Міністерство освіти і науки України
Харківський національний університет імені В.Н. Каразіна
Навчально-науковий інститут комп'ютерних наук та штучного інтелекту
Спеціальність 125 «Кібербезпека»
Освітня програма «Кібербезпека»

В.о. зав. кафедрою КІСМТ
Марина ЄСІНА
"Допущено до захисту"
« »_____2025 р.

Пояснювальна записка

до кваліфікаційної роботи бакалавра

на тему: «Дослідження, аналіз та порівняння методів і засобів для забезпечення роботи системи автентифікації у мобільних застосунках»

оцінка « _____ »

Голова ЕК

Мичуда Л. З.

Керівник: к.т.н., доцент Єсіна М. В.

Рецензент: к.т.н., професор Качко О. Г.

Виконавець: студент групи КБ-42

Колєсніков О. Є.

Харків – 2025

РЕФЕРАТ

Пояснювальна записка до проєкту бакалавра містить 60 сторінок, 15 рисунків, 3 таблиці, 4 додатки, 26 посилань на джерела.

Мета роботи полягає в дослідженні та аналізі впровадження методів автентифікації для мобільних застосунків із ОС Android, бібліотек та платформ, що використовуються для впровадження та реалізації систем автентифікації у мобільних застосунках, а також огляд існуючих методів автентифікації сьогодення.

Об’єкт дослідження – процес забезпечення безпечної автентифікації користувачів у мобільних застосунках.

Предмет дослідження – методи, інструменти та програмні засоби реалізації систем автентифікації в мобільних застосунках, їх функціональні можливості, переваги, недоліки та особливості впровадження.

Основними методами дослідження є аналіз та порівняння основних інструментів та методів запровадження системи автентифікації. Також реалізація системи автентифікації у мобільному застосунку. Тестування розробленої системи на вразливості

У роботі досліджено: різні види механізмів автентифікації, їх переваги та недоліки, платформи та бібліотеки, які можуть бути використані для реалізації системи автентифікації, спроектована та розроблена система автентифікації у мобільному застосунку, а також, тестування безпеки розробленої системи автентифікації.

Результати роботи можуть бути використані у різних наукових виданнях, а також для кращого розуміння механізмів і систем автентифікації у мобільних застосунках.

Ключові слова: SFA, 2FA, MFA, МЕХАНІЗМИ АВТЕНТИФІКАЦІЇ, АВТЕНТИФІКАЦІЯ, ВРАЗЛИВОСТІ, АТАКА, БЕЗПЕКА, ПРОЦЕС АВТЕНТИФІКАЦІЇ, КОРИСТУВАЧ, ОБЛІКОВИЙ ЗАПИС, РОЗРОБНИК.

ABSTRACT

The explanatory note to the bachelor's project contains 60 pages, 15 figures, 3 tables, 4 appendices, 26 references to sources.

The purpose of the work is to study and analyze the implementation of authentication methods for mobile applications with Android OS, libraries and platforms used to implement and implement authentication systems in mobile applications, as well as a review of existing authentication methods today.

The object of the study is the process of ensuring secure user authentication in mobile applications.

The subject of the study is methods, tools and software for implementing authentication systems in mobile applications, their functionality, advantages, disadvantages and implementation features.

The main research methods are the analysis and comparison of the main tools and methods for implementing the authentication system. Also, the implementation of the authentication system in a mobile application. Vulnerability testing of the developed system

The work investigated: different types of authentication mechanisms, their advantages and disadvantages, platforms and libraries that can be used to implement the authentication system, designed and developed an authentication system in a mobile application, as well as security testing of the developed authentication system.

The results of the work can be used in various scientific publications, as well as for a better understanding of authentication mechanisms and systems in mobile applications.

Keywords: SFA, 2FA, MFA, AUTHENTICATION MECHANISMS, AUTHENTICATION, VULNERABILITY, ATTACK, SECURITY, AUTHENTICATION PROCESS, USER, ACCOUNT, DEVELOPER.

ЗМІСТ

ПЕРЕЛІК ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

ОС	– операційна система
SSO	– Single Sign-On
TLS	– Transport Layer Security
HTTP	– HyperText Transfer Protocol
NIST	– The National Institute of Standards and Technology
FIPS	– Federal Information Processing Standards
ISO	– International Organization for Standardization
HTTPS	– HyperText Transfer Protocol Secure
API	– Application programming interface
HMAC	– Hash-based message authentication code
BaaS	– Backend as service
MVVM	– Model-View-ViewModel
UI	– User Interface
SDK	– Software Development Kit

ВСТУП

У сучасному світі людство кожен день активно використовує цифрові технології, їхній розвиток та вдосконалення все більше впливає на повсякденне життя суспільства. Мобільні пристрої стали одними із найпопулярніших і найбільш використовуваних гаджетів, які є майже у кожної людини на планеті. Їхнє використання не обмежується просто комунікацією, як це було багато років тому, а використовуючи телефон сьогодні, можна виконати майже будь-що, наприклад: оплачувати комунальні послуги через онлайн-банкінг, робити замовлення будь-яких товарів прямо із доставкою додому, керувати функціями розумного будинку або спілкуватися із друзями та родичами з будь-якої точки Землі та ще багато іншого. Незважаючи на всі переваги використання такого багатого функціоналу, є і свої нюанси, які насамперед криються у безпеці мобільних застосунків, за допомогою яких користувач і виконує усі дії. Основною ціллю кіберзлочинців є персональні дані користувачів, за допомогою яких можна отримати доступ до особистих рахунків, хмарних сховищ, даних та будь-чого, де фігурує особа. Саме тому захист персональних даних у мобільних застосунках є пріоритетним питанням сьогодення.

Кіберзлочинці використовують широкий спектр атак, які спрямовані не лише на систему користувача, але і на саму особу, наприклад: фішингові атаки, соціальна інженерія, перехоплення даних через незашифровані канали (атаки типу "людина посередині"), реверс-інжиніринг або компрометація автентифікаційних даних та багато іншого. Вразливості у застосунках розвиваються в одному потоці із розвитком самих технологій, тому кожен день ми повинні бути напоготові та захищати свої дані у різний спосіб.

Автентифікація є одним із перших бар’єрів безпеки, із яким зустрічається як користувач, так і зловмисник. Існують такі види автентифікації, як однофакторна та багатофакторна.

Однофакторна автентифікація передбачає використання пароля для доступу до ресурсу, вона є найбільш поширеною формою захисту продукту, однак в той

самий час і однією із найвразливіших. Використання простих та популярних паролів і є проблемою незахищеності систем із однофакторною автентифікацією, тому з метою посилення безпеки розробники мобільних застосунків переходять до використання багатофакторної автентифікації.

Багатофакторна автентифікація гарантує надійніший захист завдяки використанню багаторівневої системи автентифікації, а саме: паролі у поєднанні із одноразовими кодами підтвердження, біометричні дані або кодові питання та слова, та ще багато рішень, за допомогою яких можна захистити персональні дані користувача.

Таким чином, актуальність даної роботи обумовлена стрімким розвитком кіберзагроз, через які потрібно впроваджувати сучасні та надійні методи автентифікації, зручні для користувачів та стійкі перед загрозами.

Метою дипломної роботи є дослідження, аналіз та впровадження методів автентифікації для мобільних застосунків із ОС Android.

Задачі для виконання містять такі пункти:

- Аналіз існуючих методів автентифікації, їхні переваги та недоліки.
- Дослідження вразливостей систем автентифікації.
- Розробка вимог до захищеної системи автентифікації, яка відповідає сучасним стандартам.
- Розробка системи автентифікації за допомогою бібліотек Firebase, Google Authenticator та біометричних даних.
- Тестування розробленої та дослідженої системи автентифікації, та оцінка її ефективності.

1 ОГЛЯД, АНАЛІЗ ТА ДОСВІД ЗАСТОСУВАННЯ МЕХАНІЗМІВ АВТЕНТИФІКАЦІЇ У МОБІЛЬНИХ ЗАСТОСУНКАХ

1.1 Огляд історії, розвитку та використання методів автентифікації

У 1979 році Роберт Морріс і Кен Томпсон опублікували першу наукову статтю на тему безпеки на основі паролів. Вони представили експериментальний аналіз вибору паролів користувачами, провівши словникові атаки на реальну систему, відновивши 81% з 3000 паролів користувачів. Причиною такого вдалого відновлення було те, що користувачі обирали дуже короткі рядки, оскільки проблема безпеки, унікальності та індивідуальності паролів не була широко зрозумілою. Девід Фельдмайєр і Філіп Карн повторили цей аналіз через 10 років з подальшими експериментами і встановили, що вони все ще можуть зламати 30% системних паролів, використовуючи невеликі словники, що було оцінено як значне покращення [1].

Пізніше, у 1990 році, Деніел Клейн, здійснюючи подібні експерименти, зміг зламати 25% паролів, а Юджин Спаффорд у 1992 році аналогічним чином зламав ще менше – 20% паролів. Подальші експерименти показали постійне покращення у виборі паролів користувачами. Однак використання Інтернету змінило цю тенденцію. Так, у 2006 році стався витік паролів з MySpace і було експериментально виявлено, що паролі стали слабшими, а в 2007 році дослідження Казьєра і Медліна показало, що вони змогли зламати майже 99% паролів на реальному веб-сайті електронної комерції з необмеженим доступом для вгадування [1].

У той же час, способи підбору паролів стали набагато складнішими. John the Ripper створювався як універсальна бібліотека для злому паролів більше десяти років і зараз містить словники з мільйонами відомих паролів. У 2003 році Оехслін представив райдужні таблиці як значно покращений компроміс між часом і пам’яттю для злому паролів. Пізніше цей варіант був вдосконалений Нараянаном та ін. шляхом обліку реальної статистики паролів. Останні дослідження Вейра та ін. демонструють, що звичайні словники паролів можна зробити сильнішими,

автоматично генеруючи модифікації на основі спостережуваних моделей вибору паролів користувачами [1].

Відійшовши від гонки озброєнь зі злому паролів, Енн Адамс і Мартіна-Ангела Сассе в 1999 році провели низку досліджень про використання паролів і поведінку корпоративних співробітників при взаємодії з ними. Виявилося, що велика кількість користувачів записують особисті паролі в особистих щоденниках, блокнотах і т.д., і легко піддаються атакам соціальної інженерії. На основі результатів дослідження Бростофф запропонував потенційне вирішення проблеми, а саме: дати користувачам більше трьох спроб запам’ятати свої паролі, щоб запобігти дорогому ручному оновленню паролів.

У 2006 році Шеннон Райлі здійснив дослідження, схоже на те, що проводили Адамс і Сассе у 1999 році. Він встановив, що проблема використання невеликої кількості паролів для облікових записів користувачів (паролі можуть повторюватися) стає все більш значущою. Того ж року Ширлі Гау та Едвард Фелтен опублікували велике масштабне дослідження, яке виявило дуже високу частоту використання одного і того ж пароля для різних облікових записів і сторінок. Експеримент проводився в лабораторії, де Гау і Фелтен помітили, що користувачі не могли увійти до значної кількості облікових записів через те, що забували свої паролі і намагалися використовувати один і той самий ключ [1].

Епохальне дослідження, проведене Флоренсіо та Херлі у 2007 році, представило масштабні дані, зібрані через панель інструментів браузера. Цифри були вражаючими: середньостатистичний веб-користувач мав 25 окремих облікових записів і лише 6,5 паролів. Вони також припустили, що до 1% користувачів великого веб-сайту заходять, щоб змінити свій пароль протягом місяця [1].

Окрім повторного використання та забуття паролів, в дослідженнях серед користувачів постійно з’являються нові практики порушення безпеки. Значна кількість людей записують паролі, ніколи не змінюючи їх, і створюють їх на основі персональної інформації. Рівень обізнаності та використання автоматизованих

методів зберігання та введення паролів залишається вкрай низьким. Спільне використання паролів між людьми є надзвичайно поширеним явищем.

Для підвищення безпеки введення та запам’ятовування паролів було використано кілька методів. У 1991 році Вільям Хага та Моше Звіран встановили, що системи нагадування, в яких користувачеві ставлять запитання, а потім просять ввести пароль, дозволяють запам’ятовувати паролі, які складно вгадати. У 2000 році Понд та ін. розвинули цей підхід у багатослівну систему з більшою безпекою. Однак дослідження показало, що більше третини учасників забували свої паролі через два тижні, а 8% паролів були зламані зловмисниками.

Доступнішою альтернативою є мнемонічні паролі, які перетворюють речення, яке потрібно запам’ятати, на короткий пароль. Реальні дослідження 2000 року показали, що це значно зменшує ймовірність вгадування, не впливаючи на запам’ятовування та згадування. Однак подальші дослідження показали, що мнемонічні словники, засновані на базах даних цитат, фільмів і текстів пісень, були ефективними для вгадування мнемонічних паролів [1].

Більш простим підходом було переконати користувача вибрати безпечніший текстовий пароль. У 2001 році цей метод вважався найекономічнішим, а Річард Конлан та інші дослідники продемонстрували, що графічні показники значно підвищують надійність пароля. Людська пам’ять краще пристосована для запам’ятовування досвіду та емоцій, тому було запропоновано використовувати графічні паролі. Джермін та інші запропонували використовувати графічні схеми та візуальні сітки для підбору паролів [1].

Для боротьби з фішинговими атаками були запропоновані когнітивні паролі, за допомогою яких користувач міг вирахувати відповідь на запит, приховуючи при цьому певний секрет від зловмисників. Вайншалл представив практичну схему у 2006 році, але вже за рік її було зламано.

Новим етапом у безпеці парольних систем стали системи єдиного входу (Single Sign-On, SSO). Вони дозволяли користувачеві реєструвати єдиний пароль на довіреному сервері, який міг автентифікувати користувачів для входу в різні онлайн-сервіси. Історичним попередником SSO був протокол Kerberos,

розроблений для проєкту MIT Athena в 1980-х роках і стандартизований в RFC. Kerberos використовує симетричне шифрування для автентифікації користувача на довіреному сервері ключів, який надає сеансовий ключ [1].

Хоча Kerberos був оптимізований для TLS, його найпоширеніше застосування в Інтернеті використовує файли cookie та перенаправлення HTTP. Подібні протоколи, такі як Central Authentication Service (Єльський університет) і WebAuth (Стенфордський університет) були розроблені для внутрішнього використання в організаціях. Шибболет розвинув цю ідею, щоб спростити взаємодію між організаціями, але в основному вона була впроваджена в академічному середовищі.

Серед комерційних рішень SSO – OpenID, який дозволяє одному веб-серверу автентифікувати користувачів на іншому. Протокол OpenID, який підтримується некомерційним фондом, був прийнятий великими веб-сайтами, незважаючи на критику щодо його вразливості до фішингу. Подібний протокол OAuth дозволяє передавати дані користувачів між серверами і часто використовується в поєднанні з SSO.

Через складність впровадження SSO з’явилися автоматизовані системи управління паролями, які спрощують автентифікацію користувачів без зміни серверної інфраструктури. Габбер та ін. запропонували Janus у 1997 році, який використовував проксі-сервер для автоматичного заповнення надійних паролів для користувачів. Хоча такі сервіси так і не були створені, програмне забезпечення для управління паролями стало стандартним, починаючи з KeyChain від Apple у 1999 році [1].

Відсутність єдиного та універсального стандарту для використання паролів на комерційних веб-сайтах є справжньою проблемою. Різні державні стандарти, такі як FIPS 112 (1985), надають конкретні вказівки щодо шифрування паролів під час передачі та зберігання, вимагаючи періодичної зміни паролів та дотримання мінімальної довжини пароля. Однак ці стандарти з’явилися раніше, ніж споживчий Інтернет, і їх критикують за неактуальність і застарілість рекомендацій.

Стандарт NIST SP 800-63 (2005) надає більш широкі рекомендації, визначаючи чотири рівні безпеки для віддаленої автентифікації. Вимоги до безпеки

паролів відповідають рівню ризику, але деталі реалізації залишаються на розсуд розробників. Світові стандарти, такі як ISO 27001 (2005), передбачають використання надійних паролів, але не містять детальних рекомендацій [1].

Попри ці настанови, такі поширені проблеми, як процедури скидання паролю та атаки грубої сили, залишаються нерозглянутими в багатьох стандартах. Більшість з них покладаються на адміністративні служби підтримки, а не на автоматичні захисні рішення [1].

1.2 Короткий опис сучасних методів автентифікації, таких як SFA, 2FA та MFA

У сучасному світі, який стає все більше цифровим, безпека онлайн-облікових записів і персональних даних є важливішою, ніж будь-коли. Однофакторна автентифікація (SFA) є одним з найпростіших методів захисту облікових записів від несанкціонованого доступу. Хоча SFA може здатися досить простою для реалізації та розуміння, вона є невід'ємною частиною сучасної безпеки та захисту.

Однофакторна автентифікація (SFA) – це метод безпеки, який використовує лише один рівень перевірки для підтвердження особи користувача при спробі отримати доступ до ресурсу. Як правило, SFA ідентифікує щось, що відомо користувачеві, наприклад, пароль або PIN-код, що робить однофакторну автентифікацію автентифікацією на основі пароля [2].

Проте зі стрімким розвитком сучасних технологій аналогічно зростають і кіберзагрози. Для деяких програм однієї SFA може бути недостатньо, тому зараз використовується двофакторна автентифікація (2FA), яка забезпечує більш надійний захист застосунків.

Двофакторна автентифікація (2FA) – це більш безпечний спосіб підтвердження особи користувача. Замість того, щоб використовувати лише один тип облікових даних, 2FA використовує другий рівень, а саме: одноразовий пароль, підтвердження електронною поштою або SMS, електронні ключі, мобільні застосунки тощо [4].

Види автентифікації за типом «володіння»:

- те, що знає користувач, наприклад: пароль або PIN-код;

- те, чим володіє користувач, наприклад: номер телефону, електронна адреса, кредитна картка, тощо;

- те, чим є користувач, насамперед це біометричні дані, такі як: відбитки пальців, сканування обличчя або голосовий аналізатор [5].

2FA – це використання різних комбінацій наведених форм даних, і чим більш несподіваними і унікальнішими будуть ці комбінації, тим складніше буде скомпрометувати дані користувача зловмиснику та отримати доступ до чутливої інформації.

У повсякденному житті, користувачі мобільних пристроїв і не тільки, частіше за все використовують 2FA, для забезпечення безпеки своїх облікових записів, бо двофакторна автентифікація досить нескладна та надійна у порівнянні із простим використанням паролю. Також, слід зазначити, що впровадження та використання 2FA не додає більших клопотів, що також є вагомою перевагою, коли людина цінить час.

Але, якою б доброю не була двофакторна автентифікація, її все ж таки не рекомендується використовувати у якихось урядових установах або на стратегічно важливих комплексах. Для таких важливих цілей запроваджують багатофакторну автентифікацію (MFA).

Багатофакторна автентифікація (MFA) — це метод захисту облікових записів, при якому для входу потрібно не лише ввести пароль, а й підтвердити свою особу ще одним способом (рис. 1.1). Наприклад, це може бути код із SMS або пошти, сканування відбитка пальця чи відповідь на контрольне запитання. Такий підхід значно підвищує безпеку, оскільки навіть у разі компрометації пароля стороння особа не зможе отримати доступ без другого етапу перевірки.

Багатофакторна автентифікація забезпечує додатковий захисний бар’єр, що ускладнює доступ до облікового запису стороннім особам, навіть якщо їм вдалося дізнатись пароль. Такий підхід дозволяє організаціям ефективніше перевіряти особу користувача та водночас забезпечувати швидкий і зручний вхід для тих, хто має дозвіл на доступ.

Рисунок 1.1 – Складові багатофакторної автентифікації

1.3 Приклад алгоритму автентифікації у мобільних застосунках

У цьому прикладі буде продемонстровано один із алгоритмів системи автентифікації у мобільних застосунках (рис. 1.2).

Для того, щоб описати будь-яку систему автентифікації у мобільному застосунку, потрібно спочатку визначитися, який саме тип автентифікації потрібен, SFA, 2FA або MFA. Для прикладу розглянемо SFA. Уявімо, що користувач вже зареєстрований у нашому застосунку, та його облікові дані вже існують у віддаленому сховищі.

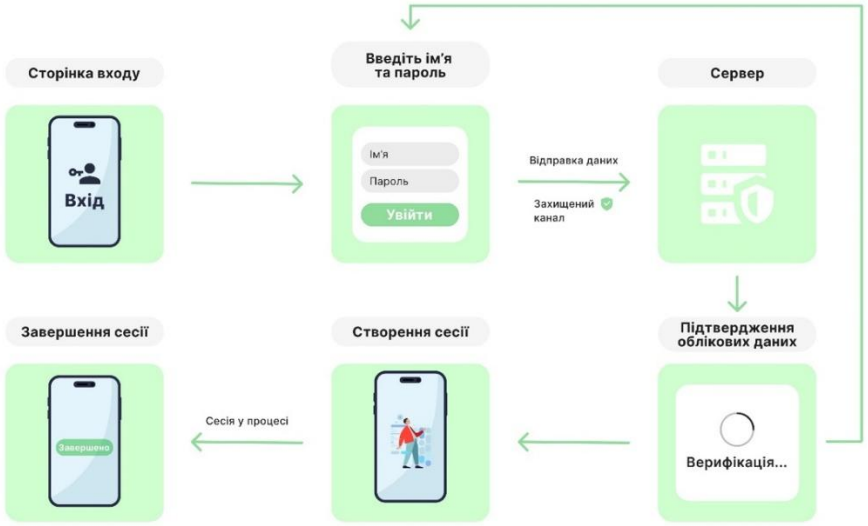

Рисунок 1.2 – Алгоритм впровадження автентифікації у мобільному застосунку

Першим етапом, користувач заходить до застосунку, де його переадресовує на сторінку із авторизацією. Система вимагає від користувача ввести його облікові дані, нехай це будуть ім’я користувача та його пароль.

Другий етап, це введення особистих облікових даних. Користувач вводить свої дані, переконавшись в їх правильності, та натискає на кнопку входу. У цьому прикладі, пароль – основний фактор автентифікації.

Третій етап, після натискання на кнопку входу, дані передаються на віддалений сервер, через захищені канали (зазвичай HTTPS).

Четвертий етап є перевіркою облікових даних користувача на сервері, а саме: сервер отримує ім’я користувача і пароль. Далі, пароль порівнюється зі збереженим гешем у БД. Якщо збіг є, то автентифікація вважається успішною. Якщо ні, користувачу буде відмовлено у доступі, та запропоновано: знову ввести облікові дані, перевіривши їх, або відновити пароль, через його скидання.

П’ятий етап, у разі успішного входу до застосунку, сервер створює сесію, в якій користувач отримує доступ до усіх захищених ресурсів особистого профілю.

Шостий етап, коли користувач хоче завершити роботу, він або виходить із застосунку, через натискання на відповідну кнопку на панелі інструментів, або закриває застосунок, і сервер потім сам завершує сесію користувача через неактивність.

1.4 Досвід застосування механізмів автентифікації від Google

Етапи автентифікації та авторизації високого рівня для API Google Workspace (рис. 1.3) [11]:

1) Налаштування свого проєкту та програми Google Cloud. Під час розробки відбувається реєстрація особистої програми в консолі Google Cloud, визначаються області авторизації та облікові дані доступу для автентифікації програми за допомогою ключа API, облікових даних кінцевого користувача або облікових даних облікового запису служби.

2) Відбувається автентифікація програми для доступу: під час запуску програми користувача оцінюються зареєстровані облікові дані доступу. Якщо програма проходить автентифікацію як кінцевий користувач, може з’явитися запрошення на вхід [11].

3) Запит ресурсів. Коли застосунок потребує доступу до ресурсів Google, він запитує у Google відповідні області доступу, які користувач раніше зареєстрував .

4) Запрошення згоди користувача. Якщо програма проходить автентифікацію як кінцевий користувач, Google відображає екран згоди OAuth, щоб користувач міг вирішити, чи надавати застосунку доступ до запитаних даних [11].

5) Надсилання затвердженого запиту на ресурси. Якщо користувач погоджується на області доступу, програма об’єднує облікові дані та затверджені користувачем області доступу до запиту. Запит надсилається на сервер авторизації Google для отримання токена доступу [11].

6) Google повертає токен доступу: токен доступу містить список областей доступу. Якщо список областей, що повертається, більш обмежений, ніж запитані області доступу, програма користувача відключає всі функції, обмежені токеном.

7) Доступ до ресурсів. Програма використовує токен доступу від Google для виклику відповідних API та доступу до ресурсів [11].

8) Отримання токену оновлення (необов’язково). Якщо ваша програма потребує доступу до API Google після закінчення терміну дії одного токена доступу, вона може отримати токен оновлення [11].

9) Запит додаткових ресурсів. Якщо необхідний додатковий доступ, ваш застосунок просить користувача надати нові області доступу, в результаті чого надсилається новий запит на отримання токена доступу (кроки 3-6) [11].

Рисунок 1.3 – Загальні етапи реалізації автентифікації та авторизації [11]

2 ДОСЛІДЖЕННЯ ТА АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ АВТЕНТИФІКАЦІЇ У МОБІЛЬНИХ ЗАСТОСУНКАХ

2.1 Парольні системи, їх переваги та недоліки

Як було зазначено у попередньому розділі, колись, окрім парольних систем, не було досить надійних механізмів автентифікації. Вони мають багато недоліків у сучасному світі, але вони ж мали і мають так само деякі переваги, тож, розглянемо їх.

Переваги парольних систем [2]:

- Простота використання: інтуїтивно зрозумілі для необізнаних користувачів, не потребують складного обладнання.

- Легке впровадження в системи: не вимагають великих витрат у додаткове обладнання чи технології.

- Універсальність: використовуються у всіх типах операційних систем, веб-сайтів, мобільних застосунках, тощо.

- Гнучкість: можуть бути адаптовані під вимоги безпеки (довжина, складність, періодична зміна).

Недоліки парольних систем [2]:

- Щільна залежність від користувача: вигадування слабких паролів, повторне використання одних й тих самих паролів для різних облікових засобів. Залежність від пам’яті користувача.

- Вразливість до атак: вразливі насамперед до соціальної інженерії, фішингових атак, атак на перехоплення. Також досить легко зламуються атаками зі словником або атаками типу «повний перебір».

- Складність управління: через те, що користувач може часто забувати особистий пароль, він може часто використовувати скидання паролю і його зміну, що підвищує навантаження на IT-відділи.

- Обмеженість безпеки: якщо сховище паролів було зламане, то всі облікові записи можуть бути під загрозою. Відсутність криптографічного захисту або погане його впровадження створюють додаткові ризики.

- Невідповідність сучасним вимогам: не відповідають сучасним реаліям загроз у світі.

