НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

«До захисту допущено»
В.о. завідувача кафедри

_____ М.В.Грайворонський
(підпис)
“___” _____2019 р.

Дипломна робота

на здобуття ступеня бакалавра

з напряму підготовки 6.170101 «Безпека інформаційних і комунікаційних систем»

на тему: Побудова та дослідження моделі безпечного способу авторизації на основі протоколу OAuth 2.0 для веб-застосунків, що використовують API

Виконав (-ла): студент (-ка) ____ курсу, групи _____
(шифр групи)

Духонченко Володимир Станіславович
(прізвище, ім’я, по батькові) (підпис)

Керівник: к.ф.-м.н. Орєхов О. А.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант _____ _____ _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Рецензент_____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ - 2019 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

Рівень вищої освіти – перший (бакалаврський)

Напрям підготовки 6.170101 «Безпека інформаційних і комунікаційних систем»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ М.В.Грайворонський
(підпис)
«___»_____2019 р.

ЗАВДАННЯ
на дипломну роботу студенту

_____Духонченку Володимиру Станіславовичу_____
(прізвище, ім'я, по батькові)

1. Тема роботи Побудова та дослідження моделі безпечного способу авторизації на основі протоколу OAuth 2.0 для веб-застосунків, що використовують API,
науковий керівник роботи доцент кафедри ІБ, к.ф.-м.н. Орєхов О.А
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «____ » 2019 р. № _____

2. Термін подання студентом роботи 10 червня 2019 р.

3. Вихідні дані до роботи _____
_____
_____

4. Зміст роботи_____
_____
_____
_____
_____

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо) _____
_____
_____
_____

6. Дата видачі завдання _____

РЕФЕРАТ

Представлена дипломна робота складається з трьох розділів, загальний обсяг роботи – 84 сторінки. Містить 13 літературних посилань, 11 ілюстрацій, 2 таблиці.

Основною метою роботи є підвищення захищеності способу авторизації веб-застосунків, які використовують протоколи авторизації. Для досягнення мети потрібно провести дослідження та аналіз існуючих протоколів авторизації, обрати найбільш безпечний для реалізації, виявити та дослідити існуючи атаки на обраний протокол авторизації – OAuth 2.0, та базуючись на складеному переліку атак побудувати модель безпечного способу авторизації на основі даного протоколу.

Об’єкт дослідження даної роботи – протоколи авторизації для веб-застосунків, що використовують API.

Предметом дослідження виступають атаки на протокол авторизації OAuth 2.0 та можливі методи захисту від них.

Під час написання роботи були проведені аналіз, дослідження та узагальнення технічної і наукової літератури, запропоновано реалізацію моделі безпечного способу авторизації на основі протоколу OAuth 2.0 для веб-застосунків.

Значення даної роботи зумовлено використанням розробниками запропонованої моделі для реалізації безпечного способу авторизації кінцевих користувачів, та побудови безпечних веб застосунків, що використовують протокол авторизації OAuth 2.0.

Ключові слова OAuth 2.0, протокол авторизації, маркери доступу, код авторизації, атака, модель, безпека, TLS, PKCE.

РЕФЕРАТ

Представленная дипломная работа состоит из трех разделов, общий объем работы – 84 страницы. Содержит 13 литературных ссылок, 11 иллюстраций, 2 таблицы.

Основной целью работы является повышение защищенности способа авторизации веб-приложений, которые используют протоколы авторизации. Для достижения цели нужно провести исследования и анализ существующих протоколов авторизации, выбрать наиболее безопасный для реализации, выявить и исследовать существующие атаки на выбранный протокол авторизации - OAuth 2.0, и основываясь на составленном перечне атак построить модель безопасного способа авторизации на основе данного протокола.

Объект исследования данной работы - протоколы авторизации для веб-приложений, использующих API.

Предметом исследования выступают атаки на протокол авторизации OAuth 2.0 и возможные методы защиты от них.

При написании работы были проведены исследование и анализ технической и научной литературы, предложено реализацию модели безопасного способа авторизации на основе протокола OAuth 2.0 для веб-приложений.

Значение данной работы обусловлено использованием разработчиками предложенной модели для реализации безопасного способа авторизации конечных пользователей, и построения безопасных веб приложений, использующих протокол авторизации OAuth 2.0.

Ключевые слова OAuth 2.0, протокол авторизации, маркеры доступа, код авторизации, атака, модель, безопасность, TLS, PKCE.

ABSTRACT

The presented thesis consists of three sections, the total volume of work is 84 pages. Contains 13 literary references, 11 illustrations, 2 tables.

The main purpose of the work is to increase security of the authorization method for web-applications that use authorization protocols. In order to achieve the goal, it is necessary to conduct research and analysis of existing authorization protocols, to select the most safe to implement, to detect and investigate existing attacks on the selected authorization protocol - OAuth 2.0, and based on a composite list of attacks to build a safe model of authorization based on this protocol.

The object of this study is authorization protocols for web applications that use the API.

The subject of the study is attacks on the authorization protocol OAuth 2.0 and possible methods of their mitigation.

During writing of this work, analysis, research and generalization of technical and scientific literature were conducted, the implementation of the safe authorization method model based on OAuth 2.0 for web applications was proposed.

The value of this work is due to the use of the proposed model by software developers to implement a secure end-user authorization method and the construction of secure web-applications that use the OAuth 2.0 authorization protocol.

OAuth 2.0 Keywords, Authorization Protocol, Access Token, Authorization Code, Attack, Model, Security, TLS, PKCE.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

OAuth – Open Authorization

HTTP - HyperText Transfer Protocol

HTTPS - Hypertext Transfer Protocol Secure

TLS - Transport Layer Security

PKCE – Proof Key for Code Exchange

HMAC - Hash-based Message Authentication Code

CSRF - Cross-Site Request Forgery

SAML - Security Assertion Markup Language

URI - Uniform Resource Identifier

URL - Uniform Resource Locator

API - Application Programming Interface

ID - Identity Document

JWT - JSON Web Token

ВСТУП

Зростаюча кількість інформації, що зберігається на різноманітних веб сервісах та «в хмарі» збільшується із кожним роком. Це сприяє збільшенню кількості програмного забезпечення, для користування яким, користувачам потрібна авторизація. Таким чином, зростає і кількість комбінацій логіна та пароля, які користувачам потрібно запам’ятовувати, для кожного з таких ресурсів.

Це представляє небезпеку з боку безпеки, так як подібна ситуація призводить до використання на всіх ресурсах однієї пари логіна та пароля, або найпростіші комбінації, такі як «login:password». Для уникнення таких ситуацій, можливе використання менеджерів паролів, або протоколів авторизації. Останні дають користувачам можливість надавати сайтам та застосункам маркери доступу, що дають доступ до матеріалів які розміщуються на потрібних сайтах-сервісах.

Існує багато протоколів авторизації та автентифікації користувачів, задачею цієї роботи буде аналіз та порівняння цих протоколів, вибір найбільш безпечного для використання в межах цілей роботи. Мають бути розглянуті можливі атаки, та дані рекомендації, що допоможуть розробникам зменшити вплив загроз, та збитки при атаках.

Актуальність роботи полягає розробці моделі авторизації на основі одного з найбільш популярних протоколів авторизації – OAuth 2.0.

Метою даної роботи є аналіз атак на протокол авторизації OAuth 2.0 для побудови та розробки моделі авторизації на основі протоколу OAuth 2.0.

Для досягнення цієї мети були поставлені такі задачі:

- проаналізувати існуючі протоколи авторизації

- проаналізувати можливі атаки на протокол авторизації OAuth 2.0, що можуть компрометувати безпеку секретної інформації кінцевих користувачів
- побудувати модель безпечного способу авторизації для протоколу OAuth 2.0

Методами дослідження обрано: опрацювання та аналіз технічної документації за даною темою, а також опрацювання наукової літератури.

Наукова новизна роботи зумовлюється у тому, що побудова моделі базується на дослідженні відомих атак на протокол OAuth 2.0.

Практична значимість роботи зумовлена можливістю створення розробниками безпечного способу авторизації кінцевих користувачів веб-застосунків.

Таким чином, **об'єкт дослідження** даної роботи – протоколи авторизації для веб-застосунків.

Предметом дослідження є атаки на протокол авторизації OAuth 2.0, можливі методи захисту від них.

1 АНАЛІЗ ПРОТОКОЛІВ АВТОРИЗАЦІЇ

1.1 Введення

Зазвичай, автентифікація проводиться при використанні клієнт-серверної моделі на стороні сервера, при використанні якої клієнт робить запит до захищеного ресурсу на сервері, використовуючи облікові дані власника ресурсу для автентифікації. У цій моделі, єдиним способом для власника ресурсу надати доступ до захищених ресурсів стороннім застосункам, є обмін обліковими даними власника ресурсу з третьою стороною.

1.1.1 Недоліки клієнт-серверної моделі

Підхід клієнт-серверної моделі має ряд проблем і обмежень:

- Третя сторона має зберігати облікові дані для майбутнього використання, часто трапляється так, що вони зберігаються у формі відкритого тексту.
- Сервери мають підтримувати автентифікацію за паролем, хоча пароль за своєю природою, може бути небезпечним, наприклад паролі «password» чи «qwerty».
- Власник ресурсу не може обмежувати тривалість або обсяг доступу, який надає програмам третіх сторін надто широкий доступ до захищених ресурсів власника ресурсів.
- Єдиним способом скасувати доступ третьої сторони до ресурсу – зміна паролю, що також скасує доступ для всіх інших застосунків третьої сторони.
- Компрометація зі сторони будь-якого застосунку, може скомпрометувати пароль користувача, а отже і всі дані, що були захищені паролем.

Для рішення цих проблем потрібна реалізація додаткових засобів авторизації. Одними із загальноприйнятих стандартів є SAML, OAuth 2.0 та OpenID Connect.

1.2 Обрані стандарти авторизації

Для аналізу в роботі були обрані найбільш актуальні, поширені та підтримувані стандарти авторизації.

Таким чином, для подальшого аналізу, згідно цих критеріїв, були обрані згадані вище стандарти авторизації:

- SAML 2.0;
- OAuth 2.0;
- OpenID Connect 1.0.

1.2.1 OAuth 2.0

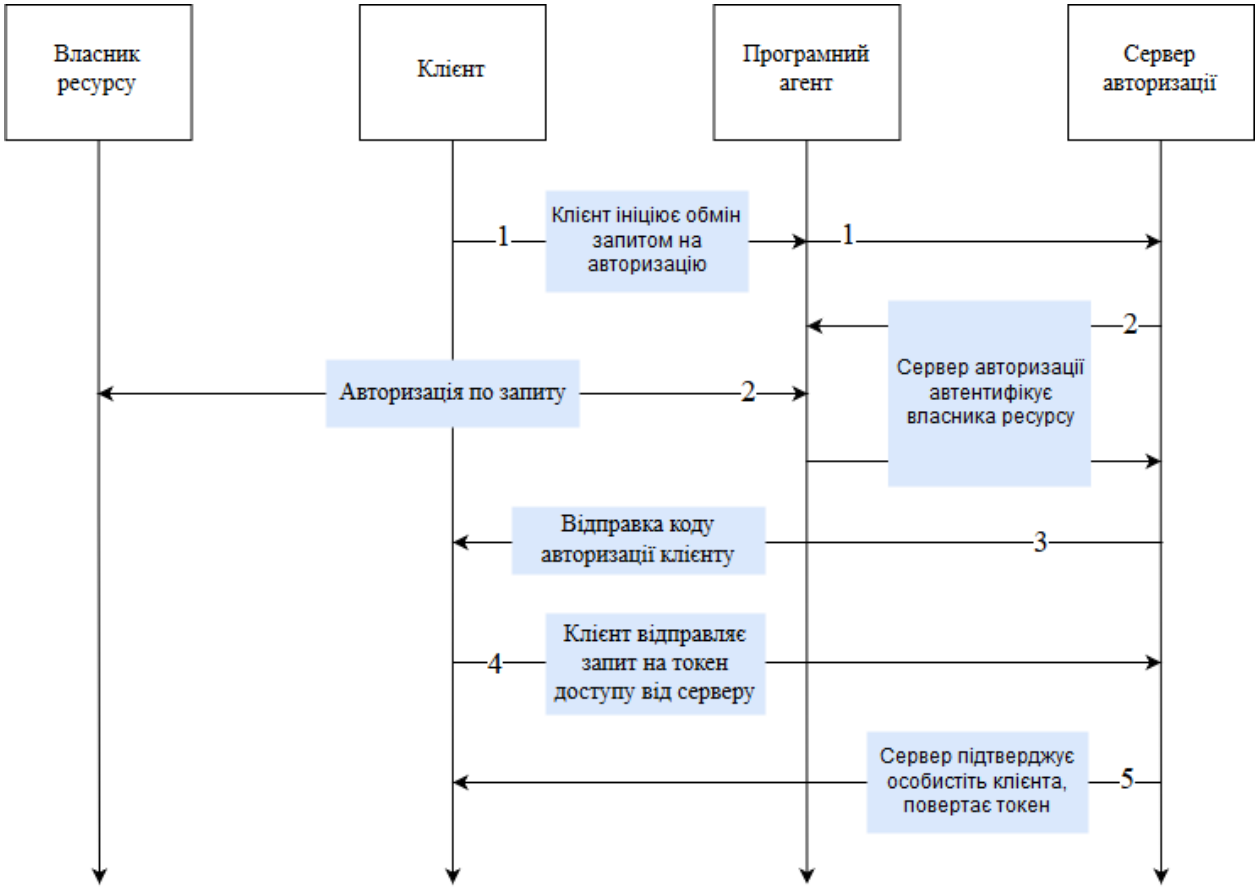

Рисунок 1.1 – Ілюстрація алгоритму роботи OAuth 2.0

Рисунок 1.1 ілюструє наступні кроки:

1. Клієнт розпочинає потік дозволу з направлення програмного агенту до серверу авторизації. При цьому, клієнт відправляє ідентифікатор, запитуваний обсяг, локальний стан та URI перенаправлення, на яке сервер авторизації відправить програмного агента після надання або відхилення права доступу.

2. Сервер авторизації автентифікує власника ресурсу, за допомогою програмного агента, та встановлює, чи власник ресурсу надає право доступу клієнту, чи відхиляє його.

3. У разі надання доступу, сервер авторизації перенаправляє програмного агента до клієнта, використовуючи URI перенаправлення, надане раніше клієнтом. URI перенаправлення включає в себе код авторизації, та локальний стан, наданий клієнтом.

4. Клієнт відправляє запит до сервера авторизації на отримання токену доступу. Під час цього, клієнт використовує URI перенаправлення із кодом верифікації.

5. Сервер авторизації автентифікує клієнта, при успішній автентифікації відповідає клієнту токеном доступу.

При цьому, OAuth 2.0 не надає застосунку третьої сторони інформацію про користувача. Клієнт OAuth 2.0 відсилає запит на токен, отримує токен и одразу використовує його для доступу до, наприклад, API. Таким чином, OAuth 2.0 дуже добре виконує завдання авторизації, але дуже лімітований щодо автентифікації.

