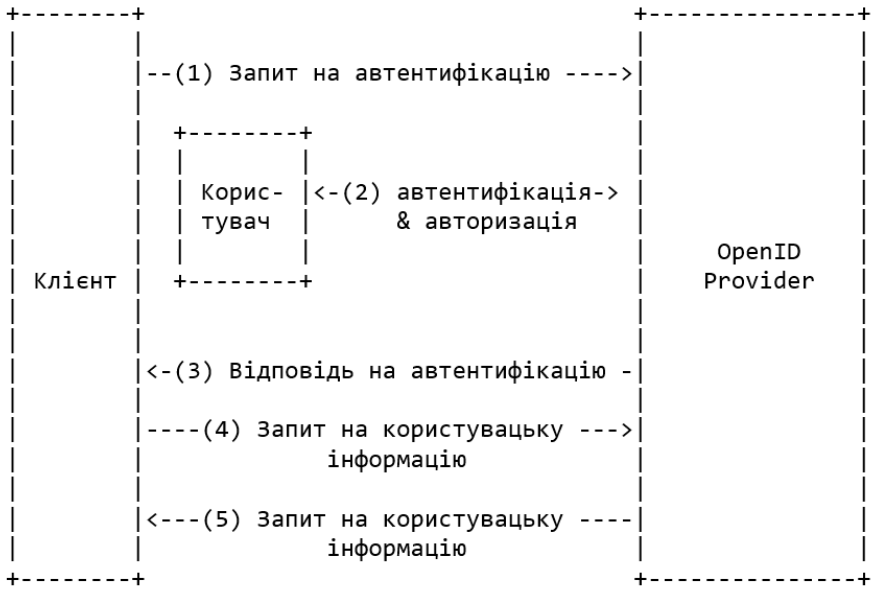

Рисунок 2.12 – Типова схема взаємодії за використання OpenID Connect 1.0 (переклад з англ. [78])

Рисунок 2.13 - Вигляд ID токенів, які є завжди підписаними за допомогою JWS

З потенційними вразливостями така ж ситуація як із OAuth 2.0. Потрібно звертатись до офіційної специфікації, аби зрозуміти, які вразливості має конкретна імплементація.

Саме тому можна зробити висновок, що, на відміну від OAuth 2.0, OpenID Connect 1.0 може слугувати гарним способом делегування ідентифікації та автентифікації користувачів і до того ж також надає гнучкість при реалізації.

Отже, для менеджерів паролів, які орієнтуються на бізнес-сегмент, дана специфікація може використовуватись для реалізації системи автентифікації користувачів, за рахунок того, що сервер авторизації не обов'язково повинен належати розробникам застосунку.

2.3.3 Автентифікація за допомогою API ключів

У даному способі автентифікації, клієнт передає певне унікальне значення при зверненні до кінцевої точки API, яке може міститись в заголовку, як параметр запиту в адресі або як cookie-значення (рисунок 2.14). [79][80] Найбільшого поширення здобув для автентифікації взаємодії між серверами.

```
GET /api/resource?api_key=some_unique_key

GET /resource HTTP/1.1
X-API-Key: some_unique_key

GET /resource HTTP/1.1
Cookie: X-API-KEY=some_unique_key
```

Рисунок 2.14 – Приклади клієнтських запитів з використанням API - ключів

API-ключі не мають механізмів протидії несанкціонованому використанню або ідентифікації користувачів й передаються у відкритому вигляді. [80] Тому в разі потрапляння значення ключа до зловмисника останній буде отримувати доступ до ресурсу, допоки даний ключ не буде анульовано сервером. Протидія атаці Man-in-the-Middle можлива лише за використання захищеного підключення до серверу за використання TLS.

Отже, цей спосіб не підходить для ідентифікації та подальшої автентифікації користувачів, особливо для такого КСЗ як менеджер паролів, проте може бути корисним для інших цілей, зокрема для збирання метрики й регулювання використання доступних ресурсів [81].

2.4 Особливості роботи розширень для Google Chrome

2.4.1 Поняття розширення. Політика Single Purpose

Розширення – це програмні рішення, що додають функціонал до браузера. Створюються з використанням браузерних технологій, по типу JavaScript, HTML, CSS. До JavaScript API, що властивий веб-сторінкам, додається ще окремий набір API, характерний тільки розширенням. [82][83]

До типових видів розширення належать: блокувальник реклами, відслідковування цін на товари в магазині, підбір купонів, активація темної теми для усіх сайтів, простенька гра.

Вище перелічені програми мають схожу філософію – усі вони призначені для виконання одної конкретної функції. Це один з наслідків того, що у 2013 році компанія Google ввела так звану політику “Single Purpose” для розробників розширень. Вона чітко проголосила правило, згідно якого застосунки повинні мати чітке одне призначення, яке можна легко зрозуміти. [83] Тобто ситуація коли розширення надає можливість змінювати колір виділеного тексту та дає змогу користувачеві робити нотатки є неприпустимою.

У випадку менеджерів-паролів, вони мають на меті надати зручний доступ до конкретного вебсервісу, тому даний принцип вони не порушують.

2.4.2 Manifest API для розробки розширення для Google Chrome

Manifest версії 2 (Manifest V2) – це сучасне API для Chrome-розширень. Він складається з набору методів та принципів за якими керується поведінка застосунку. [84]

Файл “manifest.json” – конфігураційний файл у JSON форматі. Є фундаментом усього застосунку. У ньому визначається версія Manifest (API), та усі параметри, характеристики права та файли, що будуть використані в роботі.

До архітектури розширень входить: файл-Manifest, фонові крипти (надають доступ до слухачів подій браузера, важливих для цього ПЗ), елементи UI, скрипти контенту (надають доступ до перегляду та модифікації відкритих користувачем вебсторінок) та сторінка налаштувань розширення.

Для доступу до більшості функцій визначених в “chrome.*” API необхідно здійснювати конфігурування файлу “manifest.json”: змінювати значення поля-масиву “permissions”, яке відповідає за конкретні дозволи програми. Прикладами зняття обмежень може бути: надання доступу до конкретних вебсайтів або доменних імен, дозвіл на запис даних з мікрофону, файлової системи, тощо.[84]

Розширення для Chrome розповсюджуються в магазині Chrome Web Store у вигляді архівів із розширенням “.crx”. Таким чином досягається можливість автономного їх функціонування через те, що всі ресурси задіяні можна локально визначити, в конфігураційному файлі “manifest.json”.

Таким чином, отримана клієнтська частина буде мати структуру типового SPA вебпроекту, проте з особливістю, що фундамент програми буде визначений в одному конфігураційному файлі.

РОЗДІЛ 3. ОПИС РЕАЛІЗАЦІЇ ЗАСТОСУНКУ

3.1 Аналіз технічного завдання

Сформуємо більш чіткі вимоги до фінального програмного продукту враховуючи той факт, що ціллю практичної частини є написання прототипу менеджеру паролів з урахуванням вище розглянутих теоретичних особливостей щодо компонентів та потенційних загроз.

Розширення для браузеру Google Chrome повинно мати можливість зберігати записи, які складаються з назви та даних для входу до певного облікового запису: логіну (англ. “login”) та паролю. Вище перелічені дані користувач буде вводити вручну, на відміну від вбудованих в браузери засобів зберігання даних для входу, які автоматично пропонують зберегти дані, як тільки буде задіяна HTML форма на вебсторінці. Редагування цих записів також передбачається.

Повинна бути надана можливість генерувати паролі з вказанням параметрів на довжину паролю, множину дозволених символів (латинські, кириличні букви, цифри, спеціальні символи) та їх регістр (верхній або нижній). Користувача варто попереджати, якщо він використовує пароль, який насправді є лексемою у словнику англійської мови або є довжиною менше восьми символів. [85]

Користувацький пароль ніколи не повинен покидати клієнт у нетрансформованому незахищеному вигляді, в тому числі й при реєстрації облікового запису, бо за даним паролем буде розшифровуватись весь вміст “сховища” й здобуття цього значення буде нести критичні наслідки для конфіденційності користувацьких даних входу. Тому повинно бути реалізованим надійне зберігання й передача значення для автентифікації користувача у системі.

Система авторизації повинна бути якнайменш вразливою до зловмисницьких атак й повинна мати відпрацьовані стратегії щодо уникнення потрапляння конфіденційної інформації стороннім особам. Також вона зобов’язана містити хоча б один метод двоетапної автентифікації для нових під’єднаних пристроїв.

Клієнт не повинен допускати до роботи із застосунком тих осіб, які не підтвердили свою пошту, яка у кожного користувача є унікальною і є ідентифікатором у КСЗ. Це необхідно для забезпечення вчасного сповіщення користувача, або про вхід із нових клієнтів в їх обліковий запис менеджера паролів, або про іншу критично важливу для сповіщення інформацію.

Окрім захисту від розглянутих в роботі типових атак потрібно мінімізувати можливість будь-якого перехоплення даних й перебування чутливої інформації на шляху до або з сервера. До того ж, необхідно унеможливити потрапляння даних метрик або помилок, що можуть допомогти зловмисникові у досягненні його цілей.

3.2 Архітектура КСЗ

3.2.1 Архітектура серверної частини

Оскільки, як вище в роботі було описано, клієнтська частина застосунку, розширення для браузеру, представляє собою контейнер формату “.crx”, який може дозволяє містити HTML, CSS та JavaScript файли, то на серверну частину було вирішено покласти лише дві функції: авторизація користувача та виконання бізнес-логіки. В той же час клієнт буде містити всі необхідні для його роботи компоненти локально.

Для зберігання даних облікових записів користувачів менеджера паролів було використано реляційну базу даних, бо інформація, яка буде зберігатись матиме чітку структуру, яку розраховується, якщо й змінювати в процесі

використання, то дуже мінімально, та є вимога щодо високої надійності зберігання даних, яка в реляційних базах даних чудово забезпечується чотирма складовими SQL транзакцій: Atomicity, Consistency, Isolation, Durability (ACID). [86]

Іншого підходу потребує зберігання користувацьких “сховищ” (англ. “vault”) з паролями на сервері. Будь-який прототип програми повинен враховувати шляхи для теоретичного розширення функціоналу в майбутньому. У випадку менеджерів-паролів – це можуть бути певні налаштування клієнтських програм або оновлення типів даних, що зберігаються, наприклад у записі можна додати можливість зберігати номер телефону поруч із даним для входу до певного ресурсу. До того ж кількість записів може сягати сотень або навіть тисяч, а кількість користувачів буде постійно тільки збільшуватись. А звідси випливає, одна з основних вимог до бази даних для інформації у “сховищі”: можливість до горизонтальної масштабованості. З огляду на вищесказане, маємо що дані потреби буде повністю задовольняти документно-орієнтована база даних. [87]

Перехід усієї мережної взаємодії на безпечний канал спілкування буде забезпечуватись впровадженням TLS сертифікату, котрий необхідно придбати у відповідних організацій для домену і він повинен бути підписаним спеціалізованим органом сертифікації. Конкретно, для цього прототипу було створено і використано самопідписаний варіант цього документу, аби можна було продемонструвати можливість роботи у HTTPS режимі, проте при справжньому розгортанні у комерційних цілях, необхідно впровадити обов’язково повноцінний сертифікат, аби мати можливість отримати справжній захист від Manipulator-in-the-Middle атаки.