2.2 Однофакторна автентифікація та її вразливості

SFA вже давно широко використовується як стандартний механізм доступу до більшості онлайн-платформ. Основна привабливість використання SFA серед користувачів криється у простоті впровадження та підтримки, що дозволяє швидко та без особливих проблем мати доступ до облікових записів. Ця легкість зробила однофакторну автентифікацію широко популярною у різних цифрових просторах, від електронних поштових скринь до онлайн-банкінгу і т.д. Загалом SFA служить такою собі «першою лінією оборони», яка бере участь у захисті онлайн-ідентичності та конфіденційності інформації.

Однак, як ми вже розглянули раніше, у однофакторної автентифікації є багато як переваг, так і недоліків. Одним із недоліків SFA є вразливість до кібератак.

Розглянемо основні кібератаки, до яких вразлива SFA:

- Фішингові атаки – основна вразливість криється у недостатній уважності користувача, який не може відрізнити легітимний сайт або адресу від шахрайської підробки.

- «Повний перебір» та атаки зі словником – основна вразливість криється у виборі слабких або «стандартних» паролів, які зловмисники використовують для перебору та пошуку правильного.

- Атаки повторного використання – зловмисники перехоплюють облікові дані під час автентифікації та використовують їх для повторного входу. Якщо відсутній захист проти повторного використання токенів, то виявлення таких атак ускладняється.

- Атаки з використанням кейлогерів – зловмисники поширюють заражені файли або програми, інфікуючи ПК після завантаження та встановлення відповідного ПО під видом валідного. Кейлогери зчитують та записують натискання на клавіатурі під час контакту з клавішами та пересилають отриману інформацію шахраям.

- Атаки, засновані на соціальній інженерії – шахраї, представляючись легітимними представниками організації, за допомогою маніпуляцій викрадають особисті паролі та інші дані для входу до облікових записів.

- Shoulder Surfing атаки (Спостереження через плече) – якщо людина входить до свого облікового запису в офісі або в якомусь громадському місці, зловмисник може підгледіти пароль та інші чутливі дані, які потім використає для входу до профілю користувача.

Вразливості однофакторної автентифікації роблять її малоефективною у сучасному світі. Тому відповідальність за використання систем або ресурсів із SFA лягає на самого користувача, і чи хоче він ризикувати втратити свої особисті дані.

2.3 Порівняння методів 2FA: TOTP, HOTP, push-нотифікації, біометрія

Двофакторна автентифікація (2FA), як вже було зазначено раніше, потребує два різних фактори для встановлення особи. Коротко, це означає вимогу до користувача підтвердити свою особу двома різними способами, перш ніж надавати йому доступ. Тож, щоб більш детально ознайомитися із можливими факторами автентифікації у процесі 2FA, розглянемо деякі популярні методи та потім порівняємо їх між собою.

Одноразовий пароль, заснований на часі (TOTP) – це метод автентифікації, за якого у певний проміжок часу генеруються унікальні коди на основі певного алгоритму. Цей тип автентифікації використовується як для двофакторної автентифікації, так і для багатофакторної автентифікації [7].

TOTP базується на секретному алгоритмі, який випадковим чином генерує коди. Алгоритм унікальний для кожного екземпляра, використовує поточний час як фактор. Це дозволяє створювати унікальні коди кожні 30-60 секунд. Щоразу, коли користувач ініціює створення нового TOTP для облікового запису, сервери облікових записів генеруватимуть унікальний секретний алгоритм, який зазвичай відображається у вигляді QR-коду. Сервер зберігає секрет і використовує його для генерації кодів TOTP. Далі користувачу потрібно сканувати створений QR-код за допомогою якогось інструменту автентифікації, який може бути або впроваджено в пристрій, або буде надаватися системою, де користувач намагається увійти чи

зареєструватися. Оскільки інструмент автентифікації тепер має секретний алгоритм, він обчислює ті самі шестизначні коди, що й сервер. Після синхронізації налаштувань алгоритм працює одночасно як на сервері, так і на пристрої автентифікації. Коли користувач входить, він вводить поточно згенерований код, сервер порівнює його із розрахованим кодом, і, якщо коди збігаються, то користувачу надається доступ [7].

TOTP є одним з найзручніших та найбезпечніших методів автентифікації. Навіть, якщо зловмисник зможе вкрасти облікові дані для входу до облікового запису користувача, підключений TOTP не дасть увійти, бо він буде доступний лише законному користувачу. Винятком буде випадок, коли зловмисник володітиме як особистими даними користувача, так і доступом до його апаратних ресурсів.

Наступний, дещо схожий на попередній, метод автентифікації це HOTP – одноразовий пароль на основі гешу. Він працює аналогічно до TOTP, але має трохи інший алгоритм обчислення коду. Код генерується на основі гешу HMAC. HMAC підраховує кількість запитів на код і використовує це для його обчислення. Код змінюється щоразу, коли запитується, на відміну від попереднього методу, де зміна відбувалася кожні 30-60 секунд [7].

Схематичне відображення формування HOTP та TOTP представлено на рисунку 2.1:

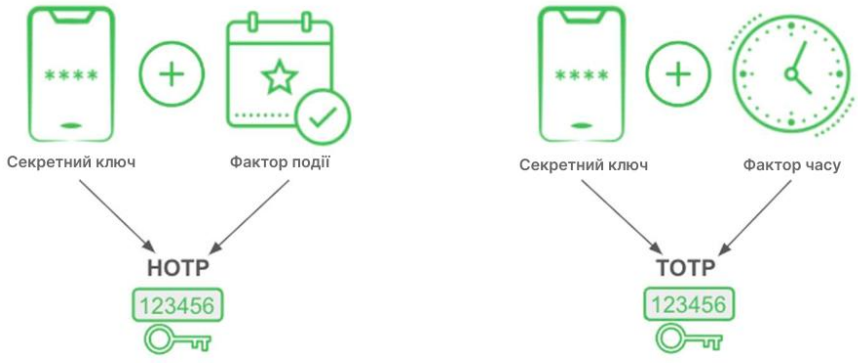

Рисунок 2.1 – Формування HOTP та TOTP

Передостанній метод, який ми розглянемо, це Push-нотифікації – метод автентифікації, який використовує безпечний сервер для подання запитів на вхід, коли користувач вводить пароль або PIN-код [8].

Сервер надсилає push-повідомлення на мобільний пристрій користувача, який був прив’язаний до облікового запису під час або після реєстрації. Це сповіщення або «виклик» зазвичай підписане закритим ключем. Сповіщення надходить із пов’язаної із застосунком програми на телефоні з використанням шифрування з відкритим ключем. Виклик, підписаний закритим ключем, буде перевірено відкритим ключем, пов’язаним із програмою та телефоном. Отримавши сповіщення, користувач може схвалити його або відхилити. Оскільки виклик перевірено, користувач може підтвердити сповіщення, щоб авторизуватися, усуваючи потребу передавати одноразові паролі або токени через загальнодоступні підключення до Інтернету. Автентифікований виклик надсилається на сервер, і користувач може отримати доступ до свого системного облікового запису та ресурсів [8].

Схема алгоритму автентифікації із використанням Push-нотифікацій представлена на рисунку 2.2:

Рисунок 2.2 – Алгоритм автентифікації із використанням Push-нотифікацій

Останнім розглянутим методом буде біометрія. Біометричні характеристики – це фізичні та біологічні властивості, унікальні для кожної людини. Вони зберігаються в базі даних і можуть бути легко порівняні з користувачем, який намагається отримати доступ до даних або пристрою. Таку біометричну автентифікацію можна розмістити в різних фізичних середовищах, таких як двері, ворота, серверні кімнати, військові бази, аеропорти, порти тощо. У наш час засоби

біометричної автентифікації стали частиною більшості споживчих пристроїв, зокрема комп'ютерів та смартфонів [12].

Розглянемо типи біометричної автентифікації (рис. 2.3):

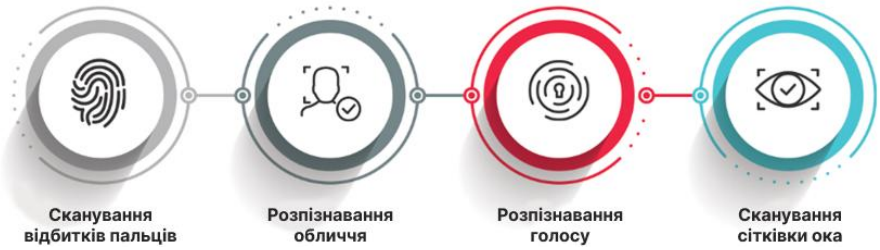

Рисунок 2.3 – Типи біометричної автентифікації

Перший тип – це сканери відбитків пальців. Найпоширеніший метод біометричної автентифікації під час використання мобільних пристроїв, адже користувачі досить часто тримають смартфон у руці і додають відбиток одного із пальців для подальшого швидкого доступу до пристрою. Сучасний технологічний прогрес призвів до появи сканерів, які виходять за рамки відбитків пальців, щоб сканувати судинні візерунки. Це допомогло зменшити помилкові спрацьовування, які час від часу трапляються з біометричними параметрами споживчого класу, що є на смартфонах. Сканери відбитків пальців продовжують залишатися найдоступнішими та найпопулярнішими [12].

Другий тип – це сканер розпізнавання обличчя. Як і сканер відбитків пальців, технологія розпізнавання обличчя також досить популярна на мобільних пристроях через наявність фронтальної камери, що робить цей тип біометричної автентифікації дуже зручним у використанні. Також слід зазначити, що на деяких моделях мобільних пристроїв Android і Apple IOS технологія сканування обличчя досить вдосконалена, а саме: технологія Face ID використовує не лише звичайну камеру, але і спеціальний набір сенсорів у модулі TrueDepth. До таких модулів входять: інфрачервона камера – зчитує глибину та форму обличчя, проєктор точок – проєктує понад 30000 невидимих інфрачервоних точок на обличчя для створення тривимірної моделі, інфрачервоний освітлювач – дає змогу використовувати технологію у темряві [12].

Параметри, які зберігаються після встановлення розпізнавання обличчя, як система біометричної автентифікації називаються відбитками. Доступ надається

лише тоді, коли їх наявна велика кількість. Незважаючи на непослідовність зіставлення обличь із параметрами під різними кутами зору або розрізнення схожих чи споріднених людей, розпізнавання обличь включено у велику кількість розумних пристроїв.

Третій тип – це технологія розпізнавання голосу. Цей тип біометричної автентифікації базується на голосових характеристиках, щоб відрізнити одну особу від іншої. Технології розпізнавання голосу більше зосереджені на формі рота та горла під час запису, а також якості звуку, ніж просто на прослуховуванні голосу. Це допомагає зменшити ймовірність неправильного прочитання спроби імітації голосу [12].

Четвертий тип – сканер сітківки та райдужної оболонки ока. Сканер сітківки проєктує яскраве світло в око, щоб виділити візерунки кровоносних судин, які сканер може зчитати. Ці показники порівнюються з інформацією, збереженою в базі даних. Сканери райдужної оболонки оцінюють унікальні візерунки в кольоровому кільці зіниці. Обидві форми сканера ідеально підходять для перевірки без використання рук. Однак вони можуть бути ненадійними, якщо людина носить контактні лінзи або окуляри. Такий метод біометричної автентифікації досить ресурсозатратний та незручний для повсякденного використання звичайним користувачем, тож його впроваджують у критично важливих підприємствах або на стратегічних об’єктах [12].

Тепер, спираючись на розглянуті методи 2FA, створимо таблицю 2.1 для швидкого та наглядного розуміння і порівняння методів.

Таблиця 2.1 – Порівняння методів 2FA

Метод 2FA	Опис	Переваги	Недоліки
TOTP	Одноразовий пароль, заснований на часі, генерується кожні 30–60 секунд на основі секретного алгоритму та часу	- Висока безпека завдяки динамічній генерації кодів. - Простота впровадження. - Широке застосування у популярних сервісах (Google Authenticator тощо).	- Потребує синхронізації часу між сервером та пристроєм. - Вразливість до атак на пристрій користувача, якщо секретний ключ збережено ненадійно.

Подовження таблиці 2.1

HOTP	Одноразовий пароль, заснований на геш-функції HMAC, генерується під час кожного запиту на основі лічильника запитів	- Незалежність від часу синхронізації. - Підходить для офлайн-доступу.	- Може бути менш зручним через залежність від лічильника запитів. - Ризик витоку секретного ключа.
Push-нотифікації	Користувач отримує сповіщення на пов'язаний пристрій, яке потрібно схвалити або відхилити. В основі лежить шифрування з відкритим і закритим ключем.	- Висока зручність для користувача. - Зниження ризику передачі паролів через Інтернет. - Підтримує перевірку унікальності викликів.	- Потребує стабільного Інтернет-з'єднання. - Залежність від надійності сервера і пристрою.
Біометрія	Використання унікальних фізичних характеристик користувача (відбитки пальців, розпізнавання обличчя, голосу тощо) для автентифікації	- Унікальність біометричних даних користувачів. - Висока швидкість і зручність. - Відсутність необхідності запам'ятовування паролів.	- Висока вартість реалізації. - Вразливість до фізичних або програмних атак (імітація відбитків пальців, розпізнавання обличчя тощо). - Неможливість змінити біометричні дані.

2.4 Аналіз вразливостей при застосуванні існуючих технологій і методів 2FA та MFA

Одним із найпоширеніших методів 2FA є SMS. Користувачу надсилається одноразовий код підтвердження у вигляді текстового повідомлення на його мобільний телефон. Активне використання частково пояснюється тим, що більшість споживачів уже мають мобільні телефони й здатні отримувати текстові повідомлення. Потенційні проблеми зі зручністю використання можуть включати затримку доставки, відсутність стільникового зв'язку (наприклад, у іншій країні чи віддаленому місці) та неправильне копіювання коду з телефону на комп'ютер. Автентифікація на основі SMS є вразливою на кількох етапах. Мобільні мережі не

шифрують повідомлення під час передачі, що дозволяє зловмисникам здійснювати атаки типу "людина посередині". Також занепокоєння викликає атака із заміною SIM-карти. Крім того, сервер (або перевіряюча сторона) має надійно зберігати одноразовий код, поки SMS-повідомлення надсилається, його отримує користувач і вводить назад на сайт для перевірки. Код можна було підсолювати та гешувати, щоб запобігти випадковій крадіжці, але зловмисник міг би легко провести атаку грубою силою на викрадений гешований код, враховуючи відносно невелику кількість кодів. Зловмисники також можуть викрасти SMS-коди за допомогою цілеспрямованих фішингових атак. Деякі способи пом’якшити ці загрози: зробити код недійсним через короткий проміжок часу та обмежити кількість невдалих спроб входу за допомогою коду [13].

Альтернативно до автентифікації за допомогою SMS можна застосувати TOTP. Перевага використання програми для створення коду TOTP полягає в тому, що після синхронізації секрету з телефоном користувачу не потрібно покладатися на постачальника мобільного зв’язку для доставки одноразових кодів, усуваючи як потенційну поверхню атаки, так і проблеми зі зручністю використання. Однак, якщо зловмисник викраде секрет TOTP з серверу або телефону, тоді він може видати себе за користувача. Для використання TOTP потрібен спільний секретний ключ між сервером та мобільним пристроєм користувача. Цей секрет має зберігатися безпечно, але механізм одностороннього гешування не є корисним, оскільки секрет є вхідними даними для генерації коду та процесу перевірки. На стороні сервера спільний секрет може бути зашифрований за допомогою пароля користувача, щоб запобігти випадковій крадіжці. Гарантуючи безпечне зберігання спільного секрету як на клієнті, так і на сервері, TOTP має значну перевагу перед SMS-кодами – він не покладається на незахищену мобільну мережу для доставки коду, таким чином усуваючи всю поверхню атаки [13].

Розглянемо автентифікацію за допомогою Push-повідомлень. Push-автентифікація потребує доступу до Інтернету. Google підтримує цю техніку (через їхню «підказку Google») і вона також доступна через комерційні програми, такі як Authy OneTouch і DUO Mobile. Перевагою цього методу є менша ймовірність

помилки користувача, оскільки немає чисел, які можна правильно скопіювати з екрана телефону. Припускається, що відсутність необхідності вводити числа, як того вимагають інші методи 2FA, є швидшою та сприймається користувачами як більш зручна. Push-автентифікація явно не вимагає зберігання секретного ключа, однак, сервер повинен переконатися, що push-повідомлення надсилаються на правильний пристрій, що передбачає певну форму двосторонньої перевірки клієнта та сервера. Крім того, зв'язок між пристроєм користувача та сервером має бути безпечним, наприклад, за допомогою TLS. Найпоширеніші методи push-автентифікації є запатентованими, що ускладнює перевірку точних заходів безпеки та вимагає неявної довіри третій стороні [13].

Наостанок розглянемо вразливості біометричної автентифікації. Однією із загроз для системи є підробка біометричних даних. У наш час, якщо є потреба, то немає складності виготовити синтетичні відбитки пальців, фотографії з високою роздільною здатністю або маску чи макет обличчя, надруковані за допомогою 3D-принтеру. Усе це є ризиком та вразливістю у системі, коли використовується біометрична система автентифікації [14].

Також ще однією вразливістю у біометричній системі є витік даних. На відміну від паролів, які можна змінити в разі зламу, біометричні дані, якщо їх викрадуть, не можна змінити. Це створює довгострокову загрозу безпеці, оскільки зловмисники можуть використовувати викрадені біометричні дані необмежено довго для несанкціонованих дій. Наприклад, у 2019 році була виявлена загальнодоступна база даних, що містить відбитки пальців понад 1 мільйона людей, а також дані розпізнавання обличь, незашифровані імена користувачів, паролі та особисту інформацію співробітників. Ця база даних належала компанії, яка обслуговувала великих клієнтів, зокрема столичну поліцію Великобританії, оборонних підрядників і банки, підкреслюючи потенційні ризики, пов'язані зі зберіганням біометричних даних [14].

3 ДОСЛІДЖЕННЯ ТА ПОРІВНЯЛЬНИЙ АНАЛІЗ ЗАСОБІВ ТА ТЕХНОЛОГІЙ ДЛЯ РЕАЛІЗАЦІЇ СИСТЕМ АВТЕНТИФІКАЦІЇ

3.1 Порівняння бібліотек і платформ, що реалізують методи автентифікації

Після того, як ми розглянули методи та технології різних факторів автентифікації тощо, перейдемо до розгляду вже визначених бібліотек та платформ, які можуть бути використані для реалізації системи автентифікації в конкретному пристрої.

Для початку розглянемо платформу Firebase. Firebase – це одне з BaaS-рішень (Backend as a Service), що дозволяє розробникам реалізувати безліч backend-технологій і полегшити створення різних застосунків швидко і без будь-яких ускладнень, пов’язаних з інтеграцією backend-частини. Firebase – це сервер, база даних, хостинг та автентифікація в одній платформі. У цій тезі ми розглянемо та використаємо Firebase Authentication та Firestore [15].

Firebase Authentication призначений для того, щоб зробити впровадження системи автентифікації простішим і безпечнішим, а також забезпечити комфортний досвід входу для користувачів. Це універсальний інструмент, який дозволяє використовувати різні способи авторизації — від класичних логіну через email і пароль до входу через популярні сервіси, як-от Google, Apple, Facebook, GitHub, X (Twitter) та інші (рис. 3.1).

Рисунок 3.1 – Різні варіанти виконання автентифікації у Firebase

FirebaseUI — це настроюване рішення з відкритим кодом, яке спрощує реалізацію процесів автентифікації, автоматизуючи створення інтерфейсів входу.

Модуль FirebaseUI Auth інтегрує перевірені практики автентифікації для мобільних додатків і вебплатформ, що дозволяє підвищити ефективність залучення користувачів під час авторизації та реєстрації в застосунку.

Створений тією ж командою, яка розробила Google Sign-in, Smart Lock і Chrome Password Manager, Firebase Security використовує власний досвід Google для управління однією з найбільших баз даних облікових записів у світі [16].

Firebase Authentication тісно інтегрується з іншими сервісами Firebase і використовує галузеві стандарти, такі як OAuth 2.0 і OpenID Connect, тому її можна легко інтегрувати з бекендом [16].

Розглянемо покроково процес автентифікації за допомогою Firebase Authentication SDK.

По-перше, як і в будь-якій системі автентифікації, нам потрібно, щоб користувач мав зареєстрований обліковий запис. Для цього в застосунку вводяться персональні дані, такі як логін чи ім’я користувача, а може і те, і інше, пароль і його підтвердження для кращого запам’ятовування і мінімізації помилок при введенні. Після реєстрації дані передаються на серверні сервіси, які шифрують їх і зберігають для подальшої автентифікації.

Тепер перейдемо безпосередньо до автентифікації. Ми отримуємо від користувача облікові дані, які були вказані при реєстрації, для входу в застосунок (або використовуємо інший доступний метод автентифікації). Потім ці дані передаються до Firebase Authentication SDK. Серверні служби перевіряють ці дані і повертають клієнту певну відповідь. Якщо відповідь задовільна, то вхід в систему відбувається успішно, якщо ні, то користувач отримує повідомлення з відповідною помилкою і далі слідує інструкціям, наданим розробником в цьому випадку. Оскільки багато технологій в Firebase тісно пов’язані між собою, при вході в обліковий запис користувач отримує доступ до даних, що зберігаються в інших продуктах платформи. У свою чергу, розробник може використовувати наданий токен автентифікації для перевірки особи користувача у власних серверних сервісах, а також контролювати доступ до цих автентифікованих користувачів, змінюючи правила роботи з базами даних Firebase та правила безпеки сховища [16].

Однак, попри всі переваги використання Firebase автентифікації, у неї є і певні недоліки, тому давайте розглянемо їх детальніше.

1) Firebase – це платформа з закритим вихідним кодом. Це означає, що ви не можете змінювати код Firebase, навіть якщо те, що надає Firebase, не відповідає вашим потребам розробки програми.

2) Не дешева у використанні – безкоштовний план пропонує лише базові функції, без розширених можливостей, які спрощують і прискорюють всі завдання розробки. Можливо, одна з причин високої вартості Firebase полягає в тому, що вона використовує пропрієтарну технологію, яка коштує дорого і потребує вигідної монетизації [17].

3) Недоступність в деяких країнах світу – оскільки Firebase є офіційним продуктом Google, а його URL-адреса використовує субдомен Google, тобто firebase.google.com, сервіс заблоковано в Китаї та інших країнах, де заблоковано сервіси Google. Тому розробники в таких країнах не можуть використовувати платформу Firebase для створення та розміщення серверної частини програми [17].

4) Обмеження на використання в середовищах Google Cloud – Firebase розміщена в Google Cloud, одному з найпотужніших хмарних сервісів в світі на сьогоднішній день. Ви не можете використовувати Firebase на інших хмарних платформах, таких як DigitalOcean, AWS або Azure. Крім того, користувачі Firebase не мають доступу до серверного рівня. Тому налаштування параметрів сервера може бути складним завданням [17].

5) Залежність від наявності підключення до Інтернету. Автентифікація Firebase вимагає стабільного Інтернет-з’єднання для автентифікації на серверах Firebase.

Далі розглянемо наступний інструмент автентифікації – Google Authenticator. Це мобільний застосунок для пристроїв на Android та IOS, який пропонує додатковий етап автентифікації користувача (2FA). На відміну від автентифікації за допомогою кодів, що надсилаються через SMS, Google Authenticator забезпечує більш безпечний метод перевірки особи користувача перед наданням доступу до

захищених ресурсів. Програма використовує алгоритм генерації кодів TOTP, який ми вже обговорювали в попередніх розділах [18].

Перейдемо до пояснення того, як встановити і використовувати Google Authenticator на вашому смартфоні.

Як і у випадку з Firebase, користувач вже повинен мати зареєстрований особистий профіль. Далі потрібно перейти до налаштувань програми та обрати параметри безпеки. У вкладці з рекомендаціями щодо безпеки потрібно вибрати опцію підключення двофакторної автентифікації. Натиснувши на цю опцію, користувач перейде на сторінку, де буде описана інструкція з підключення. В цей час користувач повинен завантажити застосунок Google Authenticator з Google Play або App Store (залежно від операційної системи пристрою). Після відкриття програми користувачеві буде запропоновано додати новий обліковий запис, відсканувавши QR-код або ввівши ключ налаштування. Перший метод вимагає, щоб користувач натиснув на знак «+» на екрані програми під час входу в застосунок Google Authenticator і відсканував QR-код з програми, в якій він хоче увімкнути двофакторну автентифікацію (рис. 3.2). Другий спосіб, з використанням ключа налаштування, схожий на перший, за тим винятком, що не потрібно нічого сканувати, а достатньо лише ввести ключ налаштування з того застосунку, в якому ви хочете увімкнути 2FA, в Google Authenticator [18].

Рисунок 3.2 – Алгоритм підключення 2FA із використанням Google Authenticator

В результаті в Google Authenticator буде додано новий профіль, для якого будуть згенеровані відповідні коди TOTP або HOTP, в залежності від налаштувань,

які потім будуть використовуватися для доступу до особистого профілю користувача під час автентифікації. Процес генерації кодів TOTP і HOTP вже був описаний в попередніх розділах.

Складний на перший погляд алгоритм підключення Google Authenticator має кілька суттєвих переваг:

1) Google Authenticator використовує TOTP-коди, які складніше зламати, ніж статичні паролі або SMS-коди.

2) Google Authenticator також може генерувати паролі локально на пристрої, що дозволяє проходити автентифікацію навіть без активного підключення до Інтернету [18].

3) Google Authenticator дозволяє користувачам легко переміщати свої облікові записи між пристроями за допомогою функції Move Accounts. Це забезпечує безперервність і безпеку при оновленні або заміні пристрою [18].

Але, як і майже будь-яка система автентифікації, Google Authenticator має свої недоліки:

- Залежність від пристрою – застосунок генерує коди локально на вашому пристрої, і частіше за все, він також може відображати деякі персональні дані. Отже, якщо втратити доступ до пристрою, то втрачається і доступ до облікових записів, до яких підключений Google Authenticator, якщо тільки у вас немає альтернативних методів відновлення [18].

- Людський фактор, а саме фішингові атаки. Користувачі можуть бути досить неуважними і помилково ввести OTP-код на нелегітимному веб-сайті або застосунку. Зловмисник викраде код і отримає доступ до облікового запису користувача.

Наостанок, перш ніж порівнювати розглянуті платформи, технології та бібліотеки, розглянемо систему Android Keystore. Система Android Keystore дозволяє зберігати криптографічні ключі в контейнері, щоб їх було складніше вилучити з пристрою. Коли ключі знаходяться в сховищі, їх можна використовувати для криптографічних операцій, в той час як ключовий матеріал залишається неекспортованим. Крім того, система зберігання ключів дозволяє обмежити, коли і

як можна використовувати ключі, наприклад, вимагаючи автентифікацію користувача для використання ключа або обмежуючи використання ключів певними криптографічними режимами [19].