1.2.2 SAML 2.0

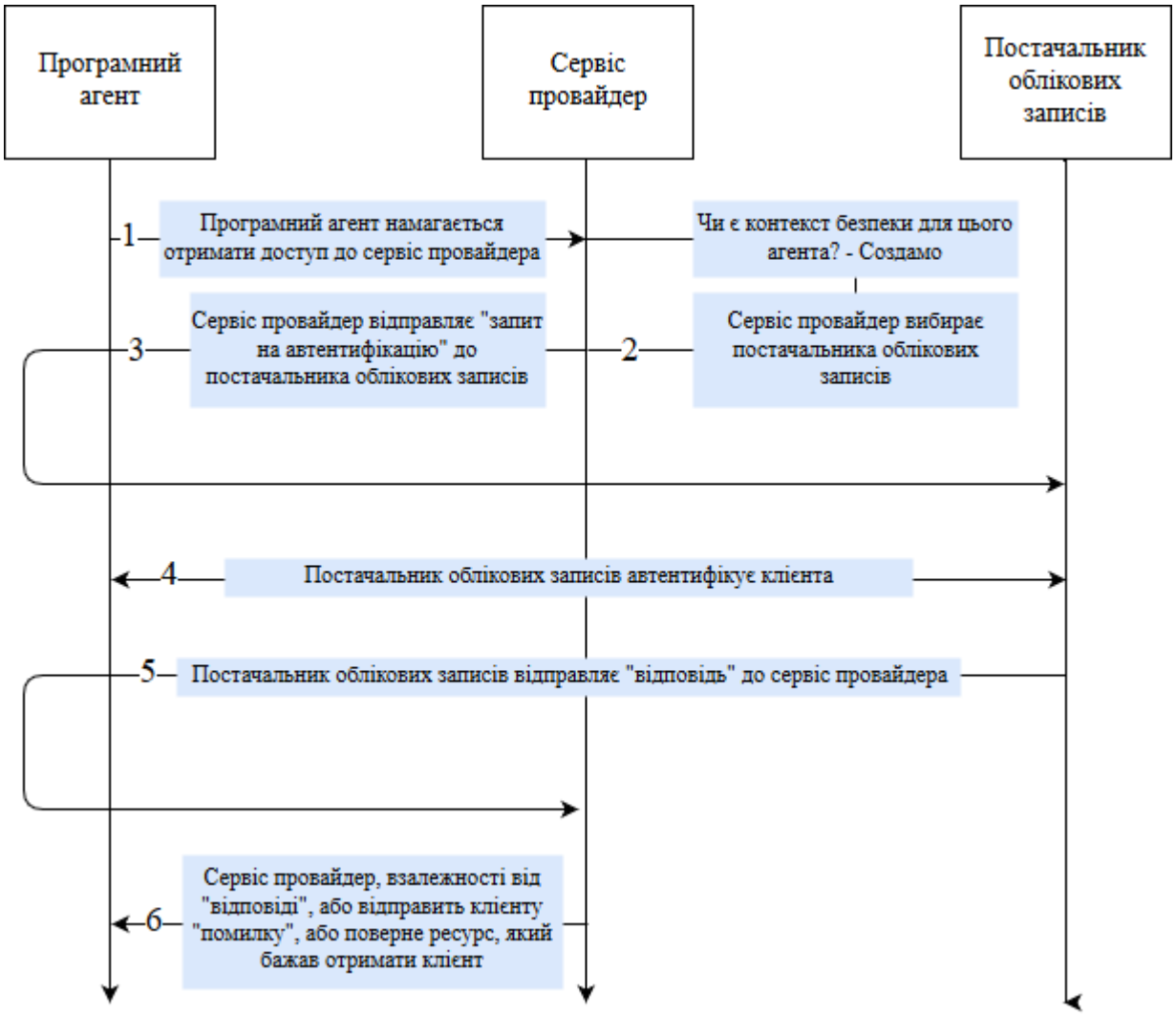

Рисунок 1.2 – Ілюстрація алгоритму роботи SAML 2.0

Рисунок 1.2 ілюструє наступні кроки:

1. Клієнт намагається отримати доступ до захищеного ресурсу через програмного агента без контексту безпеки.

2. Сервіс провайдер вибирає постачальника облікових записів, засоби, за допомогою яких це здійснюється, залежать від реалізації.

3. Сервіс провайдер відправляє «запит на автентифікацію», який доставляється програмним агентом постачальнику облікових засобів.

4. Постачальник облікових записів використовує деякий метод для аутентифікації клієнта.

5. Постачальник облікових записів через програмного агента відправляє «відповідь» до сервіс провайдера. «Відповідь» може бути як «помилкою», так и підтвердженням автентифікації.

6. Після отримання відповіді, сервіс провайдер відповідає програмному агенту чи якоюсь помилкою, якщо автентифікація була невдалою, чи ресурсом, на який програмний агент запитував.

1.2.3 OpenID Connect 1.0

OpenID Connect побудований на основі OAuth 2.0 для того, щоб надавати інформацію, не тільки потрібну для автентифікації, але й для авторизації.

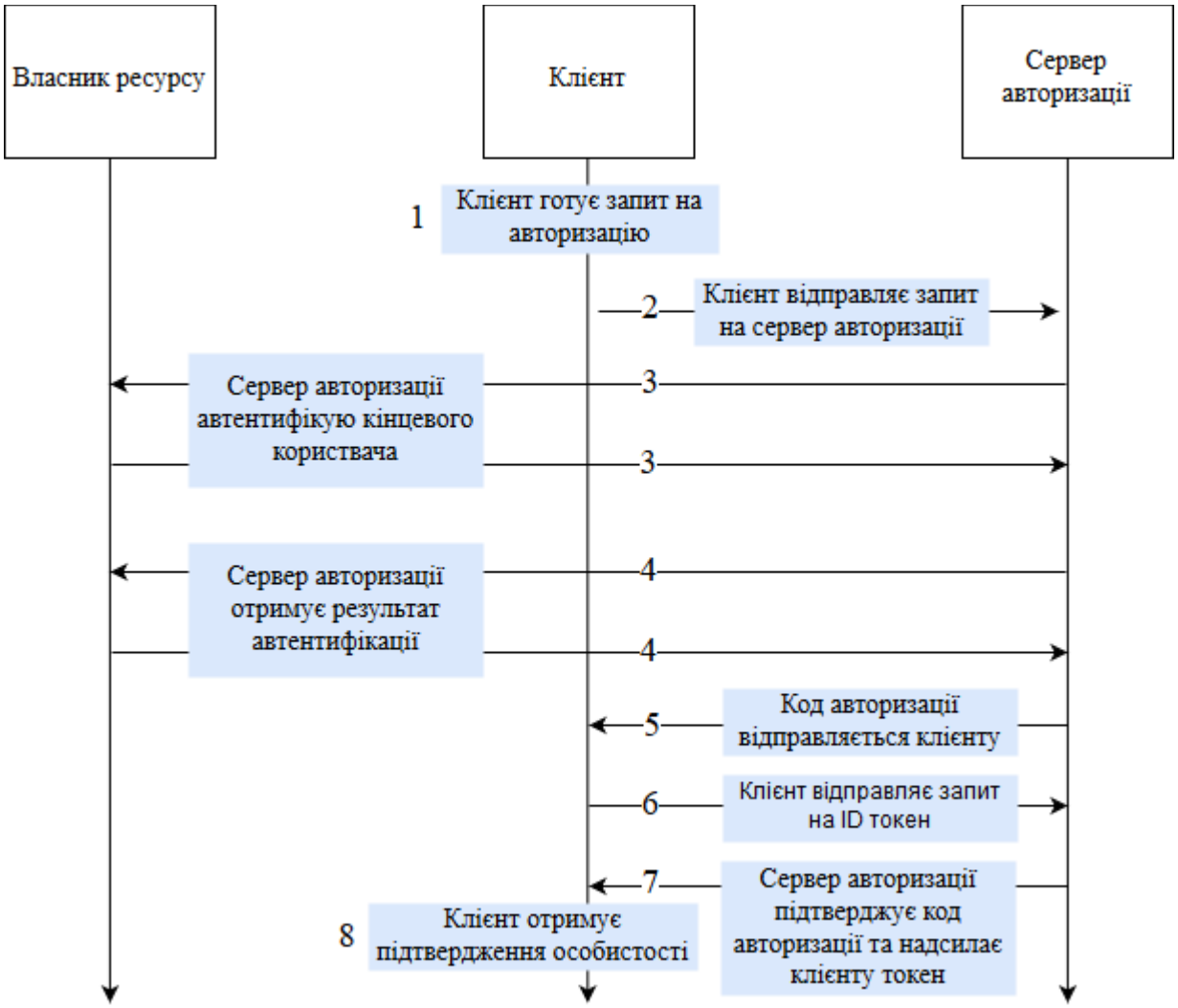

Рисунок 1.3 – Ілюстрація алгоритму роботи OpenID Connect 1.0

Рисунок 1.3 ілюструє наступні кроки:

1. Клієнт готує запит автентифікації, що містить потрібні параметри.
2. Клієнт відсилає запит до сервера авторизації.

3. Сервер авторизації автентифікує кінцевого користувача.

4. Сервер авторизації отримує згоду користувача.

5. Сервер авторизації відправляє клієнту токен доступу

6. Клієнт відправляє запит на отримання ID токена, використовуючи код авторизації.

7. Клієнт отримує відповідь від сервера, що містить ID токен, чи токен доступу.

8. Клієнт підтверджує ID токен та отримує підтвердження суб'єкта користувача.

1.3 Результати аналізу протоколів

Проаналізувавши дані протоколи, можна зробити висновок, що SAML і OpenID Connect забезпечують аутентифікацію, а також авторизацію. SAML, безумовно, є більш складним для реалізації. Але з точки зору безпеки, OAuth та OpenID Connect можуть забезпечити найбільш гнучку модель, яку можна масштабувати.

Таблиця 1.1 – Порівняння протоколів

Протокол	Кращі випадки для використання
OAuth 2.0	API застосунків, UMA
SAML 2.0	SSO на підприємстві,
OpenID Connect 1.0	SSO для клієнтів, CIAM, мобільні застосунки

OAuth 2.0 забезпечує найкращий варіант вирішення задачі авторизації для веб застосунків для даної роботи. Але також, при обираючи протокол для побудови свого рішення, треба звернути увагу на обмеження свого проектного рішення, а також на потребу автентикації в проекті.

2 АНАЛІЗ АТАК НА ПРОТОКОЛ OAUTH 2.0

2.1 Припущення щодо атаки

Зробимо припущення щодо зловмисника, та ресурсів, що йому доступні:

- зловмисник має повний доступ до мережі між клієнтом та сервером авторизації, а також клієнтом та сервером ресурсів. Зловмисник може підслуховувати будь-які зв'язки між цими сторонами. Але в нього немає доступу до зв’язку між сервером авторизації та сервером ресурсів
- зловмисник має необмежені ресурси для проведення атаки
- дві з трьох сторін, які беруть участь у протоколі OAuth, можуть вступити в змову для нападу проти третьої сторони. Наприклад, клієнт та сервер авторизації можуть перебувати під контролем зловмисника і вступити в змову, щоб обдурити користувача, для отримання доступ до ресурсів.

2.2 Припущення щодо архітектури

Зробимо припущення щодо функцій, обмежень та варіантів розробки різних об'єктів, де буде проводитись розгортання OAuth, а також, щодо конфіденційної інформації, що може оброблюватись на цих об’єктах. Ці припущення є основою аналізу загроз.

При розгортанні проектів, протокол OAuth залишає деякий рівень свободи розробникам щодо того, як саме реалізувати і застосувати стандарт. Основна специфікація визначає основні поняття сервера авторизації та сервера ресурсів. Обидва

сервери можуть бути реалізовані як в одному серверному об'єкті, так і можуть бути розділені в реалізації на два різних об’єкти. Розділення серверів авторизації та ресурсів на два різні об’єкти найчастіше зустрічається у випадку багато-сервісного провайдера, що має одну систему авторизації та автентифікації.

2.3 Сервер авторизації

На сервері авторизації може зберігатися або використовуватися наступна інформація:

- імена та паролі користувачів
- ідентифікатори та секрети клієнтів
- маркери оновлення для конкретного клієнта
- маркери доступу, специфічні для клієнта
- Сертифікат / ключ HTTPS
- під час процесу авторизації: «redirect_uri», «client_id», «код» авторизації»

2.3.1 Сервер ресурсів

На сервері ресурсів може зберігатися або використовуватися наступна інформація:

- інформація користувача (не специфікована інформація за межами дії програми)
- сертифікат / ключ HTTPS
- облікові дані сервера авторизації (дизайн на основі ручки; див. розділ 3.1) або спільний секретний / відкритий ключ сервера авторизації

- маркери доступу (на кожен запит)

Вважається, що у сервера ресурсів немає інформації про маркери оновлення, паролі користувачів та секрети клієнта.

2.3.2 Клієнт

В контексті даної роботи, клієнт – це застосунок, що робить захищені запити від імені власника ресурсу та з його авторизацією.

Існують різні типи клієнтів з різними характеристиками реалізації та безпеки, такими як веб-клієнт, клієнт на основі агенту користувача та нативні застосунки.[1]

Наступна інформація зберігається та доступна на стороні клієнта:

- Ідентифікатор/секрет клієнта
- Один чи більше маркерів оновлення (постійні), та маркери доступу (тимчасові) для кожного кінцевого користувача/контексту безпеки
- сертифікати довіреного центру сертифікації (HTTPS)
- під час процесу авторизації: «redirect_uri», «код» авторизації»

2.4 Функції безпеки

Наведемо деякі з функцій безпеки, вбудовані в протокол автентифікації OAuth 2.0, призначені для зниження ризику атак та вирішення питань безпеки.

2.4.1 Маркери (Токени)

OAuth широко використовує багато типів маркерів (маркери доступу, маркери оновлення, «коди» авторизації). Інформаційний сенс маркера може бути представлений двома способами наступним чином:

- Дескриптор(«handle») - це посилання на деяку внутрішню структуру даних на сервері авторизації; внутрішня структура даних містить атрибути маркера, такі як ідентифікатор користувача (UID), область видимості тощо. Дескриптори дозволяють легко скасувати дію маркера та не потребують криптографічного захисту змісту маркера від модифікації. З іншого боку, дескриптори потребують зв'язку між суб’єктами операції (наприклад, сервером авторизації і сервером ресурсів) для перевірки автентичності маркера, та отримання прикріпленої до маркера інформації. Але подібна комунікація між може призвести до негативного впливу на продуктивність та масштабованість, якщо обидва суб’єкти знаходяться в різних системах. Таким чином, дескриптори, як правило, використовуються, якщо суб’єкт, що дає запит на інформацію, та отримує його, є однаковим. Маркер «handle» часто називають «непрозорим» маркером, оскільки сервер ресурсів не повинен мати можливість інтерпретувати маркер безпосередньо; він просто використовує маркер.
- Твердження (assertion, автономний маркер) - інтерпретуємий маркер. Зазвичай, твердження прив’язано до деякої кількості суб’єктів, а також виділяється на фіксований час и має цифровий підпис для забезпечення цілісності та автентифікації

походження даних. Також твердження містить інформацію про користувача та клієнта. Зазвичай, твердження можуть бути безпосередньо перевірені і використані сервером ресурсів без взаємодії з сервером авторизації. Це призводить до кращої продуктивності та масштабованості, коли суб'єкт, що дає запит на інформацію, та суб'єкт, що отримує його, знаходяться в різних системах. Але реалізація скасування дії маркера при використанні тверджень, важча, ніж при використанні дескрипторів.

Маркери можна використовувати у два способи для відправлення запитів на сервері ресурсу, такі як:

- Маркер носія (bearer token) - це маркер, який може використовуватися будь-яким клієнтом, який отримав маркер [2]. Оскільки простого володіння достатньо для використання маркера, важливий зв'язок між кінцевими точками, для забезпечення того, що тільки авторизована кінцева точка може отримати маркер. Маркер носія зручно використовувати в клієнтських застосунках, не має додаткових вимог при використанні (наприклад, доказ автентичності клієнта). Маркер носія за характеристиками схожі на кукі, що використовуються технологією єдиного входу.
- Маркер перевірки (proof token) - це маркер, який може використовуватися лише певним клієнтом. Кожне використання маркера вимагає від клієнта виконання деякої дії, яка показує, що клієнт дійсно є авторизованим користувачем цього маркера. Прикладами цього є маркери доступу типу MAC, які вимагають від клієнта підписувати запит на ресурси деяким секретом, що відповідає певному маркеру, надісланому з запитом.

2.4.2 Область видимості (межі дії)

Область видимості являє собою авторизацію доступу, асоційовану з певним маркером до серверів ресурсів, ресурсів і методів на цих ресурсах. Область видимості - спосіб OAuth керувати рівнем доступу, пов'язаним з маркером доступу. Область видимості може контролюватися сервером авторизації чи кінцевим користувачем, для обмеження доступу до ресурсів для клієнтів OAuth, які вважаються менш безпечними та надійними. За бажанням, клієнт може вимагати зміни області видимості для маркера, тобто зміни рівня доступу, але тільки в області видимості меншій, ніж була надана спочатку, наприклад, для зменшення потенційного впливу, якщо цей маркер відправлений через незахищені канали. Також, зазвичай область визначення зазвичай обмежена терміном дії маркера доступу.

2.4.3 Термін дії маркера доступу

Параметр протоколу «expires_in» дозволяє серверу авторизації (або від імені кінцевого користувача) термін дії маркера доступу і передавати цю інформацію клієнту. Цей механізм може використовуватися для видачі короткоживучих маркерів для клієнтів OAuth, яких сервер авторизації вважає менш захищеними, або коли маркери передаються по незахищеним каналам.

2.4.4 Маркер доступу

Маркер доступу може використовуватись клієнтом для отримання доступу до ресурсу. Маркери доступу зазвичай мають короткий термін дії, який охоплює типовий

час одного сеансу. Маркер доступу може бути оновлений за допомогою маркера оновлення. Короткий термін служби маркера доступу, в поєднанні з використанням маркерів оновлення, дає можливість пасивного скасування авторизації доступу після закінчення терміну дії поточного маркера доступу.