Аби забезпечити виконання вимог щодо безпеки зберігання та передачі значення для автентифікації користувача будуть використовуватись вищерозглянуті в роботі функції отримання ключа через сукупність усіх їх унікальних властивостей по наданні захисту від швидкого підбору. Таким чином

клієнт буде відправляти отриманий хеш від паролю, який прибуватиме на обробки серверною логікою. Тут буде реалізований другий етап захисту – проходження даного значення знову через KDF, але вже з використанням збережених у реляційній БД при реєстрації користувача використаних параметрів та солі для застосування у функції. Якщо отриманий від клієнт хеш можна було привести після виконання цього етапу до вигляду значення у БД, то це буде значити, що користувач вказав свої дійсні дані для входу. Роль KDF на клієнтській та серверній частині буде виконувати вищеописана в теоретичній частині багатьма рекомендована Argon2id функція.

Після здійснення огляду найрозповсюдженіших методів автентифікації та авторизації користувачів стало зрозуміло, що жоден з розглянутих способів на повну міру не може виконати поставлені перед менеджером паролів вимоги. Найближче до поставлених цілей підбирався протокол OpenID Connect 1.0, проте він, як було вище описано, є додатковим шаром над OAuth 2.0, яка спеціалізується тільки на делегації доступу. У даному випадку, реалізація окремого власного Identity Provider сервера не є доцільною, оскілки ціллю даної розробки не стоїть надання можливості стороннім застосункам автентифіковувати користувачів. В той же час в OAuth 2.0 були закладено чудові способи авторизації доступу до сторонніх ресурсів, але в стандарт не входять компоненти ідентифікації та подальшої автентифікації користувача. Виходячи із всього вище сказаного, було вирішено реалізувати нижчеописану архітектуру авторизації, яка буде опиратись на рішення, які були оголошеними для OAuth 2.0 в документі RFC6749 [77]

Було вирішено реалізувати наступну Bearer HTTP-схему авторизації (рисунок 3.1), яка складається із трьох складових – клієнту (розширення), сервісу авторизації на сервері та сервісу ресурсів на сервері (всі запити, які вимагають оброблювати запити тільки від автентифікованого користувача):

1. Клієнт відправляє введені користувачем адрес електронної пошти, хеш від паролю, отриманий від введення паролю, який було вказано при реєстрації, та унікальний ідентифікатор клієнту.
2. Якщо в обліковому записі користувача було активовано двоетапну перевірку, то далі, за успішної автентифікації (вище було описано) за переданим хешем, відбувається надсилання завдання (англ. “challenge”) до користувача. Аби можна було ідентифікувати попередній виконаний запит (збереження стану автентифікації), клієнтові надається тимчасовий код, який він повинен надати разом із відповіддю на завдання.
3. Користувач вводить свій TOTP код, і клієнт передає його значення разом із тимчасовим з попереднього кроку.
4. У разі відсутності активованої двоетапної перевірки й успішної автентифікації або правильної відповіді на завдання, клієнт отримує у своє розпорядження короткотривалий (5 хв) токен-доступу та токен-оновлення (30 діб).

   Визначимо такі їх властивості:

   - Перший токен, у форматі JWT, слугує для авторизації доступу до виконання запитів та для ідентифікації користувача за його публічним ідентифікатором. Для мінімізації ризиків неавторизованого доступу, кожні п’ять хвилин після видання стає недійсним. Є типовим учасником схеми Bearer. Також надає можливість значно розвантажити сервер від постійних та дуже довгих обчислень значень отриманих від KDF, на відміну від застосування Basic HTTP-схеми, шляхом швидкої перевірки підпису та зазначених в ньому даних дійсності.
   - Другий токен використовується, оскільки існує потреба у мінімізації часу перебування похідних від паролю даних у мережі (зменшення розміру вікна для атак). Цей токен-оновлення буде стрічкою у форматі Base64, яка повинна відповідати певному ідентифікатору клієнту та буде надаватись й оновлюватись кожні 5 хв, допоки буде

активною сесія. З останнього слідує, що очікується велика кількість однотипних запитів, тому аби не перевантажувати реляційну базу даних, котра може тільки вертикально масштабуватись [88], необхідно використати представника NoSQL баз даних, який вирізняється швидкістю обробки простих запитів та можливістю до горизонтального масштабування. Після того як проходить 30 діб після його створення, то він перестати буде дійсним й автоматично вилучиться на серверній частині застосування. При відповіді про недійсний токен-оновлення необхідно буде клієнтові здійснити знову автентифікацію через хеш від паролю.

- При даній реалізації неможливими виявляються атаки Session Fixation та Session Prediction, через відсутність використання сесії взагалі.

5. При будь-якому зверненні на сервер, яке пов’язане із прямою взаємодією з ресурсами або обліковим записом, клієнтові необхідно надавати токен-доступу.
6. При наданні дійсного токену-доступу, який відсутній у кеші, JWT ідентифікаторів, з якого користувач вийшов (нижче про це описано), запит клієнта буде оброблено.
7. Клієнт передає недійсний вже токен або з невірним підписом
8. Клієнт отримує у відповіді від сервера код 403 – Forbidden.
9. Отримавши від серверу відмову в обслуговуванні запиту, клієнт надає токен-оновлення й свій ідентифікатор.
10. Якщо на сервері цьому client ID відповідає наданий токен, то клієнтові буде випущена нова пара токенів, а старий токен-оновлення буде прибрано з бази даних.

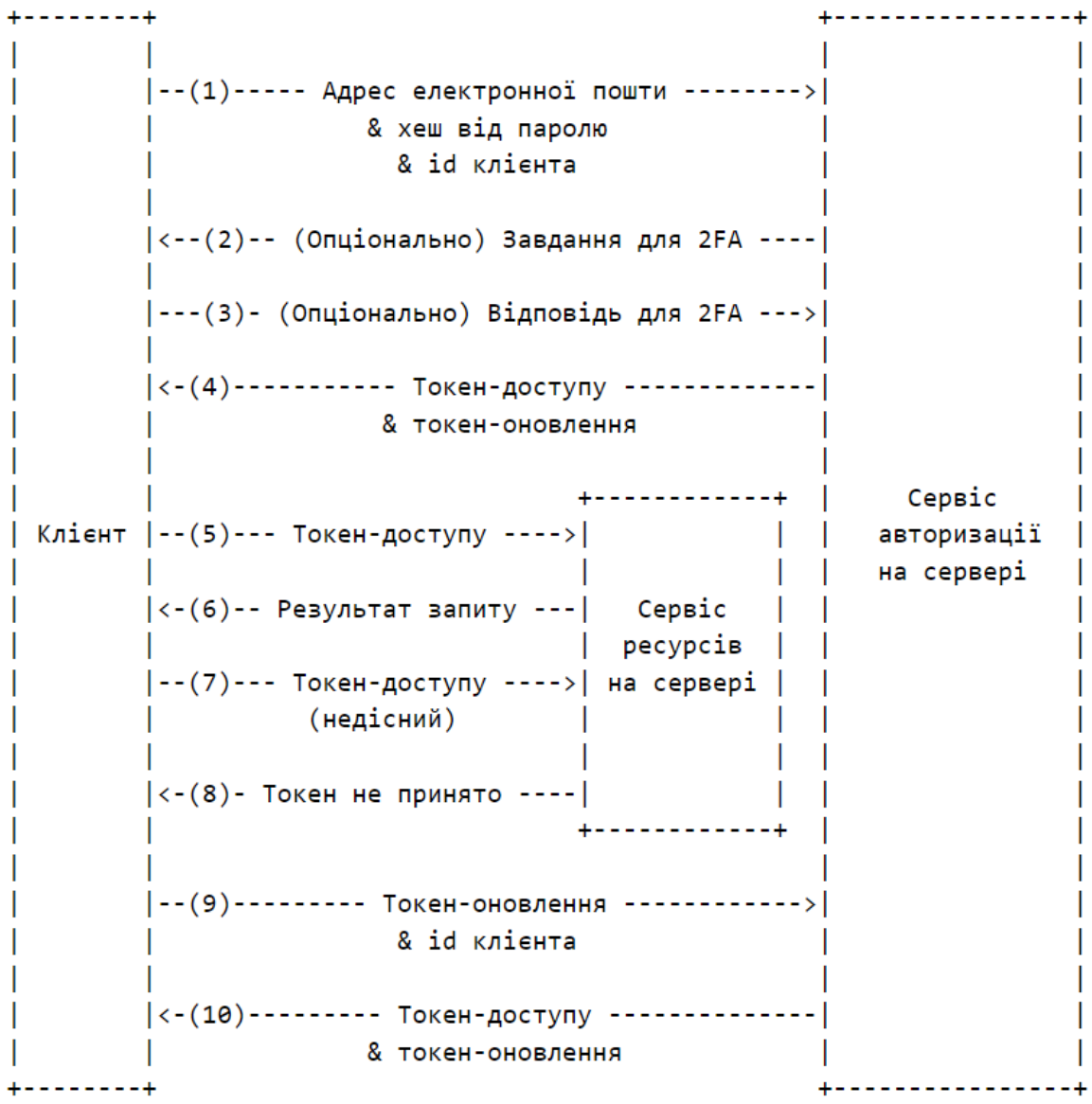

Рисунок 3.1 – Схема взаємодії клієнта та сервера

Для двоетапної автентифікації було реалізовано саме версію Time-based One-Time Password (TOTP), яка є найпоширенішим й найдоступнішим способом. [89] При звертанні на налаштування даної функцію сервер очікує від клієнта переданий хеш від паролю та токен-доступу. У базу даних записується значення секрету (англ. “secret”). Відповіддю сервера буде сформоване посилання для подальшого закодування його у QR код і зчитування мобільним застосунком. Після того як користувач додав у свій застосунок на телефоні цей запис, то він має надати розширенню код, який йому демонструє девайс на даний момент.

Клієнт передає цей код та сервер порівнює його із згенерованим на своїй стороні із врахування різниці в п’ять секунд в меншу сторону через можливу затримку у передачі значення через мережу.

Для забезпечення захисту від Brute-Force атак, які спрямовані на отриманні похідних від користувацького паролю (хешу) SQL БД повинна вести облік кількість невдалих спроб та дату та час останнього невірного входження. В той же час серверна частина повинна реагувати відповідним чином на даний лічильник: за перевищення певної кількості невдалих спроб – блокувати можливість увійти до облікового запису взагалі на певний час.

Для захисту від CSRF атаки, кожен із запитів буде перевірятись на присутність заголовку “X-PM-HEADER”, чого на даний момент достатньо, аби захиститись повністю від даної атаки. [92]

Для повного контролю над станом активації пошти, в реляційній БД необхідно зберігати згенеровані сервером коди (ідентифікатори, що є останньою частиною посилань, на які повинен перейти користувач аби підтвердити справжність скриньки) поруч із відповідними переданими на сервер електронними адресами. Зберігання часу створення надасть механізм для прибирання актуальності неактивованих скриньок.

З огляду на функціональні вимоги було розроблено нижченаведену реляційну схему даних (рисунок 3.4).

USER_DEVICES		
PK	client_id	NN char(22)
FK	user_id	NN char(22)

USERS		
PK	internal_id	NN char(22)
UK	public_id	NN char(22)
UK	email	NN varchar(320)
	hashed_pass	NN varchar(128)
	two_factor_enabled	NN tinyint(1)
	totp_secret	N varchar(32)
	email_verified	NN tinyint(1)
	time_of_last_failed_login	NN timestamp
	number_of_failed_attempts	NN smallint(6)
	en_vault_key	varchar(256)

EMAIL_ACTIVATION		
PK	code	NN varchar(64)
FK	email_address	NN varchar(320)
	created_at	NN timestamp

PK – первинний ключ

FK – зовнішній ключ

UK – унікальне значення

NN – не може набувати NULL значення

N – може набувати NULL значення

Рисунок 3.4 – реляційна схема даних для зберігання інформації про користувача

3.2.2 Архітектура клієнтської частини

Як було вже вище зазначено, клієнт буде автономним і не потребуватиме завантаження вебсторінок із інтернету, тому можна виділити лише наступні архітектурні особливості.