Ключовий матеріал ключів Android Keystore захищено від вилучення за допомогою двох заходів безпеки:

1) Ключовий матеріал ніколи не потрапляє в процес роботи програми. Коли застосунок виконує криптографічні операції з ключем Android Keystore, відкритий текст, зашифрований текст і повідомлення, які потрібно підписати або перевірити, передаються системному процесу, який виконує криптографічні операції. Якщо процес програми скомпрометований, зловмисник може використовувати ключі програми, але не може витягти ключ (наприклад, для використання за межами пристрою Android) [19].

2) Ключовий матеріал може бути прив’язаний до захищеного обладнання на пристрої Android, наприклад, до довіреного середовища виконання (Trusted Execution Environment, TEE) або захищеного елемента (Secure Element, SE). Якщо цю функцію увімкнено для ключа, його матеріали ніколи не будуть доступні за межами захищеного обладнання. Якщо ОС Android скомпрометована чи зловмисник може зчитувати внутрішню пам’ять пристрою, він також може використовувати ключі Android Keystore будь-якого застосунку на Android-пристрої, але не зможе витягти їх з пристрою. Ця функція вмикається лише в тому випадку, якщо захищене апаратне забезпечення пристрою підтримує певну комбінацію алгоритму ключа, режимів блокування, схем підстановки та дайджестів, які дозволено використовувати ключу [19].

Цю систему можна активно використовувати при впровадженні біометричної автентифікації. Ось приклад того, як Android Keystore пов’язаний з біометричною автентифікацією.

Користувач реєструється, додаючи свій відбиток пальця. Програма створює криптографічний ключ у сховищі, який буде використовуватися для автентифікації. Потім, коли користувач хоче увійти до вже створеного облікового запису, йому потрібно прикласти палець до екрану і відсканувати відбиток пальця. Біометрична

верифікація відбувається через інтерфейс Android. Після успішної автентифікації Android Keystore розблоковує ключ, необхідний для здійснення запиту до сервера або доступу до конфіденційної інформації. Раніше створений ключ можна використовувати для підписання запитів або розшифровки даних.

Тепер, коли ми розглянули технології, платформи та системи, необхідні для реалізації систем автентифікації, ми можемо їх порівняти у таблиці 3.1.

Таблиця 3.1 – Порівняння методів реалізації систем автентифікації

Платформа/ Система	Основне призначення	Переваги	Недоліки
Firebase Authentication	Платформа для інтеграції автентифікації, яка підтримує email, паролі, соціальні мережі, телефонну автентифікацію.	- Швидка інтеграція. - Підтримка різних методів автентифікації. - Інтеграція з іншими сервісами Firebase. - Використання стандартів OAuth 2.0 і OpenID Connect.	- Залежність від Інтернет-з'єднання. - Висока вартість після безкоштовного ліміту. - Не працює в країнах, де заблоковані сервіси Google (наприклад, Китай).
Google Authenticator	Мобільний застосунок для генерації TOTP/HOTP кодів, що використовуються для двофакторної автентифікації (2FA).	- Генерує коди офлайн. - Використовує стійкий до зламів алгоритм (TOTP/HOTP). - Легкий у використанні. - Підтримка перенесення облікових записів між пристроями.	- Прив'язка до пристрою: втрата пристрою веде до втрати доступу. - Вразливість до фішингових атак через людський фактор (введення коду на підробних сайтах).
Android Keystore	Система безпечного зберігання криптографічних ключів, яка інтегрується з біометрією для додаткової безпеки.	- Висока безпека через апаратне шифрування. - Можливість інтеграції з біометрією. - Неможливість витягти ключ із пристрою.	- Складність у налаштуванні. - Працює лише на пристроях Android. - Залежність від наявності підтримки TEE (Trusted Execution Environment) або Secure Element (SE).

3.2 Розгляд стандартів автентифікації

У цьому розділі ми розглянемо стандарти OAuth 2.0 та OpenID Connect.

OAuth 2.0 – це фреймворк автентифікації, який дозволяє таким сервісам, як Facebook, GitHub і DigitalOcean, отримувати обмежений доступ до облікових записів користувачів на HTTP-сервісах. Працює шляхом делегування автентифікації користувача сервісу, на якому розміщений обліковий запис користувача, і надання доступу стороннім додаткам до цього облікового запису. OAuth 2.0 забезпечує потоки авторизації для веб- і десктопних застосунків, а також мобільних пристроїв [21].

У OAuth 2.0 беруть участь різні сторони, кожна з яких відіграє певну роль. Ролі в OAuth 2.0 наступні:

1) Власник ресурсу – кінцевий користувач, який володіє даними, до яких хоче отримати доступ сторонній застосунок. Саме користувач, зазвичай фізична особа, але також і організація, контролює доступ до захищеного ресурсу, наприклад, персональних даних або послуг. Власник надає дозвіл клієнту на доступ до своїх ресурсів [20].

2) Сервер ресурсів – містить захищені ресурси, до яких клієнт хоче отримати доступ від імені власника. Це може бути API, база даних, файловий сервер або будь-яка інша система, яка керує і контролює доступ до ресурсів. Наприклад, хмарне сховище, де зберігаються особисті файли користувача.

3) Клієнт – програма або сервіс, який запитує доступ до ресурсу від імені власника. Це може бути веб-застосунок, мобільний застосунок або інше програмне забезпечення. Клієнт ініціює процес авторизації та взаємодіє з серверами авторизації та ресурсу, щоб отримати доступ до захищених матеріалів. Наприклад, веб-додаток, який сканує файли користувачів у хмарному сховищі на наявність шкідливого програмного забезпечення [20].

4) Сервер автентифікації – відповідає за автентифікацію власника ресурсу, підтвердження його авторизації та видачу токенів доступу клієнтам. Він перевіряє особу клієнта та дозволи, надані власником ресурсу. Сервер авторизації відіграє важливу роль у забезпеченні безпечного та санкціонованого доступу до захищених

ресурсів. Зазвичай сервер авторизації контролюється тією ж організацією, яка керує сервером ресурсів [20].

Тепер, коли ми розглянули ролі OAuth 2.0, проаналізуємо відповідну діаграму (рис. 3.3) і те, як ці ролі взаємодіють між собою [21]:

1) Програма запитує у користувача дозвіл на доступ до ресурсів сервісу.

2) Якщо користувач авторизує запит, програма отримує дозвіл.

3) Програма запитує токен доступу у сервера авторизації (API), що є підтвердженням її ідентичності та авторизації.

4) Якщо ідентифікатор програми автентифікований і авторизація дійсна, сервер авторизації (API) видає токен доступу до програми. Авторизація завершена.

5) Програма запитує ресурс у сервера ресурсів (API) і представляє токен доступу для автентифікації.

6) Якщо маркер доступу дійсний, сервер ресурсів (API) надає ресурс програмі.

Фактичний хід цього процесу буде відрізнятися в залежності від типу авторизації, що використовується, але це загальна ідея.

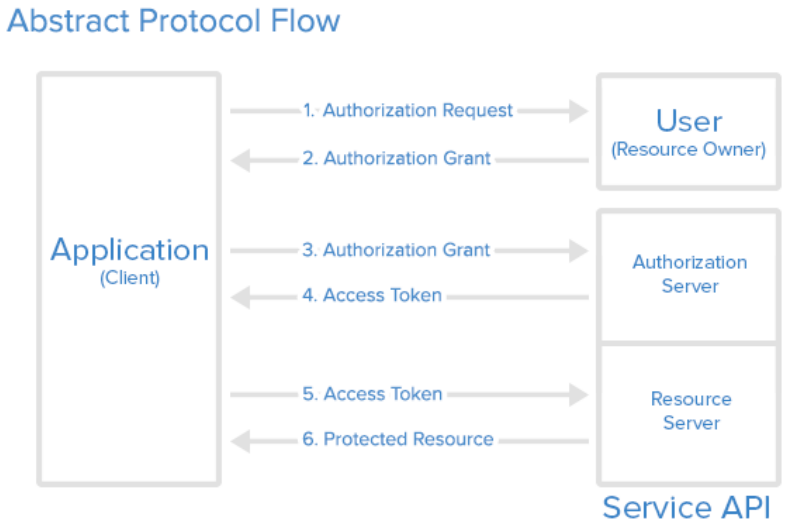

Рисунок 3.3 – Взаємодія ролей OAuth між собою [21]

OpenID Connect – це протокол делегованої автентифікації в Інтернеті: користувач може увійти в систему перевіряючої сторони (RP), пройшовши автентифікацію в так званого постачальника ідентифікаційної інформації (IdP). Наприклад, користувач може увійти на веб-сайт за допомогою свого облікового запису Google. Хоча назви можуть свідчити про інше, OpenID Connect (або

скорочено OIDC) не базується на старішому протоколі OpenID. Натомість він побудований на основі OAuth 2.0, який визначає протокол для делегованої авторизації (наприклад, користувач може надати сторонньому веб-сайту доступ до своїх ресурсів у Facebook). Хоча OAuth 2.0 не був розроблений для забезпечення автентифікації, він також часто використовувався для цієї мети, що призвело до кількох серйозних недоліків безпеки в минулому. OIDC було створено не лише для модернізації автентифікації в OAuth 2.0 за допомогою криптографічно захищених токенів і точно визначеного методу автентифікації користувача, але й для включення додаткових важливих функцій. Наприклад, використовуючи розширення Discovery, RP можуть автоматично ідентифікувати IdP, який відповідає за певну особу. Завдяки розширенню Dynamic Client Registration RP не потребує процесу налаштування вручну для роботи з конкретним IdP, натомість вони можуть зареєструватися в IdP на льоту [22].

Створений OpenID Foundation і стандартизований лише в листопаді 2014 року, OIDC вже дуже широко використовується. Він використовується та підтримується Google, Amazon, Paypal, Salesforce, Oracle, Microsoft, Symantec, Verizon, Deutsche Telekom, PingIdentity, RSA Security, VMWare та IBM. Багато рішень єдиного входу для компаній і кінцевих користувачів базуються на OIDC, наприклад, добре відомі служби, такі як Google Sign-In і Log In with Paypal. Незважаючи на його широке використання, OpenID Connect поки що не привернув особливої уваги дослідників безпеки (на відміну від OpenID та OAuth 2.0). Зокрема досі не було офіційних спроб аналізу OpenID Connect [22].

Тепер, розглянемо загально OIDC на рисунку 3.4 та пояснимо її.

Спочатку користувач просить авторизуватися на певному RP і надає свою адресу електронної пошти (A). RP отримує робочу інформацію (наприклад, деякі URL-адреси) для потоку протоколу, що залишився (виявлення, B) і реєструється в IdP (C). Після цього користувач перенаправляється до IdP, де авторизується (D), наприклад, використовуючи пароль. IdP видає токен ідентифікатора RP (E), який RP може потім перевірити, щоб переконатися в ідентичності користувача.

Спосіб, яким IdP надсилає токен ідентифікатора до RP, залежить від різних режимів OIDC. Тобто, токен ідентифікатора або передається через браузер користувача, або отримується RP безпосередньо з IdP. Ідентифікаційний токен містить ідентифікатор для IdP (емітента), ідентифікатор користувача (унікальний у відповідного IdP) і підписаний IdP. RP використовує ідентифікатор емітента та ідентифікатор користувача, щоб визначити особу користувача. Нарешті, RP може встановити файл cookie сеансу в браузері користувача, який дозволяє користувачеві отримати доступ до послуг RP (F) [22].

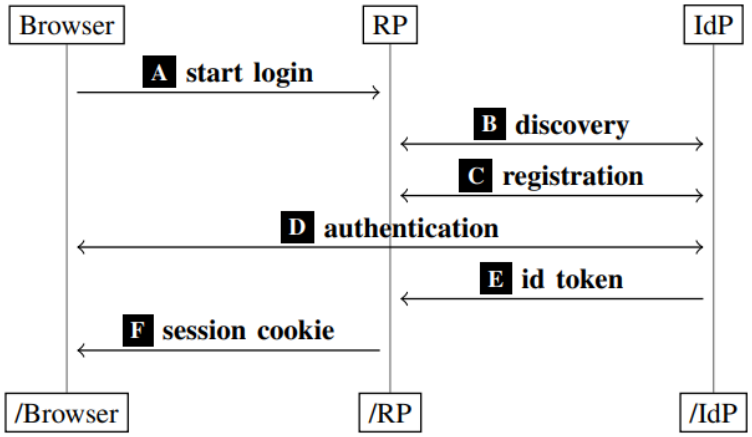

Рисунок 3.4 – Загальний огляд OIDC [22]

3.3 Вибір найбільш придатних технологій для мобільних застосунків

Коли настає час вибору технології для реалізації системи автентифікації, важливо враховувати чимало чинників. Перш за все, потрібно дивитись на можливості системи, її зручність та складність інтеграції з другорядними технологіями. Наприклад, Firebase забезпечує широкий спектр готових до використання рішень із високою інтеграцією з іншими продуктами, що є перевагою для створення швидких і безпечних застосунків, але закритість коду не дає змоги розробнику впровадити специфічні рішення. Google Authenticator підходить для впровадження 2FA з використанням кодів TOTP, що генеруються локально, підвищуючи захист користувачів, які мають нестабільне Інтернет-з'єднання. Android Keystore дозволяє безпечно зберігати криптографічні ключі, що робить його гарним вибором для забезпечення захисту чутливих даних, але він досить складний у впровадженні. Тож вибір залежить від запитів розробника, вимог до рівня безпеки, використання, та інтеграції з іншими компонентами.

4 ПРОЄКТУВАННЯ ЗАСТОСУНКУ ІЗ ВИКОРИСТАННЯМ СИСТЕМИ АВТЕНТИФІКАЦІЇ ДЛЯ МОБІЛЬНИХ ПРИСТРОЇВ

4.1 Концепція застосунку

Застосунок є мобільним менеджером нотаток із можливістю збереження, редагування та безпечного керування записами користувачів. Основна увага насамперед приділяється не лише функціональності, а й забезпеченню високого рівня безпеки, включаючи захист доступу, автентифікацію користувачів та використання двофакторної автентифікації (2FA).

Залежно від вимог до конфіденційності та доступності даних, у рамках реалізації буде розглянуто дві версії застосунку: локальну, що використовує Keystore та EncryptedSharedPreferences, та віддалену, яка інтегрується з Firebase Firestore для синхронізації даних між пристроями. Такий підхід дозволить оцінити переваги та недоліки кожного варіанта та забезпечити гнучкість у виборі оптимального рішення для кінцевого користувача.

4.2 Вибір методу 2FA

Під час розробки системи автентифікації для мобільного застосунку було обрано метод двофакторної автентифікації (2FA) на основі TOTP (Time-based One-Time Password) алгоритму у зв’язці з Google Authenticator. Це рішення забезпечує високу безпеку, зручність використання та можливість функціонування як у локальному середовищі, так і з використанням віддаленого зберігання даних.

Однією з ключових вимог до системи було забезпечення гнучкої перевірки 2FA-кодів. При локальній перевірці токени генеруються безпосередньо на пристрої користувача, не передаючи секретний ключ у мережу. У разі використання віддаленого збереження реалізовано механізм зашифрованого зберігання TOTP-секрету з можливістю його відновлення.

Також було обрано відомий застосунок для роботи із TOTP-кодами – Google Authenticator. Він дозволив налаштувати систему двофакторної автентифікації без необхідності створення власного генератора OTP-кодів на стороні сервера.

Впровадження вище переліченого методу дозволило забезпечити ряд наступних переваг:

- Відсутність необхідності в Інтернет-з’єднанні під час перевірки коду.
- Захищене зберігання TOTP-секрету локально або у зашифрованому вигляді на сервері.
- Google Authenticator підтримується на всіх сучасних мобільних платформах.
- Код перевіряється за допомогою криптографічного алгоритму HMAC-SHA(1/256/512) з використанням як локальної, так і серверної обробки.

4.3 Архітектура застосунку

Для розробки мобільного застосунку було обрано архітектуру MVVM (Model-View-View-Model), що забезпечує гнучкість, підтримуваність та розділення бізнес-процесів від UI-компонентів. Такий підхід дозволяє тестувати логіку застосунку без додаткових зусиль, масштабувати проєкт, а також забезпечувати зручність інтеграції системи із іншими допоміжними технологіями.

Архітектура мобільного застосунку складається з таких основних рівнів:

1) Model (Модель) – рівень даних. Відповідає за управління даними застосунку. На цьому рівні відбувається обробка TOTP-кодів та перевірка їх коректності.

Містить реалізацію основних сховищ, включаючи:

- Локальне збереження: використання Room Database для збереження даних користувачів.
- Безпечне збереження секретів: використання Keystore API та EncryptedSharedPreferences для шифрування та збереження TOTP-секретного ключа.
- Можливість збереження на сервері (у разі необхідності).

Основні компоненти цього рівня: База даних (Room Database), DAO (Data Access Object) – для обробки SQL-запитів, Репозиторій – забезпечує взаємодію DAO та ViewModel.

2) ViewModel – рівень бізнес-логіки. На цьому рівні міститься уся логіка обробки даних перед передачею їх у View, а саме: управління процесами реєстрації, входу та перевірка двофакторної автентифікації. Слід зазначити, що у ViewModel використовується StateFlow та MutableStateFlow – інструменти, які дозволяють автоматично оновлювати UI при зміні стану.

Основні ViewModel: LoginViewModel – перевіряє логін, пароль та інші дані користувача перед входом. SignUpViewModel – обробляє реєстрацію нового користувача. TwoAuthViewModel – працює із TOTP-кодами. SecurityViewModel – дозволяє увімкнути або вимкнути двофакторну автентифікацію.

3) View (UI-компоненти) – рівень інтерфейсу. Відповідає за відображення даних та взаємодію із користувачем. Використовує компонент Fragment для розподілу логіки між екранами. Всі UI-компоненти підключаються до своїх відповідних ViewModel (якщо вони потрібні) для отримання та обробки даних.

4.4 Архітектурні схеми застосунку

Для більш наглядного та зрозумілого сприйняття далі представлено рисунок архітектурної взаємодії класів у першій версії застосунку, а саме з локальним зберіганням даних, що відповідають за безпеку в застосунку (рис. 4.1).

Окрім архітектурної взаємодії класів, що відповідають за безпеку, важливо також розглянути загальну структуру застосунку, яка охоплює всі рівні взаємодії компонентів. Далі представлено загальну архітектурну діаграму, що ілюструє взаємозв’язки між UI-компонентами, ViewModel, репозиторіями, базою даних та системою безпеки (рис. 4.2).

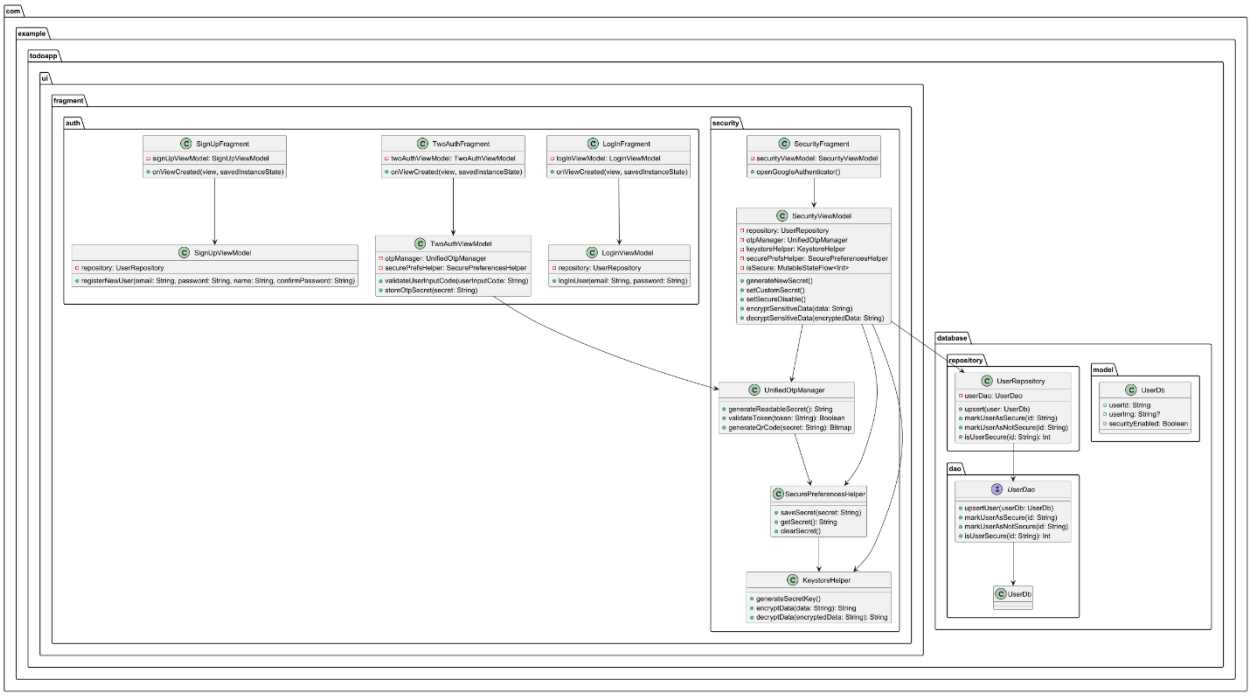

Рисунок 4.1 – Архітектура взаємодії класів безпеки у першій версії застосунку

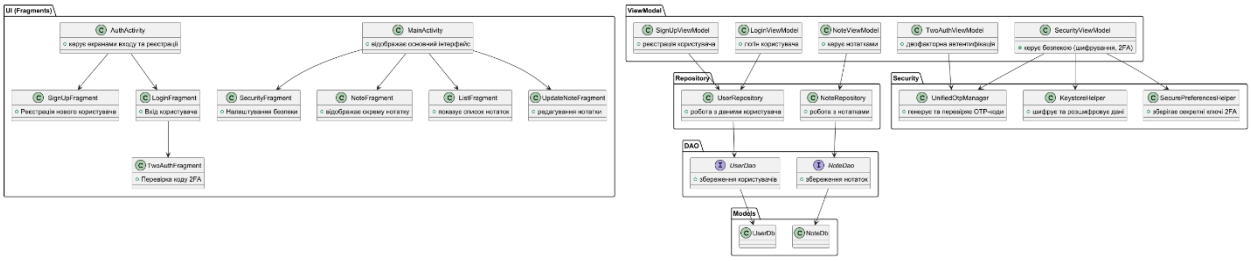

Рисунок 4.2 – Загальна структура у першій версії застосунку

Зважаючи на необхідність зберігання даних у віддаленому сховищі для покращення синхронізації між пристроями та можливості відновлення доступу до облікового запису, розглянемо другу версію, що використовує віддалене збереження даних. У цій версії двофакторна автентифікація (2FA) та користувацькі дані синхронізуються за допомогою Firebase Firestore, що забезпечує доступність інформації незалежно від пристрою користувача.

Рисунок 4.3 демонструє архітектурну взаємодію класів, що відповідають за безпеку у другій версії застосунку з використанням віддаленого сховища.

Як і в першому випадку, загальна структура застосунку з віддаленим збереженням містить ті ж основні рівні, але додатково передбачає взаємодію з Firebase Firestore для синхронізації даних між клієнтом і сервером. Далі представлено відповідну архітектурну діаграму для цієї версії (рис. 4.4).

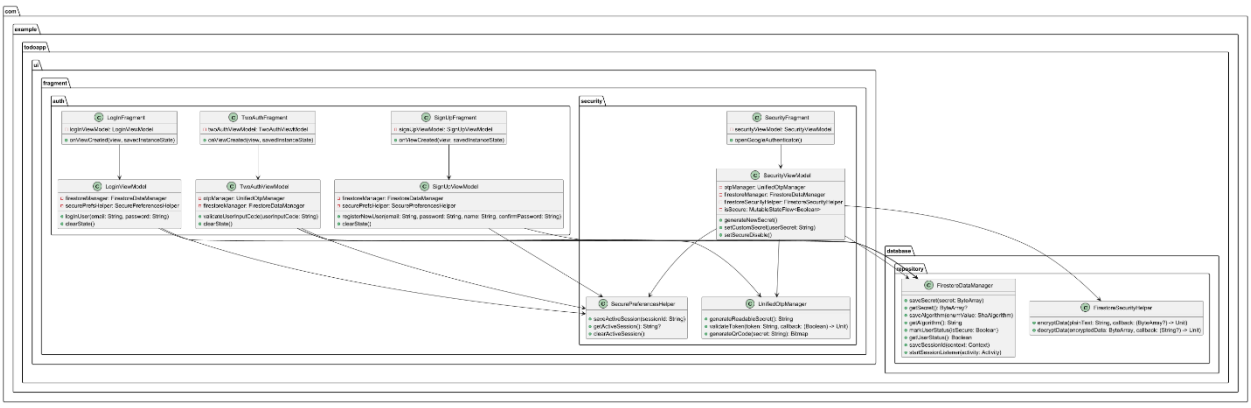

Рисунок 4.3 – Архітектура взаємодії класів безпеки у другій версії застосунку

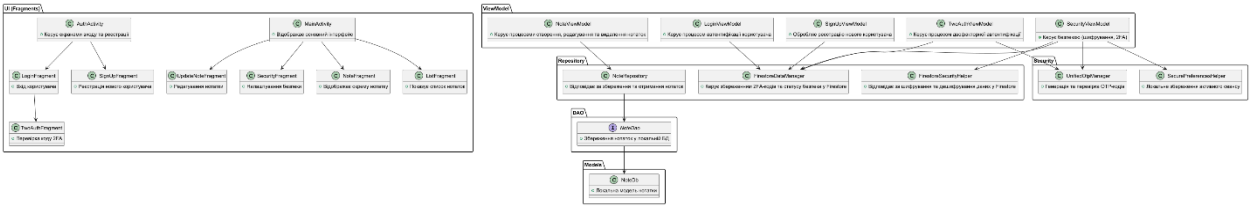

Рисунок 4.4 – Загальна структура у другій версії застосунку

4.5 Схема процесу автентифікації (реєстрація, авторизація)

Автентифікація в застосунку складається з двох основних етапів: реєстрації та авторизації. В залежності від вибраної архітектури застосунку ці процеси мають певні відмінності.

У першій версії застосунку, яка використовує локальне зберігання даних, усі операції з TOTP-секретом виконуються виключно на пристрої користувача, а процес входу здійснюється без створення унікального ідентифікатора сесії.

У другій версії, яка використовує віддалене зберігання через Firestore, після успішного входу система генерує ідентифікатор сесії, що дозволяє керувати активними сесіями користувачів та підвищує безпеку, унеможливлюючи одночасний вхід з різних пристроїв без повторної автентифікації.

Графічні схеми процесів, які ілюструють взаємодію користувача з системою у кожній з версій застосунку наведено у Додатку Г.

5 РЕАЛІЗАЦІЯ СИСТЕМИ АВТЕНТИФІКАЦІЇ НА KOTLIN

5.1 Створення проєкту Android-застосунку

Розробка мобільного застосунку для Android починається з налаштування середовища розробки, створення базової архітектури та інтеграції необхідних бібліотек. У цьому підрозділі розглянуто основні етапи створення проєкту та його підготовку до реалізації системи автентифікації.