2.4.5 Маркер оновлення

Маркер – це довготривала авторизація певного клієнта для доступу до ресурсів від імені власника ресурсу. Такими маркерами обмінюються тільки клієнт та сервер авторизації. Клієнти використовують цей тип маркера для отримання («оновлення») нових маркерів доступу, що використовуються для викликів сервера ресурсів.

Маркер оновлення, разом з коротким часом життя маркера доступу, може бути використаний для надання більш довгого доступу до ресурсів без використання авторизації кінцевого користувача. Це надає перевагу, коли сервери ресурсів і сервери авторизації не є одним і тим же об'єктом, наприклад, в розподіленому середовищі, оскільки маркер оновлення завжди обмінюється на сервері авторизації. Сервер авторизації може відкликати маркер оновлення в будь-який час, що призводить до скасування наданого доступу після закінчення терміну дії поточного маркера доступу. У зв'язку з цим, короткий термін дії маркеру є важливим, якщо в пріоритеті стоїть своєчасне скасування доступу.

Маркер оновлення також є секретом, пов'язаним з ідентифікатором клієнта та екземпляром клієнта, який спочатку запитував авторизацію; маркер оновлення також представляє дозвіл доступу зі сторони оригінального власника ресурсу. Це забезпечується процесом авторизації наступним чином:

- Власник ресурсів і агент користувача в першу чергу безпечно доставляють «код» авторизації до екземпляру клієнта.

- Клієнт негайно відправляє його по захищеному каналу зв’язку на транспортному рівні до сервера авторизації, після чого надійно зберігає маркер оновлення з довгим терміном дії.
- Клієнт завжди використовує маркер оновлення в безпечній передачі на транспортному рівні до сервера авторизації, щоб отримати маркер доступу (при необхідності пролонгувати маркера оновлення).

Таким чином, доки клієнт може забезпечити конфіденційність конкретного маркера, маркер оновлення може також використовуватися як альтернативний спосіб автентифікації самого клієнта.

2.4.6 «Код» авторизації

«Код» авторизації - проміжний результат успішного процесу авторизації кінцевого користувача, який використовується клієнтом для отримання маркерів доступу і оновлення. «Коди» авторизації надсилаються на URI клієнта для переадресації замість маркерів для двох цілей:

- Потоки через браузер, показують параметри протоколу потенційним нападникам через параметри запиту URI (HTTP referrer), кеш браузера або файли журналу браузера, що можуть бути відтворені. Для того, щоб зменшити цю загрозу, «коди» авторизації передаються замість маркерів, а потім обмінюються на маркери за допомогою більш безпечного прямого з'єднання між клієнтом і сервером авторизації.
- Набагато простіше автентифікувати клієнтів під час прямого запиту між клієнтом та сервером авторизації, аніж у контексті непрямого запиту на авторизацію. Останній варіант потребує використання цифрових підписів.

2.4.7 URI перенаправлення

URI перенаправлення допомагає виявляти шкідливих клієнтів і запобігає фішинг-атакам від клієнтів, які намагаються обдурити користувача, вважаючи, що фішер-клієнт є справжнім клієнтом. Значення фактичного URI-перенаправлення, яке використовується в запиті на авторизацію, має бути представлено і перевірено, коли «код» авторизації обмінюється на маркери. Це допомагає запобігти атакам, де «код» авторизації розкривається за допомогою перенаправників і підроблених клієнтів веб-застосунків. Сервер авторизації має вимагати, щоб публічні клієнти та конфіденційні клієнти, використовуючи тип неявного дозволу, попередньо реєстрували свої URI перенаправлення і звіряли зі зареєстрованим URI переадресації в запиті авторизації.

2.4.8 Параметр «стану»

Параметр «стану» використовується для зв'язування запитів і зворотних викликів для запобігання атак з міжсайтової підробки запитів, при яких зловмисник дає доступ до власних ресурсів, а потім приводить користувача до наступного перенаправлення з маркером атакуючого. Цей параметр повинен прив'язуватися до стану автентифікації в агенті користувача, і, відповідно до основних специфікацій OAuth, агент користувача повинен бути здатним зберігати параметр «стану» в місці, доступному тільки клієнту і агенту користувача, тобто тим, хто підлягає політиці безпеки одного походження.

2.4.9 Ідентифікатор клієнта

Протоколи автентифікації, як правило, не враховують ідентичність компонента програмного забезпечення, що діє від імені кінцевого користувача. OAuth робить це для того, щоб підвищити рівень безпеки в делегованих сценаріях авторизації і тому, що клієнт зможе діяти без присутності користувача.

OAuth використовує ідентифікатор клієнта для зіставлення пов'язаних запитів до того ж самого автора, наприклад

- конкретного процесу авторизації кінцевого користувача і відповідного запиту на кінцевій точці маркера для обміну «кодом» авторизації для маркерів, або
- початкову авторизацію та видачу маркера кінцевим користувачем конкретному клієнту, а також наступні запити цього клієнта на отримання маркерів без згоди користувача (автоматична обробка повторних дозволів)

Цей ідентифікатор може також використовуватися сервером авторизації для відображення відповідної реєстраційної інформації користувачеві при запиті на

отримання області видимості, яку запитує конкретний клієнт. Ідентифікатор клієнта може використовуватися для обмеження кількості запитів для конкретного клієнта або стягнення оплати з клієнта за кожен запит. Окрім того, може бути корисно диференціювати доступ для різних клієнтів, наприклад, в файлах журналу сервера.

OAuth визначає два типи клієнтів, конфіденційні та загальнодоступні, на підставі їхньої здатності перевіряти автентичність з сервером авторизації (тобто здатність зберігати конфіденційність своїх облікових даних клієнта). Конфіденційні клієнти здатні підтримувати конфіденційність облікових даних клієнта (тобто секрет клієнта, пов'язаного з ідентифікатором клієнта) або здатні захищати автентифікацію клієнта, використовуючи інші засоби, такі як затвердження клієнта (наприклад, SAML) або шифрування ключів. Останній вважається більш безпечним.

Сервер авторизації повинен визначити, чи здатний клієнт зберегти конфіденційність секрету або використовувати безпечну автентифікацію. Альтернативно, кінцевий користувач може перевірити ідентичність клієнта, наприклад, встановлюючи лише довірені програми. В деяких випадках, URI перенаправлення може використовуватися для запобігання видачі облікових даних підробленому клієнту після отримання авторизації кінцевого користувача, але цим не можна користуватись для перевірки та підтвердження ідентифікатора клієнта.

Клієнти можуть бути класифіковані наступним чином, виходячи з типу клієнта, профілю (наприклад, нативний чи веб-додаток) і модель розгортання:

- Незалежний від платформи «client_id» з попередньо зареєстрованими «redirect_uri» і без «client_secret». Такий ідентифікатор використовується декількома установками того ж самого пакету програмного забезпечення. Ідентифікатор такого клієнта може бути перевірений тільки за допомогою кінцевого користувача. Це прийнятний варіант для нативних застосунків для ідентифікації клієнта з метою відображення мета-інформації про клієнта користувачеві, або для того, щоб розрізняти клієнтів у файлах журналів.

Скасування прав, пов'язаних з таким ідентифікатором клієнта, вплине на всі установки відповідного програмного забезпечення.

- Залежний від платформи "client_id" з попередньо зареєстрованими "redirect_uri" та з "client_secret". Процес реєстрації клієнта забезпечує перевірку властивостей клієнта, наприклад URI перенаправлення, URL веб-сайту та ім'я веб-сайту. Такий ідентифікатор клієнта може бути використаний для всіх відповідних випадків використання, наведених вище. Цей рівень може бути досягнутий для веб-додатків у поєднанні з процедурою реєстрації користувачем. Досягнення цього рівня для нативних застосунків набагато складніше. Встановлення програми здійснюється адміністратором, який перевіряє автентичність клієнта. Скасування прав вплине лише на одне розгортання застосунка.
- Залежний від платформи "client_id" з "client_secret" без перевірених властивостей. Такий клієнт може бути розпізнаний сервером авторизації в транзакціях з наступними запитами: авторизація і видача маркера, запит на оновлення маркера та оновлення маркера доступу. Сервер авторизації не може забезпечити будь-яку з властивостей клієнта кінцевим користувачам. Також може бути дозволена автоматична обробка повторних авторизацій. Такі облікові дані клієнта можуть генеруватися автоматично без перевірки властивостей клієнта, що може бути застосовано для нативних програм. Скасування прав вплине лише на одне розгортання застосунку.

2.5 Атаки на протокол OAuth 2.0

2.5.1 Недостатня перевірка URI перенаправлення

Деякі сервери авторизації дозволяють клієнтам реєструвати шаблони URI переадресації замість повних URI перенаправлень. У цих випадках сервер авторизації під час виконання порівнює фактичне значення параметра URI переадресації в кінцевій точці авторизації з цим шаблоном. Цей підхід дозволяє клієнтам закодувати стан транзакції в додаткові параметри URI перенаправлення або зареєструвати лише один шаблон для декількох URI перенаправлення. З однієї сторони, це збільшує швидкість перевірки, з іншої ж, такий варіант виявився більш складним для реалізації і також призводить до збільшення кількості помилок, ніж перевірка звичайних URI перенаправлень. Також було виявлено декілька успішних атак, що використовували недоліки в реалізації шаблону. Такий недолік ефективно порушує ідентифікацію або автентифікацію клієнта (залежно від типу гранту та клієнта) і дозволяє зловмиснику отримати код авторизації або маркер доступу:

- шляхом безпосередньої відправки агента користувача до URI під управлінням атакуючих або
- розкриваючи зловмисникам облікові дані OAuth, використовуючи відкрите перенаправлення на клієнті в поєднанні з тим, як агенти користувача обробляють фрагменти URL-адрес.

2.5.1.1 Атака на надання коду авторизації

Для загальнодоступного клієнта, що використовує код типу гранту, атака виглядатиме так:

Припустимо, що URL-адреса перенаправлення «https://*.website.example/*» зареєстровано для клієнта «d3GfVHdmt7». Цей шаблон дозволяє перенаправляти URI, які вказують на будь-який хост, що знаходиться в домені website.example. Отже, якщо зловмисникові вдається встановити хост або субдомен у website.example, він може видати себе за законного клієнта. Припустимо, що зловмисник встановлює хост «eviler.website.example».

1. Зловмисник повинен обдурити користувача відкриттям підробленого URL у його браузері, який запускає сторінку під контролем атакуючого, скажімо, «https://www.eviler.example».

2. Ця URL-адреса ініціює запит авторизації з ідентифікатором клієнта законного клієнта до кінцевої точки авторизації. В цьому прикладі запиту на авторизацію:

```
GET
/authorize?response_type=code&client_id=d3GfVHdmt7&state=xyz&redirect_uri=https%3
A%2F%2Feviler.website.example% 2Fcb HTTP/1.1
Host: server.website.example
```

3. Сервер авторизації перевіряє URI перенаправлення для ідентифікації клієнта. Оскільки шаблон дозволяє довільні імена доменів у «website.example», запит на авторизацію обробляється відповідно до ідентичності законного клієнта. Це включає в себе спосіб подання запиту на згоду користувача кінцевому користувачеві. Якщо дозволено автоматичне схвалення запитів(яке не рекомендується для публічних клієнтів [1]), атака може бути виконана ще простіше.

4. Якщо користувач не розпізнає атаку, «код» авторизації видається і надсилається безпосередньо клієнту зловмисника.

5. Оскільки зловмисник уособлює публічний клієнт, він може безпосередньо обміняти код на маркери на відповідній кінцевій точці маркера.

Але ця атака не буде безпосередньо працювати для конфіденційних клієнтів, оскільки обмін кодами вимагає автентифікації з легітимним секретом клієнта. Зловмисник повинен використовувати законного клієнта для отримання коду (наприклад, шляхом виконання ін'єкції коду).

2.5.1.2 Атака на неявний грант

Описана вище атака працює також для неявного надання гранту. Якщо зловмисник може надіслати відповідь серверу авторизації до URI під його контролем, він безпосередньо отримає доступ до фрагмента з маркером доступу.

Крім того, неявні клієнти можуть піддаватися подальшому виду атак. Цей вид атак використовує той факт, що агенти користувачів приєднують фрагменти до цільової URL переадресації, якщо заголовок розташування не містить цих фрагментів [3]. Ця атака поєднує в собі цю поведінку із тим, що клієнт виступає в ролі відкритого переадресувача. Це дозволяє обходити навіть строгі шаблони URI переадресації (але не строгу перевірку URL).

Припустимо, що шаблон для клієнта «d3GfVHdmt7» є «https://client.website.example/cb?*», Тобто будь-який параметр допускається для перенаправлення в «https://client.website.example/cb». На жаль, клієнт який виступає в ролі відкритого переадресувача, відкриває себе для атаки. Ця кінцева точка підтримує параметр "redirect_to", який приймає цільову URL-адресу, і надсилатиме браузер до цієї URL-адреси за допомогою HTTP заголовка перенаправлення Location 302.

1. Так само, як з атакою вище, зловмисник повинен обдурити користувача відкриттям підробленого URL в його браузері, який запускає сторінку під контролем атакуючого, скажімо "https: //www.eviler.example".

2. URL ініціює запит авторизації, який дуже схожий на атаку на потік коду. Як відмінності, він використовує відкритий переадресувач, кодуючи «redirect_to=https://client.evil.example» в URI перенаправлення, і використовує тип відповіді «маркер»:

```
GET /authorize?response_type=token&client_id=d3GfVHdmt7&state=xyz&
redirect_uri=https%3A%2F%2Fclient.website.example%2Fcb%26redirect_to%253Dhttps%253A%252
F%252Fclient.eviler.example%252Fcb HTTP/1.1
Host: server.website.example
```

3. Оскільки URI перенаправлення відповідає зареєстрованому шаблону, сервер авторизації дозволяє запит і надсилає отриманий маркер доступу з перенаправленням 302

```
HTTP/1.1 302 Found
Location: https://client.somesite.example/cb?redirect_to%3Dhttps%3A%2F%2F
client.eviler.example%2Fcb#access_token=2YotnFZFEjr1zCsicMWpAA&...
```

4. На сайті example.com запит надходить до відкритого переадресувача. Він буде читати параметр перенаправлення і видасть HTTP заголовок перенаправлення 302 на URL "https://eviler.example.com/cb".

HTTP/1.1 302 Found

Location: https://client.eviler.example/cb

5. Оскільки переадресація на client.somesite.example не включає фрагмент у заголовку Location, агент користувача повторно приєднає оригінальний фрагмент «#access_token=3FsYndGDVjr4dXgfdCGkdFF&...» до URL і перейдіть до наступної URL-адреси:

https://client.eviler.example/cb#access_token=3FsYndGDVjr4dXgfdCGkdFF&...

6. Сторінка зловмисника на client.evil.example може отримати доступ до фрагмента і отримати маркер доступу.

2.5.2 Витік «кодів» або станів через сторони клієнта або сервера авторизації через Referrer заголовки

Коди авторизації або значення стану можуть бути ненавмисно розкриті для зловмисників через Referrer заголовок, при наявності витоку інформації з веб-сайту клієнта або з веб-сайту сервера авторизації.

1. Витік з клієнта OAuth: це вимагає, щоб клієнт, в результаті успішного запиту на авторизацію, надав сторінку, що

- містить посилання на інші сторінки під контролем атакуючого (оголошення, FAQ тощо) і користувач натискає на таке посилання, або
- включає вміст третіх сторін (зображення), наприклад, якщо сторінка містить вміст, створений користувачем (блог).

Як тільки браузер переходить на сторінку зловмисника або завантажує вміст третіх сторін, зловмисник отримує URL-адресу відповіді на авторизацію і може витягувати код або стан.

2. Витік з серверу авторизації: Подібним чином зловмисник може дізнатися стан, якщо кінцева точка авторизації на сервері авторизації містить посилання або вміст третіх сторін, як зазначено вище.

Наслідки: зловмисник, який дізнається дійсний код через Referrer заголовок, може виконувати ті ж атаки, що описані в Розділі 4.1.1. Якщо зловмисник дізнається стан, то захист CSRF, досягнутий за допомогою стану, втрачається, що призводить до атак CSRF.

2.5.3 Атаки через історію браузера

Коди авторизації та маркери доступу можуть опинитися в історії відвідуваних URL-адрес веб-переглядача, дозволяючи атаки, описані нижче.

2.5.3.1 Код в історії браузера

Коли браузер переходить до "client.com/redirection_endpoint?code=asdf" в результаті перенаправлення з кінцевої точки авторизації провайдера, URL-адреса, що включає код авторизації, може опинитися в історії браузера. Зловмисник, який має доступ до пристрою, може отримати код і спробувати його повторити.