Зовнішнє оформлення буде створено за допомогою звичайного HTML та CSS підходу з використанням Bootstrap [93].

Для зручності роботи із даними було обрано формат JSON для зберігання користувацького “сховища”, а для забезпечення конфіденційності користувацької інформації алгоритм AES-256 CBC буде використано для перетворення записів із зашифрованого стану в дешифрований в оперативній пам’яті і для шифрування поданих від користувача нових даних відповідно. За допомогою цього ж алгоритму буде зашифровано та відправлено в реляційну БД ключ шифрування, оптимальної довжини в 32 байти. Саме сховище буде зберігатись в chrome.storage.local, але не в chrome.storage.sync, бо не можна допустити синхронізацію таких конфіденційних даних серед будь-якого пристрою.

Ідентифікатор клієнту буде створюватись після першого запуску розширення або після примусовій очистки chrome.storage.local. Дозволятиме серверу вести облік авторизованих пристроїв і за ним видавати нові токени-оновлення. В цьому ж місці будуть також зберігатись токени доступу та токени-оновлення.

3.3 Обґрунтування вибору технологій

3.3.1 Використані технології для серверної частини застосунку

Для розробки було використано інтегроване середовище Inteliji IDEA 2021.1. Git було використано у ролі системи керування версій програми.

Реляційною СКБД стала MariaDB, яку було обрано через той факт, що вона є відгалуженням від популярного рішення в індустрії – MySQL [94], з відкритим кодом, активною спільнотою та безкоштовним доступом до усього функціоналу. [95]

MongoDB - представник документно-орієнтованої система керування базами даних, був використаний, оскільки є дуже розповсюдженим, перевіреним багатьма розробниками та має широку підтримку від спільноти. [96]

Роль сховища для токенів-оновлення дісталась теж документно-орієнтованій базі даних Redis. Вона має дуже високу швидкість виконання запитів, здобула широку популярність та відповідає усім поставленим вимогам, у тому числі має механізми збереження поточного стану БД у довготривалу пам’ять. [97]

Для розробки серверної частини було обрано мову програмування Kotlin, котра стрімко набирає популярності [98] та має підтримку роботи всесвітньо відомої та популярної в сегменті розробки клієнт-серверних застосувань - Java Virtual Machine (JVM). [99] Через обмеження у сумісності з певними використаними у проекті бібліотеками було обрано версію 1.8 для JVM.

Утилітою збирання проекту виступатиме Gradle, котрий було обрано за зручний синтаксис мови Groovy та швидкість збирання проектів у порівнянні з Maven. [100]

Використаний каркас (англ. “framework”) – Ktor. Це бібліотека від компанії JetBrains, розробників мови Kotlin, версії 1.5.4. Її було обрано через простий декларативний спосіб написання клієнт-серверних застосунків, широке використання усіх переваг цієї мови, модульний підхід, достатній набір вбудованого функціоналу для реалізації повноцінного застосунку та персональний великий досвід роботи з ним. [101]

Двигуном роботи усього серверу обрано Netty, яка вбудована у відповідний модуль каркасу Ktor. Він призначений для розробки високопродуктивних асинхронних вебсерверів на JVM. [102]

Журнал запитів і помилок буде забезпечувати Logback, який рекомендовано використовувати у поєднанні із Ktor через повну інтеграцію. [103]

Робота з Argon2 буде забезпечена бібліотекою Jargon2-API, котра надає зручний інтерфейс взаємодії й враховує усі особливості роботи JVM та самого алгоритму. [104]

Для TOTP двоетапної автентифікації з підтримкою правил, що визначені Google Authenticator було використано бібліотеку kotlin-onetimepassword, яка повністю покриває специфікації RFC. [105]

Для реалізації роботи з токенами-доступу у форматі JWT було під’єднано відповідний Ktor модуль, який є тісно інтегрованим з цим каркасом.[106]

Для використанням мапи, у якій кожний запис містить регулювання часу на життя після його запису, Caffeine бібліотека виявилась єдиним кандидатом на цю роль, оскільки періодично під час обробки запитів звертається до випадкових пар мапи і таким чином має механізми для автономного очищення пам’яті. [107]

Для впровадження захисту від CSRF було використано бібліотеку з прямою інтеграцією з Ktor - ktor-csrf [108]. А для протидії Denial-of-Service та Brute-Force атакам – Resilience4j [109], котра надає можливість використовувати функціонал лімітування кількості звернень від одного користувача.

3.3.2 Використані технології для клієнтської частини застосунку

WebStorm – інтегроване середовище розробки було використане разом із системою керування версій Git.

Для перетворення користувацького паролю в хеш за допомогою Argon2 було використано бібліотеку argon2-browser, яка є відносно популярною й одночасно має можливість виконуватись у браузері за рахунок компілювання в Web Assembly. [110]

Для спрощення роботи із запитами на клієнті було використано бібліотеку ky, яка є обгорткою над Fetch API браузерного JavaScript.

Шифрування клієнтської інформації буде використовувати aes-js [111] бібліотеку, котра на відміну від популярної раніше Crypto-js компанії Google [112], підтримується на даний момент і може напряму працювати в браузері.

Генерування паролів буде використовувати можливості бібліотеки generate-password, яка підтримує роботу у браузері й містить весь необхідний функціонал для створення сильних паролів. [113]

Оскільки, для роботи Google Authenticator необхідно просканувати QR код, то буде використано популярну бібліотеку qrcodejs. Вона може працювати із усіма поточними браузерами і генерує на HTML canvas елементі самі зображення закодованої інформації. [114]

3.4 Особливості реалізації КСЗ

3.4.1 Важливі моменти в реалізації серверної частини

Розглянемо структуру отриманої серверної частини (рисунок 3.5):

- тека resources – містить конфігураційний файл application.conf у форматі HOCON. В ньому конфігуруються наступні компоненти системи: параметри KDF Argon2, час зберігання даних у кеші для входу через двоетапну перевірку, кількість запитів й проміжок часу для захисту від Denial-of-service та Brute-force атак, час дійсності токенів, налаштування пошти, TLS сертифікатів, БД, MongoDB, Redis та параметри для розгортання застосунку;

- тека db – тека з Kotlin класами, в яких визначена взаємодія з MariaDB, Redis та MongoDB;
- тека dev – містить єдиний виконуваний клас, який генерує самопідписані сертифікати, призначений для імітування роботи справжніх TLS сертифікатів;
- тека exceptions – містить класи помилок, які використовуються по всьому проектові;
- тека http – визначає класи, які використовуються при транспорті від і до клієнта;
- тека routes – визначає шляхи та контролерну логіку, для кожного з них;
- тека services – складається із класів, що визначають усю бізнес логіку серверної частини. Логічно можна розділити на авторизаційний сервіс та сервіс ресурсів;
- тека utils – визначає функції, які використовуються по всій програмі;
- файл Application.kt – виконуваний файл серверу, який ініціює та налаштовує усі компоненти застосунку.

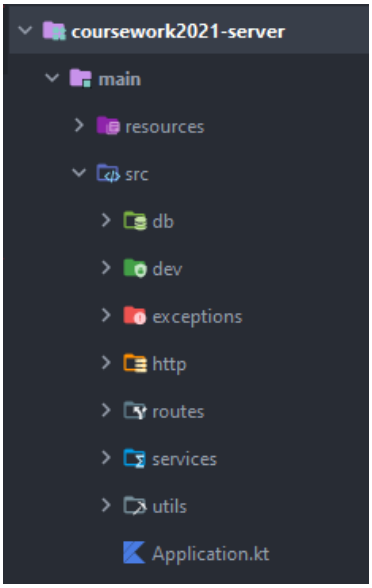

Рисунок 3.5 – Структура проекту

Для захисту від Manipulator-in-the-Middle буде використано автоматичний редірект клієнта з HTTP з’єднання на HTTPS та, до того ж, буде додано до кожної

відповіді заголовок “HTTP Strict-Transport-Security”, який не дасть браузерові лишній раз пробувати підключиться по незахищеному з’єднанні (рисунок 3.6). Нижче на тому ж рисунку вказаний захист від CSRF – вимагання вказування спеціального заголовку “X-PM-HEADER”.

```
install(HSTS)
install(HttpsRedirect) {  this: HttpsRedirect.Configuration
    sslPort = 443
    permanentRedirect = true
}
install(CsrfProtection) {  this: CsrfProtection.Configuration
    applyToAllRoutes()
    validate(HeaderPresent( name: "X-PM-HEADER"))
}
```

Рисунок 3.6 – Налаштування захисту від CSRF та Man-in-the-Middle

SQL ін’єкції за використання PreparedStatement з драйвером JDBC Java драйвером є неможливими, а MongoDB має вбудований захист від NoSQL ін’єкцій за використання Binary JSON (BSON) об’єктів.

Захист від Denial-of-Service було реалізовано за допомогою компоненти resilient4j - Ratelimiter. Вона дозволяє виставити параметри максимального навантаження від користувача за заданий момент часу. Клас ConnLimitingService використовує кеш від бібліотеки Caffeine для реєстрації усіх Ratelimiter екземплярів. Вони створюються для кожної ip-адреси, яка звертається до даного сервісу. Використання зазначено на рисунку 3.7.

```
route( path: "/token") {  this: Route
    post {  this: PipelineContext<Unit, ApplicationCall>
        val host = call.request.host()
        ConnLimitingService.getAuthLoginLimiter(host).executeSuspendFunction {
            val postParameters = call.receiveParameters()
```

Рисунок 3.7 – Приклад використання Ratelimiter-ів

Токени-оновлення та коди доступу до даних автентифікації у кеші (елемент двоетапної перевірки) покладаються на унікальність та непередбачуваність згенерованого значення, проте UUID не рекомендовано використовувати для даних цілей. Натомість необхідно використовувати криптографічно безпечні генератори випадкових значень. В JVM таким буде SecureRandom клас [49]. Для використання на ОС Windows рекомендують використовувати Windows-PRNG, а не SHA1PRNG, котрий ставиться за замовчуванням, а для ОС Linux – NativePRNG, який вже буде автоматично вибрано правильно. При реалізації цього генератора випадкових стрічок, необхідно спочатку виділити масив байтів, в які буде дане значення записане, а потім його конвертувати в Base64 (рисунок 3.8).

```
object SecureRandomString {
    private val random = when {
        System.getProperty("os.name").startsWith( prefix: "Windows")
        → SecureRandom.getInstance( algorithm: "Windows-PRNG")
        else → SecureRandom()
    }
    private val encoder = Base64.getUrlEncoder().withoutPadding()

    fun generate(byteArrayLength: Int): String {
        val buffer = ByteArray(byteArrayLength)
        random.nextBytes(buffer)
        return encoder.encodeToString(buffer)
    }

    fun generatedLength(byteArrayLength: Int)
        = ceil( x: byteArrayLength.toDouble() * 1.33).toInt()
```

Рисунок 3.8 – Лістинг коду з генеруванням криптографічно безпечних випадкових стрічок

Розглянемо додаток 1. В ньому проілюстровано код, який відповідає з основну компоненту авторизації доступу. Структура токену виходить такою, як наведено у наступному рисунку 3.9 і будується методом . Детальніше опишемо,

що значить кожне поле у отриманому JWT токені, який підписано за допомогою HMACSHA512 та значенням секрету “mysecret”.