Для розробки Android-застосунку використовується Android Studio – програмне забезпечення, яке надає зручний інструментарій для створення та тестування застосунків.

Попередньо необхідно завантажити останню версію Android Studio з офіційного сайту та виконати інсталяцію відповідно до офіційної документації. Після цього слід завантажити останні версії SDK та необхідні компоненти (Android SDK, Emulator, Platform Tools).

Далі на рисунку 5.1 створюється новий проєкт із використанням мови програмування Kotlin та встановлюється мінімальна версія API, рекомендована для підтримки сучасних функцій – Android 7.0 (API 24). Також обирається мова конфігурації збірки – Kotlin DSL.

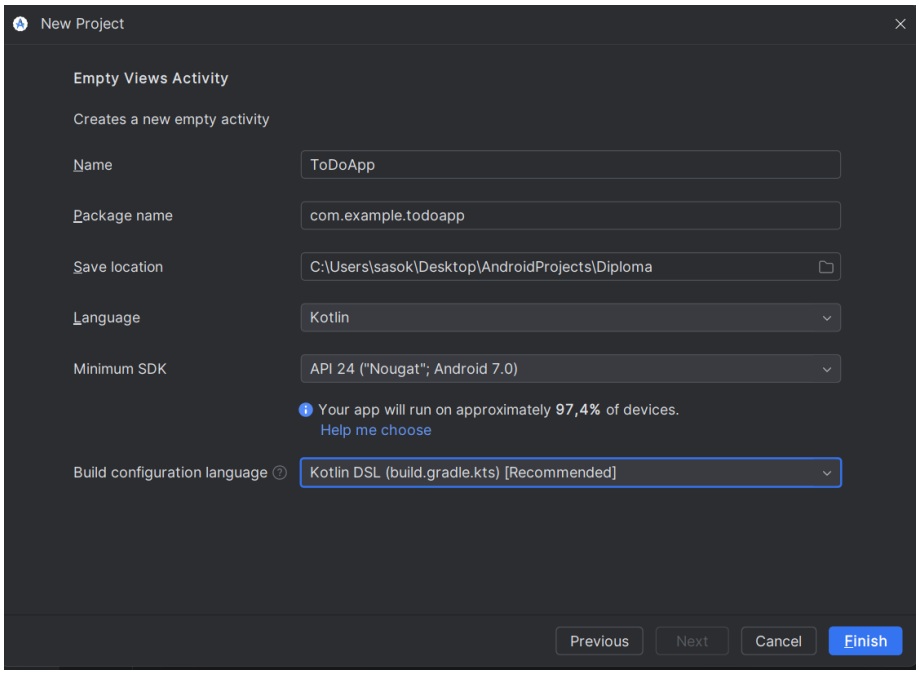

Рисунок 5.1 – Налаштування середовища розробки

Для коректної роботи застосунку до модульного файлу build.gradle.kts (Module: app) додаються відповідні бібліотеки. Нижче наведено список використаних бібліотек та їх призначення:

- androidx.core.ktx – Kotlin-розширення для стандартних API Android, що спрощують розробку;
- androidx.appcompat – забезпечує підтримку старих версій Android для нових компонентів;
- material – бібліотека Material Design для сучасного та гнучкого UI;
- androidx.activity – допомагає керувати життєвим циклом активностей у застосунку;
- androidx.constraintlayout – потужний менеджер розмітки для створення складних UI без вкладених ViewGroup;
- junit – стандартна бібліотека для модульного тестування;
- androidx.junit – інтеграція JUnit для тестування Android-компонентів;
- androidx.espresso.core – інструмент для тестування UI, що дозволяє автоматизувати взаємодію з елементами інтерфейсу.

Вище показані залежності, які є основою для роботи Android-застосунку, забезпечуючи підтримку основних функціональностей, UI-компонентів та тестування.

Окрім базових залежностей, до проєкту додаються бібліотеки, які розширюють його можливості, покращують безпеку, зручність роботи з базою даних та інтеграцію з хмарними сервісами:

- AndroidX Navigation Component (androidx.navigation.fragment, androidx.navigation.ui) – забезпечує зручну навігацію між екранами застосунку.
- Room Database (androidx.room.runtime, androidx.room.ktx) – ORM для роботи з локальною базою даних SQLite.
- Hilt (hilt.android, hilt.android.compiler) – фреймворк для ін’єкції залежностей, що спрощує управління компонентами.
- Firebase SDK (firebase.auth, firebase.firestore) – використовується для автентифікації користувачів та роботи з віддаленим сховищем.

- Android Security (androidx.security.crypto) – забезпечує шифрування конфіденційних даних у застосунку.
- AndroidX Biometric (androidx.biometric) – API для інтеграції біометричної автентифікації (Face ID, відбиток пальця).

Після додавання всіх залежностей проєкт синхронізується, а також виконується тестовий запуск для перевірки коректної роботи застосунку.

5.2 Опис процесу генерації та перевірки одноразових паролів (TOTP)

У Додатку А представлений вихідний код класу UnifiedOtpManager, який реалізує механізм генерації та перевірки одноразових паролів (TOTP) згідно зі стандартом RFC 6238. Цей код є частиною версії застосунку, що використовує локальне збереження даних, де всі операції зберігання секретного ключа виконуються безпосередньо на пристрої користувача.

Варто зазначити, що сама логіка генерації та перевірки TOTP-коду не відрізняється між версіями застосунку, незалежно від того, використовується локальне сховище чи віддалене (Firebase Firestore). Основна відмінність між цими двома підходами полягає у методах зберігання та шифрування секретного ключа:

- У версії з локальним сховищем використовується KeystoreHelper (Додаток А) для шифрування секретного ключа перед його збереженням у SecurePreferencesHelper (Додаток А).
- У версії з віддаленим сховищем дані шифруються перед тим, як зберігаються у Firestore через FirestoreSecurityHelper (Додаток А).

Генерація OTP-коду здійснюється на основі секретного ключа, який закодовується у форматі Base32 і не передається через мережу, що забезпечує високий рівень безпеки.

Для визначення поточного стану генерації OTP використовується концепція «timestamp». Час поділяється на рівні проміжки, кожен з яких триває 30 секунд. Відповідний часовий інтервал визначається за формулою:

$$T = (currentUnixTime - T_0) / timeStep,$$

де T_0 – початковий час (зазвичай 0, тобто 1 січня 1970 року), а timeStep – фіксований часовий інтервал.

Коли необхідно отримати OTP-код, використовується функція generateTokenForCounter() (Додаток А), яка виконує такі дії:

1) Обчислює поточний часовий інтервал T.

2) Використовує HMAC-алгоритм (HMAC-SHA1, HMAC-SHA256 або HMAC-SHA512) для гешування T за допомогою секретного ключа.

3) Виконує Truncate-функцію, яка вибирає частину отриманого гешу та приводить її до числового OTP-коду заданої довжини (6 цифр).

4) Код повертається у вигляді рядка та може бути використаний для автентифікації користувача.

Перевірка OTP виконується у функції validateToken() (Додаток А). Вона отримує введений користувачем код і порівнює його з OTP, згенерованим для поточного та сусідніх часових інтервалів (±1 інтервал). Це дозволяє компенсувати можливі розбіжності в часі.

Щоб система була більш безпечною та зручною у використанні, реалізація TOTP у цьому класі враховує кілька важливих моментів. По-перше, OTP-код оновлюється автоматично кожні 30 секунд, що забезпечує його актуальність та унеможливлює повторне використання. Крім того, передбачено можливість генерації спеціального URI, який можна використовувати для швидкого додавання ключа до Google Authenticator. Щоб спростити цей процес, реалізована функція створення QR-коду, що дозволяє користувачеві миттєво сканувати або натиснути на нього та підключити двофакторну автентифікацію. У версії з локальним зберіганням також реалізовано шифрування секретного ключа за допомогою KeystoreHelper, що гарантує його безпеку навіть у разі компрометації пристрою.

5.3 Порівняння використаних методів зберігання секрету. Локальне сховище та Firebase Firestore

Для збереження секретного ключа у застосунку було розглянуто два підходи: локальне сховище, що використовує Room Database, EncryptedSharedPreferences та Android Keystore, та віддалене зберігання у Firebase Firestore. Кожен із цих методів має свої переваги та обмеження, які впливають на вибір відповідного підходу залежно від потреб застосунку.

Локальне сховище забезпечує вищий рівень конфіденційності, оскільки секретний ключ залишається на пристрої користувача та не передається через мережу. Цей підхід також дозволяє працювати в автономному режимі без необхідності постійного Інтернет-з’єднання.

Firebase Firestore спрощує керування даними, дозволяє отримати доступ до них із різних пристроїв і забезпечує централізоване управління без необхідності вручну налаштовувати локальну базу даних.

Нижче наведена порівняльна таблиця 5.1, яка демонструє основні відмінності між двома підходами.

Таблиця 5.1 – Порівняння методів зберігання секрету

Критерій	Локальне сховище	Віддалене сховище
Доступність без Інтернету	Повністю автономна робота	Потребує підключення до мережі*
Безпека секретного ключа	Використовує Keystore для шифрування	Шифрування перед передачею у Firestore
Синхронізація між пристроями	Не підтримується	Дані доступні на будь-якому пристрої
Контроль сесій	Відсутній	Генерація унікального ідентифікатора сесії
Швидкість доступу	Миттєвий доступ без затримок	Можливі затримки через мережеві запити
Надійність зберігання	Дані залишаються навіть після виходу з облікового запису	Вимагає повторного підключення при зміні пристрою
Простота реалізації	Вимагає налаштування бази та шифрування	Інтегрується через SDK Firebase

*Важливо зауважити, Firestore підтримує офлайн-кешування, що дозволяє тимчасово працювати без підключення.

5.4 Використання біометрії, як додаткового способу автентифікації

Біометрична автентифікація використовується у застосунку як додатковий рівень захисту облікового запису користувача. Вона інтегрована таким чином, що її використання стає необхідним при виконанні критично важливих дій, пов’язаних із безпекою облікового запису.

У проєкті біометрична автентифікація активується лише для користувачів, які увімкнули двофакторну автентифікацію у налаштуваннях. Після її активації будь-

які зміни в параметрах безпеки, такі як вимкнення двофакторної автентифікації або оновлення секретного ключа TOTP, потребують додаткового підтвердження особи. Це дозволяє запобігти несанкціонованим змінам безпеки в разі компрометації пристрою.

Реалізація біометричної автентифікації заснована на API AndroidX Biometric, що підтримує такі методи перевірки особи, як сканер відбитків пальців, розпізнавання обличчя або використання загального паролю пристрою. У разі, якщо пристрій не підтримує біометричну автентифікацію, система автоматично пропонує використати альтернативний спосіб підтвердження через PIN-код, графічний ключ або пароль, встановлений на пристрої.

Процес біометричної перевірки реалізований за допомогою BiometricPrompt, який ініціює запит до користувача. Якщо автентифікація проходить успішно, користувач отримує доступ до зміни параметрів безпеки. У разі невдалої спроби автентифікації система надає кілька спроб, після чого блокує можливість повторної перевірки на певний час.

Якщо пристрій не підтримує біометричні методи або користувач не налаштував біометрію на своєму смартфоні, застосунок пропонує налаштувати загальну систему безпеки пристрою через налаштування Android. Це забезпечує можливість використання PIN-коду або пароля для доступу до налаштувань безпеки.

Біометрична автентифікація значно підвищує рівень безпеки, забезпечуючи захист налаштувань двофакторної автентифікації, та дозволяє використовувати сучасні методи перевірки особи для спрощення взаємодії користувача із застосунком без втрати рівня захисту.

5.5 Обробка помилок та забезпечення безпеки

У процесі автентифікації користувачів важливо не лише забезпечити зручність взаємодії, а й передбачити ефективну систему обробки помилок. У застосунку передбачені різні механізми обробки помилок у процесах входу, реєстрації та двофакторної автентифікації. Вони дозволяють коректно реагувати на помилки введення даних, некоректні коди OTP, невірні облікові дані, а також інші

можливі проблеми, пов’язані із взаємодією з Firebase або локальним збереженням даних.

Перед тим як здійснити реєстрацію або вхід, застосунок виконує валідацію введених даних. Наприклад:

- Перевіряється, чи заповнені всі обов’язкові поля (email, пароль, ім’я користувача).
- У разі реєстрації перевіряється збіг пароля та підтвердження пароля.
- Використовуються регулярні вирази для перевірки коректності email-адреси.

Якщо користувач вводить некоректні дані, система одразу відображає відповідне повідомлення з поясненням, що саме було введено неправильно.

У процесі автентифікації через Firebase обробляються різні типи помилок:

- Невірний пароль або email – якщо користувач вводить неправильні облікові дані, отримується помилка FirebaseAuthInvalidCredentialsException, яка обробляється у LoginViewModel і виводиться як підказка для користувача.
- Обліковий запис не існує – якщо користувач намагається увійти до неіснуючого облікового запису, отримується FirebaseAuthInvalidUserException, що повідомляє про необхідність реєстрації.
- Системні помилки Firebase – якщо виникають проблеми зі з’єднанням або сервером автентифікації, передбачено загальне повідомлення про неможливість входу.

Помилки автентифікації обробляються в LoginViewModel та передаються у AuthenticationState, після чого вони відображаються у LogInFragment через Snackbar.

У процесі двофакторної автентифікації (2FA) користувачеві надається три спроби введення коду. У TwoAuthViewModel реалізовано обмеження на кількість спроб введення некоректного OTP-коду. Якщо користувач тричі вводить неправильний код, система виводить попередження про блокування, а сесія завершується, змушуючи користувача повторно увійти в обліковий запис.

Якщо користувач забув пароль, застосунок пропонує функцію відновлення доступу. Користувачеві надсилається лист на електронну пошту із посиланням для скидання пароля.

Щоб уникнути несанкціонованого доступу, застосунок використовує механізм управління сесіями. У разі використання Firebase Firestore після входу генерується унікальний ідентифікатор сесії, що дозволяє слідкувати за активними сеансами та завершувати їх за потреби.

5.6 Тестування системи: покрокові інструкції

Для перевірки коректної роботи системи автентифікації та безпеки було виконано тестування на версії застосунку, яка використовує віддалене зберігання у Firebase Firestore. Саме ця версія дозволяє продемонструвати всі аспекти функціоналу, включаючи керування сесіями, двофакторну автентифікацію (2FA), біометричну автентифікацію та відновлення доступу до облікового запису. Нижче представлено покрокову інструкцію тестування із відповідними скріншотами, які ілюструють основні етапи взаємодії користувача із застосунком.

Після відкриття застосунку користувач потрапляє на головний екран, де він може виконати вхід або зареєструвати новий обліковий запис. Якщо користувач ще не має облікового запису, йому необхідно пройти процедуру реєстрації. Головний екран застосунку наведено у Додатку Б.

Для реєстрації необхідно заповнити всі обов’язкові поля. Якщо всі дані введені коректно, користувач успішно створює обліковий запис і отримує доступ до основного функціоналу застосунку. Форма введення даних для реєстрації та екран основного функціоналу застосунку після успішної реєстрації наведено у Додатку Б.

У випадку введення некоректних або порожніх даних при реєстрації, система повідомляє користувача про помилку. Валідація застосовується до всіх полів, що запобігає створенню облікового запису з некоректною інформацією. Тестування введення помилкових даних під час реєстрації наведено у Додатку Б.

При вході в систему користувач вводить свої облікові дані. У разі успішного введення застосунок відкриває головний екран із доступними функціями. Форму

входу та екран основного функціоналу застосунку після успішного входу наведено у Додатку Б.

Якщо користувач вводить неправильний пароль або email, система повідомляє про помилку. Також передбачено обробку ситуацій, коли поле email залишається порожнім, або, коли користувач вводить облікові дані, яких не існує у базі. Тестування введення помилкових даних під час авторизації наведено у Додатку Б.

Користувач має можливість увімкнути 2FA у налаштуваннях безпеки. Система генерує секретний ключ, який можна додати у Google Authenticator за допомогою QR-коду. Після цього користувач вводить одноразовий код для перевірки.

Для налаштування двофакторної автентифікації користувач переходить у налаштування безпеки, де він може створити секретний ключ або вручну ввести свій власний. Процес переходу до фрагменту із налаштуваннями 2FA та генерацію нового секрету із переходом до Google Authenticator наведено у Додатку Б.

Після створення секретного ключа користувач може додати його до Google Authenticator, просканувавши QR-код або натиснувши на нього. Це дозволяє синхронізувати генерацію OTP-кодів між застосунком і Google Authenticator. Додавання синхронізованого токену до Google Authenticator наведено у Додатку Б.

Користувач може вибрати алгоритм генерації TOTP-секрету (SHA1, SHA256 або SHA512). Якщо 2FA вже було налаштовано, то зміна алгоритму потребує повторного підключення секрету. Вибір потрібного алгоритму для створення секрету та заміна алгоритму, коли секрет до цього вже існував, а також приклади згенерованих секретів з використанням алгоритмів SHA1/SHA256/SHA512 послідовно наведено у Додатку Б.

При спробі встановити некоректний секрет система видає помилку. Якщо користувач хоче вимкнути двофакторну автентифікацію, він може зробити це в налаштуваннях безпеки. Тестування помилкового вводу секрету із використанням ручної установки та результат вимкнення 2FA наведено у Додатку Б.

Якщо 2FA активована, після введення логіна і пароля система запитує одноразовий пароль із Google Authenticator. Тільки після успішного введення OTP-коду користувач отримує доступ до застосунку. Авторизацію із підключеною двофакторною автентифікацією наведено у Додатку Б.

У випадку введення невірного OTP-коду система повідомляє про помилку. Після вичерпання доступних спроб користувач автоматично виходить із системи. Тестування помилкового вводу одноразового паролю, та вичерпання усіх доступних спроб наведено у Додатку Б.

Для підвищення безпеки для доступу до налаштувань безпеки було використано біометричну автентифікацію. Якщо пристрій підтримує цей метод, доступ до критичних налаштувань буде можливий лише після успішної біометричної перевірки або введення пароля пристрою. Тестування біометричної автентифікації, при успішному вводі паролю – успішна, при скасуванні – помилкова, наведено у Додатку Б.

Якщо користувач забув пароль, він може скористатися функцією відновлення доступу. Для цього потрібно ввести email, після чого на вказану пошту буде відправлено лист із посиланням для зміни пароля. Процес відновлення пароля в застосунку у разі його втрати або забуття та повідомлення на пошті користувача під час відновлення пароля наведено у Додатку Б.

Користувач може створювати, редагувати та переглядати нотатки у своєму обліковому записі. Це основний функціонал застосунку, який дозволяє зберігати важливу інформацію в захищеному середовищі. Усі скріншоти тестування функціоналу застосунку наведено у Додатку Б.

6 ТЕСТУВАННЯ ТА ОЦІНКА БЕЗПЕКИ

6.1 Тестування на вразливості

Для забезпечення належного рівня безпеки мобільного застосунку було проведено комплексне тестування на вразливості з використанням сучасних інструментів аналізу. Зокрема, застосовувалися такі засоби, як MobSF (Mobile Security Framework) – для статичного та динамічного аналізу Android-застосунку, Snyk – для виявлення вразливостей у використаних бібліотеках та залежностях, а також Burp Suite – для перехоплення мережевих запитів, перевірки автентифікаційного механізму та оцінки можливих атак на рівні API.

У процесі тестування було охоплено такі ключові аспекти:

- Аналіз програмного коду, написаного на Kotlin, з метою виявлення вразливостей, пов’язаних із неправильним керуванням ресурсами, небезпечними API, або порушеннями принципів безпечної розробки.
- Перевірка використаних бібліотек і сторонніх залежностей на наявність відомих вразливостей за допомогою інструменту Snyk, з метою виявлення потенційно застарілих або небезпечних компонентів.
- Оцінка механізмів автентифікації тестування на надійність захисту від перебору або інших способів обходу захисту.
- Тестування на перехоплення конфіденційних даних, логінів і паролів, через аналіз HTTP/HTTPS трафіку з використанням Burp Suite, включаючи перевірку захисту від MITM-атак.
- Перевірка відновлення пароля, включаючи тестування доставки електронної пошти та оцінку захищеності процесу від компрометації.

Особливу увагу було приділено версії застосунку з локальним зберіганням, оскільки саме вона потенційно містить більше точок атаки через збереження даних безпосередньо на пристрої користувача. У такій архітектурі підвищується ризик доступу до зашифрованих секретів, маніпуляцій із файлами конфігурацій або зламів при недостатньому захисті Keystore.

Далі буде представлено скріншоти виконання тестів та пояснення.

Результати загального тестування, проведеного з використанням MobSF, показали, що застосунок отримав оцінку безпеки 44 зі 100, що відповідає середньому рівню ризику (оцінка "B"). Це означає, що виявлені вразливості не є критичними, але потребують уваги. Загальний розподіл виявлених проблем показує наявність трьох високих ризиків і десяти середнього рівня. Також виявлено два потенційних трекери, які можуть мати вплив на приватність користувача. Один із компонентів був позначений як безпечний, що свідчить про наявність певних реалізацій безпеки, однак загальна оцінка вказує на необхідність подальшого вдосконалення захисту застосунку. Загальний результат статичного тестування .apk файлу застосунку за допомогою MobSF наведено у Додатку В.

Далі розглянемо окремо кожну з виявлених проблем.

Однією з серйозних вразливостей є те, що GenericIdpActivity (компонент Firebase Authentication) позначена як android:exported="true". Це означає, що ця Activity є доступною для інших застосунків на пристрої, і в теорії може бути викликана сторонніми програмами, що створює ризик для безпеки. У контексті даного застосунку, ця Activity виконує роль точки входу у процес автентифікації, що є вимогою Firebase SDK. Тому зміна параметра exported на false призведе до того, що застосунок не зможе запускатися, що унеможливлює застосування простого вирішення. Це вимагає додаткових заходів захисту, таких як обмеження викликів до цієї Activity лише з перевірених джерел або впровадження проміжної логіки перевірки. Вразливість через відкриту Activity GenericIdpActivity наведено у Додатку В.

Схожа проблема виявлена у Activity com.google.firebase.auth.internal.RecaptchaActivity. Вона є частиною Firebase Auth і також вимагає параметра exported="true" для коректної роботи. Зміна цього значення призведе до неможливості запуску частини механізмів автентифікації, тому в рамках проєкту ця вразливість є вимушено допустимою. Вразливість відкритої RecaptchaActivity наведено у Додатку В.

Ще однією серйозною проблемою стало використання режиму шифрування CBC (Cipher Block Chaining) у поєднанні з паддінгом PKCS5/PKCS7. Така

конфігурація вразлива до padding oracle атак, що дозволяють зловмиснику розшифрувати зашифровані дані без знання ключа, шляхом аналізу відповіді системи на змінені блоки даних. Це особливо небезпечно, коли передаються чутливі дані, і потребує перегляду криптографічної реалізації із переходом на більш безпечні режими шифрування, такі як GCM (Galois/Counter Mode). Вразливість CBC з PKCS5/PKCS7 Padding наведено у Додатку В.

Аналогічною до попередніх є вразливість RevocationBoundService. Цей компонент також має exported="true", що відкриває його для зовнішніх викликів. Зміна параметра на false унеможливить роботу служби, оскільки вона є точкою входу у механізм авторизації Google Sign-In. Це створює вимушений компроміс між функціональністю та безпекою, який потребує додаткових перевірок на рівні дозволів. Вразливість через невизначений рівень дозволу у службі наведено у Додатку В.

Подібна ситуація зафіксована і з компонентом androidx.profileinstaller.ProfileInstallerReceiver, який має атрибут exported="true" і захищений дозволом із невизначеним рівнем. Якщо дозвіл не має рівня "signature", сторонні застосунки можуть взаємодіяти з цим Broadcast Receiver, що створює ризик зловмисного використання. Вразливість ненадійного рівню дозволу у Broadcast Receiver наведено у Додатку В.

Під час тестування було виявлено, що застосунок використовує SQLite з виконанням сирих SQL-запитів. Це відкриває потенційну можливість SQL-ін’єкцій, але у програмному коді ця проблема частково нівельована через обробку виключень. Також зафіксовано відсутність шифрування в базі, хоча чутливі дані зберігаються окремо і шифруються. Вразливість використання сирих SQL-запитів наведено у Додатку В.

Окрема вразливість пов’язана з використанням ненадійного генератора випадкових чисел, що може призводити до передбачуваних значень, особливо у випадку токенів, OTP чи криптографічних ключів. Водночас, у програмному коді використовується криптографічно стійкий генератор – SecureRandom, що вказує на

можливу похибку сканера. Вразливість ненадійного генератору випадкових чисел наведено у Додатку В.

Ще одна проблема стосується зберігання жорстко закодованих чутливих даних у файлах, таких як логіни, паролі або ключі доступу. Такі практики загалом порушують принципи безпеки, проте у цьому випадку дані надійно захищені за допомогою Android Keystore та зашифровані. Вразливість жорстко закодованих чутливих даних у файлах наведено у Додатку В.

На завершення розгляду цього виду тестування, у застосунку виявлено два трекери, які можуть здійснювати збір даних про дії користувача або ідентифікувати пристрій. Це не є критичною вразливістю, але створює ризики для конфіденційності. У цьому випадку йдеться про аналітику Firebase Crashlytics, яка використовується для відстеження критичних помилок. Вразливість наявності трекерів конфіденційності наведено у Додатку В.

У результаті перевірки бібліотек за допомогою Snyk було виявлено низку вразливостей, значна частина яких пов’язана з неактуальними версіями бібліотек – зокрема kotlin-stdlib, protobuf, netty, guava та інші. Хоча Snyk надає конкретні рекомендації щодо оновлення, практична реалізація цього виявилася проблематичною. Оновлення деяких залежностей призводить до конфліктів між бібліотеками, через що неможливо виконати збірку проєкту (build). Таким чином, повне автоматичне оновлення всіх компонентів наразі неможливе без перебудови частини архітектури або переходу на інші стабільні версії бібліотек. Поточні результати тестування Snyk та перелік вразливих залежностей наведено у Додатку В.

Під час перевірки на стійкість до брутфорс-атак було використано інструмент Burp Suite Intruder, налаштований на перебір найбільш популярних паролів для стандартного сценарію входу через Firebase Authentication. Для атаки використовувався метод Sniper, з підстановкою типових слабких значень паролю у поле password у POST-запиті до Google Identity Toolkit.

У результаті спроб перебору, сервер повернув код помилки 400 із повідомленням "TOO_MANY_ATTEMPTS_TRY_LATER", що свідчить про

наявність механізму обмеження кількості запитів. Це вказує на те, що сторонній механізм авторизації (у даному випадку Google/Firebase) реалізує захист від грубої сили. Таким чином, ризик брутфорс-атак знижується до мінімуму, оскільки подальші запити блокуються ще до повного завершення словника. Вразливість до стандартного перебору не виявлена – механізм блокування спрацьовує коректно. Налаштування передвиконання Brute Force – атака та її результат наведено у Додатку В.