2.5.3.2 Маркер доступу в історії браузера

Маркер доступу може опинитися в історії веб-переглядача, якщо клієнт або просто веб-сайт, який вже має маркер, навмисно переходить на сторінку, наприклад, «provider.com/get_user_profile?access_token=qwerty». Насправді подібна практика не рекомендується, бажано передати маркери через заголовок [2], але на практиці веб-сайти часто просто передають маркер доступу до параметрів запиту.

У випадку неявного гранту, URL-адреса, подібна до «client.com/redirection_endpoint#access_token=qwerty», може також опинитися в історії веб-переглядача в результаті перенаправлення з кінцевої точки авторизації провайдера.

2.5.4 Підміна

Підміна - це сценарій атаки, коли клієнт OAuth взаємодіє з декількома серверами авторизації, при цьому, як це часто буває, використовується динамічна реєстрація. Метою атаки є отримання коду авторизації або маркера доступу, при цьому вимусити клієнта відправити ці дані до зловмисника замість того, щоб використовувати їх у відповідній кінцевій точці на сервері авторизації/ресурсу.

Передумови: Для того, щоб атака працювала, ми припускаємо, що

1. для надання неявного гранту або коду авторизації використовуються декілька серверів авторизації, один з яких вважається «чесним» (honest authorization server, H-AS), а інший керується зловмисником (attacker’s authorization server, A-AS)
2. клієнт зберігає обраний користувачем сервер авторизації у сеансі, прив'язаному до браузера користувача, і використовує той же URI кінцевої точки перенаправлення для кожного сервера авторизації
3. зловмисник може маніпулювати першою парою запиту/відповіді від браузера користувача до клієнта (в якому користувач вибирає певний сервер авторизації і потім перенаправляється клієнтом до цього серверу).

Деякі з описаних нижче варіантів атаки вимагають різних передумов.

Нижче ми припускаємо, що клієнт зареєстрований за допомогою H-AS (URI: «https://honest.as.example», id клієнта: 2FZSJNLieQ) і з A-AS (URI: «https://attacker.example», id клієнта: EV666DIL).

Атака на надання коду авторизації:

1. Користувач вибирає почати надання гранту за допомогою H-AS (наприклад, натиснувши кнопку на веб-сайті клієнта).
2. Зловмисник перехоплює цей запит і змінює вибір користувача на "A-AS".

3. Клієнт зберігає у сесії користувача, що користувач вибрав "A-AS" і перенаправляє користувача на кінцеву точку авторизації A-AS, надіславши наступну відповідь:

```
HTTP/1.1 302 Found
Location:
https://attacker.example/authorize?response_type=code&client_id=666RVZJTA
```

4. Тепер зловмисник перехоплює цю відповідь і змінює перенаправлення таким чином, що користувач перенаправляється на H-AS. Зловмисник також замінює ідентифікатор клієнта в A-AS ідентифікатором клієнта в H-AS, в результаті чого до веб-переглядача надсилається наступна відповідь:

```
HTTP/1.1 302 Found
Location:
https://honest.as.example/authorize?response_type=code&client_id=2FZSJNLieQ
```

5. Тепер користувач дає клієнту доступ до своїх ресурсів у H-AS. H-AS видає код і відправляє його (через веб-переглядач) назад до клієнта.

6. Оскільки клієнт все ще припускає, що код був виданий A-AS, він спробує отримати код на кінцевій точці маркера A-AS.

7. Отже, зловмисник отримує код і може або обміняти код на маркер доступу (для публічних клієнтів), або виконати ін'єкцію коду.

Варіанти:

1. Непрямий грант

При використанні неявного гранту, зловмисник отримує маркер доступу замість коду; решта атаки працює як описано вище.

2. Підміна без перехоплення

Варіант вищезазначеної атаки працює, навіть якщо першу пару запит/відповідь не можна перехопити (наприклад, тому що для захисту цих повідомлень використовується TLS): Тут ми припускаємо, що користувач хоче почати надання гранту за допомогою A-AS (і не H-AS). Після того, як клієнт перенаправив користувача до кінцевої точки авторизації в A-AS, зловмисник негайно перенаправляє користувача на H-AS (змінюючи ідентифікатор клієнта «2FZSJNLieQ»). (У цьому випадку пильний користувач може

виявити, що він намагається використовувати A-AS замість H-AS.) Після цього атака протікає так само, як у першому кроці вище.

3. Різні URI переспрямування для серверів авторизації

Якщо клієнти використовують різні URI перенаправлення для різних серверів авторизації, не зберігають вибрані сервери авторизації в сесії користувача, і сервери авторизації не перевіряють URI перенаправлення ретельно, зловмисники можуть провести атаку під назвою " Міжсайтова підробка запиту" [4].

2.5.5 Ін'єкція коду

У такій атаці зловмисник, використовуючи на пристрої, що знаходиться під його контролем, намагається провести ін’єкцію вкраденого «коду» авторизації до справжнього клієнта. У найпростішому випадку зловмисник хоче використовувати код у своєму клієнті. Але є ситуації, коли це може бути неможливим або не бажаним. Прикладами є:

- Зловмисник хоче отримати доступ до певних функцій цього конкретного клієнта. Наприклад, зловмисник хоче видати себе за свою жертву в певному застосунку або на певному веб-сайті.
- Код прив'язаний до певного конфіденційного клієнта, і зловмисник не може отримати необхідні облікові дані клієнта, щоб сам отримати «код» авторизації.
- Сервери авторизації або ресурсів обмежені певними мережами, до яких зловмисники не можуть отримати безпосередній доступ.

Як виглядає атака?

1. Зловмисник отримує код авторизації, виконуючи будь-які з описаних вище атак.

2. Він виконує звичайний процес авторизації OAuth зі справжнім клієнтом на своєму пристрої.

3. Зловмисник вводить вкрадений код авторизації у відповідь сервера авторизації справжньому клієнту.

4. Клієнт надсилає код до кінцевої точки маркера сервера авторизації, разом з ідентифікатором клієнта, секретом клієнта і фактичним URI перенаправлення.

5. Сервер авторизації перевіряє секрет клієнта, чи був він виданий цьому конкретному клієнту, і чи відповідає фактичний URI перенаправлення параметру redirect_uri.

6. Якщо всі перевірки є успішними, сервер авторизації видає клієнту доступ та інші маркери, тож зловмисник може видати себе за законного користувача.

Очевидно, що перевірка на етапі (5) не відбудеться, якщо «код» авторизації був виданий на інший id клієнта, наприклад, клієнту, налаштованому зловмисником. Перевірка також зазнає невдачі, якщо код авторизації вже був отриманий справжнім користувачем і мав термін дії тільки на одне використання.

Спроба ін'єкції коду, отриманого через шкідливе програмне забезпечення, що прикидається справжнім клієнтом, також має бути виявлена, якщо сервер авторизації зберігає повний URI перенаправлення, використовуваний у запиті авторизації, і порівнює його з параметром redirect_uri.

Також, при виконанні рекомендацій [1], у сценарії атаки, описаному вище, законний клієнт буде використовувати правильний URI перенаправлення, який завжди використовується для запитів авторизації. Але цей URI не відповідає URI, що використовується зловмисником для перенаправлення (в іншому випадку переадресація не приземлиться на сторінку зловмисників). Таким чином, сервер авторизації виявить атаку і відмовиться від обміну кодом.

Примітка: ця перевірка також може виявити спробу ін’єкції коду, який був отриманий з іншого екземпляра цього клієнта на іншому пристрої, якщо виконуються певні умови:

- URI переспрямування сам повинен містити псевдовипадкове число, що утворене протоколом автентифікації (nonce), або інший вид секретних даних одноразового використання
- клієнт прив'язав ці дані до цього конкретного екземпляра.

Але цей підхід конфліктує з ідеєю примусового узгодження URI перенаправлення в кінцевій точці авторизації. Більше того, спостерігалося, що провайдери дуже часто ігнорують вимогу перевірки redirect_uri на цьому етапі, можливо, тому, що вона не є критично важливою для читання специфікацій.

Інші провайдери лише звіряють шаблон redirect_uri проти зареєстрованого URI шаблону переадресації. Це дає серверу авторизації можливість не зберігати зв’язок між URI перенаправлення та відповідним «кодом» авторизації для кожної транзакції. Але така перевірка, очевидно, не відповідає меті специфікації, оскільки URI непрямого перенаправлення не розглядається. Таким чином, будь-яка спроба ін'єкції коду, отриманого за допомогою клієнт-ідентифікатора законного клієнта або використовуючи законний клієнт на іншому пристрої, не буде виявлена у відповідних розгортаннях.

Також передбачається, що вимоги, визначені в [1], збільшують складність реалізації клієнта, оскільки клієнтам необхідно запам'ятати або повторно побудувати правильний URI переадресації для виклику кінцевої точки маркерів.

Тому цей документ рекомендує прив'язувати кожен «код» авторизації до певного екземпляра клієнта на певному пристрої (для певного агенту користувача) в контексті певної транзакції.

2.5.6 Міжсайтова підробка запиту

Атака міжсайтової підробки запиту на URI перенаправлення клієнта дозволяє зловмиснику вводити свій власний «код» авторизації або маркер доступу, що може призвести до того, що клієнт використовує маркер доступу, пов'язаний із захищеними ресурсами атакуючого, а не з жертвою (наприклад, збереження інформацію про банківський рахунок жертви до захищеного ресурсу, керований зловмисником).

Як виглядає атака?

1. Обрати клієнта, в даному випадку example.website.com, розпочати процес автентифікації, отримати перенаправлення від сервера автентифікації, але не переходити за ним.

2. Не відвідуючи останній URL, зберегти його і додати до зображення, iframe тощо:

http://example.website.com/connect/facebook/?code=AQCOtAVov1Cu316rpqPfsR…

3. Після цього потрібно змусити користувача відправити HTTP запит до вашого URL перенаправлення. Цього можна досягти, відправивши користувачу email, чи твіт, чи примусити його відвідати деякий веб-сайт http://example.website.com/anypage.html, який буде містити iframe src=URL. При цьому, користувач має бути в системі, при відправленні запиту.

4. Тепер ваш OAuth аккаунт прив’язаний до аккаунта справжнього користувача на example.website.com, отже ви можете авторизуватись в обліковому аккаунті цього користувача.

2.5.7 Виток маркерів доступу на сервері ресурсів

За деяких обставин може відбуватись виток маркерів доступу з серверу ресурсів.

2.5.7.1 Фішинг маркерів доступу шляхом підроблення серверу ресурсів

Зловмисник може налаштувати свій власний сервер ресурсів і обдурити клієнта надсиланням маркерів доступу до нього, які діють для інших серверів ресурсів. Якщо клієнт надішле на цей сервер підробленого ресурсу дійсний маркер доступу, зловмисник, у свою чергу, зможе використовувати цей маркер для доступу до інших служб від імені власника ресурсу.

Ця атака передбачає, що клієнт не пов'язаний з певним сервером ресурсів (і відповідним URL) під час розробки, але екземпляри клієнта пов’язуються з URL сервера ресурсів під час виконання. Таке пізнє зв'язування є типовим у ситуаціях, коли клієнт використовує стандартний API, наприклад, для електронної пошти, календаря або банківської діяльності, і налаштований користувачем або адміністратором для використання на основі стандартів, використанні в компанії, або використанні конкретним користувачем.

2.5.7.1.1 Метадані

Сервер авторизації може надати клієнту додаткову інформацію про місцезнаходження, де безпечно використовувати його маркери доступу.

У найпростішій формі це вимагає, щоб сервер авторизації публікувала список своїх відомих серверів ресурсів, проілюстрованих у наступному прикладі, використовуючи параметр метаданих resource_servers:

```
HTTP/1.1 200 OK
Content-Type: application/json
{
  “issuer”:“https://server.website.example”,
  “authorization_endpoint”: “https://server.website.example/authorize”,
  “resource_servers”:[
    “email.website.example”,
    ”storage.website.example”,
    ”video.website.example”]
  ...
}
```