- sub – унікальний ідентифікатор користувача, який використовується при знаходженні поза межами серверної логіки;
- expires – кількість секунд, після яких токен перестане бути дійсним;
- iss – значення від видавця токену;
- exp – дата та час, після яких токен перестане бути дійсним;
- iat – дата та час створення токену;
- jti – унікальний ідентифікатор, який використовується при реалізації можливості моментального виходу із облікового запису.

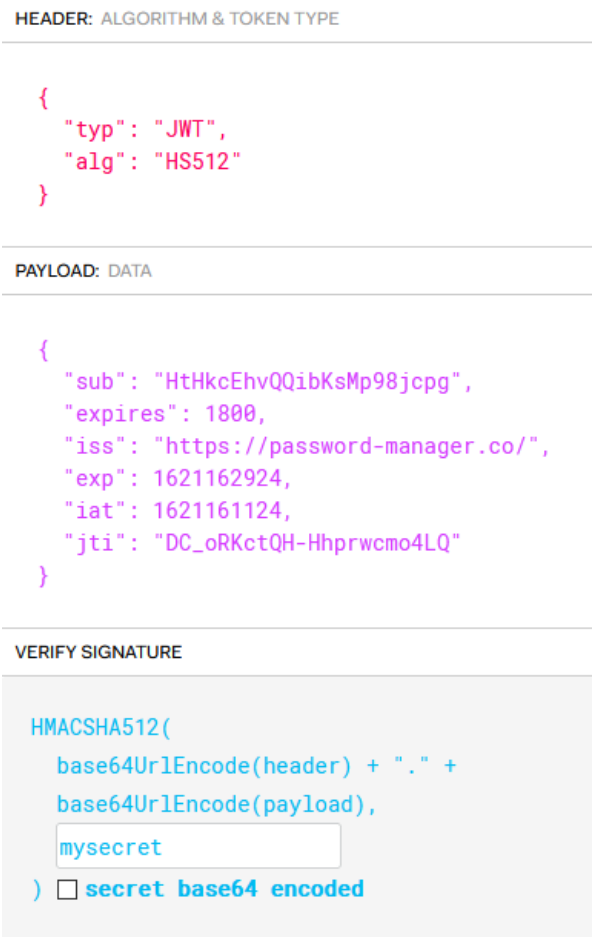

Рисунок 3.9 – Значення даних запакованих в JWT токен-доступу

Також в тому додаткові вказані наступні функції: validatorFunc, яка використовується для того, щоб отримати від запита значення токена за умови,

що його не має серед забракованих (після виходу із запису) та verifierFunc, який здійснює перевірку підпису за алгоритмом та дійсність по часу токена-доступу.

Розглянемо додаток 2. Система автентифікації складається із двох кешів, з тимчасовими ключами, один призначений до легкого блокування подальших спроб на вхід до системи (після п’ять спроб необхідно зробити перерву на 15 хв), а і інший до тяжкого (після п’ятнадцяти спроб необхідно зробити перерву на 6 год). Вони були введені для того щоб протидіяти Brute-Force атаці. При кожній невірній спробі автентифікуватися паролем, в реляційній БД буде збільшено значення лічильника на одиницю або воно зміниться на нуль. При вдалій автентифікації значення теж стає нулем.

Також в систему автентифікації вбудовано автоматичне сповіщення на пошту про новий вхід до системи, якщо було використано новий унікальний ідентифікатор клієнта.

За увімкненої двоетапної автентифікації на обліковому записі користувача, клієнт отримає у відповідь JSON файл, який міститиме тип завдання та згенерований код, за яким потім можна клієнтові отримати токен-доступу та токен-оновлення.

Варто також прокоментувати метод створення та збереження токенів-оновлення в Redis. Розглянемо код проілюстрований на рисунку 3.10.

Як було вже вище пригадано, використовувати UUID не є гарною практикою для значень токенів оновлень, тож тут використовується 256-бітне значення згенероване криптографічного безпечним генератором випадкових чисел і, до того ж, немає необхідності токени оновлення обов’язково суворо робити унікальними, оскільки їх збереження в Redis відбувається за ключем, яким є унікальний ідентифікатор клієнта.

Використовується підключення – пул, бо воно забезпечує підвищення ефективності виконання запитів, якщо одночасно буде надходити їх велика кількість.

Після успішного завершення транзакції відбувається, або оновлення існуючого рядку в USER_DEVICES, або за відсутності – створення нового. В основному ця реляція призначена для відстеження користування обліковим записом та служить додатковим шаром захисту, бо потрібно знати зловмисникові не тільки значення токену-оновлення, а й актуального ідентифікатора клієнта.

```
fun generateAndSaveToken(clientId: String, userId: String): String {
    val newOpaqueToken = SecureRandomString.generate( byteArrayLength: 32)

    redisHandler.lettucePool.acquire().thenCompose { connection →
        val asyncCmd: RedisAsyncCommands<String, String> = connection.async()
        asyncCmd.del(clientId)
        asyncCmd.setex(clientId,  seconds: validityInMin * 60, newOpaqueToken)
        asyncCmd.exec()
            .whenComplete { s, throwable →
                redisHandler.lettucePool.release(connection)
            } ^thenCompose
    }

    val userDevice = UserDevice(clientId, userId)
    try {
        UserDevicesDAO.create(userDevice)
    } catch (e: SQLException) {
        UserDevicesDAO.update(userDevice)
    }
    return newOpaqueToken
}
```

Рисунок 3.10 – Лістинг коду генерування токенів-оновлення

Розглянемо додаток 3 – реалізацію роботи двоетапної автентифікації. В ній можна виділити декілька наступних особливостей.

По-перше, використовується кеш з лічильником на існування з бібліотеки Caffeine. Ключем виступає код, отриманий при проходженні першого етапу автентифікації (нагадує Authorization Grant визначений в OAuth 2.0) [77], а значенням – структура PostponedAuthInfo, яка складається із внутрішнього імені

користувача userId, пари з токенів доступу та оновлення – tokenPair, IP-адреси, за якою було здійснено автентифікацію, та назви програмного агенту – userAgent.

Для того щоб пройти саму перевірку, користувач повинен надати згенерований код у відповідному записі на мобільному пристрої, а клієнт – повернутий із запиту попередньої автентифікації код. Якщо код авторизації виявиться дійсним, то буде ініційована перевірка вказаного користувач TOTP значення. Для перевірки використовується бібліотека kotlin-onetimepassword, й за допомогою неї, через передання їй секрету користувача з БД, буде створено клас-перевіряльник, який зможе за поданим часом встановити чи насправді наданий TOTP є дійсним. Також в реалізації враховано випадок мережної затримки при надходженні запита – якщо перша перевірка не спрацювала, то створюється екземпляр часу з врахуванням запізнювання на п’ять секунд, чого цілком є достатньо.

3.4.2 Важливі моменти в реалізації клієнтської частини

Розглянемо структуру отриманої клієнтської частини (рисунок 3.11):

- тека background містить HTML файл, який збирає усі фонові процеси, які спрямовані на те щоб слухати події в браузері, аби визначати поточну сторінку, на якій знаходиться користувач;
- тека icons містить усю використану графіку;
- тека popup складається з одного HTML-файлу, який буде змінювати свій вигляд в залежності від поточного меню, та набору під’єднаних CSS, JS файлів;
- файл manifest.json визначає роботу усього розширення (додаток 4).

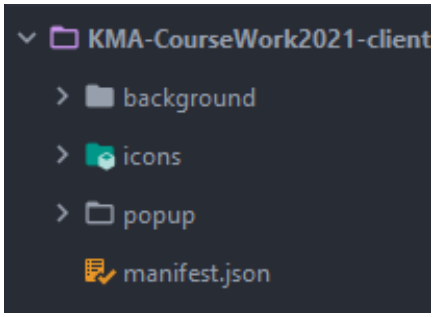

Рисунок 3.11 – Структура клієнтської частини

Розглянемо рисунок 3.12. На ньому зображена конфігурація звернень до сервісу ресурсів, яка враховує зазначено вище архітектурну логіку виконання авторизації доступу. Також варто зазначити, що заголовок X-PM-HEADER - відповідальний за захист від CSRF атак. Таким чином його необхідно додавати до будь-яких запитів до серверу, інакше буде отримано помилку з кодом 400 – Bad Request.

```
const resourceCall = ky.extend({
    hooks: {
        beforeRequest: [
            async request => {
                let accessToken = await getStorageValue( key: 'access_token');
                request.headers.set('X-PM-HEADER', '');
                request.headers.set('Authorization', `Bearer ${accessToken}`);
            }
        ],
        beforeRetry: [
            async ({request, options, error, retryCount}) => {
                if (error.response === 403) {
                    let clientId = await getStorageValue( key: 'client_id');
                    let prevRefreshToken = await getStorageValue( key: 'refresh_token');
                    let tokenPair = await ky('https://192.168.1.95/token/refresh',
                        {json: {refresh_token: prevRefreshToken, client_id: clientId}}).json();
                    onTokenPairUpdate(tokenPair.access_token, tokenPair.refresh_token);
                    request.headers.set('Authorization', `Bearer ${token}`);
                }
            }
```

Рисунок 3.12 – Лістинг коду, який відповідає за типове звернення до серверу

При реєстрації, якщо користувач буде вводити пароль довжини менше 8 символів, то було зроблено так, аби на екрані з’являлась відповідна помилка

(рисунок 3.13). Також на екрані входу при введенні некоректної пошти або паролю виводиться абстрактна помилка, яка просто вказує на неправильність введених даних (рисунок 3.14).

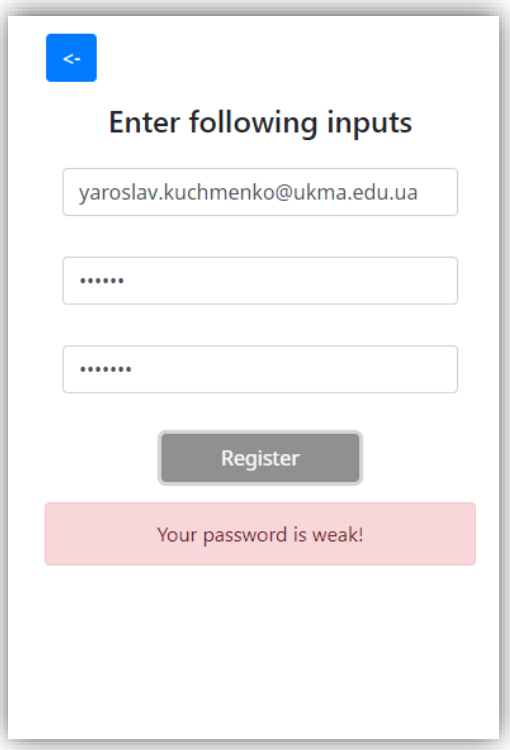

Рисунок 3.13 – Ілюстрація реєстрації при введені криптографічно простого паролю

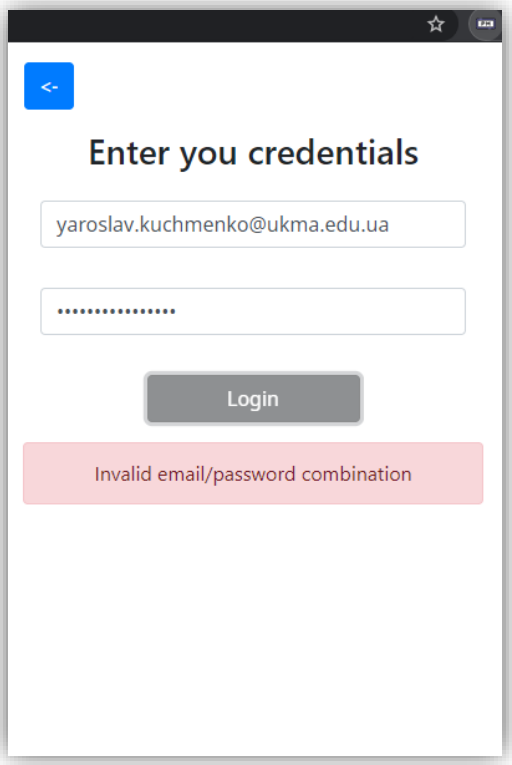

Рисунок 3.14 – Ілюстрація неправильно введених даних при спробі увійти в обліковий запис