Під час моделювання спроби повторної реєстрації існуючого користувача було здійснено запит до API Firebase через Burp Suite з новим паролем, але використанням тієї ж адреси електронної пошти. У відповідь сервер повернув статус-код 400 та повідомлення про помилку: "EMAIL_EXISTS" – що свідчить про те, що реєстрація з уже наявним email заблокована на рівні бекенду. Це є позитивним результатом з точки зору безпеки: система не дозволяє обійти механізм автентифікації шляхом повторної реєстрації з тим самим обліковим записом, але іншим паролем. Отже, ризик захоплення облікового запису через повторну реєстрацію виключено. Спробу повторної реєстрації існуючого користувача наведено у Додатку В.

Було проведено другу серію брутфорс-атак, під час якої одночасно варіювалися як email-адреси, так і паролі. Тест виконано з використанням типу атаки Cluster Bomb у Burp Suite Intruder – для симуляції повного перебору комбінацій логін/пароль. На відміну від попереднього тесту, в якому спрацьовував захист після певної кількості запитів, цього разу система заблокувала навіть валідний пароль після кількох неправильних спроб. У відповідь на запити із правильною парою облікових даних сервер повернув код 400 з повідомленням:

message: "TOO_MANY_ATTEMPTS_TRY_LATER".

Це вказує на наявність ефективного механізму блокування за кількістю спроб, незалежно від того, чи коректний пароль. Така поведінка значно ускладнює brute-force-атаки та є позитивною з точки зору безпеки. Система успішно виконує обмеження за частотою запитів і не допускає зловмисника до тестування навіть

валідних паролів після перевищення порогу. Налаштування другої вже ускладненої Brute Force атаки та приклад запиту та відповіді наведено у Додатку В.

Для перевірки наявності захисту від user enumeration (визначення наявності облікового запису за email-адресою) було здійснено кілька запитів до Firebase API із вказанням різних адрес електронної пошти, зокрема як зареєстрованих, так і вигаданих. Усі запити супроводжувалися навмисно некоректним паролем. У відповідь сервер у всіх випадках повернув однаковий статус-код. Це вказує на те, що API не розкриває, чи існує користувач з вказаною адресою. Незалежно від того, чи email дійсний, відповідь є уніфікованою, що унеможливлює ідентифікацію зареєстрованих користувачів шляхом перебору адрес. Такий підхід відповідає рекомендаціям OWASP щодо захисту від user enumeration. Результати перевірки на user enumeration наведено у Додатку В.

Після багаторазових невдалих спроб входу та перебору паролів на одному з облікових записів, система активувала механізм захисту від підозрілої активності. У результаті:

Увійти в обліковий запис стало неможливо навіть із валідними даними, оскільки спроби авторизації блокуються повідомленням: "Authentication failed: We have blocked all requests from this device due to unusual activity." Крім того, спроба відновлення пароля через email-посилання також завершилася невдачею, оскільки система повідомила, що посилання або вже використано, або його термін дії закінчився.

Це підтверджує, що Firebase реалізує мультикомпонентний захист – не лише блокує brute-force спроби, але й вводить тимчасові обмеження на дії з боку підозрілих користувачів або пристроїв, включаючи функціонал відновлення доступу. Зазначений підхід підвищує загальний рівень безпеки системи та унеможливлює автоматизоване зловживання. Результати блокування через підозрілу активність наведено у Додатку В.

6.2 Оцінка ефективності захисту

На основі проведеного тестування можна зробити висновок, що мобільний застосунок демонструє загалом прийнятний рівень безпеки, з реалізованими

базовими механізмами захисту як на рівні клієнтської частини, так і бекенду. Статичний аналіз виявив низку структурних вразливостей середнього рівня ризику, багато з яких обумовлені специфікою роботи Firebase SDK та необхідністю використання exported компонентів. Динамічне тестування через Burp Suite підтвердило ефективність вбудованих механізмів захисту від brute-force атак, повторної реєстрації, user enumeration та надмірної активності. Аналіз бібліотек за допомогою Snyk засвідчив наявність потенційно небезпечних залежностей, проте їх усунення ускладнене через конфлікти сумісності між версіями. Водночас, критичних вразливостей, які дозволяли б пряме втручання або компрометацію даних, виявлено не було. Таким чином, рівень захищеності застосунку можна оцінити як середній із тенденцією до покращення, за умови усунення виявлених недоліків і впровадження оновлень у відповідності до сучасних практик безпечної розробки.

6.3 Можливі сценарії атак і реакція системи

У рамках тестування було змодельовано низку поширених атак (brute-force, повторна реєстрація, user enumeration, зловживання API), на які система адекватно реагує завдяки вбудованим обмеженням доступу, уніфікованим повідомленням про помилки та тимчасовим блокуванням пристроїв після підозрілої активності. Проте існують і потенційні сценарії атак, які можуть становити загрозу за певних умов.

Одним із таких сценаріїв є фізичний доступ до пристрою, на якому встановлено застосунок. Навіть у цьому випадку ризик компрометації даних значно знижено завдяки кільком рівням захисту. По-перше, застосунок зберігає чутливу інформацію у внутрішній файловій системі пристрою, доступ до якої можливий лише при root-доступі або спеціальних інструментах. По-друге, навіть якщо зловмиснику вдасться витягти ці файли, дані будуть зашифровані за симетричним алгоритмом AES, що унеможливлює пряме їх використання без знання ключа.

Більше того, ключі, що використовуються для шифрування, не зберігаються в явному вигляді – вони захищені через Android Keystore, який апаратно ізольований і не дозволяє доступ до ключа без проходження автентифікації. Навіть якщо зловмисник зможе отримати частину зашифрованих даних, йому доведеться

повністю відтворити алгоритм розшифрування, що реалізований у програмному коді. Без знання внутрішньої логіки, послідовності операцій і додаткових параметрів (ініціалізаційного вектору, режиму шифрування тощо), ймовірність успішного дешифрування надзвичайно низька.

Ще одним важливим елементом захисту є двофакторна автентифікація. Навіть за умови доступу до пристрою або облікового запису, зловмисник не зможе змінити параметри безпеки (наприклад, вимкнути 2FA або змінити секретний ключ), оскільки доступ до налаштувань автентифікації захищено біометричною перевіркою. Це забезпечує надійний захист навіть у випадку викрадення або тимчасового використання пристрою.

До можливих додаткових сценаріїв атак можна також віднести:

- MITM-атаки під час публічних Wi-Fi-з’єднань – однак трафік у застосунку проходить через HTTPS із коректно налаштованим TLS, тому перехоплення даних унеможливлено без компрометації сертифікатів.
- Reverse Engineering застосунку – застосунок містить мінімум жорстко закодованої інформації, а всі чутливі операції виконуються через захищені API Firebase, що знижує ефективність такого підходу.
- Атаки на сторонні бібліотеки – хоча деякі з них містять відомі вразливості, вони не використовуються у критичних зонах застосунку, і їх вплив обмежено.

Таким чином, навіть у випадку складних, комбінованих сценаріїв, застосунок демонструє високий рівень стійкості до атак, завдяки комплексному захисту на рівні зберігання, автентифікації, мережевого трафіку та внутрішньої логіки взаємодії з даними.

ВИСНОВКИ

У межах виконання дипломної роботи було досліджено, реалізовано та протестовано сучасні підходи до автентифікації в мобільних застосунках на платформі Android. На тлі постійного зростання цифрових загроз і активності кіберзлочинців, захист персональних даних користувачів став пріоритетним завданням розробників мобільного ПЗ. Як було показано у вступі, з огляду на широке поширення мобільних пристроїв та їхній доступ до критично важливої інформації, питання забезпечення надійної автентифікації є вкрай актуальним.

У ході роботи проведено ґрунтовний аналіз методів автентифікації, зокрема однофакторної та багатофакторної, їх переваг і недоліків. Особливу увагу приділено дослідженню вразливостей, які можуть виникати при використанні застарілих або недостатньо захищених механізмів автентифікації. На основі проведеного аналізу сформовано вимоги до безпечної системи автентифікації, адаптованої до реалій мобільного середовища.

У результаті практичної частини проєкту реалізовано багатофакторну систему автентифікації, що базується на технологіях Firebase Authentication, Google Authenticator та біометричній автентифікації, яка дозволяє значно посилити захист користувацьких облікових записів. Застосунок був протестований за допомогою інструментів MobSF, Snyk та Burp Suite, що дозволило виявити потенційні вразливості й оцінити реакцію системи на типові сценарії атак.

Отримані результати засвідчили, що реалізована система автентифікації забезпечує високий рівень безпеки, демонструє ефективну стійкість до brute-force атак, спроб підбору паролів, атак на API та інших типових загроз. Усі критичні операції додатково захищені біометрією, а конфіденційні дані – надійно зашифровані за допомогою AES та Android Keystore, що унеможливлює їх компрометацію навіть у разі фізичного доступу до пристрою.

Таким чином, поставлені завдання були повністю виконані, а досягнуті результати підтверджують доцільність впровадження багаторівневих, зручних та захищених методів автентифікації в сучасних мобільних застосунках.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. The password thicket: technical and market failures in human authentication on the web. [Електронний ресурс]: https://jbonneau.com/doc/BP10-WEIS-password_thicket.pdf.

2. Single Factor Authentication (SFA) – Exploring its role. [Електронний ресурс]. – Режим доступу: https://www.fraud.com/post/single-factor-authentication#Benefits_of_Single_Factor_Authentication.

3. What is a Time-based One-time Password (TOTP)? [Електронний ресурс]. – Режим доступу: https://www.twilio.com/docs/glossary/totp.

4. Two-factor authentication (2FA) comparison of methods and applications. [Електронний ресурс]: https://bibliotekanauki.pl/articles/31233162.pdf.

5. THE BENEFITS OF TWO-FACTOR-AUTHENTIACTION (2FA). [Електронний ресурс]: https://cerrt.ng/Uploads/a07518d3-0be6-48a5-90b6-383e15a88b46.pdf.

6. What are the most common two-factor authentication vulnerabilities? [Електронний ресурс]. – Режим доступу: https://www.linkedin.com/advice/3/what-most-common-two-factor-authentication-yz2we.

7. What is a Time-Based One-Time Password (TOTP)? [Електронний ресурс]. – Режим доступу: https://www.keepersecurity.com/resources/glossary/what-is-a-time-based-one-time-password/.

8. What Is Push Authentication (2FA)? [Електронний ресурс]. – Режим доступу: https://www.1kosmos.com/authentication/push-authentication/.

9. What is 3-factor authentication (3FA)? [Електронний ресурс]. – Режим доступу: https://www.infobip.com/glossary/3-factor-authentication.

10. What Is 3FA? Three-Factor Authentication for Beginners. [Електронний ресурс]. – Режим доступу: https://techjury.net/blog/what-is-3fa/.

11. Google Workspace. Learn about authentication and authorization. [Електронний ресурс]. – Режим доступу: https://developers.google.com/workspace/guides/auth-overview.

12. What Is Biometric Authentication? [Електронний ресурс]. – Режим доступу: https://www.spiceworks.com/it-security/identity-access-management/articles/what-is-biometric-authentication-definition-benefits-tools/.

13. A Usability Study of Five Two-Factor Authentication Methods. [Електронний ресурс]: https://www.usenix.org/system/files/soups2019-reese.pdf.

14. Biometric Security Risks. [Електронний ресурс]: https://socradar.io/biometric-security-risks-beyond-fingerprints-and-facial-recognition/.

15. Що таке Firebase? [Електронний ресурс]. – Режим доступу: https://avada-media.ua/services/firebase/.

16. Firebase [Електронний ресурс]. – Режим доступу: https://firebase.google.com/products/auth.

17. Firebase Advantages and Disadvantages. [Електронний ресурс]. – Режим доступу: https://blog.back4app.com/firebase-advantages-and-disadvantages/.

18. What is Google Authenticator? [Електронний ресурс]. – Режим доступу: https://www.manageengine.com/products/self-service-password/blog/mfa/what-is-google-authenticator.html.

19. Android Keystore system. [Електронний ресурс]. – Режим доступу: https://developer.android.com/privacy-and-security/keystore.

20. Formal Analysis and Verification of OAuth 2.0 in SSO. [Електронний ресурс]. – Режим доступу: https://aaltodoc.aalto.fi/server/api/core/bitstreams/063d7790-3f2b-4df4-ac0b-2a6e8fc87855/content.

21. An Introduction to OAuth 2. [Електронний ресурс]. – Режим доступу: https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2.

22. The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines. [Електронний ресурс]. – Режим доступу: https://arxiv.org/pdf/1704.08539.

23. PlantUML. [Електронний ресурс]. – Режим доступу: https://plantuml.com/en/class-diagram.

24. PortSwigger. BurpSuite. [Електронний ресурс]. – Режим доступу: https://portswigger.net/burp.

25. MobSF. [Електронний ресурс]. – Режим доступу: https://github.com/MobSF/Mobile-Security-Framework-MobSF.

26. Snyk. The AI-powered developer security platform. [Електронний ресурс]. – Режим доступу: https://snyk.io/.

ДОДАТОК А

ВИХІДНИЙ КОД КЛАСУ UNIFIEDOTPMANAGER

```
class UnifiedOtpManager @Inject constructor(
    @ApplicationContext private val context: Context,
) {
    private val codeDigits = 6
    private val timeStep = 30L
    private val timeUnit = TimeUnit.SECONDS

    private val sizeOfSecretInBytes = MutableStateFlow(0)

    private val secretFlow = MutableStateFlow<ByteArray?>(null)

    private val algorithm = MutableStateFlow("")

    init {
        initAlgorithmFromPrefs()
    }

    private fun initAlgorithmFromPrefs() {
        val savedAlgorithm = SecurePreferencesHelper.getEnum(context)
        if (savedAlgorithm.isNotEmpty()) {
            algorithm.value = savedAlgorithm
            sizeOfSecretInBytes.value = ShaAlgorithm.entries.find { it.algorithm ==
algorithm.value }?.sizeOfSecret ?: 32
        }
    }

    fun updateAlgorithm(newAlgorithm: ShaAlgorithm) {
        algorithm.value = newAlgorithm.algorithm
        sizeOfSecretInBytes.value = newAlgorithm.sizeOfSecret
        SecurePreferencesHelper.saveEnum(context, newAlgorithm)
    }

    fun clearAlgorithm(){
        algorithm.value = ""
    }

    fun updateSecret(newSecret: String) {
        val encodedSecret = Base32().decode(newSecret.uppercase().replace("=", ""))
        secretFlow.value = encodedSecret
    }

    fun generateReadableSecret(): String {
        if (algorithm.value == ""){
            algorithm.value = ShaAlgorithm.SHA256.algorithm
            sizeOfSecretInBytes.value = ShaAlgorithm.SHA256.sizeOfSecret
        }
        val allowedChars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ234567"
        val random = SecureRandom()
        return (1..sizeOfSecretInBytes.value)
            .map { allowedChars[random.nextInt(allowedChars.length)] }
            .joinToString("")
    }

    @OptIn(ExperimentalCoroutinesApi::class)
    fun tokenFlow(): Flow<String?> {
        return algorithm.flatMapLatest {
            secretFlow.flatMapLatest { secret ->
```

```
                flow {
                    if (secret != null) {
                        if (secret.isEmpty()) {
                            emit("")
                            return@flow
                        }
                    }
                    while (true) {
                        val currentTime = System.currentTimeMillis()
                        val currentCounter = currentTime / timeUnit.toMillis(timeStep)

                        val currentToken = secret?.let { generateTokenForCounter(it,
currentCounter) }
                        emit(currentToken)

                        val nextExecutionTime =
                            ((currentTime / timeUnit.toMillis(timeStep)) + 1) *
timeUnit.toMillis(
                                timeStep
                            )
                        delay(nextExecutionTime - currentTime)
                    }
                }
            }
        }.flowOn(Dispatchers.Default)
    }

    fun takeSecret(): ByteArray {
        val secret = SecurePreferencesHelper.getSecret(context) ?: return ByteArray(0)
        val decryptedSecret = KeystoreHelper.decryptData(secret)

        return try {
            val decodedSecret = Base32().decode(decryptedSecret)
            secretFlow.value = decodedSecret
            decodedSecret
        } catch (e: Exception) {
            ByteArray(0)
        }
    }

    fun validateToken(inputCode: String): Boolean {
        val secretKey = takeSecret()

        if (secretKey.isEmpty()) return false
        val currentCounter = currentCounter()

        return (-1..1).any { offset ->
            inputCode == generateTokenForCounter(secretKey, currentCounter + offset)
        }
    }

    private fun generateTokenForCounter(secretKey: ByteArray, counter: Long): String {
        val hash = calculateHmac(secretKey, counter)
        val offset = (hash.last() and 0x0F).toInt()
        val otp =
            (ByteBuffer.wrap(hash.copyOfRange(offset, offset + 4)).int and 0x7FFFFFFF) %
10.0.pow(
                codeDigits
            ).toInt()

        return otp.toString().padStart(codeDigits, '0')
    }
```

```
    fun buildOtpUri(account: String, issuer: String): String {
        val secretKey = takeSecret()
        if (secretKey.isEmpty()) return ""
        val encodedSecret = Base32().encodeToString(secretKey).replace("=", "")
        val algorithm = algorithm.value.removePrefix("Hmac")
        return
"otpauth://totp/$issuer:$account?secret=$encodedSecret&issuer=$issuer&algorithm=$algorithm&
digits=$codeDigits&period=$timeStep"
    }

    fun generateQrCode(content: String): Bitmap? {
        if (content.isEmpty()) return null
        return try {
            val size = 180
            val qrCodeWriter = QRCodeWriter()
            val bitMatrix = qrCodeWriter.encode(content, BarcodeFormat.QR_CODE, size, size)
            Bitmap.createBitmap(size, size, Bitmap.Config.RGB_565).apply {
                for (x in 0 until size) {
                    for (y in 0 until size) {
                        setPixel(x, y, if (bitMatrix[x, y]) Color.BLACK else Color.WHITE)
                    }
                }
            }
        } catch (e: Exception) {
            null
        }
    }

    private fun calculateHmac(key: ByteArray, counter: Long): ByteArray {
        if (algorithm.value == ""){
            algorithm.value = ShaAlgorithm.SHA256.algorithm
            sizeOfSecretInBytes.value = ShaAlgorithm.SHA256.sizeOfSecret
        }
        val mac = Mac.getInstance(algorithm.value)
        mac.init(SecretKeySpec(key, algorithm.value))
        return mac.doFinal(ByteBuffer.allocate(8).putLong(counter).array())
    }

    private fun currentCounter(): Long {
        return System.currentTimeMillis() / timeUnit.toMillis(timeStep)
    }
}
```

ВИХІДНИЙ КОД КЛАСУ KEYSTOREHELPER

```
object KeystoreHelper {

    private const val KEY_ALIAS = "TOTP_SECRET_KEY"
    private const val ANDROID_KEYSTORE = "AndroidKeyStore"
    private const val TRANSFORMATION = "AES/GCM/NoPadding"

    fun generateSecretKey() {
        val keyStore = KeyStore.getInstance(ANDROID_KEYSTORE).apply { load(null) }
        if (!keyStore.containsAlias(KEY_ALIAS)) {
            val keyGenerator =
KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES, ANDROID_KEYSTORE)
            val keyGenParameterSpec = KeyGenParameterSpec.Builder(
                KEY_ALIAS,
                KeyProperties.PURPOSE_ENCRYPT or KeyProperties.PURPOSE_DECRYPT
            ).setBlockModes(KeyProperties.BLOCK_MODE_GCM)
```

```
                .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
                .build()
            keyGenerator.init(keyGenParameterSpec)
            keyGenerator.generateKey()
        }
    }

    private fun getSecretKey(): SecretKey {
        val keyStore = KeyStore.getInstance(ANDROID_KEYSTORE).apply { load(null) }
        return keyStore.getKey(KEY_ALIAS, null) as SecretKey
    }

    fun encryptData(plainText: String): ByteArray {
        val cipher = Cipher.getInstance(TRANSFORMATION)
        cipher.init(Cipher.ENCRYPT_MODE, getSecretKey())
        val iv = cipher.iv
        val encryptedBytes = cipher.doFinal(plainText.toByteArray(Charsets.UTF_8))
        return iv + encryptedBytes
    }

    fun decryptData(encryptedData: ByteArray): String {
        val iv = encryptedData.copyOfRange(0, 12)
        val encryptedBytes = encryptedData.copyOfRange(12, encryptedData.size)
        val cipher = Cipher.getInstance(TRANSFORMATION)
        cipher.init(Cipher.DECRYPT_MODE, getSecretKey(), GCMParameterSpec(128,
iv))
        return String(cipher.doFinal(encryptedBytes), Charsets.UTF_8)
    }
}
```

ВИХІДНИЙ КОД КЛАСУ SECUREPREFERENCEHELPER

```
object SecurePreferencesHelper {

    private const val PREFS_NAME = "secure_prefs"

    private var auth = FirebaseAuth.getInstance()

    private fun getPrefs(context: Context): SharedPreferences {
        return context.getSharedPreferences(PREFS_NAME, Context.MODE_PRIVATE)
    }

    fun saveSuccess(context: Context, success: String) {
        val successBytes = success.toByteArray(Charsets.UTF_8)
        val encryptedSuccess = Base32().encodeToString(successBytes).replace("=",
"")
        getPrefs(context).edit()

.putString("encrypted_success_${FirebaseAuth.getInstance().currentUser?.uid}",
encryptedSuccess)
            .apply()
    }

    fun getSuccess(context: Context): String {
        val encryptedSuccess =
getPrefs(context).getString("encrypted_success_${FirebaseAuth.getInstance().curren
tUser?.uid}", null)
```

```
        if (encryptedSuccess.isNullOrEmpty()) return ""

        val decodedBytes = Base32().decode(encryptedSuccess)
        val decodedString = String(decodedBytes, Charsets.UTF_8)

        return decodedString
    }

    fun saveSecret(context: Context, secret: ByteArray) {
        val encryptedSecret = secret.toHex()

getPrefs(context).edit().putString("encrypted_secret_${auth.currentUser?.uid}",
encryptedSecret).apply()
    }

    fun getSecret(context: Context): ByteArray? {
        val encryptedSecretHex =
getPrefs(context).getString("encrypted_secret_${auth.currentUser?.uid}", null)
        return encryptedSecretHex?.decodeHex()
    }

    fun saveEnum(context: Context, enumValue: ShaAlgorithm) {
        val encryptedEnum =
Base32().encodeToString(enumValue.algorithm.toByteArray(Charsets.UTF_8)).replace("
=", "")

getPrefs(context).edit().putString("encrypted_enum_${auth.currentUser?.uid}",
encryptedEnum).apply()
    }

    fun getEnum(context: Context): String {
        val encryptedEnum =
getPrefs(context).getString("encrypted_enum_${auth.currentUser?.uid}", null)

        return try {
            val decodedBytes = Base32().decode(encryptedEnum)
            String(decodedBytes, Charsets.UTF_8)
        } catch (e: Exception) {
            ""
        }
    }

    fun clearEnum(context: Context) {

getPrefs(context).edit().remove("encrypted_enum_${auth.currentUser?.uid}").apply()
    }

    fun clearSecret(context: Context) {
        val prefs = getPrefs(context)
        val editor = prefs.edit()
        val userId = auth.currentUser?.uid
        if (userId != null) {
            editor.remove("encrypted_secret_$userId")
            editor.apply()
        }
    }


    private fun ByteArray.toHex(): String = joinToString("") { "%02x".format(it) }
```

```
    private fun String.decodeHex(): ByteArray {
        val result = ByteArray(length / 2)
        for (i in indices step 2) {
            result[i / 2] = ((this[i].digitToInt(16) shl 4) + this[i +
1].digitToInt(16)).toByte()
        }
        return result
    }
}
```

ВИХІДНИЙ КОД КЛАСУ FIRESTORESECURITYHELPER

```
object FirestoreSecurityHelper {

    private const val TRANSFORMATION = "AES/GCM/NoPadding"
    private const val FIRESTORE_COLLECTION = "users"
    private const val SECRET_FIELD = "secret_key_aes"
    private const val KEY_SIZE = 256

    @SuppressLint("StaticFieldLeak")
    private val firestore = FirebaseFirestore.getInstance()
    private val auth = FirebaseAuth.getInstance()

    private fun getOrCreateSecretKey(callback: (SecretKey?) -> Unit) {
        val userId = auth.currentUser?.uid
        if (userId == null) {
            callback(null)
            return
        }

        val userDoc = firestore.collection(FIRESTORE_COLLECTION)
            .document(userId)
        userDoc.get()
            .addOnSuccessListener { document ->
            if (document.exists() && document.contains(SECRET_FIELD)) {
                val keyString = document.getString(SECRET_FIELD)
                if (!keyString.isNullOrEmpty()) {
                    val keyBytes = Base64.decode(keyString, Base64.DEFAULT)
                    val secretKey = SecretKeySpec(keyBytes, "AES")
                    callback(secretKey)
                    return@addOnSuccessListener
                }
            }
            val newKey = generateSecretKey()
            val encodedKey = Base64.encodeToString(newKey.encoded, Base64.DEFAULT)
            userDoc.set(mapOf(SECRET_FIELD to encodedKey), SetOptions.merge())
                .addOnSuccessListener { callback(newKey) }
                .addOnFailureListener { callback(null) }
        }.addOnFailureListener {
            callback(null)
        }
    }

    private fun generateSecretKey(): SecretKey {
        val keyGenerator = KeyGenerator.getInstance("AES")
        keyGenerator.init(KEY_SIZE)
        return keyGenerator.generateKey()
    }
```

```
    fun encryptData(plainText: String, callback: (ByteArray?) -> Unit) {
        getOrCreateSecretKey { secretKey ->
            if (secretKey == null) {
                callback(null)
                return@getOrCreateSecretKey
            }
            try {
                val cipher = Cipher.getInstance(TRANSFORMATION)
                cipher.init(Cipher.ENCRYPT_MODE, secretKey)
                val iv = cipher.iv
                val encryptedBytes =
cipher.doFinal(plainText.toByteArray(Charsets.UTF_8))
                callback(iv + encryptedBytes)
            } catch (e: Exception) {
                e.printStackTrace()
                callback(null)
            }
        }
    }

    fun decryptData(encryptedData: ByteArray, callback: (String?) -> Unit) {
        getOrCreateSecretKey { secretKey ->
            if (secretKey == null) {
                callback(null)
                return@getOrCreateSecretKey
            }
            try {
                val iv = encryptedData.copyOfRange(0, 12)
                val encryptedBytes = encryptedData.copyOfRange(12,
encryptedData.size)
                val cipher = Cipher.getInstance(TRANSFORMATION)
                cipher.init(Cipher.DECRYPT_MODE, secretKey, GCMParameterSpec(128,
iv))
                val decryptedText = String(cipher.doFinal(encryptedBytes),
Charsets.UTF_8)
                callback(decryptedText)
            } catch (e: Exception) {
                e.printStackTrace()
                callback(null)
            }
        }
    }
```