Сервер авторизації також може повернути URL-адреси, для яких маркер доступу є корисним для маркера відповіді, що ілюструється прикладом повернення параметра access_token_resource_server:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  “access_token”: “3FsYndGDVjr4dXgfdCGkdFF”,
  “access_token_resource_server”: “https://hostedres.website.example/path1”,
   ...
}
```

2.5.7.1.2 Маркери обмеженого доступу відправника

Як випливає з назви, маркер обмеженого доступу обмежує область визначення маркеру доступу для конкретного відправника. Цей відправник зобов'язаний продемонструвати знання певної таємниці як необхідну умову для прийняття цього маркера на ресурсному сервері.

Типовий потік виглядає так:

1. Сервер авторизації асоціює дані з маркером доступу, який прив’язує цей конкретний маркер до певного клієнту. Прив'язка може використовувати ідентифікацію клієнта, але в більшості випадків сервер авторизації використовує матеріал ключа (або дані, отримані з матеріалу ключа), відомий клієнту.

2. Цей матеріал ключа повинен бути деяким чином розподілений. Або матеріал ключа вже існує, перш ніж сервер авторизації створює прив'язку, або сервер авторизації створює ефемерні ключі. Спосіб розподілення існуючого матеріалу ключа залежить від різних підходів. Наприклад, можна використовувати сертифікати X.509, і в цьому випадку розподіл відбувається явно під час процесу реєстрації. Або матеріал ключа створюється і поширюється на шарі TLS, в цьому випадку він може автоматично відбуватися під час встановлення з'єднання TLS.

3. На сервері ресурсів повинен бути реалізований перевірка доказу володіння. Зазвичай це робиться на прикладному рівні, але ця перевірка може використовувати можливості і транспортного рівню (наприклад, TLS[5]).

2.5.7.1.3 Маркери обмеженого доступу для аудиторії

Обмеження аудиторії встановлює певний маркер доступу, який може бути використаний на сервері ресурсів. Сервер авторизації асоціює маркер доступу з певним сервером ресурсів, і кожен сервер ресурсів зобов'язаний перевірити для кожного запиту, чи маркер, надісланий з цим запитом, може використовуватися на певному сервері ресурсів. Якщо ні, сервер ресурсів повинен відмовити в обслуговуванні відповідного запиту. У загальному випадку обмеження аудиторії обмежують вплив витоку маркерів. У випадку з підробленим сервером ресурсів, це обмеження може також запобігти зловживанню маркером доступу, отриманого шляхом фішингу, на справжньому сервері ресурсів.

Аудиторія може бути виражена в основному за допомогою логічних імен або фізичних адрес (наприклад, URL). Щоб запобігти фішингу, необхідно використовувати фактичний URL, на який клієнт буде надсилати запити. У випадку фішингу ця URL-адреса вказує на підроблений сервер ресурсів. Якщо зловмисник намагається використати маркер доступу на справжньому сервері ресурсів (який має іншу URL-адресу), сервер ресурсів виявить невідповідність (неправильна аудиторія) і відмовиться виконувати запит.

У розгорнутих проектах, де сервер авторизації знає URL-адреси всіх серверів ресурсів, сервер авторизації може просто відмовитися видавати маркери доступу для URL-адрес невідомого сервера ресурсів.

Клієнт повинен повідомити серверу авторизації, на якій URL-адресі він буде використовувати маркер доступу, на який він запитує. Він міг би закодувати інформацію в значенні області видимості.

Замість URL-адреси також можна використовувати відбиток сертифіката X.509 сервера ресурсів як значення для аудиторії. Цей варіант також дозволить виявити спробу

крадіжки URL-адресу справжнього серверу ресурсів, використовуючи дійсний сертифікат TLS, отриманий з іншого центру сертифікації. Також, переважним для конфіденційності вважається приховування URL-адреси серверу ресурсів від серверу авторизації.

Обмеження аудиторії здається простим у використанні, оскільки не вимагає виконання жодних криптографічних операцій на стороні клієнта. Але оскільки кожен маркер доступу пов'язаний з певним сервером ресурсів, клієнт також повинен отримати різні маркери доступу, специфічні для серверу ресурсів, якщо він хоче отримати доступ до декількох серверів ресурсів.

Слід зазначити, що обмеження аудиторії, або взагалі кажучи вказівка від клієнта до серверу авторизації, де він хоче використовувати маркер доступу, має додаткові переваги, що виходять за рамки запобігання витоку маркера. Вона дозволяє серверу авторизації створювати різні маркери доступу, формат і вміст яких специфічно викарбувані для відповідного сервера. Це має величезні переваги у функціонуванні та конфіденційності при розгортанні з використанням структурованих маркерів доступу.

2.5.7.2 Скомпрометований сервер ресурсів

Зловмисник може скомпрометувати сервер ресурсів, щоб отримати доступ до його ресурсів та інших ресурсів відповідного розгортання. Така компрометація може варіюватися від часткового доступу до системи, наприклад, файлів журналу серверу, до повного управління відповідним сервером.

Якщо зловмисник зможе взяти на себе повний контроль, включаючи доступ до оболонки, він зможе обійти всі контролі на місці та отримати доступ до ресурсів без контролю доступу. Вона також отримає доступ до маркерів доступу, які надсилаються до скомпрометованої системи і які потенційно є дійсними для доступу до інших серверів

ресурсів. Навіть якщо зловмисник «тільки» має доступ до лог-файлів або баз даних серверної системи, він може отримати доступ до дійсних маркерів доступу.

Запобігання порушенням сервера шляхом зміцнення та моніторингу серверних систем вважається стандартною операційною процедурою і, отже, виходить за рамки цього документа. Цей розділ буде зосереджено на впливі таких порушень на частини екосистеми, пов'язані з OAuth, що є відтворенням захоплених маркерів доступу на скомпрометованому сервері ресурсів та інших серверах ресурсів відповідного розгортання.

Наступні заходи повинні бути враховані виконавцями для того, щоб впоратися з відтворенням маркера доступу:

- Сервер ресурсів повинен розглядати маркери доступу, як і будь-які інші облікові дані. Вважається хорошою практикою не реєструвати їх, а не зберігати їх у вигляді звичайного тексту.
- Токени доступу обмеження відправника, як описано в розділі 3.7.1.2, не дозволять зловмисникам відтворювати маркери доступу на інших серверах ресурсів. Залежно від тяжкості проникнення, це також запобігає відтворенню компрометованої системи.
- Обмеження аудиторії, як описано у Розділі 3.7.1.3, може бути використано для запобігання відтворення захоплених маркерів доступу на інших серверах ресурсів.

2.5.8 Відкриті перенаправлення

Наступні атаки можуть виникати, коли сервер авторизації або клієнт має відкрите перенаправлення, тобто URL, який викликає HTTP перенаправлення на веб-сайт, керований зловмисником.

2.5.8.1 Сервер авторизації як відкрите перенаправлення

Зловмисники можуть спробувати використати довіру користувача до сервера авторизації (зокрема, його URL) для виконання фішингових атак.

Серверу авторизації рекомендовано не перенаправляти агента користувача автоматично у випадку недійсного поєднання client_id і redirect_uri. [1]

Однак, зловмисник може також використовувати правильно зареєстроване URI перенаправлення для виконання фішингових атак. Він може, наприклад, зареєструвати клієнта за допомогою реєстрації динамічного клієнта і навмисно відправити помилковий запит на авторизацію, наприклад, використовуючи недійсне значення області видимості, щоб викликати автоматичне перенаправлення агент користувача сервером авторизації на фішинговий сайт зловмисника.

Сервер авторизації повинен вживати запобіжних заходів, щоб запобігти цій загрозі. На основі своєї оцінки ризику сервер авторизації має вирішити, чи може він довіряти URI перенаправленню, чи необхідно тільки автоматично перенаправляти агента користувача, якщо він довіряє URI перенаправленню. Якщо ні, сервер авторизації має повідомити користувача про те, що він збирається перенаправити його на інший сайт, і покладатися на користувача для прийняття рішення, чи може просто повідомити користувача про те, що сталася помилка.

2.5.8.2 Клієнти як відкрите перенаправлення

Клієнт не повинен розкривати, які URL-адреси можна використовувати як відкриті перенаправлення. Зловмисники можуть використовувати відкритий переадресувач для створення URL-адрес, які, як здається, вказують на клієнта, що може змусити користувачів довіряти URL-адресі та перейти за нею в своєму веб-переглядачі. Інший

випадкок зловживання - створення URL-адрес, що вказують на клієнта, і використання їх для того, щоб видати себе за клієнта, при зв’язку з сервером авторизації.

Щоб запобігти відкритому перенаправленню, клієнти повинні відкривати таку функцію лише якщо цільові URL-адреси є в «білому списку», або якщо джерело запиту може бути автентифіковано.

Висновки до розділу 2

В даному розділі були розглянуті припущення щодо реалізації атак, архітектура протоколу OAuth 2.0. Також були розглянуті та проаналізовані можливі атак на протокол авторизації OAuth 2.0. З результатів аналізу можна сказати, що наведені вище атаки порушують безпеку протоколу OAuth 2.0.

Таблиця 2.1 – Перелік атак на протокол авторизації OAuth 2.0

Назва атаки	Порушення протоколу OAuth 2.0
Недостатня перевірка URI перенаправлення	Авторизація та автентифікація
Витік кодів або станів від клієнта або сервера авторизації через Referrer заголовки	Цілісність сеансу
Атаки через історію браузера	Цілісність сеансу
Підміна	Авторизація та автентифікація
Ін'єкція коду	Авторизація та автентифікація
Міжсайтова підробка запиту	Цілісність сеансу
Виток маркерів доступу на сервері ресурсів	Авторизація та автентифікація
Відкриті перенаправлення	Цілісність сеансу

3 ПОБУДОВА МОДЕЛІ БЕЗПЕЧНОЇ АВТОРИЗАЦІЇ

В цьому розділі буде проведено побудову та описання моделі у вигляді протоколу, який складається з наступних компонентів.

3.1 Який грант OAuth 2.0 використовувати

Розглянемо грант як метод отримання маркеру.

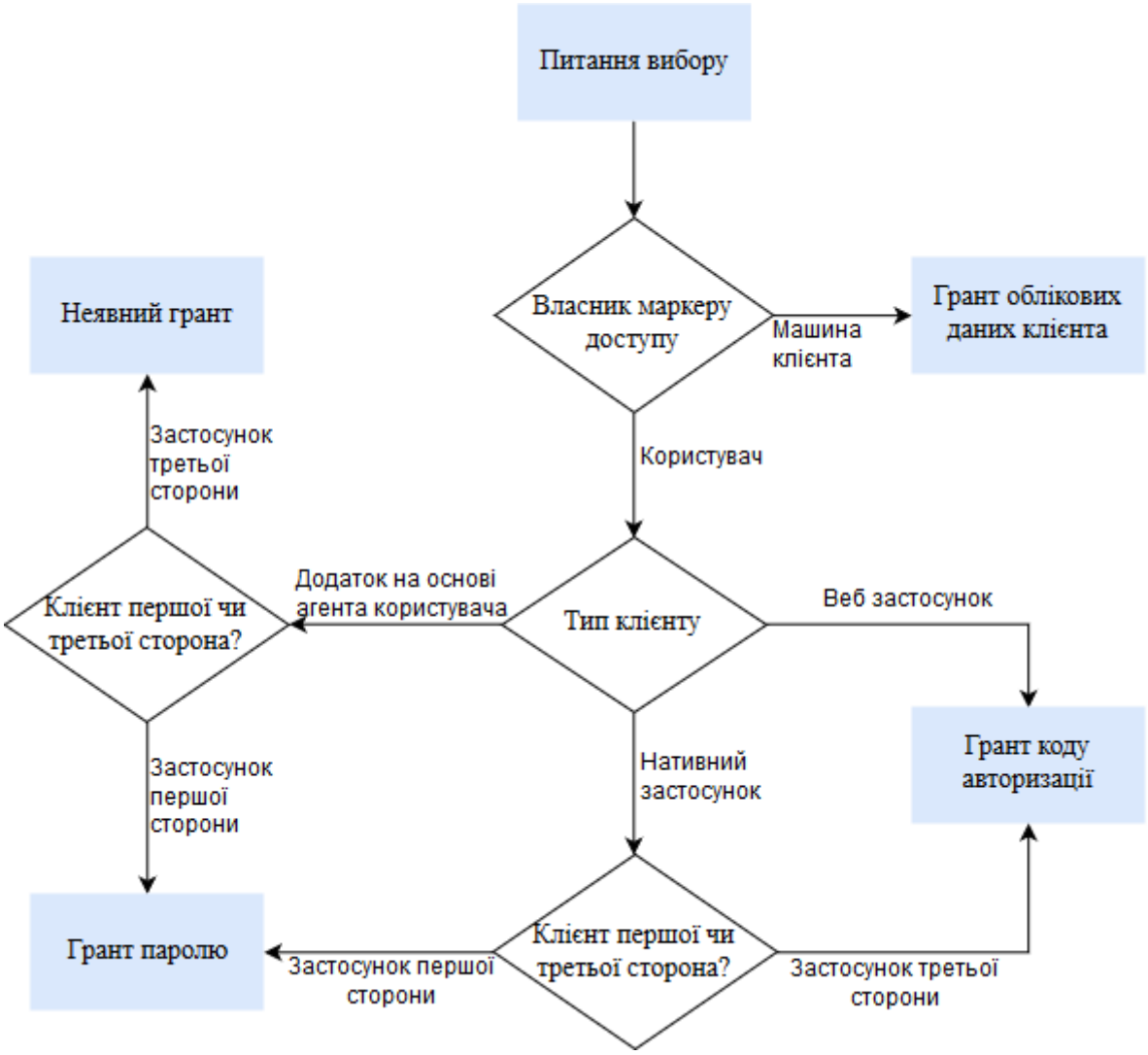

Рисунок 3.1 – Діаграма отримання гранту

1. Клієнт першої чи третьої сторони.
   Клієнт першої сторони - клієнт, якому ви довіряєте щоб оброблювати облікові дані кінцевого користувача. Клієнт третьої сторони - клієнт, якому ви не довіряєте оброблювати особові дані користувача.

2. Власник маркера доступу.
   Маркер доступу являє собою дозвіл, наданий клієнту для доступу до деяких захищених ресурсів. Якщо ви авторизуєте машину для доступу до ресурсів, і ви не вимагаєте дозволу користувача на доступ до цих ресурсів, рекомендується реалізувати грант облікових даних клієнта. Якщо для доступу до ресурсів потребується дозвіл користувача, необхідно визначити тип клієнта.

3. Тип клієнта.
   Те, який грант повинен використовувати клієнт залежить від того, чи може клієнт не розголошувати секрет.
   Якщо клієнт - це веб-застосунок, що має компонент на стороні сервера, тоді потрібно реалізувати грант коду авторизації.
   Якщо клієнт - це веб-застосунок, що повністю працює на стороні frontend (наприклад, односторінкові веб-застосунки), рекомендується запровадити грант паролю для клієнтів першої сторони, а для клієнтів третьої сторони - неявний грант.
   Якщо клієнт є нативним застосунком, наприклад, мобільний застосунок, потрібно грант паролю.
   Нативні застосунки третіх сторін повинні використовувати грант коду авторизації (через нативний, а не вбудований до застосунку, веб-переглядач).

Також, з одного боку, модель має включати ті типи грантів, які підтримують безпечне використання. З іншої, в межах цієї роботи ми розглядаємо використання протоколу OAuth 2.0 для веб застосунків.

Таким чином, для використання в цій роботі було обрано використання гранту коду авторизації.

Для захисту від атак витоку коду авторизації, важливо використовувати одноразові коди. Подібні дії також допомагають захиститись від атак з ін’єкцією коду авторизації, але для більш безпечної реалізації потрібно прив’язувати кожен код авторизації до конкретного екземпляру клієнту в межах однієї транзакції. Також, захист від атак ін’єкції коду може бути реалізований шляхом зв’язування кодів авторизації з параметрами стану.

При цьому, якщо код авторизації використовується більше ніж для одного запиту, сервер авторизації має відхилити запит, і скасувати всі видані маркери, що пов’язані з цим кодом авторизації.

Прикладом доброї практики є заміна маркерів оновлення:

Сервер авторизації може видати новий маркер оновлення, в такому випадку клієнт повинен видалити і замінити старий маркер новим маркером оновлення. Сервер авторизації має скасувати старий маркер оновлення після видачі клієнту нового маркера оновлення. Якщо видано новий маркер оновлення, область видимості маркера має бути ідентичною маркеру оновлення, включеному клієнтом у запиті.

3.2 Взаємна автентифікація клієнта TLS

Так як протокол OAuth 2.0 не має механізму захисту трафіку між сервером та клієнтом, вбудований в його реалізацію, розробники мають впроваджувати цей функціонал, інакше, без захисту трафіка, інформація авторизації, що передається, буде надсилатися у відкритому вигляді, отже у випадку перехвату трафіка, вона буде повністю доступна зловмиснику.

Для захисту трафіку часто використовується криптографічний протокол TLS. Використання цього протоколу також рекомендовано документацією OAuth 2.0. В той же час, впровадження TLS вимагає додаткових знань та зусиль від розробника, отримання сертифікату, налаштування серверу на підтримку та перевірку TLS сертифікатів.

Одним з популярних варіантів впровадження TLS є конфігурація, коли перед сервером застосунка знаходиться проксі сервер, що використовується для обробки вхідних з’єднань TLS, розшифровуючи TLS та передаючи незашифровані запити до сервера застосунку. Але такий також відкриває деякі кути для атаки.

У деяких ситуаціях проксі-сервер повинен передавати дані, пов'язані з безпекою, на інші сервери застосунків для подальшої обробки. Як приклад можна взяти відправку IP-адресу одержувача запиту, ідентифікатори зв'язування маркерів і підтверджені клієнтські сертифікати TLS.

Якщо проксі передає будь-який заголовок, надісланий ззовні, зловмисник може спробувати безпосередньо надіслати підроблені значення заголовка через проксі-сервер на сервер додатків, щоб уникнути контролю безпеки. Стандартна практика для проксі-серверів - приймати заголовки forwarded_for і додавати до них інформацію про походження вхідного запиту. Залежно від логіки, виконуваної на сервері застосунків, якщо зловмиснику відомі IP-адреси з білого списку серверу застосунка, він може може просто додати IP-адресу до заголовка і таким чином обійти перевірку білого списку IP.

Також, якщо зловмисник зможе увійти до внутрішньої мережі між проксі-сервером і сервером додатків, він також зможе обійти методи контролю безпеки.

Таким чином, при впровадженні TLS протоколу, розробник має звернути увагу на реалізацію перевірки автентичності сутностей, які беруть участь у зв’язку. Більш того, зв’язок між проксі сервером та сервером застосунку має бути захищений від ін’єкцій.

3.3 Маркер носія

Доступ до сервера ресурсів, наприклад, веб-API, з використанням маркерів є досить простою задачею. Маркери доступу в OAuth 2.0 зазвичай мають тип маркера носія, тобто клієнт просто повинен передавати маркер з кожним запитом. Заголовок HTTP Authorization є рекомендованим методом передачі маркерів.

```
GET /resource/v1 HTTP/1.1
Host: api.example.com
Authorization: Bearer oab3thieWohyai0eoxibaequ0Owae9oh
```

Сервер ресурсів повинен визначити, чи є маркер дійсним, та перевірити його термін дії. У випадку, якщо маркер дійсний і термін його дії не закінчився, сервермає продовжити обслуговування запиту. В іншому випадку сервер має повернути відповідну помилку в заголовку відповіді HTTP WWW-Authenticate.

Приклад помилки для недійсного маркера, або маркера термін дії якого закінчився:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="api.example.com",
                         error="invalid_token",
                         error_description="The access token is invalid or expired"
```

Якщо клієнт отримує помилку invalid_token для дійсного маркера доступу, це сигнал, що він повинен відправити запит на отримання нового маркера носія з сервера авторизації.

Треба мати на увазі те, що володіння маркером носія, за своєю природою, рівносильно доступу до серверу ресурсів. Тому маркери носія повинні бути захищені - по-перше, їх потрібно зберігати у безпечному сховищі на стороні клієнти, по-друге, для передачі маркерів носія використовувати описаний вище протокол TLS. Використання маркерів з коротким терміном дії рекомендується для зменшення ризику витоку маркерів.

3.4 Ключ підтвердження для обміну кодами (PKCE)

Публічні клієнти OAuth 2.0 піддаються до варіанту атаки перехоплення коду авторизації. У цій атаці зловмисник перехоплює код авторизації, що надіслали з кінцевої точки авторизації в межах частини зв'язку, не захищеного TLS, наприклад, під час зв’язку між застосунками в операційній системі клієнта.
Як тільки зловмисник отримав доступ до коду авторизації, він може використовувати його для отримання маркера доступу.

Для пом'якшення цієї атаки, потрібно використати динамічно створений криптографічний випадковий ключ, який називається «верифікатор коду». Для кожного запиту на авторизацію треба створити унікальний верифікатор коду, і його перетворене значення, зване "виклик коду", надсилати серверу авторизації для отримання коду авторизації. Отриманий код авторизації потім надсилається до кінцевої точки маркера за допомогою "верифікатора коду", і сервер має порівняти його з раніше отриманим кодом запиту, для підтвердження володіння клієнтом "верифікатора коду". Цей вид пом’якшення атаки базується на тому, що зловмисник не знає одноразового ключа, оскільки він надсилається через TLS і не може бути перехоплений.