Оскільки, ми реалізовуємо підтримку Google Authenticator версії TOTP, то нам необхідно мати механізми налаштування його, яке відбувається за надання секрета. Розробники ОС Android та iOS надають можливість вказувати URI адреси, які дають можливість налаштувати розробникам прями перехід в потрібний застосунок і виконання закодованої в дану адресу функції. Сервер повинен згенерувати секрет та відправити до клієнта. Далі викликається функція обробник-запиту generateQrCode (рисунок 3.15), яка генерує Qr – код із потрібним URI значенням (рисунок 3.16).

```
function generateQrCode(secret) {
    let domElem = document.getElementById("qrcode");
    new QRCode(domElem, "otpauth://totp/PasswordManager?secret="+secret)
}
```

Рисунок 3.15 – генерування QR на клієнт URI для TOTP алгоритму

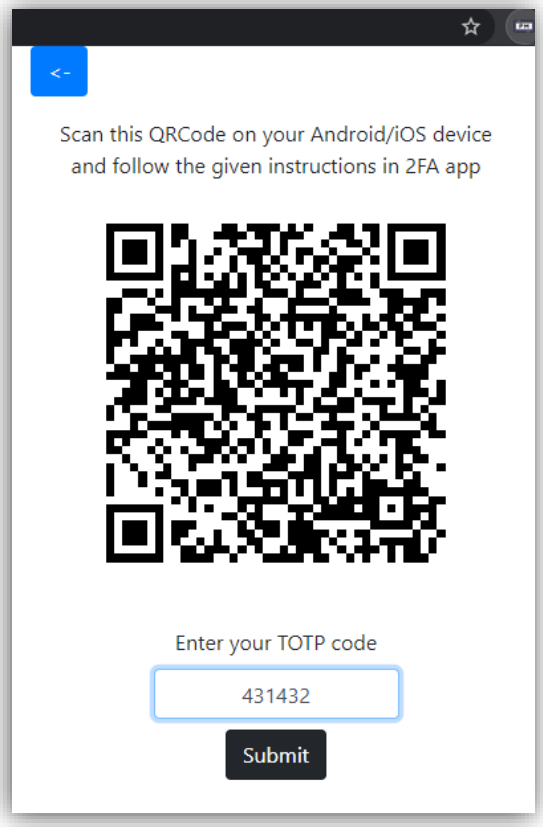

Рисунок 3.16 – Ілюстрація етапу встановлення двоетапної перевірки

При увімкненій двоетаповної автентифікації, після передачі клієнтом, правильної комбінації хеша та адреси електронної пошти, користувача переводять на сторінку, яка зображена на рисунку 3.16, а за умови введення коректного значення TOTP - буде направлено на сторінку із поточними записами (рисунок 3.18).

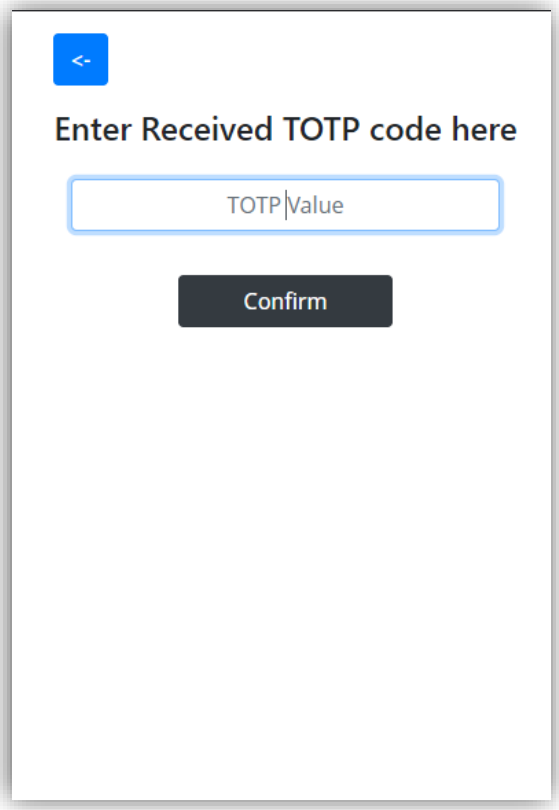

Рисунок 3.17 – Ілюстрація етапу проходження двоетапної перевірки

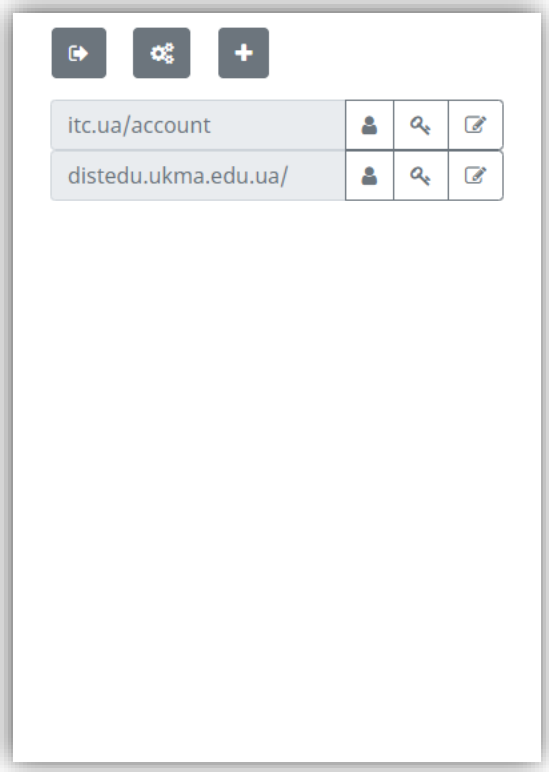

Рисунок 3.18 – Ілюстрація екрану переліку усіх записів менеджера паролів

Як вище вже було зазначено, генерування паролів потребує використання криптографічно безпечного генератору випадкових значень. Оскільки, застосунок зобов'язаний містити цей функціонал засобів (рисунок 3.20), як було поставленому в технічному завданні, то для криптографічного генерування паролів на клієнтському застосунку необхідно звертатись до вбудованих в браузер засобів (рисунок 3.19), а саме компоненту вікна: до об'єкта crypto, що і реалізувала використана бібліотека generate-password.

```
let arr = new Uint8Array( length: 16);
window.crypto.getRandomValues(arr);
```

Рисунок 3.19 – Приклад, коду, який потрібно було використати для випадково згенерованих значень

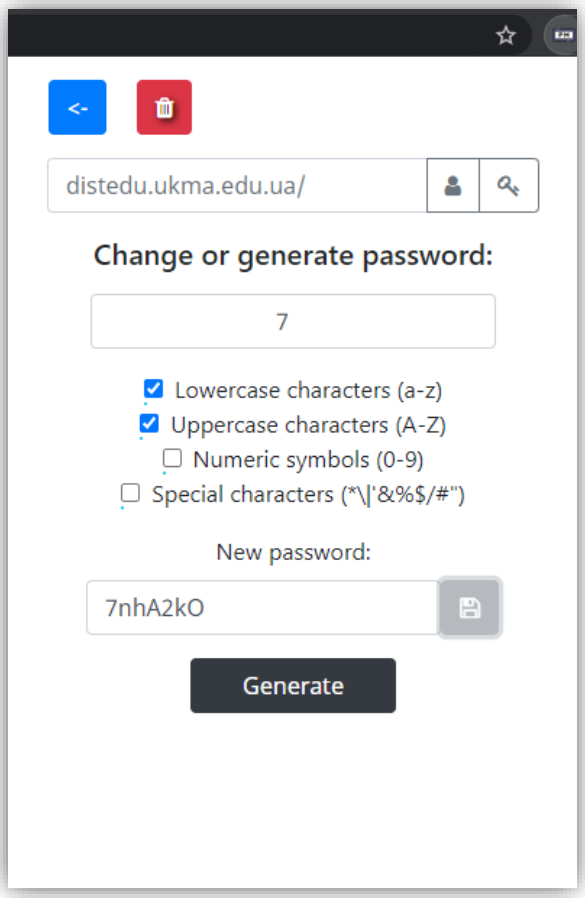

Рисунок 3.20 – Вигляд екрану перегляду запису, яка містить генерування паролю

У додатку 5 розглядаються ключові моменти реалізації в шифруванні, дешифруванні користувацьких даних та їх збереження на дискові (chrome.storage.local). Варто зазначити, що вектором ініціалізації та ключем для AES256 CBC для функції шифрування виступає KDF Argon2, яка ділиться навпіл, а на вхід бере пошту користувача та отриманий після введення паролю ключ шифрування.

3.5 Висновки до розділу 3

У цьому розділі було описано повний цикл розробки отриманого застосунку.

Спочатку було проаналізовано технічне завдання, яке поєднало у собі поставлене в першому розділі завдання та отримані знання про особливості розробки, які були описані у другому.

Наступним завданням було створити таку архітектуру застосунку, яка б повністю відповідала поставленим вимогам. Для виконання даної задачі було витрачено чимало часу у зв’язку із комплексним підходом до кожної складової роботи, які є обов’язковими учасниками таких серверів та клієнтів та вимагають детальної уваги. Таким чином було запропоноване цілісне бачення архітектури й деталей роботи для кожної з критичних компонент.

Наступним етапом в розробці став вибір технологій, який не став проблемою, оскільки на даний момент для розробників даного типу КСЗ є велика кількість популярних бібліотек.

Під кінець розділу було розглянуто особливості реалізації за використання обраних технологій та ключові частини зовнішнього вигляду клієнтського застосунку.

ВИСНОВКИ

Під час виконанні курсової роботи була розглянута чимала кількість інформації щодо тих особливостей, які вимагає предметна область, та у її результаті було отримано клієнт-серверний застосунок, в архітектурі якого було враховано захист від найбільш типових загроз. До того ж реалізований клієнт та сервер надають весь той мінімальний функціонал, який необхідно мати, аби можна було називати це застосування менеджером паролів.

Також отриманий КСЗ має можливість до успішного розвитку, розширення та покращення користувацького досвіду без значних змін до вже налагодженої архітектури.

Розгляд використання хеш-функцій та функцій шифрування в криптографії надав можливість більш свідомо проектувати та розробляти застосунки, які вимагають високий рівень захисту конфіденційних даних, а теоретичні відомості про поширенні вразливості, на прикладі менеджера паролів, заставляють більш обачніше підходити до розробок.

Зрештою, отримана реалізація зуміла надати чітке представлення про розробку сучасних клієнт-серверних застосунків із серйозними вимогами.