ДОДАТОК Б

СКРІНШОТИ ВИКОНАННЯ ТЕСТУВАННЯ СИСТЕМИ

Рисунок Б.1 – Головний екран застосунку

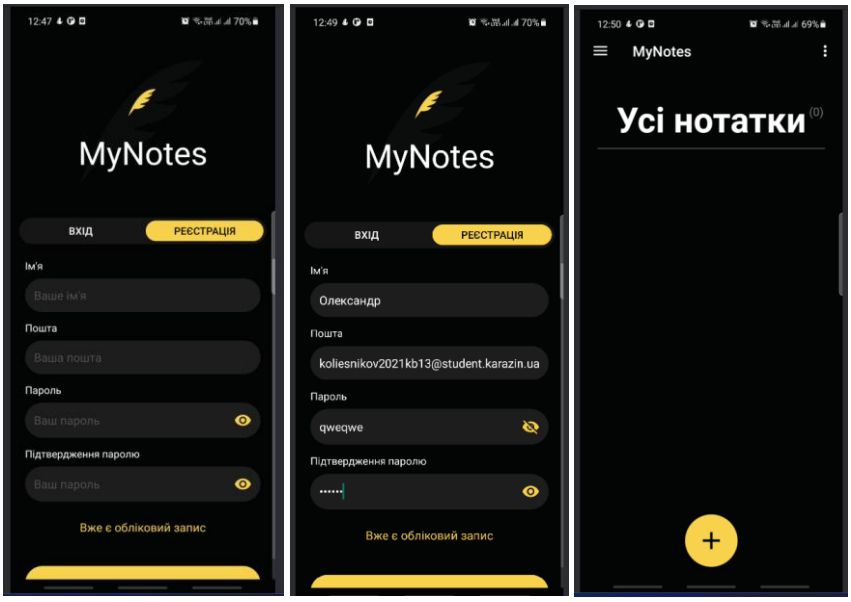

Рисунок Б.2 – Форма введення даних для реєстрації та екран основного функціоналу застосунку після успішної реєстрації

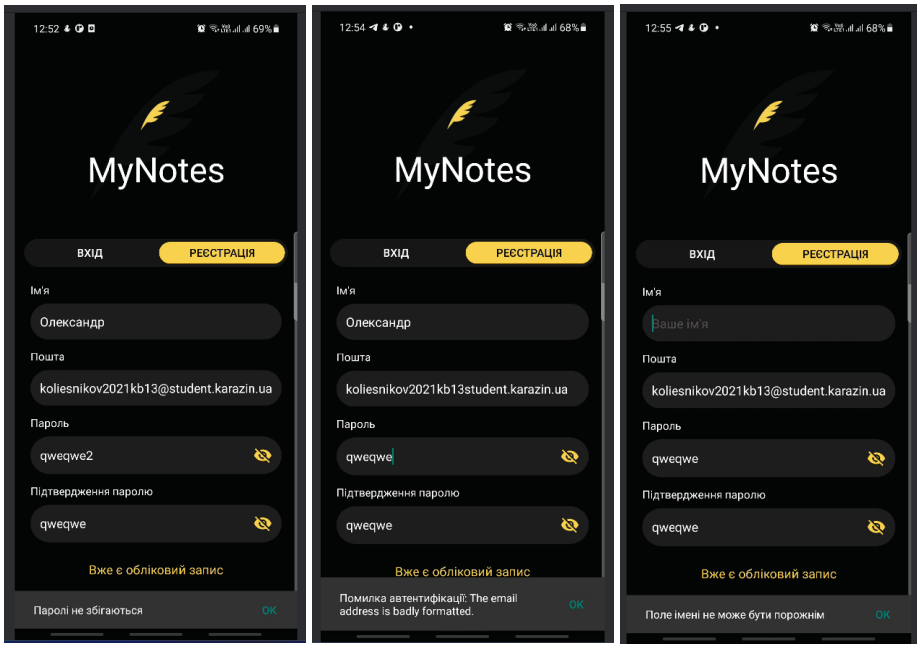

Рисунок Б.3 – Тестування введення помилкових даних під час реєстрації

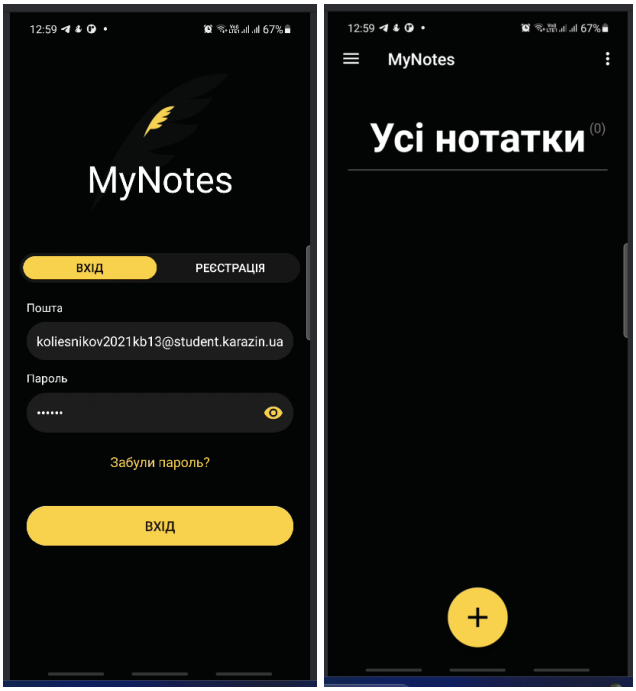

Рисунок Б.4 – Форма входу та екран основного функціоналу застосунку після успішного входу

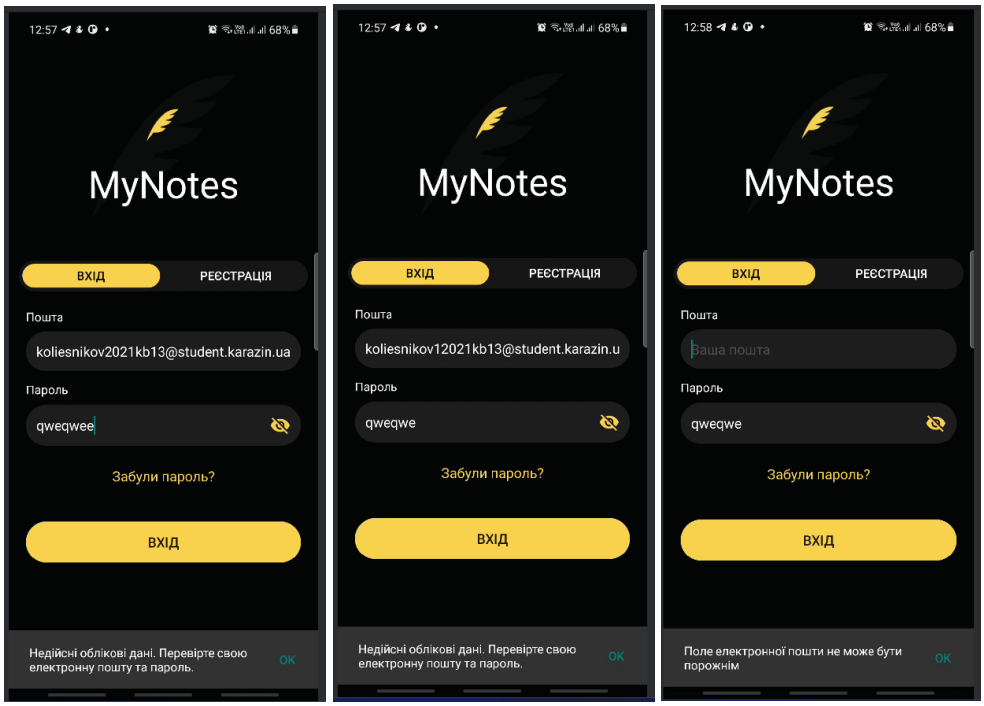

Рисунок Б.5 – Тестування введення помилкових даних під час авторизації

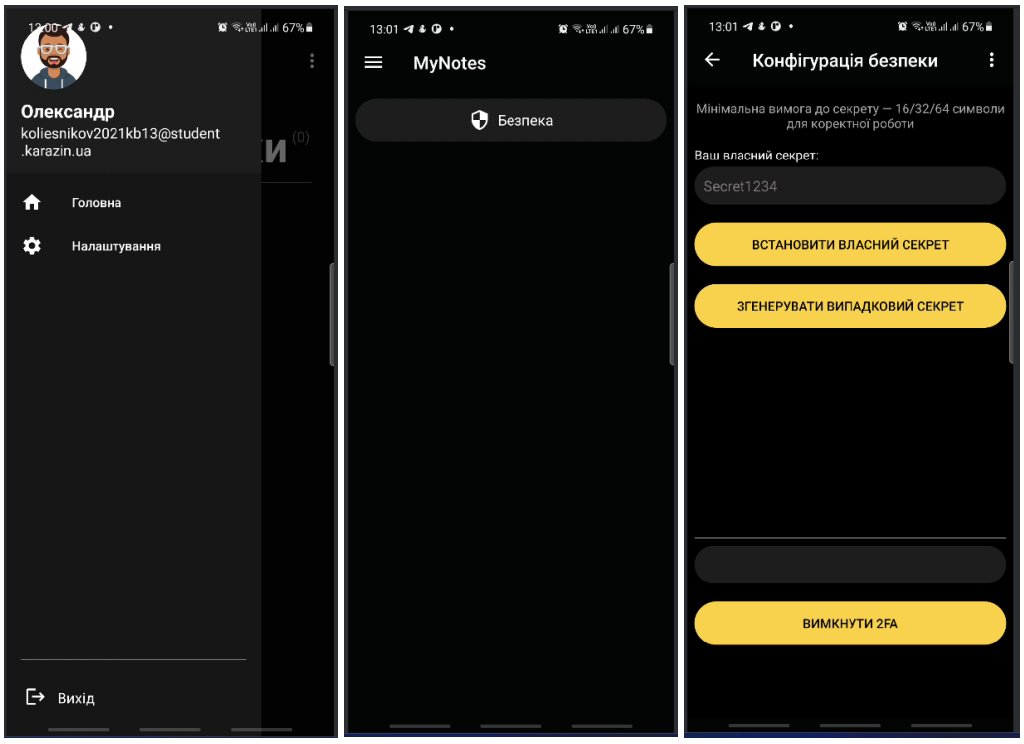

Рисунок Б.6 – Процес переходу до фрагменту із налаштуваннями 2FA

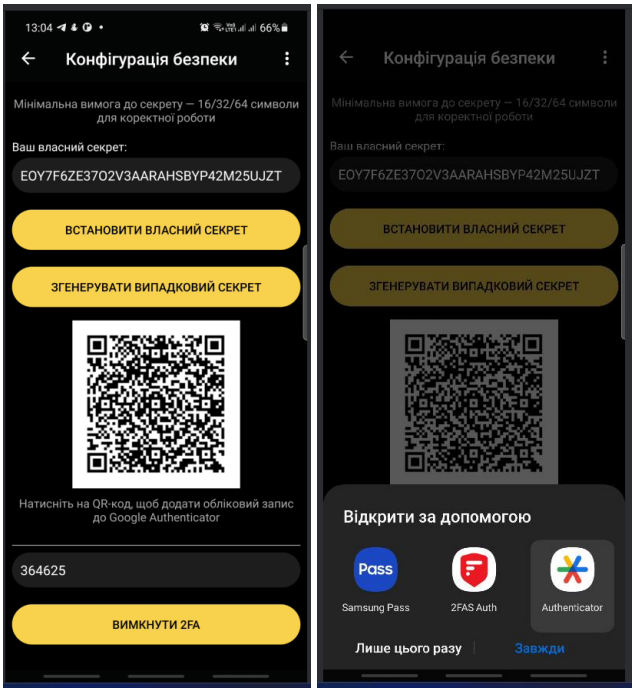

Рисунок Б.7 – Генерація нового секрету та перехід до Google Authenticator

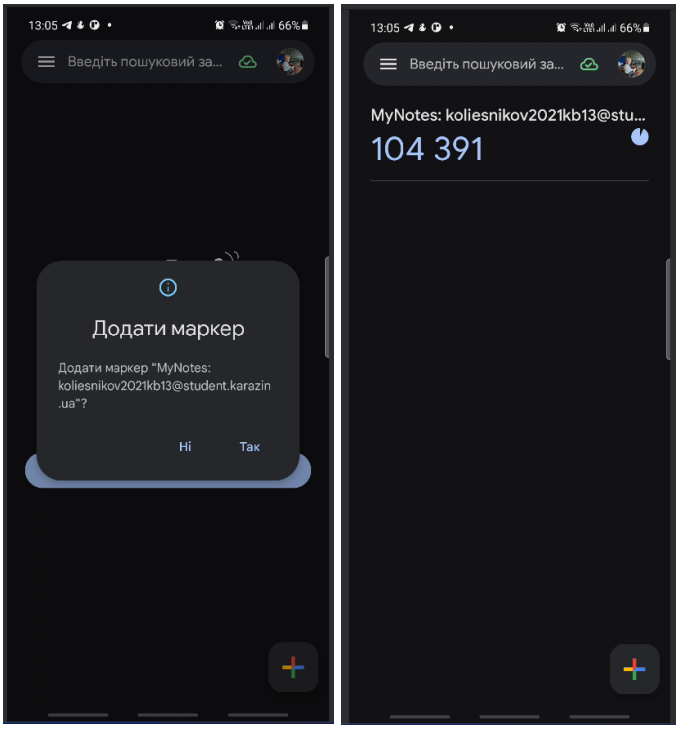

Рисунок Б.8 – Додавання синхронізованого токену до Google Authenticator

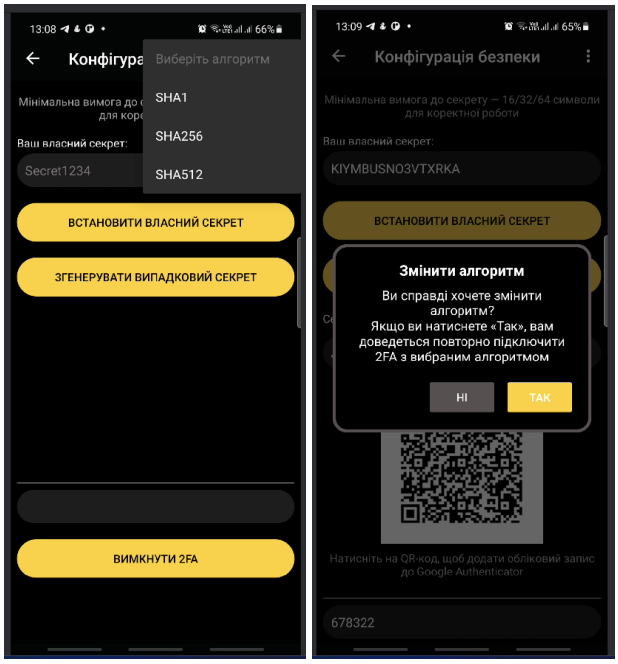

Рисунок Б.9 – Вибір потрібного алгоритму для створення секрету та заміна алгоритму, коли секрет до цього вже існував

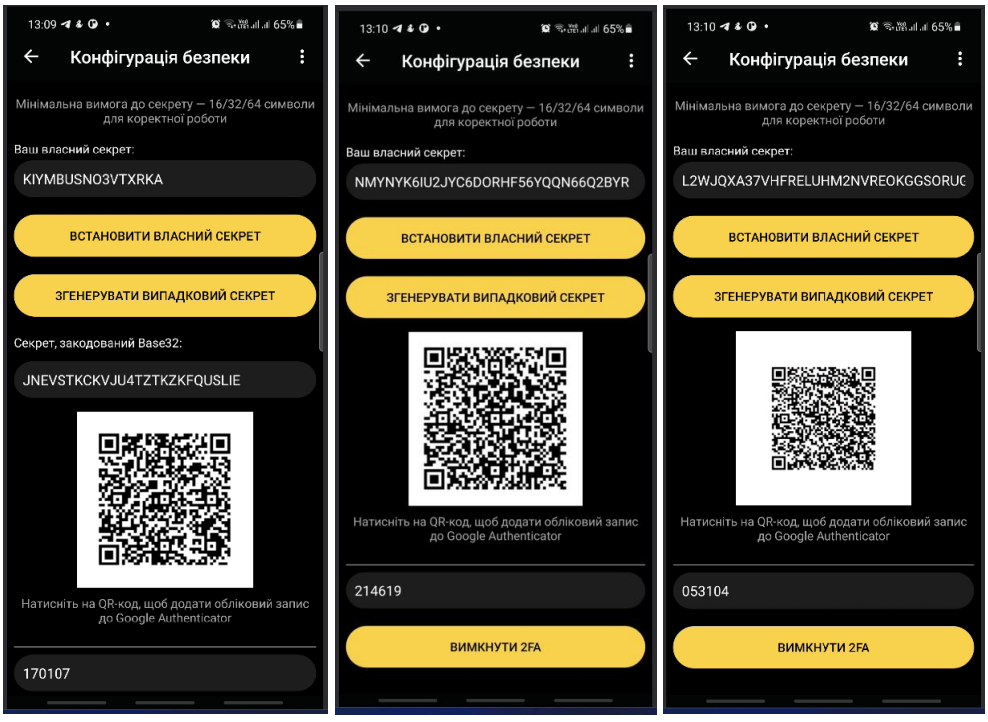

Рисунок Б.10 – Приклади згенерованих секретів з використанням алгоритмів SHA1/SHA256/SHA512 послідовно

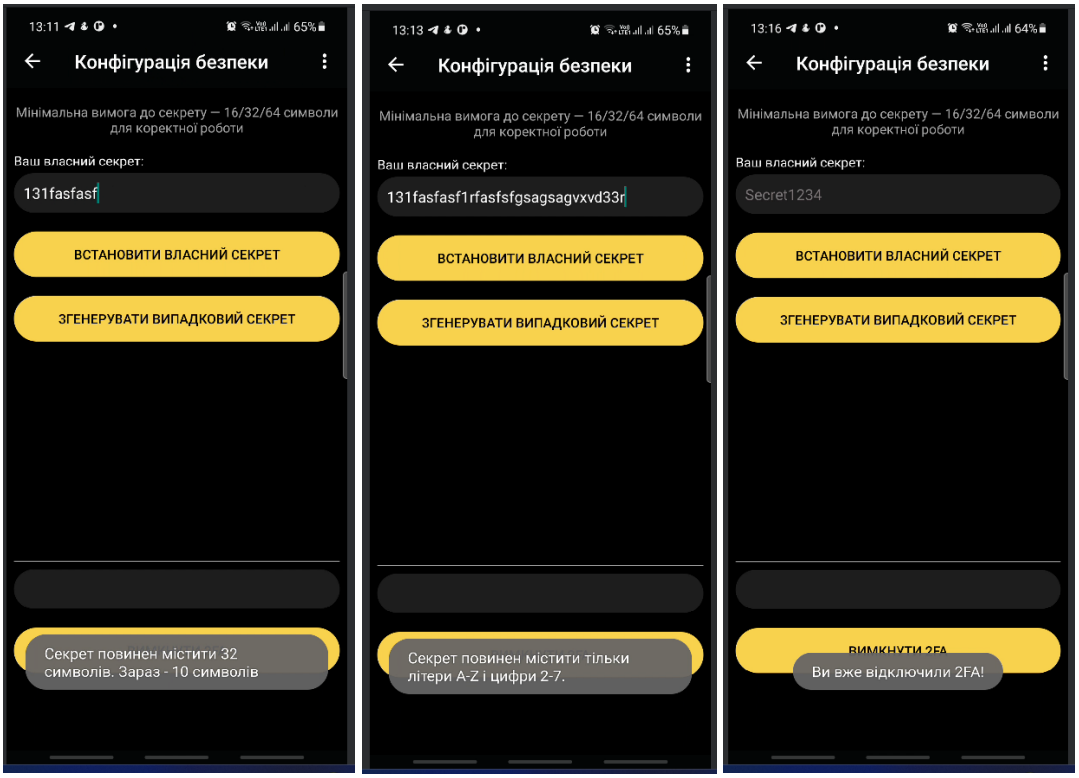

Рисунок Б.11 – Тестування помилкового вводу секрету із використанням ручної установки. Результат вимкнення 2FA

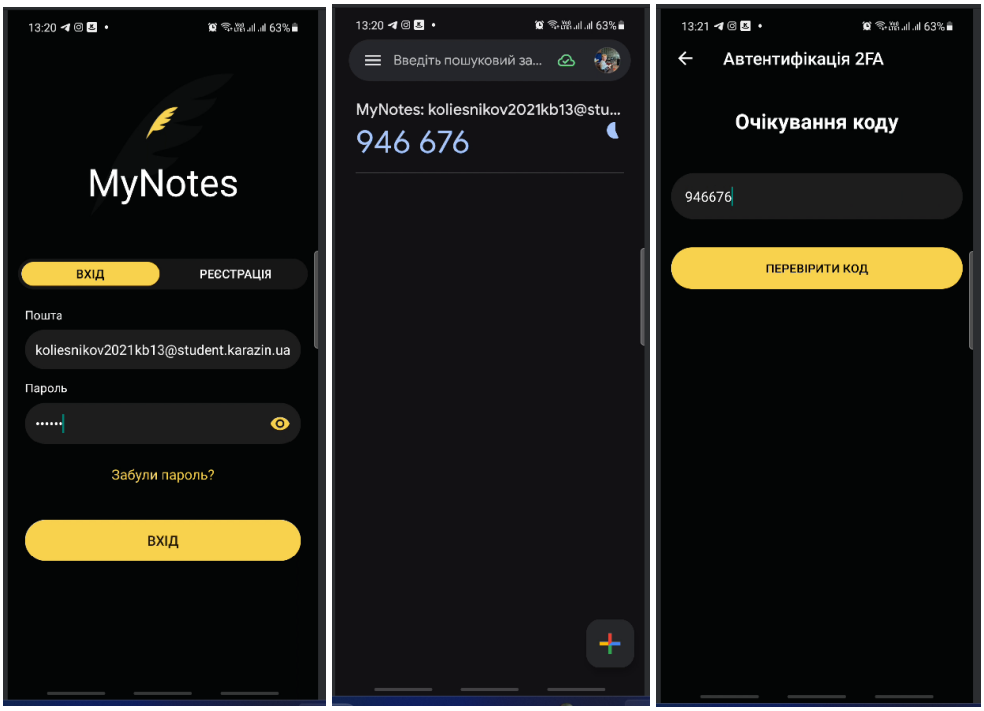

Рисунок Б.12 – Авторизація із підключеною двофакторною автентифікацією

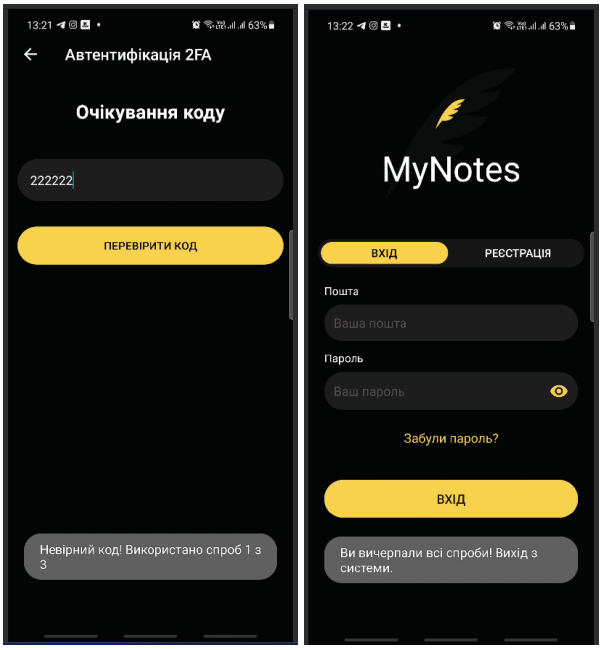

Рисунок Б.13 – Тестування помилкового вводу одноразового паролю, та вичерпання усіх доступних спроб

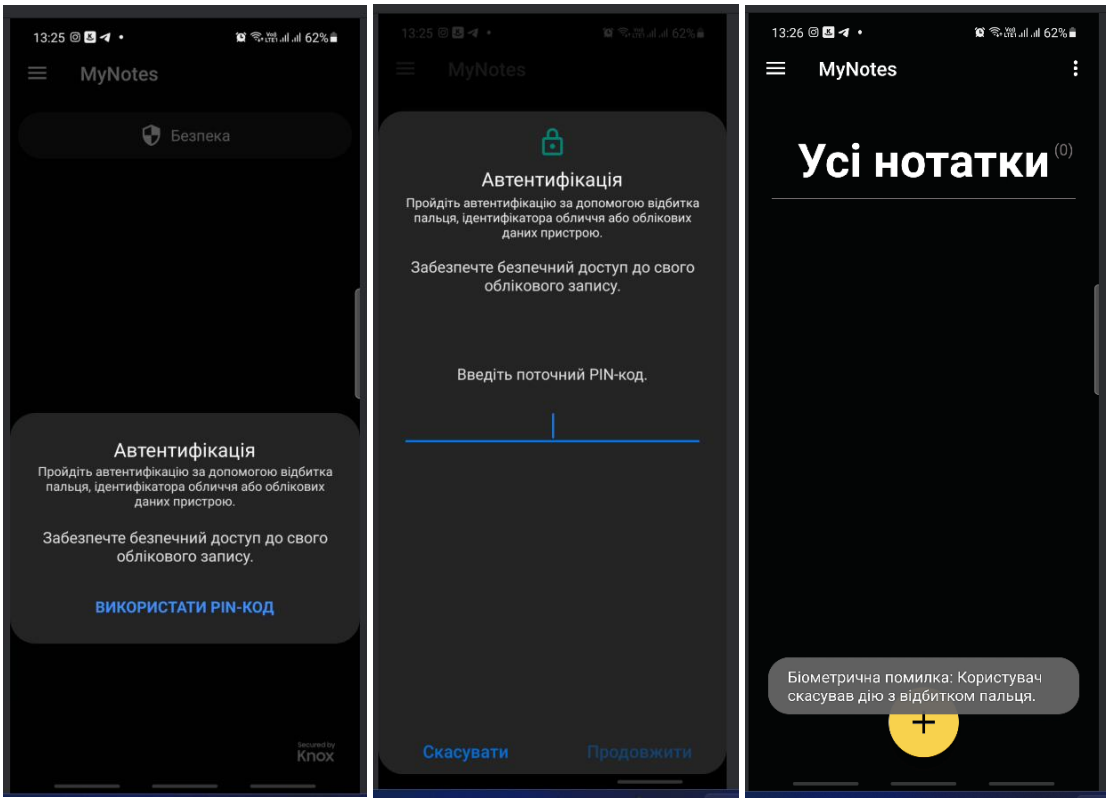

Рисунок Б.14 – Тестування біометричної автентифікації, при успішному вводі паролю – успішна, при скасуванні – помилкова