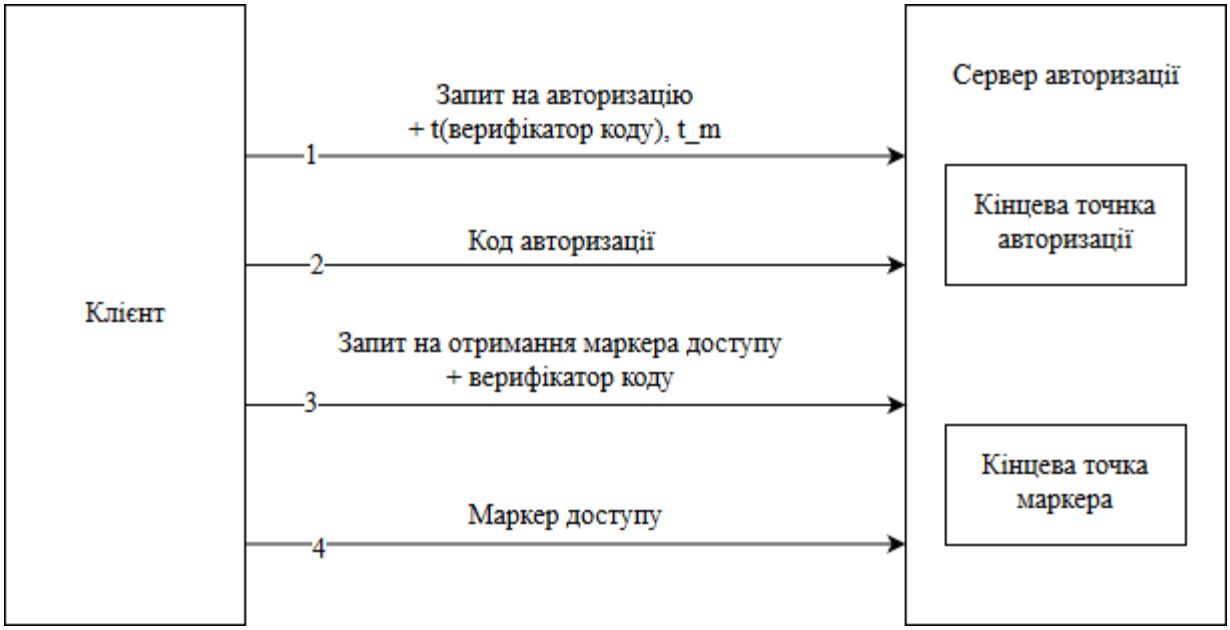

Рисунок 3.2 – Потік протоколу авторизації з PKCE

1. Клієнт створює та записує секрет, який називається «code_verifier», після цього отримує перетворену версію «t (code_verifier)» (називається «code_challenge»), яка надсилається в запиті авторизації OAuth 2.0 разом з методом трансформації «t_m».

2. Кінцева точка авторизації надсилає звичайний варіант відповіді, але зберігає перетворений варіант верифікатора коду(«code_challenge») і метод перетворення.

3. Клієнт потім відправляє код авторизації в запиті на отримання маркеру доступу, як звичайно, але додає до запиту секрет «code_verifier», створений на першому кроці.

4. Сервер авторизації перетворює «code_verifier» і порівнює його з «t (code_verifier)» з другого кроку. Доступ забороняється, якщо вони не рівні.

Таким чином, зловмисник, який перехоплює код авторизації на другому кроці, не може не може використати його для отримання маркера доступу, оскільки він не знає секрету «code_verifier».

3.5 Кодування HTTP запитів використовуючи HMAC

Можна покращити рівень безпеки моделі при використанні HMAC. Головна задумка використання HMAC в тому, що клієнт та сервер знають секретний ключ, але цей секретний ключ ніколи не надсилається «через дріт». Він використовується тільки в поєднанні з іншими фрагментами даних, які надсилаються. Таким чином, коли ми використовуємо наш секретний ключ і будь-які інші дані – скажімо, відкритий ключ, що ідентифікує користувача, поточну мітку часу або будь-що інше, що ми хочемо використовувати – і ми пропустимо ці дані через, наприклад, алгоритм SHA-1, ми можемо створити той самий хеш як на клієнті, так і на сервері.

Такий підхід дозволяє захиститися від пасивних атак – зловмисник не зможе використовувати «підслуханий» маркер доступу з обміну між клієнтом та сервером. До того ж, цей механізм допомагає захиститись від витоку маркер доступу при відправці через захищений канал на неправильний сервер ресурсів, якщо клієнт надав інформацію про сервер ресурсів, з яким він хоче взаємодіяти в запиті до сервера авторизації.

3.6 Автентифікація клієнтів

Найбільш безпечно для реалізації використовувати механізм автентифікації JWT – це механізм розширення, в якому твердження JWT буде використовуватися як облікові дані клієнта. Клієнт повинен включати твердження і пов’язану інформацію за допомогою наступних параметрів HTTP-запиту – client_assertion_type та client_assertion.

Твердження JWT повинно бути підписано з використанням симетричного алгоритму підпису або алгоритму асиметричної підпису. Якщо клієнт вибирає підписати JWT, використовуючи алгоритм симетричної підпису, такий як HS256, в якості криптографічного ключа слід використовувати секрет клієнта.

Якщо клієнт підписує JWT з використанням асиметричного алгоритму підпису, такого як RS256, клієнт повинен використовувати свій секрет як криптографічний ключ і повинен завантажити відповідний сертифікат відкритого ключа на платформі Akana, щоб перевірити підпис в Token API.

3.7 Параметр стану

Протоколи авторизації надає параметр стану, який дозволяє відновити попередній стан програми. Параметр стану зберігає деякий об’єкт стану, встановлений клієнтом у запиті авторизації, і робить його доступним для клієнта у відповідь.

Основна причина для використання параметра стану – пом’якшення CSRF атак. При використані параметра в кінцевій точці перенаправлення для пом’якшення CSRF атаки, значенні стану – унікальне значення, пов’язане з кожним запитом автентифікації, яке не зможе вгадати зловмисник. Це унікальне значення, яке дозволяє запобігти атаці, підтвердивши, що значення, отримане у відповіді, відповідає тому, який ви очікуєте (те, яке ви створили під час ініціювання запиту).

3.8 Реалізація безпечної моделі авторизації

Для реалізації моделі авторизації була обрана одна з найбільш популярних мов для написання веб-застосунків – Javascript [11]. Також, реалізація моделі в межах цієї роботи буде проводитись на платформі Cloud Datastore[12], зважаючи на високу масштабовність цієї бази даних. Зазвичай, не дуже здоровою практикою при

використанні криптографічних алгоритмів вважається написання таких алгоритмів самому. Рекомендується використовувати вже реалізовані бібліотеки та фреймворки, що містять потрібні криптографічні алгоритми. Тому, для цієї реалізації будуть використовуватись такі фреймворки:

- crypto [13]
- crypto.js [14]

Функція auth відповідає за представлення сторінки, на якій користувачі можуть авторизувати клієнтів для доступу до їхньої інформації:

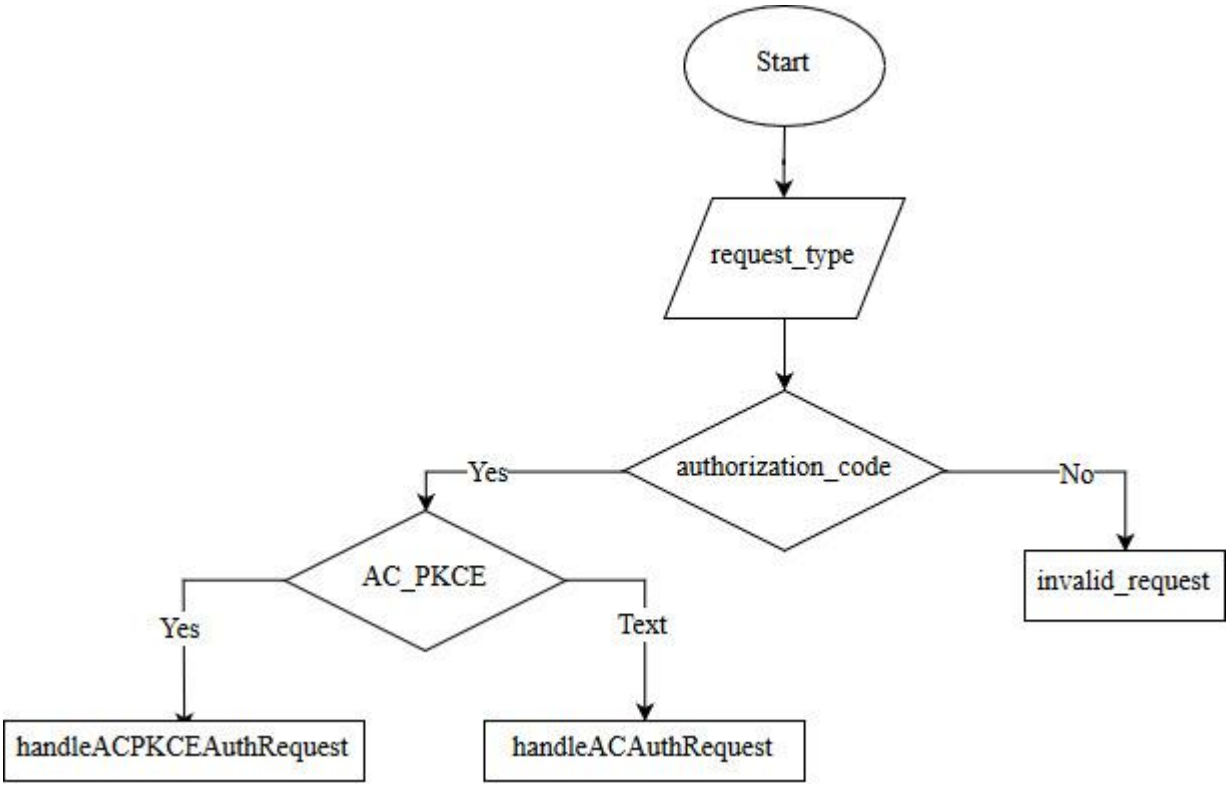

Рисунок 3.3 – Блок-схема функції auth

Запити з параметром response_type мають параметр code_challenge і code_challenge_method, що ініціюють потік коду авторизації (PKCE), обробляються ***функцією handleACPKCEAuthRequest***:

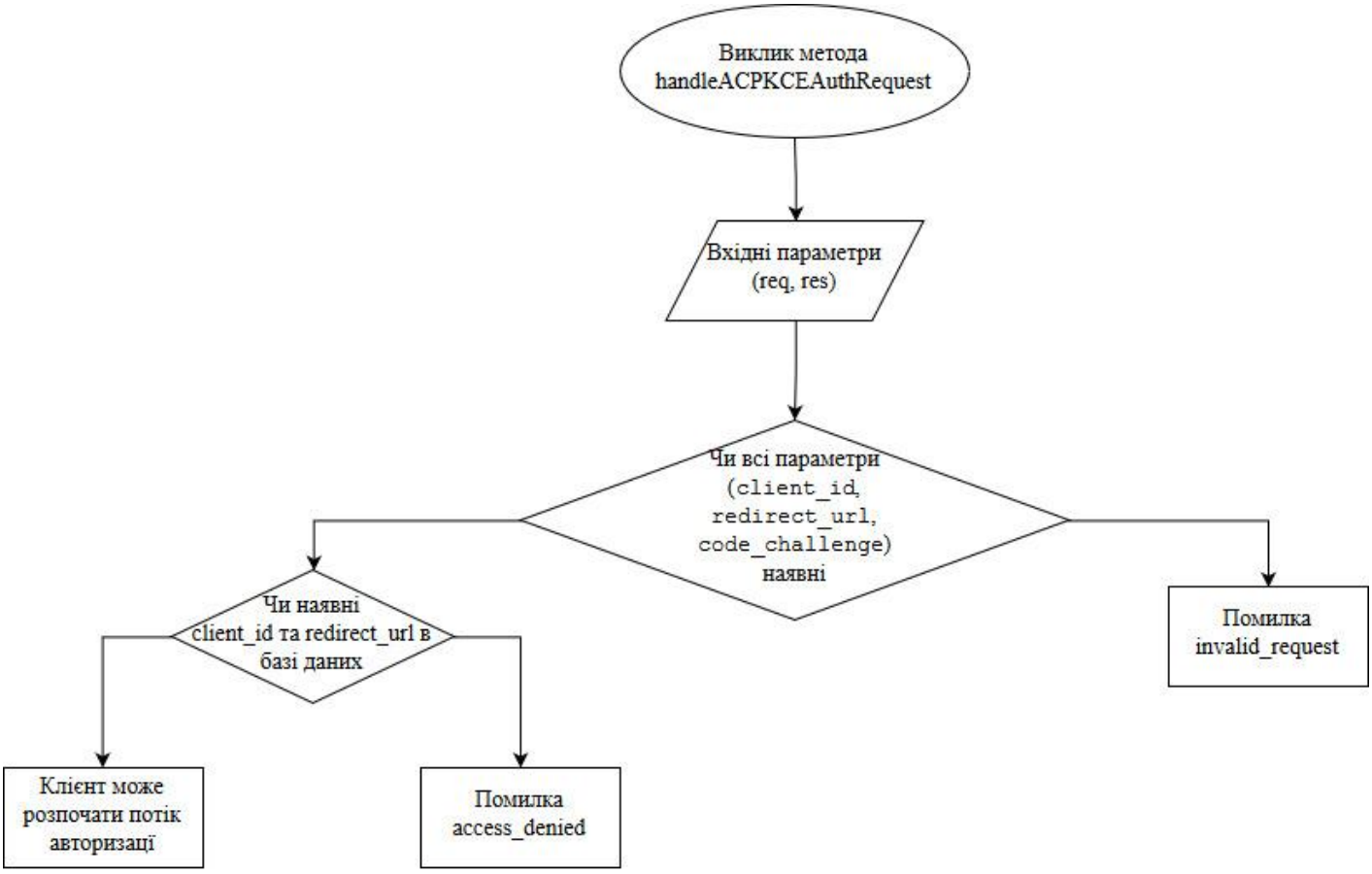

Рисунок 3.4 – Блок-схема функції handleACPKCEAuthRequest

Ця функція спочатку перевіряє, чи присутні всі необхідні параметри (client_id, redirect_url, code_challenge), а потім - що ідентифікатор клієнта та URL перенаправлення існують у базі даних, і клієнт має право ініціювати потік. Нарешті, видає сторінку де користувач може ввійти до свого облікового запису.

Також, в цьому зразку параметр code_challenge_method, хоча і необхідний, не використовується, оскільки передбачається, що всі виклики коду шифруються за допомогою SHA-256.

Так як операції механізму PKCE можуть бути ресурсоємкими, можлива реалізація потоку авторизації окремо.

Запити з параметром response_type задаються кодом, але не мають параметрів code_challenge і code_challenge_method, ініціюють потік коду авторизації і обробляються

функцією handleACAuthRequest, яка за своєю структурою дуже схожа на handleACPKCEAuthRequest.

Функція signin отримує облікові дані користувача і перенаправляє користувача назад до клієнта з кодом авторизації (або маркером доступу, у випадку неявного потоку):

Подібно до функції auth, signin використовує функції handleACPKCESigninRequest та handleACSigninRequest для обробки запитів з потоку коду авторизації з PKCE та потоку коду авторизації відповідно.