Режим доступу до розробки:

https://github.com/lemonderon/KMA-CourseWork2021-server

https://github.com/lemonderon/KMA-CourseWork2021-client

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Zviran, M., & Haga, W. J. (1999). Password Security: An Empirical Study. Journal of Management Information Systems, 15(4), 161–185. [Електронний ресурс] https://doi.org/10.1080/07421222.1999.11518226 (дата звернення – 01.05.2021).
2. FIDO2: WebAuthn & CTAP [Електронний ресурс] – https://fidoalliance.org/fido2/ (дата звернення – 01.05.2021).
3. Beyond Passwords: FIDO2 and WebAuthn in Practice [Електронний ресурс] https://www.inovex.de/blog/fido2-webauthn-in-practice/ (дата звернення – 01.05.2021).
4. SP 800-63 – Electronic Authentication Guideline. NIST, 2004, ст.47 [Електронний ресурс] https://web.archive.org/web/20040712152833/http://csrc.nist.gov/publications/nistpubs/800-63/SP800-63v6_3_3.pdf (дата звернення – 01.05.2021).
5. A Large-Scale Study of Web Password Habits. Dinei Florˆencio and Cormac Herley. Microsoft Research. One Microsoft Way. Redmond, WA. [Електронний ресурс] https://www.microsoft.com/en-us/research/wp-content/uploads/2006/11/www2007.pdf (дата звернення – 02.05.2021).
6. Uncovering Password Habits: Are Users" Password Security Habits Improving? (Infographic) [Електронний ресурс] https://digitalguardian.com/blog/uncovering-password-habits-are-users-password-security-habits-improving-infographic (дата звернення – 01.05.2021).
7. America"s Password Habits: 2020 [Електронний ресурс] https://www.security.org/resources/online-password-strategies/ (дата звернення – 02.05.2021).
8. Cain, A. A., Edwards, M. E., & Still, J. D. (2018). An exploratory study of cyber hygiene behaviors and knowledge. Journal of Information Security and

Applications, 42, 36–45. [Електронний ресурс] https://doi.org/10.1016/j.jisa.2018.08.002 (дата звернення – 02.05.2021).

9. C. Wang, S. T. K. Jan, H. Hu, D. Bossart, і G. Wang, «The Next Domino to Fall», в Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, 2018, doi: 10.1145/3176258.3176332 [Електронний ресурс] - http://dx.doi.org/10.1145/3176258.3176332 (дата звернення – 02.05.2021).
10. Security Update for the LastPass Extension [Електронний ресурс] https://blog.lastpass.com/2017/03/security-update-for-the-lastpass-extension/ (дата звернення – 02.05.2021).
11. Google Trends [Електронний ресурс] https://trends.google.com/trends/explore?date=all&q=password%20manager (дата звернення – 02.05.2021).
12. Browser Market Share Worldwide. April 2021 [Електронний ресурс] https://gs.statcounter.com/browser-market-share/desktop/worldwide (дата звернення – 02.05.2021).
13. Microsoft launches first Chromium Edge builds for Windows 10 [Електронний ресурс] https://venturebeat.com/2019/04/08/microsoft-launches-first-chromium-edge-builds-for-windows-10/ (дата звернення – 02.05.2021).
14. The Chromium-Powered Opera Is Finally Here [Електронний ресурс] https://www.webpronews.com/the-chromium-powered-opera-is-finally-here (дата звернення – 02.05.2021).
15. How to install Chrome extensions in Chromium-based browsers. 2019 [Електронний ресурс] https://www.addictivetips.com/web/install-chrome-extensions-in-chromium-based-browsers/ (дата звернення – 02.05.2021).
16. Best password managers in 2021: Free and paid software to secure your passwords [Електронний ресурс] https://www.techradar.com/best/password-manager (дата звернення – 02.05.2021).

17. Norton - How a password manager can help secure your passwords [Електронний ресурс] https://us.norton.com/internetsecurity-how-to-password-managers-are-the-key-to-secure-passwords.html (дата звернення – 02.05.2021).
18. Why You Need a Password Manager. Yes, You - Andrew Cunningham [Електронний ресурс] https://www.nytimes.com/wirecutter/blog/why-you-need-a-password-manager-yes-you/ (дата звернення – 02.05.2021).
19. The Economic Times - Definition of "Authentication" [Електронний ресурс] https://economictimes.indiatimes.com/definition/authentication (дата звернення – 02.05.2021).
20. Sharma, Seema, "Location Based Authentication" (2005). University of New Orleans Theses and Dissertations. 141. [Електронний ресурс] https://scholarworks.uno.edu/td/141 (дата звернення – 02.05.2021).
21. RFC 6238 – TOTP: Time-Based One-Time Password Algorithm [Електронний ресурс] https://tools.ietf.org/html/rfc6238 (дата звернення – 02.05.2021).
22. Password Managers: Under the Hood of Secrets Management. 2019 [Електронний ресурс] https://www.ise.io/casestudies/password-manager-hacking/ (дата звернення – 02.05.2021).
23. Medium - Before You Use a Password Manager. Stuart Schechter [Електронний ресурс] https://stuartschechter.medium.com/before-you-use-a-password-manager-9f5949ccf168 (дата звернення – 02.05.2021).
24. Bitwarden. Help. Article. Vault data. [Електронний ресурс] https://bitwarden.com/help/article/vault-data/ (дата звернення – 02.05.2021).
25. Security Concerns in Password Managers. Investigation and comparison of password management tools based on security concerns: Matthew Grant, Jamie

Kennedy, Jiechen Zhu, Jayden Tan, Stephanie Markovski. Supervisor: Claudiu Popa. 2021. [Електронний ресурс] https://www.researchgate.net/publication/350818744_Security_Concerns_in_Password_Managers (дата звернення – 02.05.2021).

26. That Was Then, This Is Now: A Security Evaluation of Password Generation, Storage, and Autofill in Thirteen Password Managers. Sean Oesch, Scott Ruoti [Електронний ресурс] https://arxiv.org/abs/1908.03296 (дата звернення – 04.05.2021).
27. Z. Li, W. He, D. Akhawa, і D. Song, «The Emperor's New Password Manager: Security Analysis of Web-based Password Managers», Defense Technical Information Center, Лип 2014 [Електронний ресурс] http://dx.doi.org/10.21236/ada614474 (дата звернення – 04.05.2021).
28. Attacks on Software Application Security | OWASP Foundation [Електронний ресурс] https://owasp.org/www-community/attacks/ (дата звернення – 04.05.2021).
29. OWASP Foundation | Open Source Foundation for Application Security [Електронний ресурс] https://owasp.org/ (дата звернення – 04.05.2021).
30. B. B. Gupta, S. Gupta, S. Gangwar, M. Kumar, і P. K. Meena, «Cross-Site Scripting (XSS) Abuse and Defense: Exploitation on Several Testing Bed Environments and Its Defense», Journal of Information Privacy and Security, вип. 11, вип. 2, с. 118–136, Квіт 2015, doi: 10.1080/15536548.2015.1044865. [Електронний ресурс] http://dx.doi.org/10.1080/15536548.2015.1044865 (дата звернення – 04.05.2021).
31. Cross Site Scripting (XSS) Software Attack | OWASP Foundation [Електронний ресурс] https://owasp.org/www-community/attacks/xss/(дата звернення – 04.05.2021).
32. SQL Injection | OWASP [Електронний ресурс] https://owasp.org/www-community/attacks/SQL_Injection (дата звернення – 04.05.2021).

33. What is NoSQL Injection Attack and How to Prevent It? - Security Boulevard [Електронний ресурс] https://securityboulevard.com/2021/03/what-is-nosql-injection-attack-and-how-to-prevent-it/ (дата звернення – 04.05.2021).
34. No SQL, No Injection? Examining NoSQL Security Aviv Ron, Alexandra Shulman-Peleg, Emanuel Bronshtein. 2015 [Електронний ресурс] https://arxiv.org/ftp/arxiv/papers/1506/1506.04082.pdf (дата звернення – 04.05.2021).
35. NOSQL INJECTION. Presentation [Електронний ресурс] https://owasp.org/www-pdf-archive/GOD16-NOSQL.pdf (дата звернення – 04.05.2021).
36. What is NoSQL Injection Attack and How to Prevent It? [Електронний ресурс] https://www.netsparker.com/blog/web-security/what-is-nosql-injection/ (дата звернення – 04.05.2021).
37. Cross Site Request Forgery (CSRF) | OWASP Foundation [Електронний ресурс] https://owasp.org/www-community/attacks/csrf(дата звернення – 04.05.2021).
38. Brute Force Attack Software Attack | OWASP Foundation [Електронний ресурс] https://owasp.org/www-community/attacks/Brute_force_attack(дата звернення – 05.05.2021).
39. G. Avoine, P. Junod, і P. Oechslin, «Characterization and Improvement of Time-Memory Trade-Off Based on Perfect Tables», ACM Trans. Inf. Syst. Secur., вип. 11, вип. 4, с. 1–22, Лип 2008, doi: 10.1145/1380564.1380565. [Електронний ресурс] http://dx.doi.org/10.1145/1380564.1380565(дата звернення – 05.05.2021).
40. Denial of Service Software Attack | OWASP Foundation [Електронний ресурс] https://owasp.org/www-community/attacks/Denial_of_Service (дата звернення – 05.05.2021).
41. What is an ICMP Flood DDoS Attack? | NETSCOUT [Електронний ресурс] https://www.netscout.com/what-is-ddos/icmp-flood(дата звернення – 05.05.2021).

42. SYN Flood DDoS Attack | Cloudflare [Електронний ресурс] https://www.cloudflare.com/learning/ddos/syn-flood-ddos-attack/(дата звернення – 05.05.2021).
43. "Understanding Denial-of-Service Attacks". US-CERT [Електронний ресурс] https://us-cert.cisa.gov/ncas/tips/ST04-015(дата звернення – 05.05.2021).
44. Manipulator-in-the-middle attack | OWASP [Електронний ресурс] https://owasp.org/www-community/attacks/Manipulator-in-the-middle_attack(дата звернення – 05.05.2021).
45. F. Callegati, W. Cerroni, і M. Ramilli, «Man-in-the-Middle Attack to the HTTPS Protocol», IEEE Secur. Privacy Mag., вип. 7, вип. 1, с. 78–81, Січ 2009, doi: 10.1109/msp.2009.12. [Електронний ресурс] http://dx.doi.org/10.1109/MSP.2009.12(дата звернення – 05.05.2021).
46. Session Prediction Software Attack | OWASP Foundation [Електронний ресурс] https://owasp.org/www-community/attacks/Session_Prediction(дата звернення – 05.05.2021).
47. Session fixation Software Attack | OWASP Foundation [Електронний ресурс] https://owasp.org/www-community/attacks/Session_fixation(дата звернення – 05.05.2021).
48. Password Managers: Attacks and Defenses. David Silver, Suman Jana, Eric Chen, Collin Jackson, and Dan Boneh [Електронний ресурс] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.649.2706&rep=rep1&type=pdf(дата звернення – 05.05.2021).
49. Practical Cryptography for Developers[Електронний ресурс] https://cryptobook.nakov.com(дата звернення – 05.05.2021).
50. P. Rogaway і T. Shrimpton, «Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance», в Fast Software Encryption, Springer Berlin Heidelberg, 2004, с. 371–388 [Електронний ресурс] http://dx.doi.org/10.1007/978-3-540-25937-4_24(дата звернення – 05.05.2021).