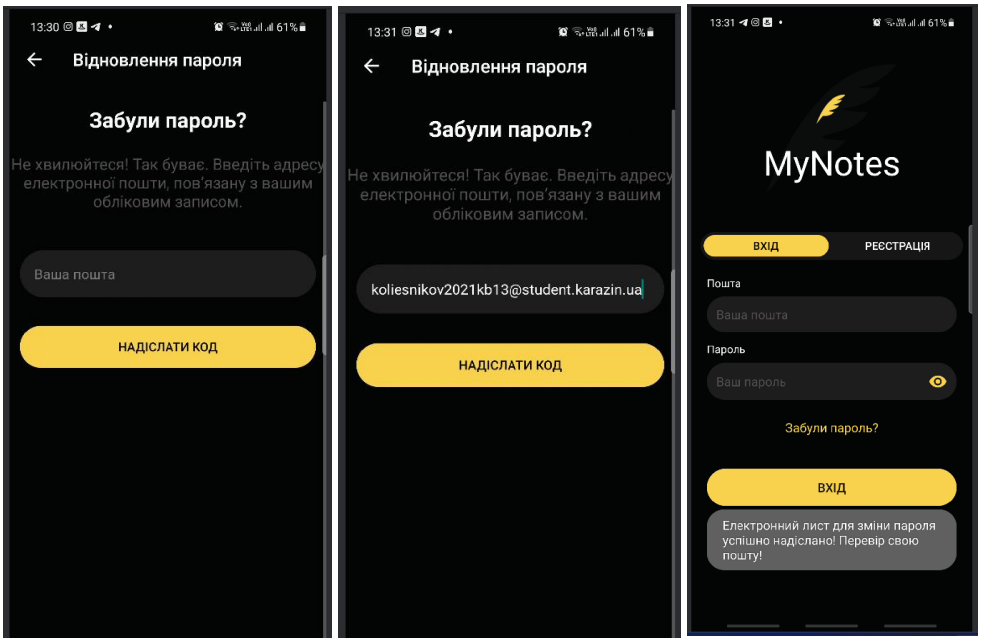

Рисунок Б.15 – Процес відновлення пароля в застосунку у разі його втрати або забуття

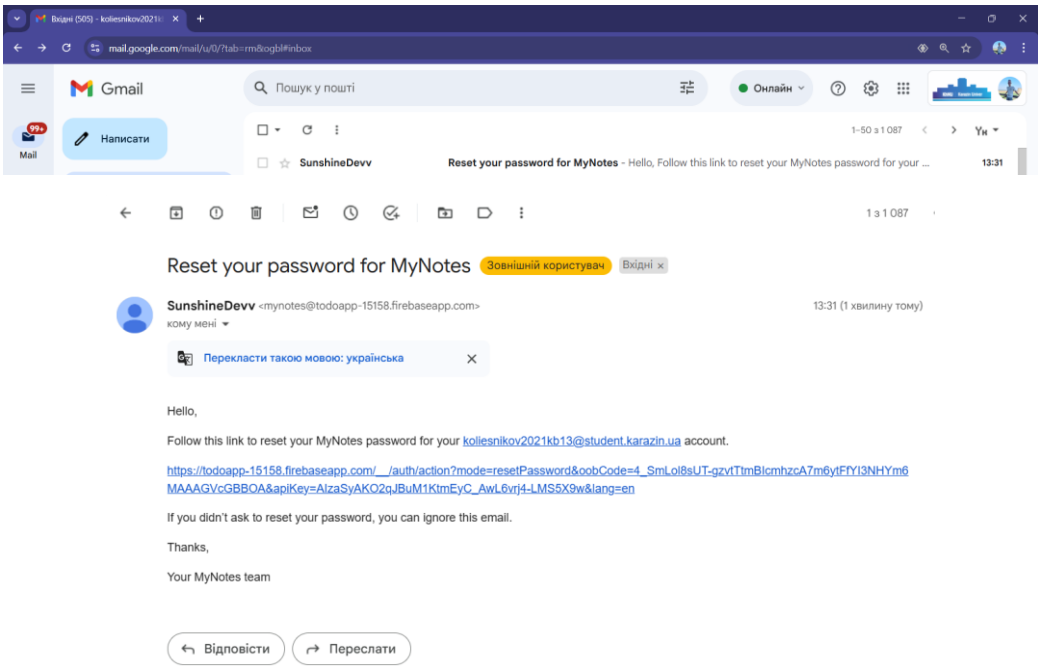

Рисунок Б.16 – Повідомлення на пошті користувача під час відновлення пароля

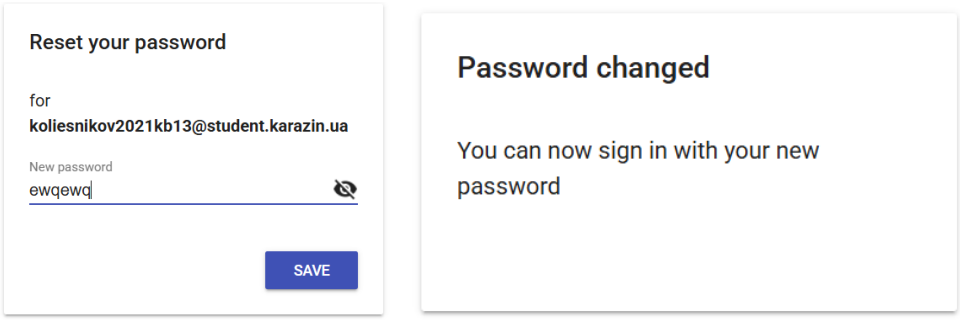

Рисунок Б.17 – Успішне відновлення пароля користувача

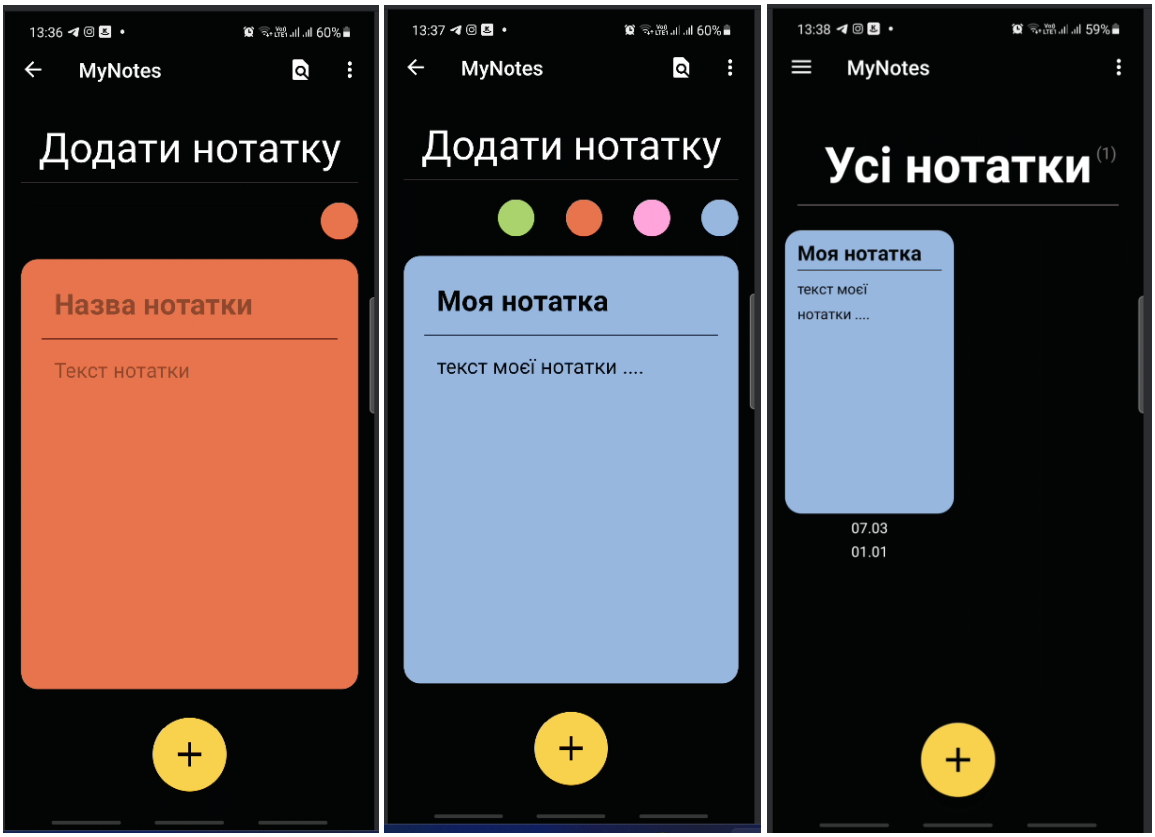

Рисунок Б.18 – Процес додавання нової нотатки

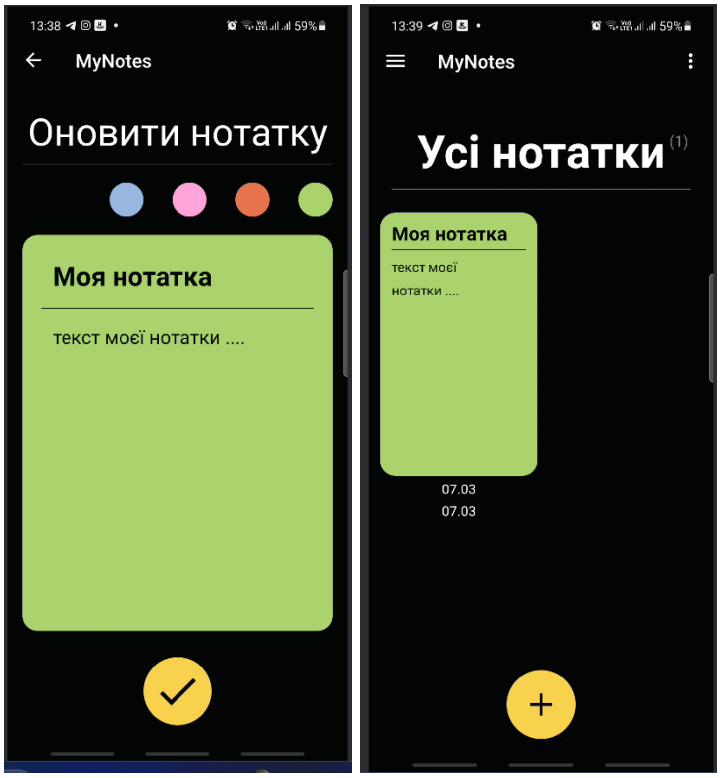

Рисунок Б.19 – Процес редагування існуючої нотатки

ДОДАТОК В

СКРІНШОТИ ТЕСТУВАННЯ БЕЗПЕКИ ЗАСТОСУНКУ

Рисунок В.1 – Загальний результат статичного тестування .apk файлу застосунку за допомогою MobSF