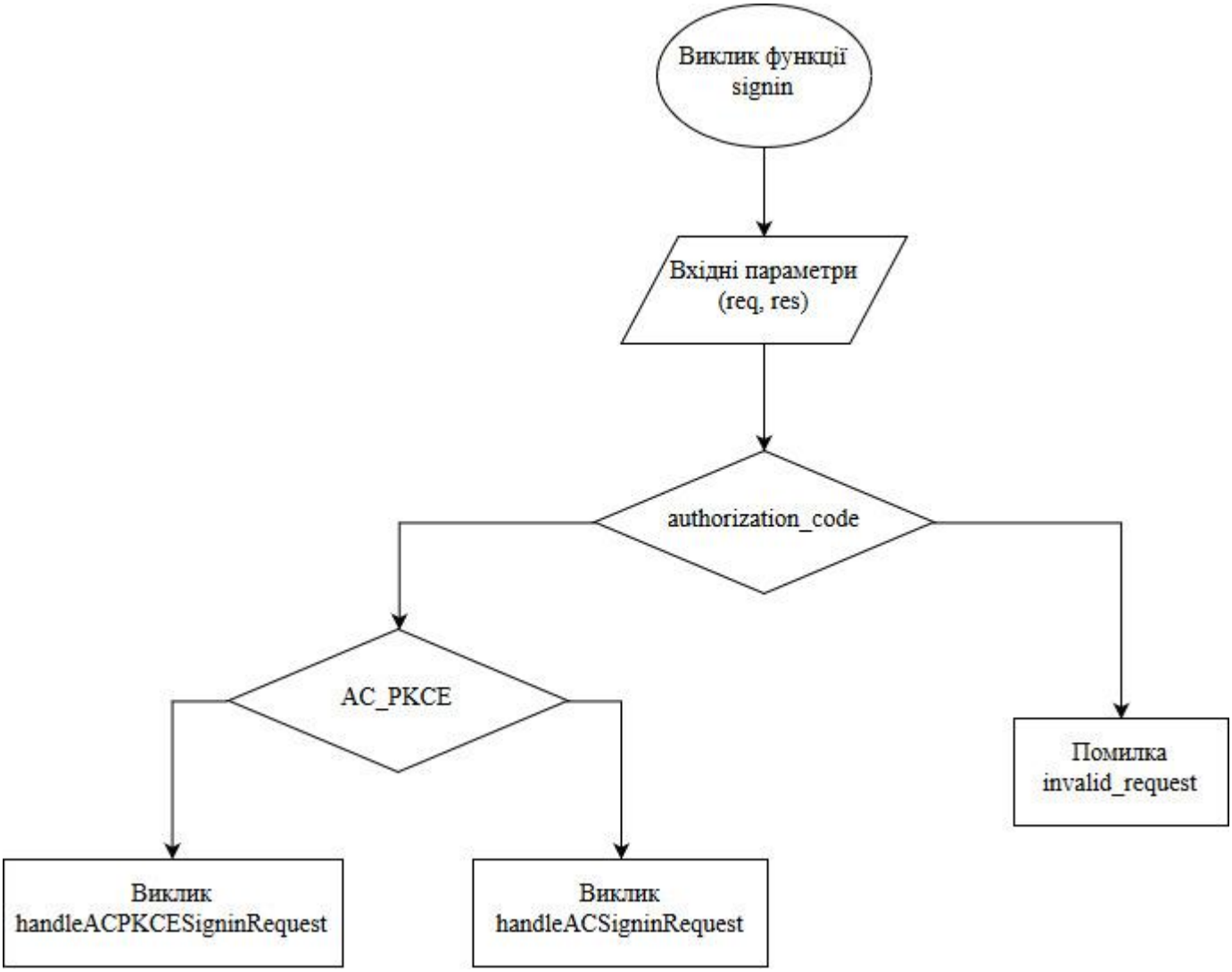

Рисунок 3.5 – Блок-схема функції signin

Функція token, як випливає з назви, відповідає за видачу маркерів.

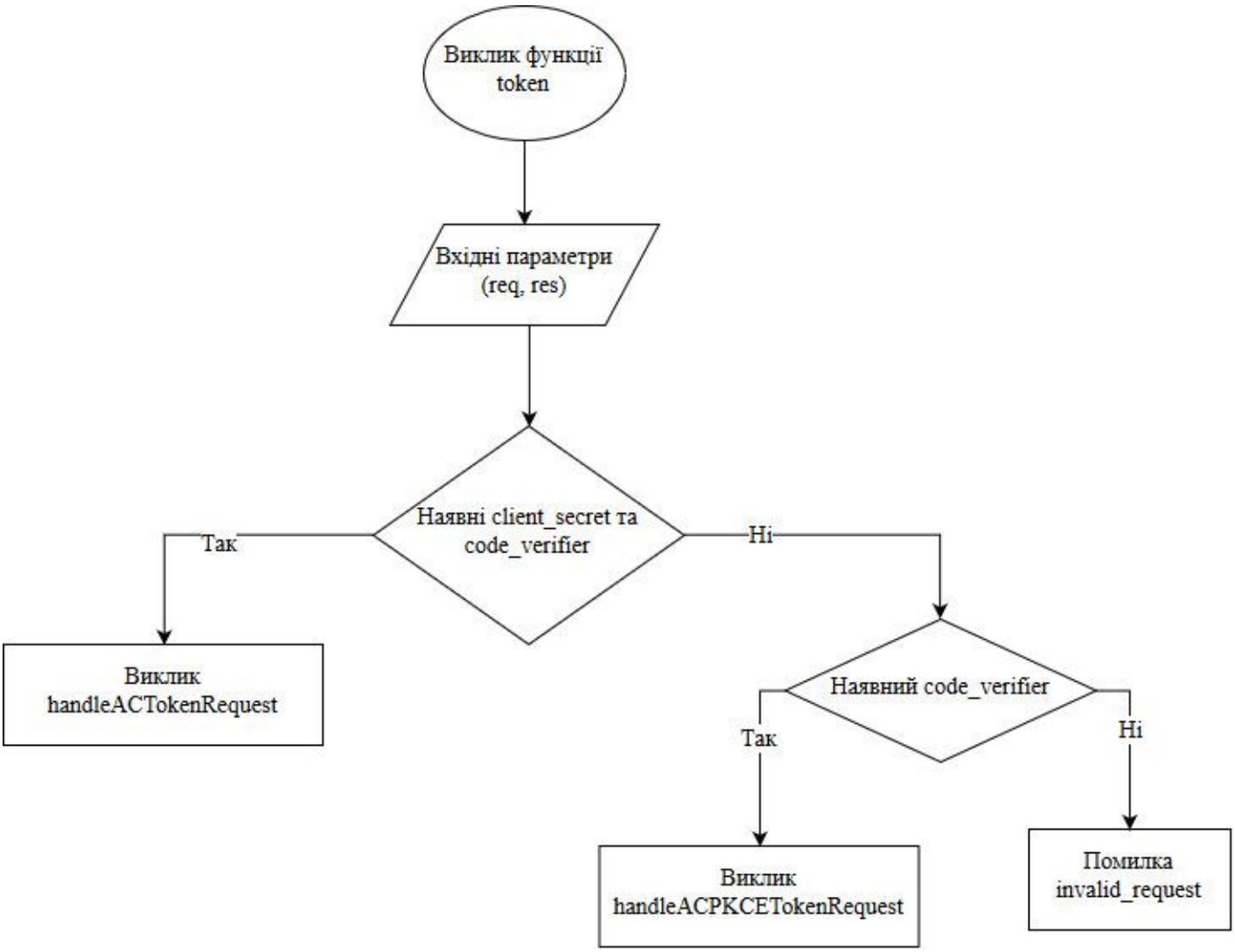

Рисунок 3.6 – Блок-схема функції token

Потік коду авторизації та потік коду авторизації PKCE використовують тип гранту authorization_code, причому перший обробляється ***handleACTokenRequest***, а другий – ***handleACPKCETokenRequest***:

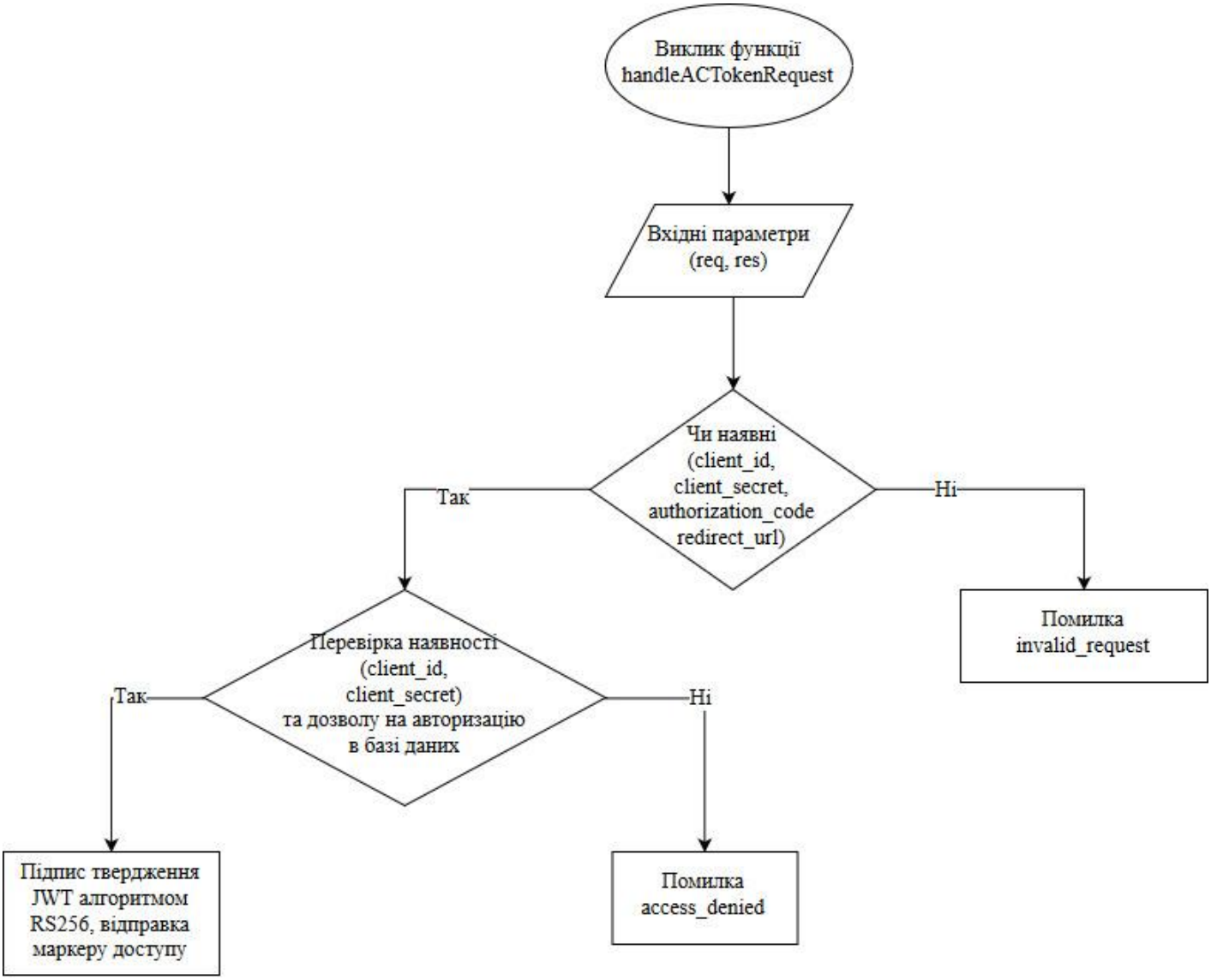

Рисунок 3.7 – Блок-схема функції handleACTokenRequest

Код авторизації перевіряється за допомогою **verifyAuthorizationCode**. При цьому, розшифровувати код авторизації не потрібно; перевірки значень client_id і redirect_url повинно бути достатньо.

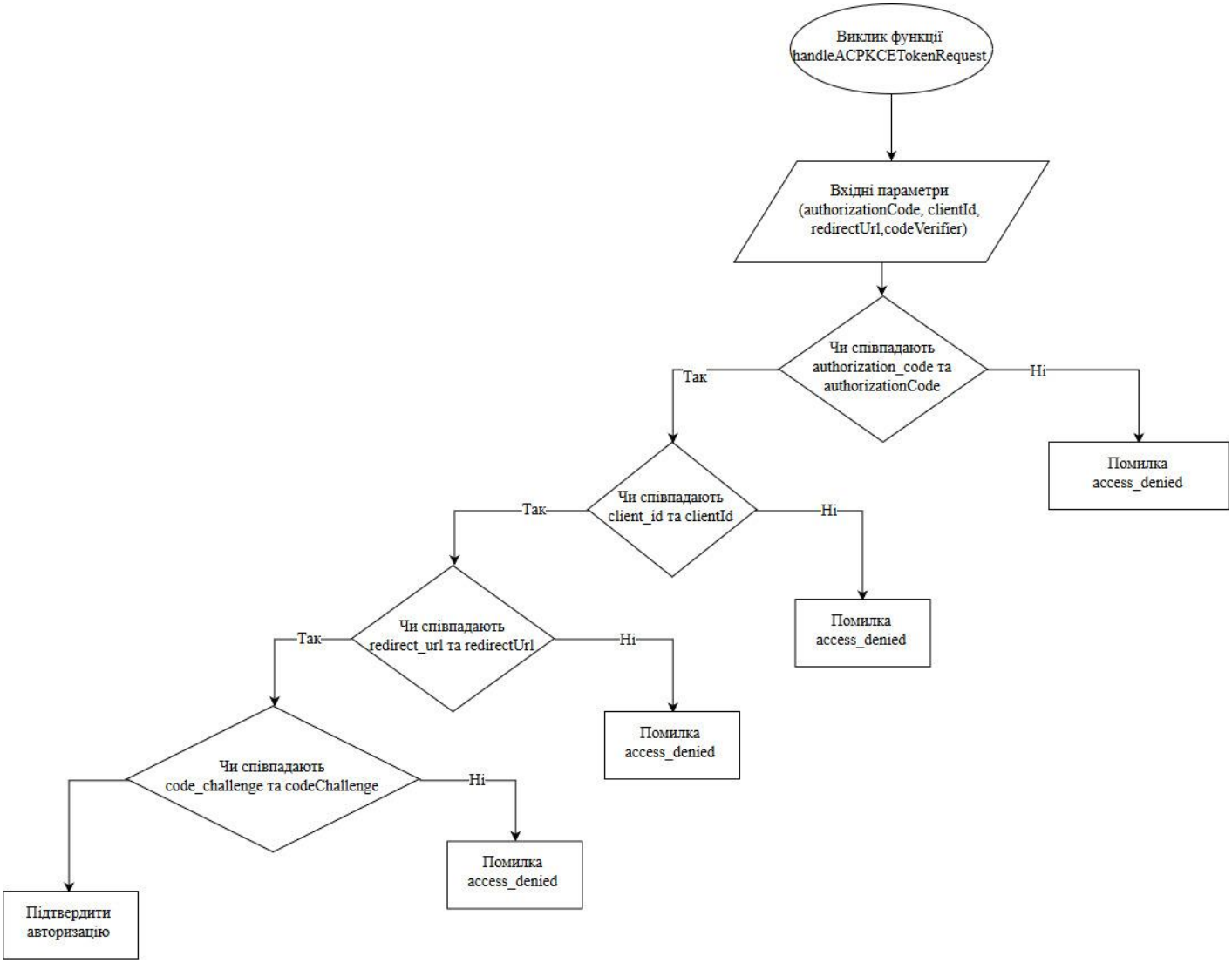

Рисунок 3.8 – Блок-схема функції verifyAuthorizationCode

Висновки до розділу 3

У цьому розділі було проведене дослідження та побудована модель авторизації на основі протоколу OAuth 2.0 для веб-застосунків. Дана модель заснована на аналізі можливих загроз та атак на протокол авторизації OAuth 2.0 та включає вимоги до реалізації протоколу.

Також була проведена реалізація моделі мовою Javascript, яка може застосовуватись та масштабуватись розробниками. Дана модель може бути розвинута при появі нових атак на протокол, або при появі нових загроз.

ВИСНОВКИ

У даній роботі було розглянуто питання безпечного способу авторизації на основі протоколу авторизації OAuth 2.0 для веб-застосунків, що використовують API.

Було проведення дослідження та аналіз існуючих протоколів, що можуть використовуватись для авторизації веб-застосунків, розглянуті алгоритми роботи цих протоколів, а також обраний найбільш безпечний для реалізації моделі авторизації для веб-застосунків.

Згідно поставленим завданням, були досліджені та проаналізовані атаки на протокол авторизації OAuth 2.0. Були виявлені та вивчені методи захисту від існуючих атак, та виявлено необхідність розробки більш ефективних і надійних аналогів, щоб забезпечити безпеку веб-застосунків та кінцевих користувачів.

При побудові моделі авторизації були оглянуті попередні дослідження моделей загроз, атак. Був проведений аналіз існуючих методів пом’якшення атак, що можуть допомогти розробниками створити більш захищені реалізації веб-застосунків.

Спираючись на технічну та професійну літературу були визначені загрози, сформовані основні вимоги до захисту веб-застосунків, на основі чого була сформована та запропонована модель авторизації, що має поліпшити рівень захищеності реалізацій протоколу OAuth 2.0.