51. C. S. Wright, «Bitcoin: A Peer-to-Peer Electronic Cash System», SSRN Journal, 2008, doi: 10.2139/ssrn.3440802. [Електронний ресурс] http://dx.doi.org/10.2139/ssrn.3440802(дата звернення – 05.05.2021).
52. B. Preneel, «Cryptographic hash functions», Eur. Trans. Telecomm., вип. 5, вип. 4, с. 431–448, Вер 2010, doi: 10.1002/ett.4460050406. [Електронний ресурс] http://dx.doi.org/10.1002/ett.4460050406(дата звернення – 05.05.2021).
53. P. Gauravaram, «Security Analysis of salt||password Hashes», в 2012 International Conference on Advanced Computer Science Applications and Technologies (ACSAT), 2012, doi: 10.1109/acsat.2012.49 [Електронний ресурс] http://dx.doi.org/10.1109/ACSAT.2012.49(дата звернення – 05.05.2021).
54. P.-H. Kamp, «LinkedIn Password Leak: Salt Their Hide», Queue, вип. 10, вип. 6, с. 20–22, Чер 2012, doi: 10.1145/2246036.2254400. [Електронний ресурс] http://dx.doi.org/10.1145/2246036.2254400(дата звернення – 05.05.2021).
55. F. F. Yao і Y. L. Yin, «Design and Analysis of Password-Based Key Derivation Functions», IEEE Trans. Inform. Theory, вип. 51, вип. 9, с. 3292–3297, Вер 2005, doi: 10.1109/tit.2005.853307. [Електронний ресурс] http://dx.doi.org/10.1109/TIT.2005.853307(дата звернення – 05.05.2021).
56. A. Biryukov, D. Dinu, і D. Khovratovich, «Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications», в 2016 IEEE European Symposium on Security and Privacy (EuroS&P), 2016, doi: 10.1109/eurosp.2016.31 [Електронний ресурс] http://dx.doi.org/10.1109/EuroSP.2016.31 (дата звернення – 05.05.2021).
57. Password Storage - OWASP Cheat Sheet Series [Електронний ресурс] https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (дата звернення – 06.05.2021).
58. P. Fanfara, E. Dankova, і M. Dufala, «Usage of asymmetric encryption algorithms to enhance the security of sensitive data in secure communication», в 2012 IEEE 10th International Symposium on Applied Machine Intelligence and Informatics (SAMI), 2012, doi: 10.1109/sami.2012.6208959 [Електронний ресурс] http://dx.doi.org/10.1109/SAMI.2012.6208959 (дата звернення – 06.05.2021).

59. S. Chandra, S. Paira, S. S. Alam, і G. Sanyal, «A comparative survey of Symmetric and Asymmetric Key Cryptography», в 2014 International Conference on Electronics, Communication and Computational Engineering (ICECCE), 2014, doi: 10.1109/icecce.2014.7086640 [Електронний ресурс] http://dx.doi.org/10.1109/ICECCE.2014.7086640 (дата звернення – 06.05.2021).
60. E. B. Barker і Q. H. Dang, «Recommendation for Key Management Part 3: Application-Specific Key Management Guidance», National Institute of Standards and Technology, Січ 2015 [Електронний ресурс] http://dx.doi.org/10.6028/NIST.SP.800-57pt3r1(дата звернення – 06.05.2021).
61. «Advanced encryption standard (AES)», National Institute of Standards and Technology, Лис 2001 [Електронний ресурс] http://dx.doi.org/10.6028/NIST.FIPS.197(дата звернення – 06.05.2021).
62. Best Practices for Client Side Encryption - Services for Research - CSC Company Site [Електронний ресурс] https://research.csc.fi/best-practices-for-client-side-encryption (дата звернення – 06.05.2021).
63. ChaCha, a variant of Salsa20. Daniel J. Bernstein [Електронний ресурс] http://cr.yp.to/chacha/chacha-20080120.pdf (дата звернення – 06.05.2021).
64. Commerce Secretary Announces New Standard for Global Information Security [Електронний ресурс] https://www.nist.gov/news-events/news/2001/12/commerce-secretary-announces-new-standard-global-information-security (дата звернення – 06.05.2021).
65. Transport Layer Protection - OWASP Cheat Sheet Series [Електронний ресурс] https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Protection_Cheat_Sheet.html (дата звернення – 06.05.2021).
66. A. E. W. Eldewahi, T. M. H. Sharfi, A. A. Mansor, N. A. F. Mohamed, і S. M. H. Alwahbani, «SSL/TLS attacks: Analysis and evaluation», в 2015 International Conference on Computing, Control, Networking, Electronics and Embedded

Systems Engineering (ICCNEEE), 2015, doi: 10.1109/iccneee.2015.7381362 [Електронний ресурс] http://dx.doi.org/10.1109/ICCNEEE.2015.7381362 (дата звернення – 07.05.2021).

67. CRIME Attack Uses Compression Ratio of TLS Requests as Side Channel to Hijack Secure Sessions [Електронний ресурс] https://threatpost.com/crime-attack-uses-compression-ratio-tls-requests-side-channel-hijack-secure-sessions-091312/77006/ (дата звернення – 07.05.2021).
68. HTTP & WWW: Website URLs Explained [Електронний ресурс] https://wpengine.com/resources/http-vs-www-urls-for-seo/ (дата звернення – 07.05.2021).
69. Authentication vs. Authorization | Okta [Електронний ресурс] https://www.okta.com/identity-101/authentication-vs-authorization/ (дата звернення – 07.05.2021).
70. HTTP authentication - HTTP | MDN [Електронний ресурс] https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication (дата звернення – 07.05.2021).
71. Reschke, J., "The 'Basic' HTTP Authentication Scheme", RFC 7617, DOI 10.17487/RFC7617, September 2015 [Електронний ресурс] https://tools.ietf.org/html/rfc7617 (дата звернення – 07.05.2021).
72. The Java EE 6 Tutorial, Volume I. Chapter 25 Getting Started Securing Web Applications. [Електронний ресурс] https://docs.oracle.com/cd/E19226-01/820-7627/6nisfjn89/index.html (дата звернення – 07.05.2021).
73. Usability: Allow users to clear HTTP Basic authentication details ('Logout') . [Електронний ресурс] https://bugzilla.mozilla.org/show_bug.cgi?id=540516 (дата звернення – 07.05.2021).

74. Shekh-Yusef, R., Ed., Ahrens, D., and S. Bremer, "HTTP Digest Access Authentication", RFC 7616, DOI 10.17487/RFC7616, September 2015 [Електронний ресурс] https://tools.ietf.org/html/rfc7616 (дата звернення – 07.05.2021).
75. Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012 [Електронний ресурс] https://tools.ietf.org/html/rfc6750 (дата звернення – 07.05.2021).
76. RFC 7519 - JSON Web Token (JWT) - IETF Tools [Електронний ресурс] https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення – 07.05.2021).
77. The OAuth 2.0 Authorization Framework (RFC) [Електронний ресурс] https://datatracker.ietf.org/doc/html/rfc6749 (дата звернення – 07.05.2021).
78. Final: OpenID Connect Core 1.0 incorporating errata set 1 [Електронний ресурс] https://openid.net/specs/openid-connect-core-1_0.html (дата звернення – 07.05.2021).
79. Swagger API Keys [Електронний ресурс] https://swagger.io/docs/specification/authentication/api-keys/ (дата звернення – 07.05.2021).
80. The RapidBlog API. API Key – What is an API Key? [Електронний ресурс] https://rapidapi.com/blog/api-glossary/api-key/ (дата звернення – 07.05.2021).
81. Why and when to use API keys. [Електронний ресурс] https://cloud.google.com/endpoints/docs/openapi/when-why-api-key (дата звернення – 09.05.2021).
82. What are extensions? - Mozilla | MDN [Електронний ресурс] https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/What_are_WebExtensions (дата звернення – 09.05.2021).
83. Chrome Developers. What are extensions? - Chrome Developers [Електронний ресурс] https://developer.chrome.com/docs/extensions/mv2/overview/ (дата звернення – 09.05.2021).
84. About Manifest V2 - Chrome Developers Developers [Електронний ресурс]

https://developer.chrome.com/docs/extensions/mv2/
(дата звернення – 09.05.2021).

85. Strong Passwords - IS&T Contributions - Hermes [Електронний ресурс] http://kb.mit.edu/confluence/display/istcontrib/Strong+Passwords(дата звернення – 09.05.2021).
86. ACID properties of transactions - IBM [Електронний ресурс] https://www.ibm.com/docs/en/cics-ts/5.4?topic=processing-acid-properties-transactions (дата звернення – 09.05.2021).
87. Document Database | MongoDB [Електронний ресурс] https://www.mongodb.com/document-databases (дата звернення – 09.05.2021).
88. Database Scaling : Horizontal and Vertical Scaling | Hacker Moon [Електронний ресурс] https://hackernoon.com/database-scaling-horizontal-and-vertical-scaling-85edd2fd9944 (дата звернення – 09.05.2021).
89. 10 Most Popular Two-Factor Authentication Apps Compared [Електронний ресурс] https://www.protectimus.com/blog/10-most-popular-2fa-apps-on-google-play/ (дата звернення – 09.05.2021).
90. Set-Cookie - HTTP | MDN [Електронний ресурс] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie (дата звернення – 09.05.2021).
91. Using HTTP cookies - HTTP | MDN Moon [Електронний ресурс] https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies (дата звернення – 09.05.2021).
92. Cross-Site Request Forgery Prevention - OWASP Cheat Sheet [Електронний ресурс] https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html (дата звернення – 09.05.2021).

93. Bootstrap - The most popular HTML, CSS, and JS library in the world [Електронний ресурс] https://getbootstrap.com/ (дата звернення – 09.05.2021).
94. MySQL Database Service is a fully managed database service to deploy cloud native applications. [Електронний ресурс] https://www.mysql.com/ (дата звернення – 09.05.2021).
95. MariaDB Foundation - MariaDB.org [Електронний ресурс] https://mariadb.com/ (дата звернення – 09.05.2021).
96. MongoDB: The most popular database for modern apps [Електронний ресурс] https://www.mongodb.com/ (дата звернення – 09.05.2021).
97. Redis https://redis.io/ (дата звернення – 09.05.2021).
98. Stack Overflow Developer Survey 2020 - Stack Overflow Insights Wirecutter [Електронний ресурс] https://insights.stackoverflow.com/survey/2020 (дата звернення – 09.05.2021).
99. JVM Ecosystem Report 2020 – Snyk Wirecutter [Електронний ресурс] https://snyk.io/wp-content/uploads/jvm_2020.pdf (дата звернення – 09.05.2021).
100. Gradle vs Maven: Performance Comparison - Gradle [Електронний ресурс] https://gradle.org/gradle-vs-maven-performance/ (дата звернення – 09.05.2021).
101. Ktor: Build Asynchronous Servers and Clients in Kotlin [Електронний ресурс] https://ktor.io/ (дата звернення – 09.05.2021).
102. Netty: Home [Електронний ресурс] https://netty.io/ (дата звернення – 10.05.2021).
103. Logging | Ktor [Електронний ресурс] https://ktor.io/docs/logging.html (дата звернення – 10.05.2021).
104. kosprov/jargon2-api: Fluent Java API for Argon2 – GitHub [Електронний ресурс] https://github.com/kosprov/jargon2-api (дата звернення – 10.05.2021).
105. marcelkliemannel/kotlin-onetimepassword – GitHub [Електронний ресурс]

https://github.com/marcelkliemannel/kotlin-onetimepassword
(дата звернення – 10.05.2021).

106. Authentication and authorization | Ktor [Електронний ресурс] https://ktor.io/docs/authentication.html (дата звернення – 10.05.2021).

107. ben-manes/caffeine: A high performance caching library – GitHub [Електронний ресурс] https://github.com/ben-manes/caffeine (дата звернення – 10.05.2021).