Рисунок В.2 – Вразливість через відкриту Activity GenericIdpActivity

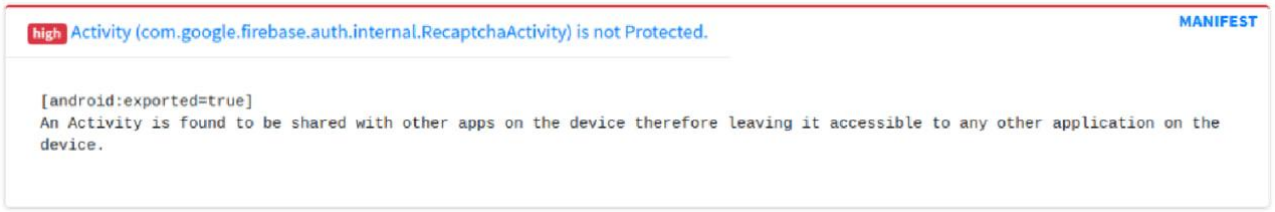

Рисунок В.3 – Вразливість відкритої RecaptchaActivity

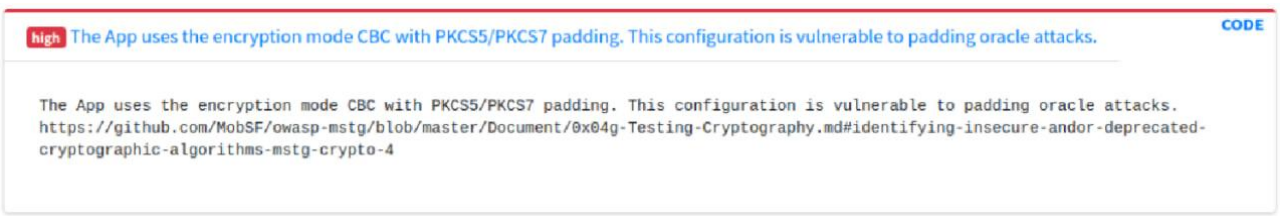

Рисунок В.4 – Вразливість CBC з PKCS5/PKCS7 Padding

Рисунок В.5 – Вразливість через невизначений рівень дозволу у службі

Рисунок В.6 – Вразливість ненадійного рівню дозволу у Broadcast Receiver

Рисунок В.7 – Вразливість використання сирих SQL-запитів

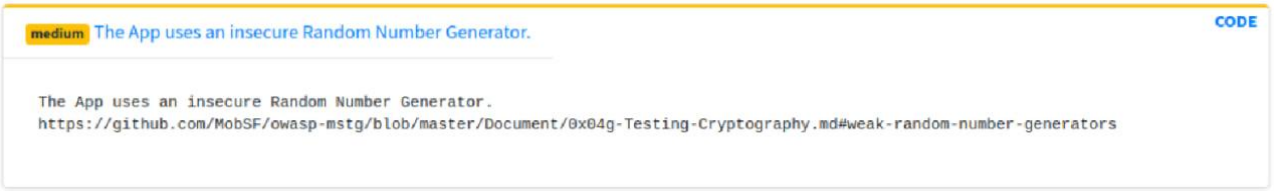

Рисунок В.8 – Вразливість ненадійного генератору випадкових чисел

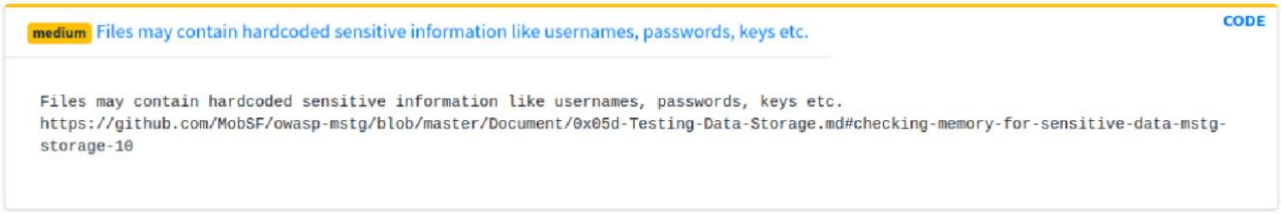

Рисунок В.9 – Вразливість жорстко закодованих чутливих даних у файлах

Рисунок В.10 – Вразливість наявності трекерів конфіденційності

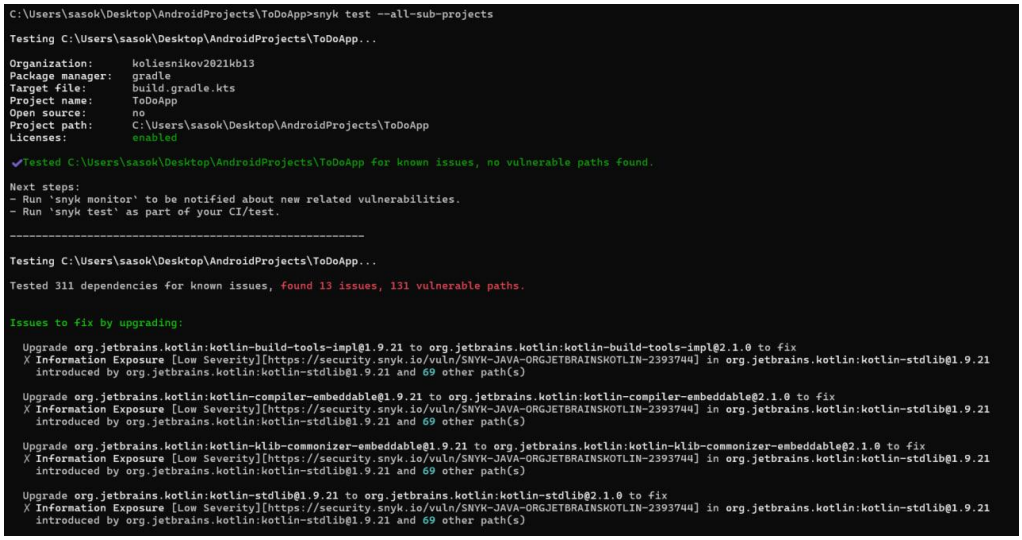

Рисунок В.11 – Поточні результати тестування Snyk

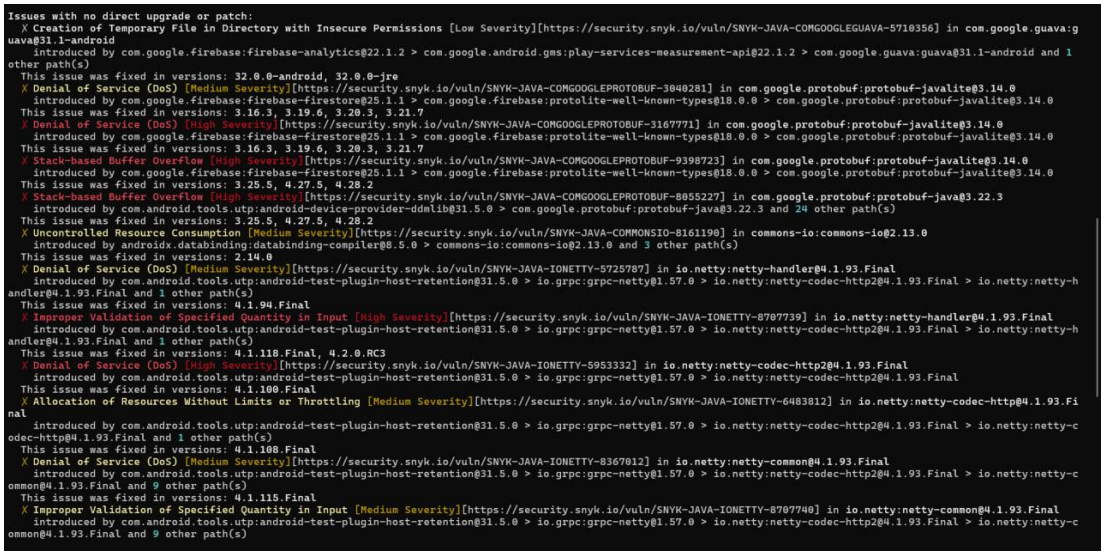

Рисунок В.12 – Перелік вразливих залежностей

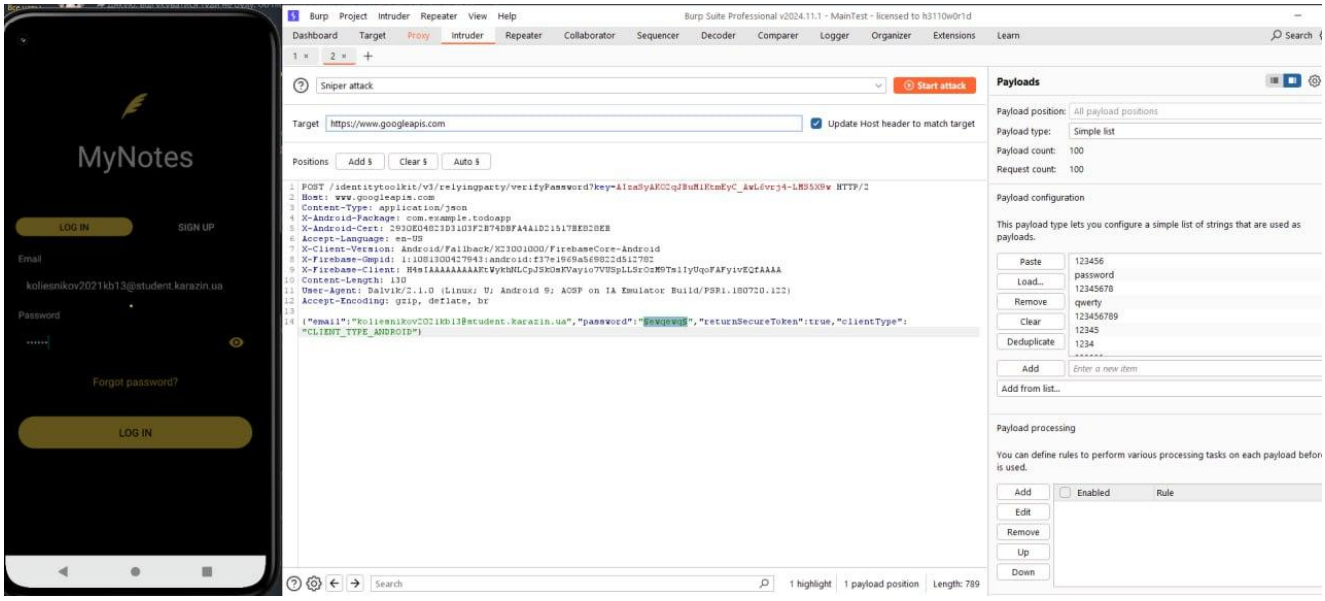

Рисунок В.13 – Налаштування передвиконання Brute Force атаки

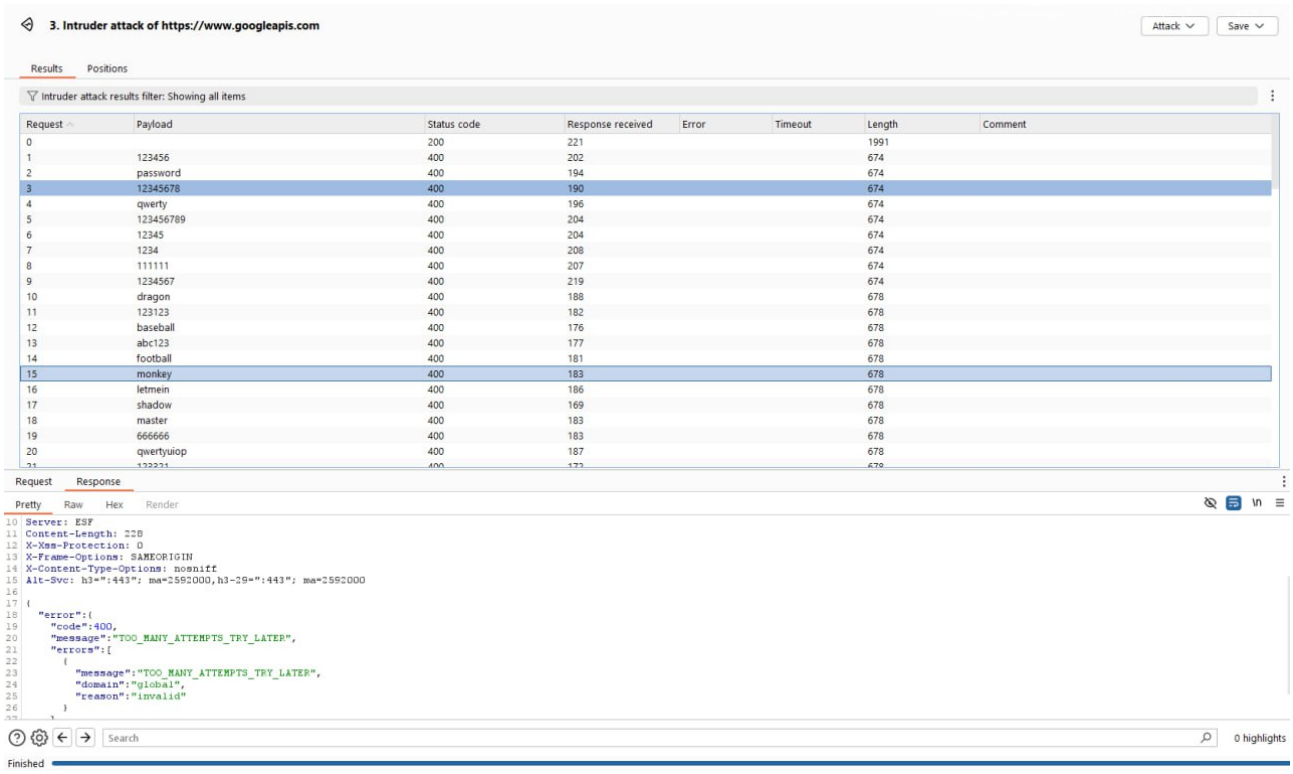

Рисунок В.14 – Результат виконання атаки Brute Force

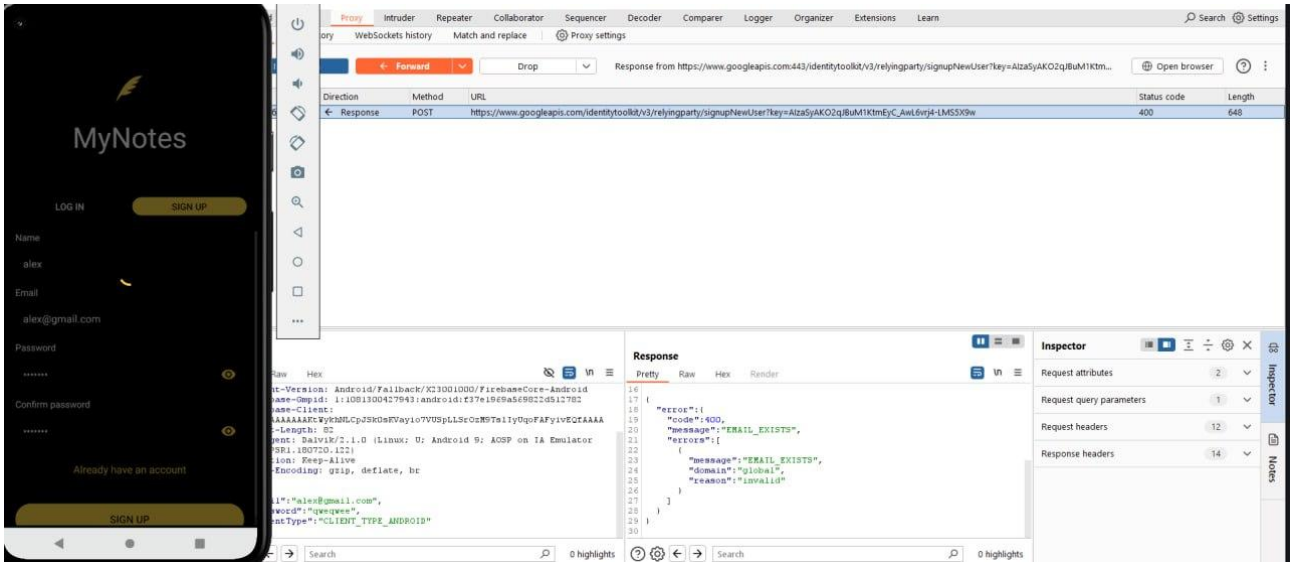

Рисунок В.15 – Спроба повторної реєстрації існуючого користувача

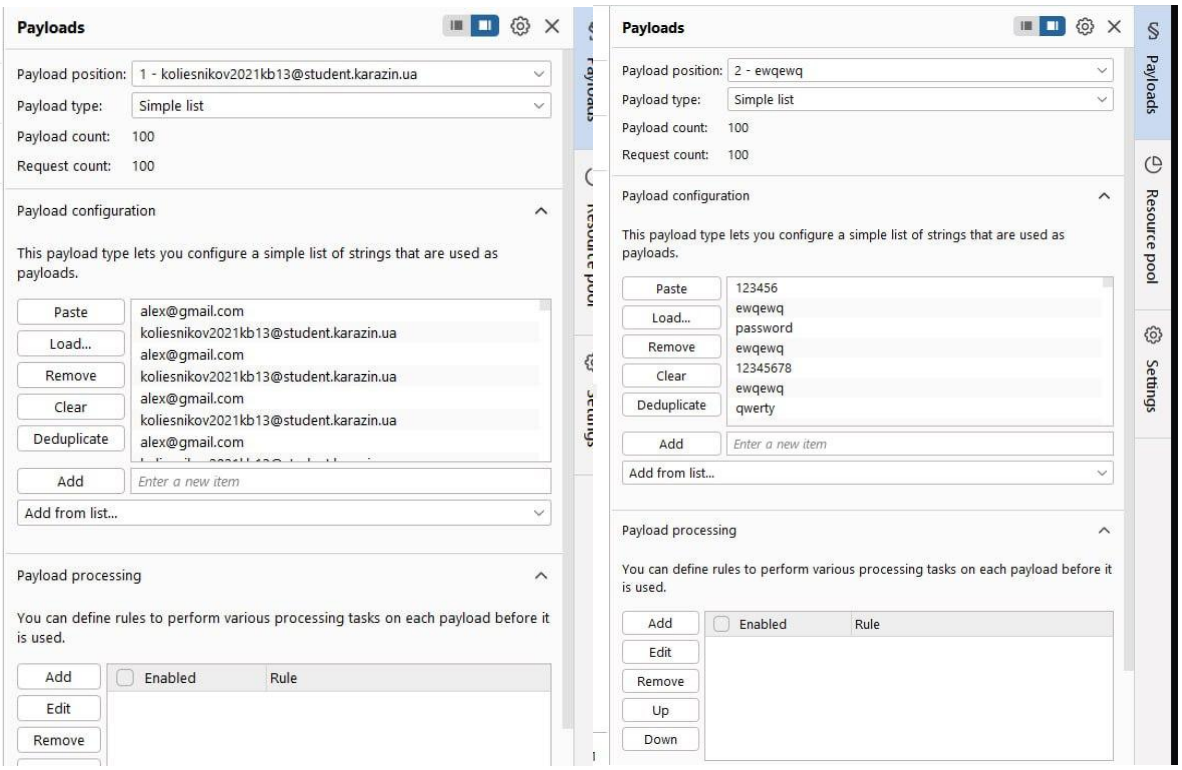

Рисунок В.16 – Налаштування другої вже ускладненої Brute Force атаки

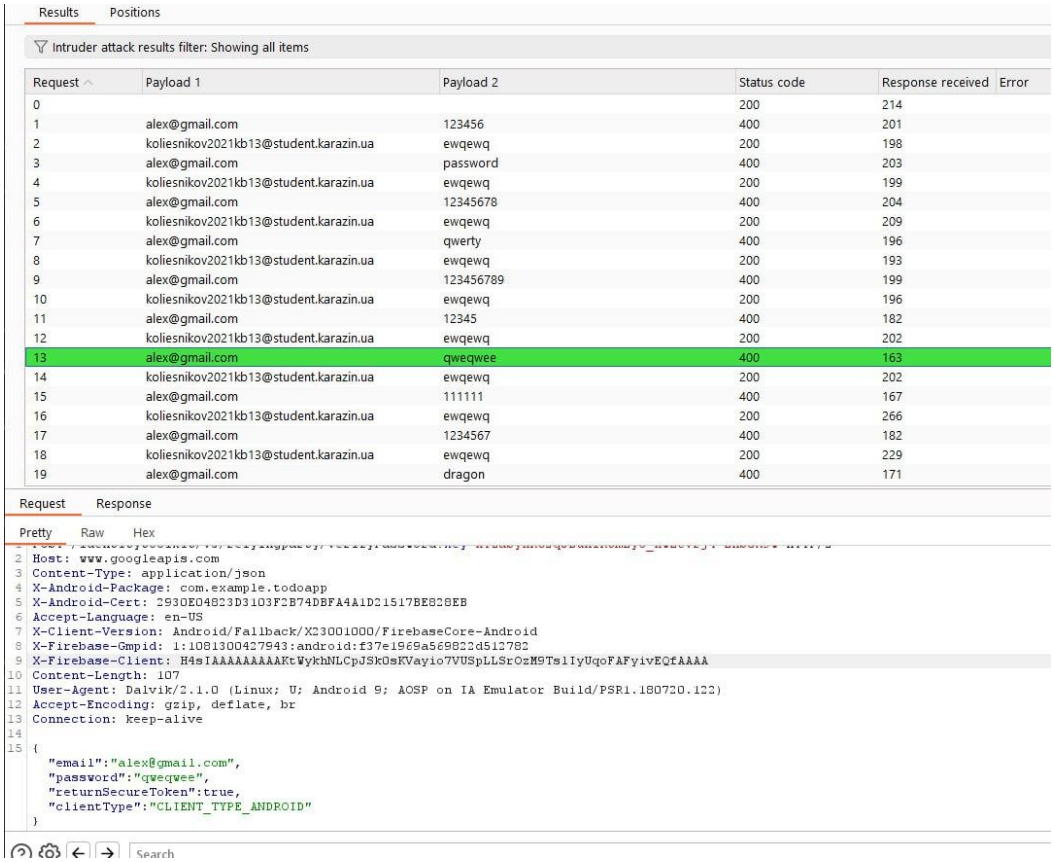

Рисунок В.17 – Приклад запиту у процесі атаки

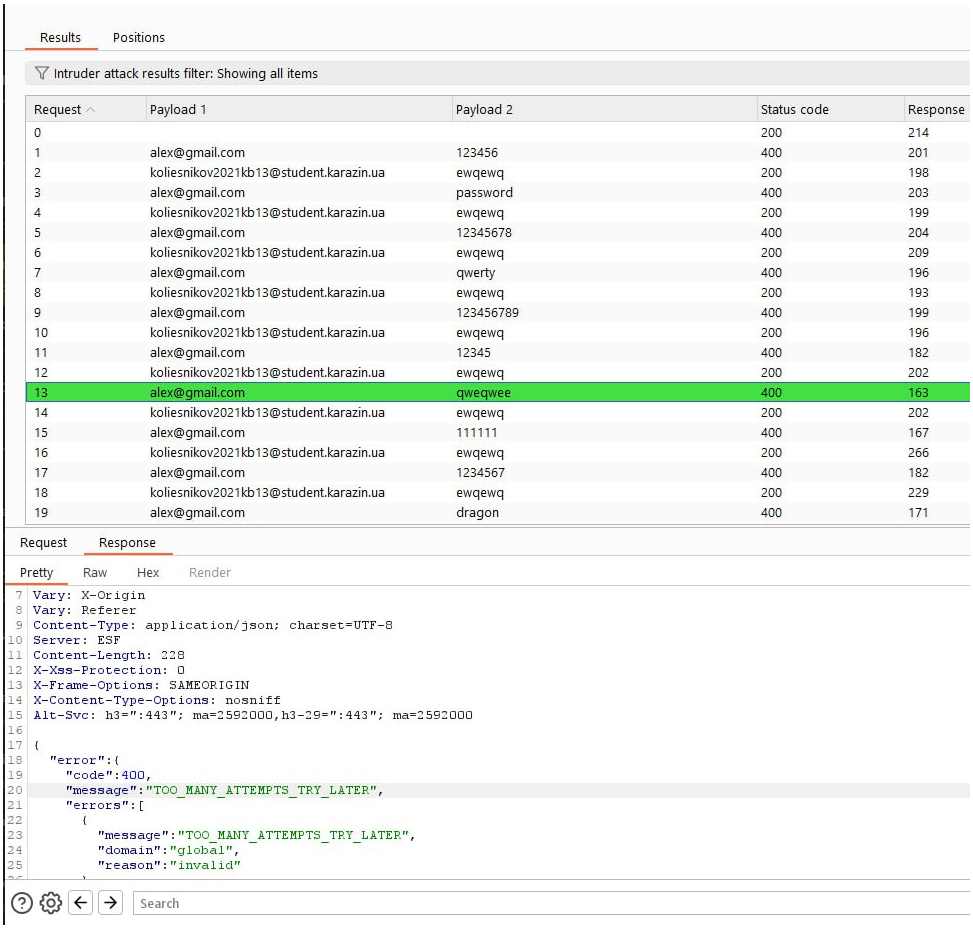

Рисунок В.18 – Відповідь на попередній запит у процесі атаки

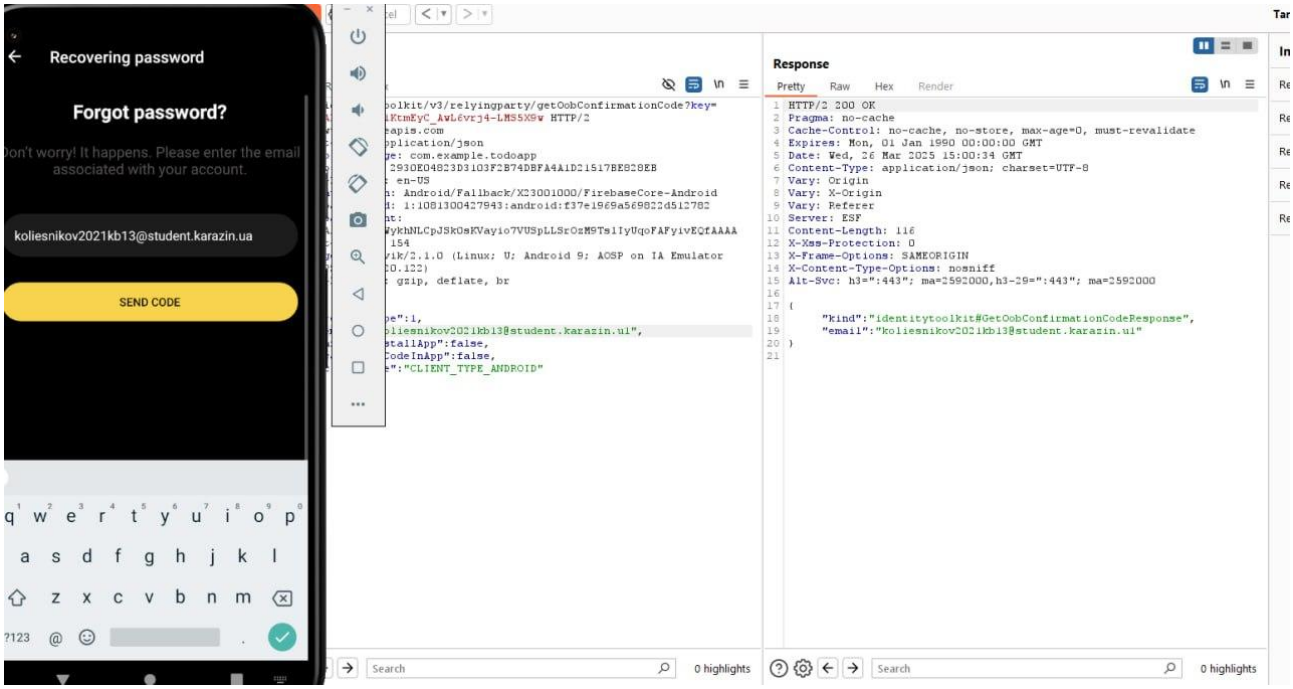

Рисунок В.19 – Результати перевірки на user enumeration із вигаданою поштою

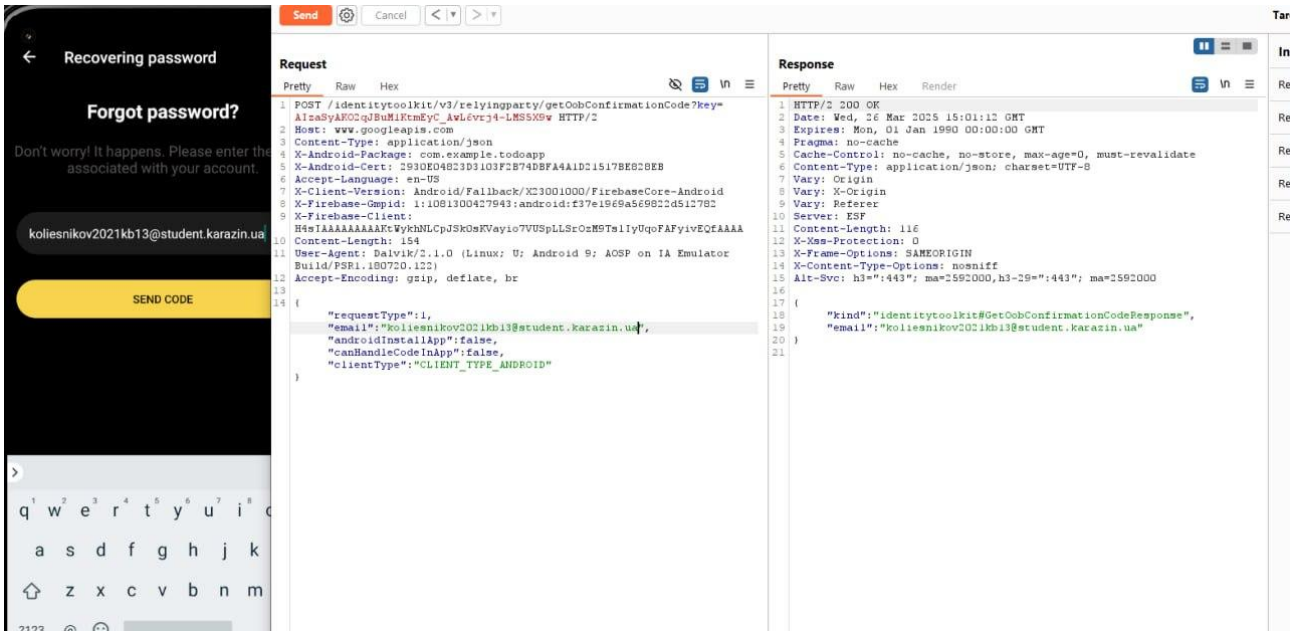

Рисунок В.20 – Результати перевірки на user enumeration із валідною поштою

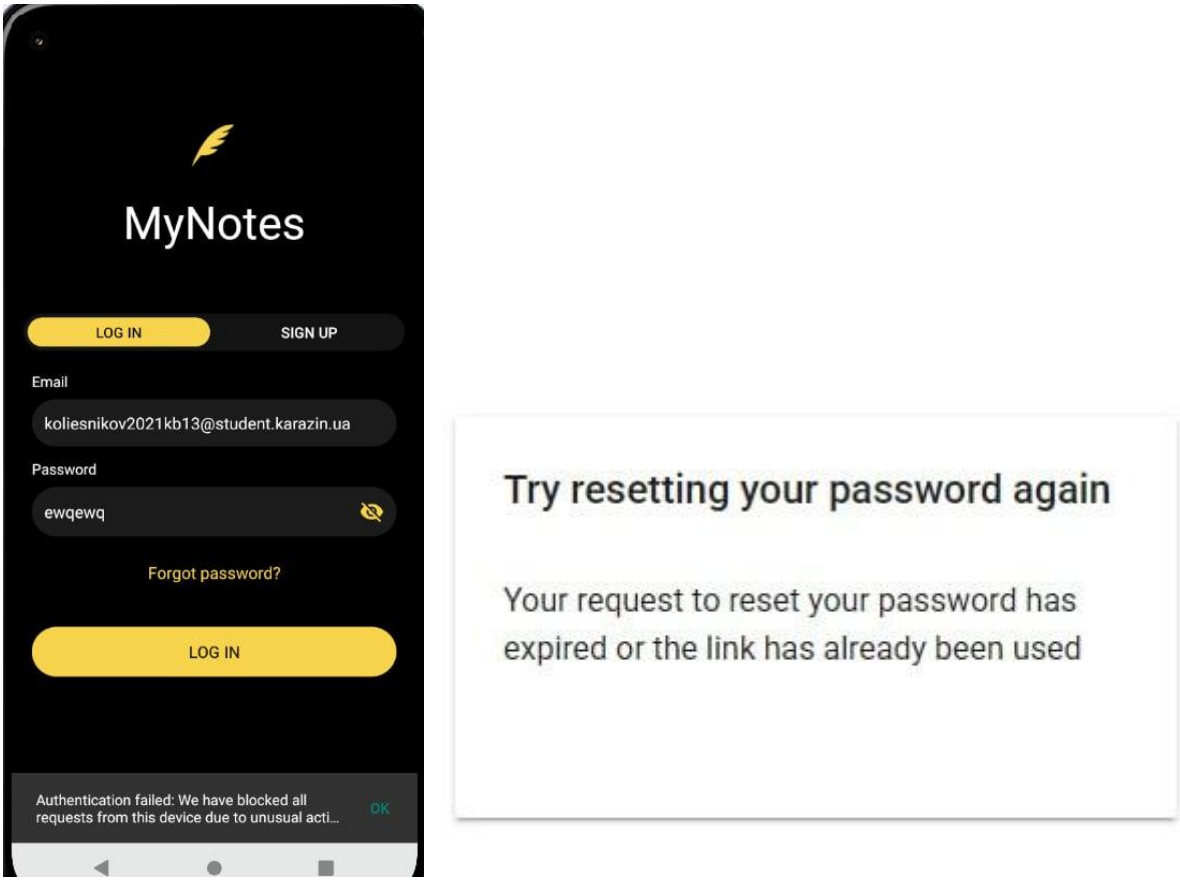

Рисунок В.21 – Блокування виконання процесів автентифікації через підозрілу активність

ДОДАТОК Г

СКРІНШОТИ ГРАФІЧНИХ СХЕМ ВЗАЄМОДІЇ

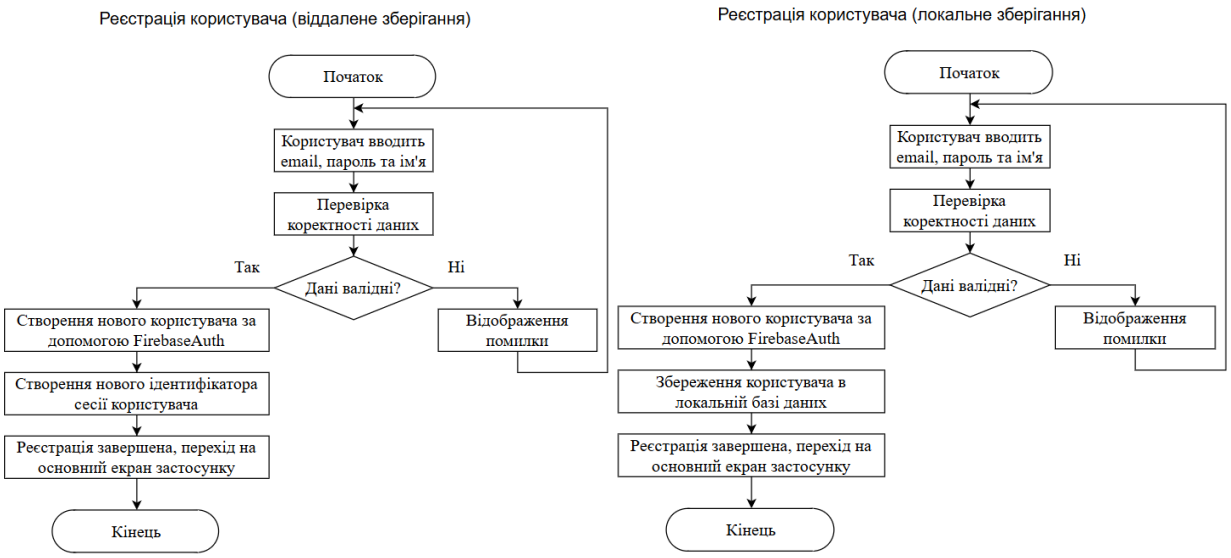

Рисунок Г.1 – Схеми процесу реєстрації нового користувача в різних версіях застосунку

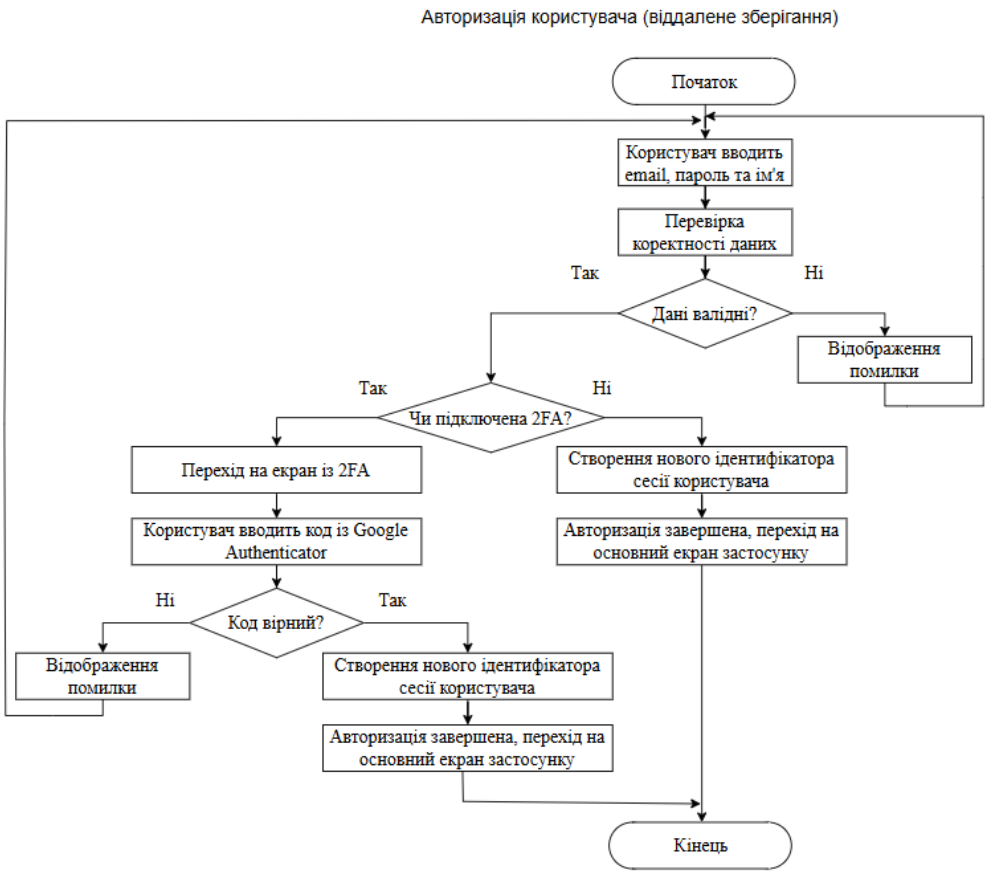

Рисунок Г.2 – Схема процесу авторизації користувача у версії з віддаленим зберіганням даних

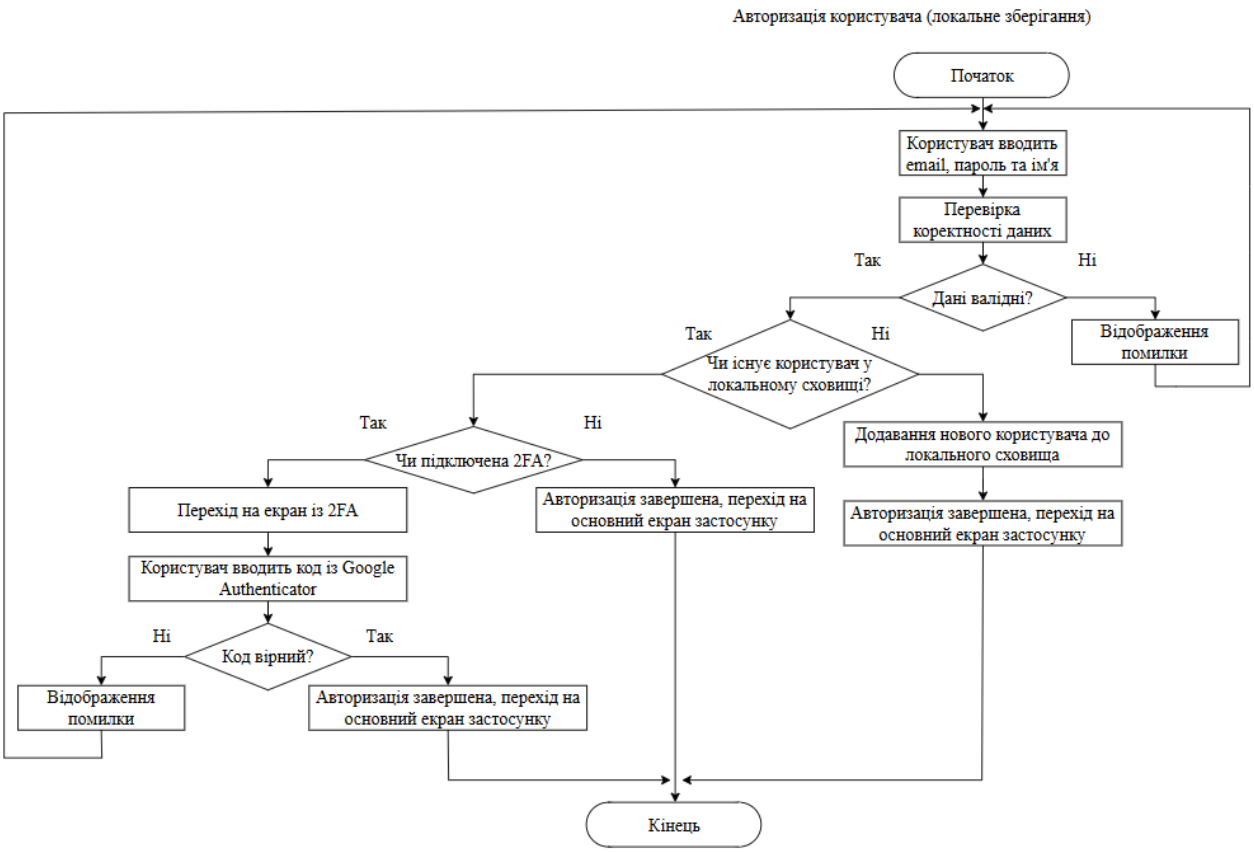

Рисунок Г.3 – Схема процесу авторизації користувача у версії з локальним зберіганням даних