108. marshallpierce / ktor-csrf — Bitbucket [Електронний ресурс] https://bitbucket.org/marshallpierce/ktor-csrf (дата звернення – 10.05.2021).

109. Resilience4j is a fault tolerance library for Java™ [Електронний ресурс] https://resilience4j.readme.io/ (дата звернення – 10.05.2021).

110. antelle/argon2-browser: Argon2 library compiled for browser runtime – GitHub [Електронний ресурс] (дата звернення – 10.05.2021). https://github.com/antelle/argon2-browser (дата звернення – 10.05.2021).

111. ricmoo/aes-js: A pure JavaScript implementation of the AES block cipher – GitHub [Електронний ресурс] https://github.com/ricmoo/aes-js (дата звернення – 10.05.2021).

112. Crypto-JS - Google Code Archive [Електронний ресурс] https://code.google.com/archive/p/crypto-js/#AES (дата звернення – 10.05.2021).

113. brendanashworth/generate-password: NodeJS - GitHub [Електронний ресурс] https://www.npmjs.com/package/generate-password
(дата звернення – 14.05.2021).

114. davidshimjs/qrcodejs: Cross-browser QRCode Generator – GitHub https://github.com/davidshimjs/qrcodejs (дата звернення – 14.05.2021).

ДОДАТОК 1 – ЛІСТИНГ КОДУ СЕРВІСУ ТОКЕНІВ-ДОСТУПУ

JWTAccessTokenService.kt

```
17  class JWTAccessTokenService() {
18      private val envConf = HoconApplicationConfig(ConfigFactory.load())
19      private val jwtSecret = envConf.property( path: "ktor.security.jwt.access_token
20      private val jwtIssuer = envConf.property( path: "ktor.jwt.issuer").getString()
21      private val validityInMin = envConf.property( path: "ktor.jwt.validity_in_min")
22      private val algorithm = Algorithm.HMAC512(jwtSecret)
23
24      val blackList: Cache<String, Any?> = Caffeine.newBuilder()
25          .expireAfterWrite(validityInMin, TimeUnit.MINUTES)
26          .build()
27
28      fun generateToken(publicId: String): String {
29          val currentTime = Instant.now()
30          val expiration = getExpiration(currentTime.toEpochMilli())
31          val JWTId = generateUUIDv4InBase64WithoutPadding()
32          return JWT.create()
33              .withJWTId(JWTId)
34              .withExpiresAt(expiration)
35              .withNotBefore(Date.from(currentTime))
36              .withIssuedAt(Date.from(currentTime))
37              .withSubject(publicId)
38              .withIssuer(jwtIssuer)
39              .withClaim( name: "expires",  value: validityInMin * 60)
40              .sign(algorithm)
41      }
42      fun verifierFunc(): JWTVerifier {
43          return JWT
44              .require(algorithm)
45              .withIssuer(jwtIssuer)
46              .build()
47      }
48      fun validateFunc(cred: JWTCredential): JWTPrincipal? {
49          val jwtId: String = cred.payload.id
50          if (blackList.getIfPresent(jwtId) ≠ null) {
51              return null
52          }
53          return JWTPrincipal(cred.payload)
54      }
55      private fun getExpiration(currentTimeInMs: Long) =
56          Date( date: currentTimeInMs + Duration.ofMinutes(validityInMin).toMillis())
```

ДОДАТОК 2 – ЛІСТИНГ КОДУ СЕРВІСУ ПЕРШОГО ЕТАПУ АВТЕНТИФІКАЦІЇ

```
private val lightBlockForAuth: Cache<String, Any?> = Caffeine.newBuilder()
    .expireAfterWrite(lightBlockDurationInMinutes, TimeUnit.MINUTES)
    .build()
private val hardBlockForAuth: Cache<String, Any?> = Caffeine.newBuilder()
    .expireAfterWrite(hardBlockDurationInMinutes, TimeUnit.MINUTES)
    .build()

fun onUserLogin(userAuthInfo: RecOb.UserAuthInfo, ipAddress: String, userAgent: String
        : ResOb.Login {
    val userFromDB = UsersDAO.getUserByEmail(userAuthInfo.email)
        ?: throw InvalidCredentialsException()
    validateAuthAllowance(userAuthInfo.email, userFromDB.timeOfLastFailedLogin)

    try {
        passwordCheck(userFromDB, userAuthInfo.passHash)
    } catch (e: InvalidCredentialsException) {
        onInvalidLoginAttempt(userFromDB)
    }
    val clientIdIsNew = !UserDevicesDAO.getAllDevicesForUser(userFromDB.internalId)
        .any { it.clientId == userAuthInfo.clientId }
    if (!userFromDB.emailVerified)
        throw EmailNotVerifiedException()
    if (clientIdIsNew)
        EmailService.sendEmailNewClientNotification(
            userAuthInfo.email, ipAddress, userAgent
        )
    failedAttemptsZeroing(userFromDB)
    val tokenPair = updatedTokensForUser(userFromDB, userAuthInfo.clientId)
    if (clientIdIsNew && userFromDB.twoFactorEnabled) {
        val randomGeneratedCode = SecureRandomString.generate( byteArrayLength: 32)
        totpService.addEntryToPostponedAuth(
            randomGeneratedCode,
            userFromDB.internalId,
            tokenPair,
            ipAddress,
            userAgent
        )
        return ResOb.Login(`2fa_type` = "totp", code = randomGeneratedCode)
    }
    return ResOb.Login(tokenPair.accessToken, tokenPair.refreshToken)
```

ДОДАТОК 3 – ЛІСТИНГ КОДУ СЕРВІСУ ДРУГОГО ЕТАПУ АВТЕНТИФІКАЦІЇ

TotpService.kt

```kotlin
class TotpService(private val authService: AuthService) {
    private val envConf = HoconApplicationConfig(ConfigFactory.load())

    class PostponedAuthInfo(
        val userId: String, val tokenPair: AuthService.TokenPair,
        val ipAddress: String, val userAgent: String
    )
    private val postponedAuthCache: Cache<String, PostponedAuthInfo> =
        Caffeine.newBuilder()
            .expireAfterWrite(
                envConf.property( path: "ktor.postponed_auth.validity_in_min")
                    .getString().toLong(),
                TimeUnit.MINUTES
            )
            .build()
    fun totpAuth(code: String, totpValue: String): ResOb.Login {
        try {
            val cacheEntry = postponedAuthCache.getIfPresent(code)!!
            val user = UsersDAO.read(UserKey(cacheEntry.userId))!!
            totpCheck(user, totpValue)
            postponedAuthCache.invalidate(code)
            EmailService.sendEmailNewClientNotification(
                user.email,
                cacheEntry.ipAddress, cacheEntry.userAgent
            )
            return ResOb.Login(
                cacheEntry.tokenPair.accessToken, cacheEntry.tokenPair.refreshToken
            )
        } catch (e: Exception) {
            throw IllegalArgumentException()
        }
    }
    private fun totpCheck(userFromDB: User, totpValue: String) {
        val totpSecret = userFromDB.totpSecret!!
        val twoFAGenerator = GoogleAuthenticator(totpSecret)
        if (!(twoFAGenerator.isValid(totpValue, Date(System.currentTimeMillis())) ||
                    twoFAGenerator.isValid(totpValue,
                        Date( date: System.currentTimeMillis() - 5000)))) {
            throw AuthenticationException()
        }
```

ДОДАТОК 4 – ЛІСТИНГ КОДУ З ФАЙЛУ MANIFEST.JSON

```
{
  "manifest_version": 2,
  "name": "PM",
  "description": "Advanced password-manager with advanced password-manager backend",
  "version": "0.1.0",
  "icons": {
    "16": "icons/favicon-16×16.png",
    "32": "icons/favicon-32×32.png",
    "48": "icons/favicon-48×48.png",
    "128": "icons/favicon-128×128.png"
  },
  "permissions": [
    "activeTab",
    "tabs",
    "storage",
    "unlimitedStorage",
    "https://*/*"
  ],
  "browser_action": {
    "default_icon": "icons/favicon-32×32.png",
    "default_title": "Password Manager",
    "default_popup": "popup/popup.html"
  },
  "background": {
    "persistent": true,
    "scripts": ["background/background.js"]
  },
  "content_security_policy": "script-src https://cdn.jsdelivr.net/ 'self';"
}
```

ДОДАТОК 5 – ЛІСТИНГ КОДУ ШИФРУВАННЯ, ДЕШИФРУВАННЯ ТА ЗБЕРІГАННЯ КОРИСТУВАЦЬКОГО СХОВИЩА НА КЛІЄНТІ

popup.js

```
async function encryptPass(email, password, enKey) {
    let hash = (await argon2.hash(
        {
            pass: enKey, salt: email, hashLen: 32,
            time: 1, mem: 64, parallelism: 2,
            type: argon2.ArgonType.Argon2id
        })).hash;
    let iv = hash.splice(0, 16);
    let key = hash.splice(-16);
    let passInBytes = aesjs.utils.utf8.toBytes(password);
    let aesCbc = new aesjs.ModeOfOperation.cbc(key, iv);
    let encryptedBytes = aesCbc.encrypt(passInBytes);
    let encryptedHex = aesjs.utils.hex.fromBytes(encryptedBytes);
    return encryptedHex;
}

async function decryptPass(email, passwordInHex, enKey) {
    let encryptedBytes = aesjs.utils.hex.toBytes(passwordInHex);
    let hash = (await argon2.hash(
        {
            pass: enKey, salt: email, hashLen: 32,
            time: 1, mem: 64, parallelism: 2,
            type: argon2.ArgonType.Argon2id
        })).hash;
    let iv = hash.splice(0, 16);
    let key = hash.splice(-16);
    let aesCbc = new aesjs.ModeOfOperation.cbc(key, iv);
    let decryptedBytes = aesCbc.decrypt(encryptedBytes);
    let decryptedText = aesjs.utils.utf8.fromBytes(decryptedBytes);
    return decryptedText;
}

async function creatNewRecord(name, username, password) {
    let enPass = encryptPass(password);
    let updatedRecord = {name: name, username: username, password: enPass};
    let newId = "record-" + uuidv4();
    setNewValueToStorage(newId, updatedRecord);
```

```
222  function updateRecord(recId, name, username, password, isGivenPassEncrypted) {
223      let updatedRecord = {
224          name: name, username: username,
225          password: isGivenPassEncrypted ? password : encryptPass(password)
226      }
227      chrome.storage.local.set({recId: updatedRecord}, function () {
228          onStorageChange();
229      });
230  }
231
232  async function getRecordById(recordId) {
233      return new Promise( executor: (resolve, reject) => {
234          chrome.storage.local.get([recordId], function (result) {
235              if (Object.values(result)[0] !== undefined) {
236                  resolve(Object.values(result)[0].val);
237              } else {
238                  reject();
239              }
240          });
241      });
242  }
243
244  async function getRecords() {
245      return new Promise( executor: (resolve, reject) => {
246          chrome.storage.local.get(null, function (items) {
247              if (Object.values(result)[0] !== undefined) {
248                  let allKeys = Object.keys(items);
249                  let res = allKeys.filter((e : string ) =>
250                      e.value.substring(0, 7) === "record-");
251                  resolve(res);
252              } else {
253                  reject();
254              }
255          });
256      });
257  }
```

```
259  function setNewValueToStorage(recordId, newRecord) {
260      chrome.storage.local.set({recordId: newRecord}, function () {
261          onStorageChange();
262      });
263  }
264
265  function remValueFromStorage(recordId) {
266      chrome.storage.local.remove(recordId, function () {
267          onStorageChange();
268      });
269  }
```