Отриманий під час дослідження та побудови моделі досвід допоміг реалізувати основні вимоги запропонованої моделі авторизації. Отримана модель може використовуватись розробниками для реалізації авторизації веб-застосунків, а також може бути надалі масштабована та розширена при появі нових загроз та атак на протокол авторизації OAuth 2.0.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. The OAuth 2.0 Authorization Framework [Електронний ресурс]. – 2012. – Режим доступу до ресурсу: https://tools.ietf.org/html/rfc6749#section-2.1.

2. M. Jones. The OAuth 2.0 Authorization Framework: Bearer Token Usage [Електронний ресурс] / M. Jones, D. Hardt. – 2012. – Режим доступу до ресурсу: https://tools.ietf.org/html/rfc6750.

3. R. Fielding. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content [Електронний ресурс] / R. Fielding, J. Reschke. – 2014. – Режим доступу до ресурсу: https://tools.ietf.org/html/rfc7231.

4. Discovering concrete attacks on websiteauthorization by formal analysis [Електронний ресурс] / Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Sergio Maffeis // IOS Press. – 2014. – Режим доступу до ресурсу: https://www.doc.ic.ac.uk/~maffeis/papers/jcs14.pdf.

5. Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations [Електронний ресурс] / Tim Polk, Kerry McKay, Santosh Chokhani, Santosh Chokhani // NIST Special Publication. – 2014. – Режим доступу до ресурсу: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-52r1.pdf.

6. M. McGloin. OAuth 2.0 Threat Model and Security Considerations [Електронний ресурс] / M. McGloin, T. Lodderstedt, P. Hunt. – 2013. – Режим доступу до ресурсу: https://tools.ietf.org/html/rfc6819.

7. B. Campbell. OAuth 2.0 Form Post Response Mode [Електронний ресурс] / B. Campbell, M. Jones. – 2015. – Режим доступу до ресурсу: https://openid.net/specs/oauth-v2-form-post-response-mode-1_0.html.

8. Douglas Purdy. Platform Updates: Operation Developer Love [Електронний ресурс] / Douglas Purdy // 2018 – Режим доступу до ресурсу: https://developers.facebook.com/blog/post/552/.

9. N. Sakimura. The OAuth 2.0 Authorization Framework: JWT Secured Authorization Request [Електронний ресурс] / N. Sakimura, J. Bradley. – 2018. – Режим доступу до ресурсу: https://tools.ietf.org/html/draft-ietf-oauth-jwsreq-16.

10. OAuth 2.0 Token Binding [Електронний ресурс] / M. Jones, B. Campbell, J. Bradley, W. Denniss. – 2018. – Режим доступу до ресурсу: https://tools.ietf.org/html/draft-ietf-oauth-token-binding-06.

11. Developer Survey Results 2019 [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://insights.stackoverflow.com/survey/2019.

12. Cloud Datastore [Електронний ресурс] // Google – Режим доступу до ресурсу: https://cloud.google.com/datastore/.

13. Node.js v12.4.0 Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://nodejs.org/api/crypto.html.

14. Evan Vosberg. JavaScript library of crypto standards. [Електронний ресурс] / Evan Vosberg. – 2013. – Режим доступу до ресурсу: https://www.npmjs.com/package/crypto-js.

ДОДАТКИ

ДОДАТОК А
function.js

```
'use strict';

const appendQuery = require('append-query');
const crypto = require('crypto');
const Datastore = require('@google-cloud/datastore');
const fernet = require('fernet');
const fs = require('fs');
const jwt = require('jsonwebtoken');
const path = require('path');
const pug = require('pug');

const ISSUER = 'sample-issuer';
const JWT_LIFE_SPAN = 1800 * 1000;
const CODE_LIFE_SPAN = 600 * 1000;
/*
 * Linux and macOS users: prepare a base64-encoded 256-bit random seq. with
 *
 * dd if=/dev/urandom bs=32 count=1 2>/dev/null | openssl base64
 *
 * Python package, cryptography, also provides methods to generate keys.
 *
 * See https://cryptography.io/en/latest/fernet/ for more information.
 *
 */
const SECRET = new fernet.Secret(
  'YHD1m3rq3K-x6RxT1MtuGzvyLz4EWIJAEkRtBRycDHA=');

const datastore = Datastore();
const fernetToken = new fernet.Token({
  secret: SECRET
});

let privateKey;
fs.readFile('private.pem', 'utf8', function (error, data) {
  if (error) {
    console.log(`An error has occurred when reading the key file: ${error}`);
  } else {
    privateKey = data;
  }
});

function handleACPKCEAuthRequest (req, res) {
  if (req.query.client_id === undefined ||
      req.query.redirect_url === undefined ||
      req.query.code_challenge === undefined) {
    return res.status(400).send(JSON.stringify({
      'error': 'invalid_request',
      'error_description': 'Required parameters are missing in the request.'
    }));
  }

  const clientQuery = datastore
```

```
    .createQuery('client')
    .filter('client-id', '=', req.body.client_id)
    .filter('redirect-url', '=', req.body.redirect_url)
    .filter('acpkce-enabled', '=', true);

  datastore
    .runQuery(clientQuery)
    .then(result => {
      if (result[0].length === 0) {
        return Promise.reject(new Error('Invalid client/redirect URL.'));
      }
    })
    .then(() => {
      const html = pug.renderFile(path.join(__dirname, 'auth.pug'), {
        response_type: 'code',
        client_id: req.query.client_id,
        redirect_url: req.query.redirect_url,
        code_challenge: req.query.code_challenge
      });
      res.status(200).send(html);
    })
    .catch(error => {
      if (error.message === 'Invalid client/redirect URL.') {
        res.status(400).send(JSON.stringify({
          'error': 'access_denied',
          'error_description': error.message
        }));
      } else {
        throw error;
      }
    });
}

function handleACAuthRequest (req, res) {
  if (req.query.client_id === undefined ||
      req.query.redirect_url === undefined) {
    return res.status(400).send(JSON.stringify({
      'error': 'invalid_request',
      'error_description': 'Required parameters are missing in the request.'
    }));
  }

  const clientQuery = datastore
    .createQuery('client')
    .filter('client-id', '=', req.body.client_id)
    .filter('redirect-url', '=', req.body.redirect_url)
    .filter('ac-enabled', '=', true);

  datastore
    .runQuery(clientQuery)
    .then(result => {
      if (result[0].length === 0) {
        return Promise.reject(new Error('Invalid client/redirect URL.'));
      }
    })
    .then(() => {
      const html = pug.renderFile(path.join(__dirname, 'auth.pug'), {
        response_type: 'code',
```

```
        client_id: req.query.client_id,
        redirect_url: req.query.redirect_url,
        code_challenge: req.query.code_challenge
      });
      res.status(200).send(html);
    })
    .catch(error => {
      if (error.message === 'Invalid client/redirect URL.') {
        res.status(400).send(JSON.stringify({
          'error': 'access_denied',
          'error_description': error.message
        }));
      } else {
        throw error;
      }
    });
}

exports.auth = (req, res) => {
  console.log(req.query);
  switch (req.query.response_type) {
    case ('code'):
      if (req.query.code_challenge && req.query.code_challenge_method) {
        handleACPKCEAuthRequest(req, res);
      } else if (!req.query.code_challenge &&
                 !req.query.code_challenge_method) {
        handleACAuthRequest(req, res);
      } else {
        res.status(400).send(JSON.stringify({
          'error': 'invalid_request',
          'error_description': 'Required parameters are missing in the request.'
        }));
      }
      break;


    default:
      res.status(400).send(JSON.stringify({
        'error': 'invalid_request',
        'error_description': 'Grant type is invalid or missing.'
      }));
      break;
  }
};

function handleACPKCESigninRequest (req, res) {
  if (req.body.username === undefined ||
      req.body.password === undefined ||
      req.body.client_id === undefined ||
      req.body.redirect_url === undefined ||
      req.body.code_challenge === undefined) {
    return res.status(400).send(JSON.stringify({
      'error': 'invalid_request',
      'error_description': 'Required parameters are missing in the request.'
    }));
  }

  const userQuery = datastore
```

```
    .createQuery('user')
    .filter('username', '=', req.body.username)
    .filter('password', '=', req.body.password);

  const clientQuery = datastore
    .createQuery('client')
    .filter('client-id', '=', req.body.client_id)
    .filter('redirect-url', '=', req.body.redirect_url)
    .filter('acpkce-enabled', '=', true);

  datastore
    .runQuery(userQuery)
    .then(result => {
      if (result[0].length === 0) {
        return Promise.reject(new Error('Invalid user credentials.'));
      }
    })
    .then(() => {
      return datastore.runQuery(clientQuery);
    })
    .then(result => {
      if (result[0].length === 0) {
        return Promise.reject(new Error('Invalid client and/or redirect URL.'));
      }
    })
    .then(() => {
      const authorizationCode = fernetToken
        .encode(JSON.stringify({
          'client_id': req.body.client_id,
          'redirect_url': req.body.redirect_url
        }))
        .toString('base64')
        .replace(/\+/g, '-')
        .replace(/\//g, '_')
        .replace(/=/g, '');

      const exp = Date.now() + CODE_LIFE_SPAN;

      const codeKey = datastore.key(['authorization_code', authorizationCode]);
      const data = {
        'client_id': req.body.client_id,
        'redirect_url': req.body.redirect_url,
        'exp': exp,
        'code_challenge': req.body.code_challenge
      };

      return Promise.all([
        datastore.upsert({ key: codeKey, data: data }),
        Promise.resolve(authorizationCode)
      ]);
    })
    .then(results => {
      res.redirect(appendQuery(req.body.redirect_url, {
        authorization_code: results[1]
      }));
    });
}
```

```
function handleACSigninRequest (req, res) {
  if (req.body.username === undefined ||
      req.body.password === undefined ||
      req.body.client_id === undefined ||
      req.body.redirect_url === undefined) {
    return res.status(400).send(JSON.stringify({
      'error': 'invalid_request',
      'error_description': 'Required parameters are missing in the request.'
    }));
  }

  const userQuery = datastore
    .createQuery('user')
    .filter('username', '=', req.body.username)
    .filter('password', '=', req.body.password);

  const clientQuery = datastore
    .createQuery('client')
    .filter('client-id', '=', req.body.client_id)
    .filter('redirect-url', '=', req.body.redirect_url)
    .filter('ac-enabled', '=', true);

  datastore
    .runQuery(userQuery)
    .then(result => {
      if (result[0].length === 0) {
        return Promise.reject(new Error('Invalid user credentials.'));
      }
    })
    .then(() => {
      return datastore.runQuery(clientQuery);
    })
    .then(result => {
      if (result[0].length === 0) {
        return Promise.reject(new Error('Invalid client and/or redirect URL.'));
      }
    })
    .then(() => {
      const authorizationCode = fernetToken
        .encode(JSON.stringify({
          'client_id': req.body.client_id,
          'redirect_url': req.body.redirect_url
        }))
        .toString('base64')
        .replace(/\+/g, '-')
        .replace(/\//g, '_')
        .replace(/=/g, '');

      const exp = Date.now() + CODE_LIFE_SPAN;

      const key = datastore.key(['authorization_code', authorizationCode]);
      const data = {
        'client_id': req.body.client_id,
        'redirect_url': req.body.redirect_url,
        'exp': exp
      };

      return Promise.all([
```

```
        datastore.upsert({ key: key, data: data }),
        Promise.resolve(authorizationCode)
      ]);
    })
    .then(results => {
      res.redirect(appendQuery(req.body.redirect_url, {
        authorization_code: results[1]
      }));
    });
}


exports.signin = (req, res) => {
  switch (req.body.response_type) {
    case ('code'):
      if (!req.body.code_challenge) {
        handleACSigninRequest(req, res);
      } else {
        handleACPKCESigninRequest(req, res);
      }
      break;

    default:
      res.status(400).send(JSON.stringify({
        'error': 'invalid_request',
        'error_description': 'Grant type is invalid or missing.'
      }));
      break;
  }
};


function verifyAuthorizationCode (authorizationCode, clientId, redirectUrl,
                                  codeVerifier = undefined) {
  const transaction = datastore.transaction();
  const key = datastore.key(['authorization_code', authorizationCode]);

  return transaction
    .run()
    .then(() => transaction.get(key))
    .then(result => {
      const entry = result[0];
      if (entry === undefined) {
        return Promise.reject(new Error('Invalid authorization code.'));
      }

      if (entry.client_id !== clientId) {
        return Promise.reject(new Error('Client ID does not match the record.'));
      }

      if (entry.redirect_url !== redirectUrl) {
        return  Promise.reject(new  Error('Redirect  URL  does  not  match  the
record.'));
      }

      if (entry.exp <= Date.now()) {
        return Promise.reject(new Error('Authorization code expired.'));
      }
```

```
      if (codeVerifier !== undefined &&
          entry.code_challenge !== undefined) {
        let codeVerifierBuffer = Buffer.from(codeVerifier);
        let codeChallenge = crypto
                              .createHash('sha256')
                              .update(codeVerifierBuffer)
                              .digest()
                              .toString('base64')
                              .replace(/\+/g, '-')
                              .replace(/\//g, '_')
                              .replace(/=/g, '');
        if (codeChallenge !== entry.code_challenge) {
          return  Promise.reject(new  Error('Code  verifier  does  not  match  code
challenge.'));
        }
      } else if (codeVerifier === undefined ||
                 entry.code_challenge === undefined) {
        // Pass
      } else {
        return Promise.reject(
          new Error('Code challenge or code verifier does not exist.'));
      }

      return transaction.delete(key);
    })
    .then(() => transaction.commit())
    .catch(error => {
      transaction.rollback();
      throw error;
    });
}

function handleACTokenRequest (req, res) {
  if (req.body.client_id === undefined ||
      req.body.client_secret === undefined ||
      req.body.authorization_code === undefined ||
      req.body.redirect_url === undefined) {
    return res.status(400).send(JSON.stringify({
      'error': 'invalid_request',
      'error_description': 'Required parameters are missing in the request.'
    }));
  }

  const clientQuery = datastore
      .createQuery('client')
      .filter('client-id', '=', req.body.client_id)
      .filter('client-secret', '=', req.body.client_secret)
      .filter('ac-enabled', '=', true);

  datastore
    .runQuery(clientQuery)
    .then(clientQueryResult => {
      if (clientQueryResult[0].length === 0) {
        return Promise.reject(new Error('Invalid client credentials.'));
      }
    })
    .then(() => {
```

```
      return verifyAuthorizationCode(req.body.authorization_code,
        req.body.client_id, req.body.redirect_url);
    })
    .then(() => {
      const token = jwt.sign({}, privateKey, {
        algorithm: 'RS256',
        expiresIn: JWT_LIFE_SPAN,
        issuer: ISSUER
      });
      res.status(200).send(JSON.stringify({
        access_token: token,
        token_type: 'JWT',
        expires_in: JWT_LIFE_SPAN
      }));
    })
    .catch(error => {
      if (error.message === 'Invalid client credentials.' ||
          error.message === 'Invalid authorization code.' ||
          error.message === 'Client ID does not match the record.' ||
          error.message === 'Redirect URL does not match the record.' ||
          error.message === 'Authorization code expired.') {
        res.status(400).send(JSON.stringify({
          'error': 'access_denied',
          'error_description': error.message
        }));
      } else {
        throw error;
      }
    });
}

function handleACPKCETokenRequest (req, res) {
  if (req.body.client_id === undefined ||
      req.body.authorization_code === undefined ||
      req.body.redirect_url === undefined ||
      req.body.code_verifier === undefined) {
    return res.status(400).send(JSON.stringify({
      'error': 'invalid_request',
      'error_description': 'Required parameters are missing in the request.'
    }));
  }

  verifyAuthorizationCode(req.body.authorization_code, req.body.client_id,
    req.body.redirect_url, req.body.code_verifier)
    .then(() => {
      const token = jwt.sign({}, privateKey, {
        algorithm: 'RS256',
        expiresIn: JWT_LIFE_SPAN,
        issuer: ISSUER
      });
      res.status(200).send(JSON.stringify({
        access_token: token,
        token_type: 'JWT',
        expires_in: JWT_LIFE_SPAN
      }));
    })
    .catch(error => {
      if (error.message === 'Invalid authorization code.' ||
```

```
          error.message === 'Client ID does not match the record.' ||
          error.message === 'Redirect URL does not match the record.' ||
          error.message === 'Authorization code expired.' ||
          error.message === 'Code verifier does not match code challenge.') {
        res.status(400).send(JSON.stringify({
          'error': 'access_denied',
          'error_description': error.message
        }));
      } else if (error.msg === 'Code challenge does not exist.') {
        res.status(400).send(JSON.stringify({
          'error': 'invalid_request',
          'error_description': error.message
        }));
      } else {
        throw error;
      }
    });
}

exports.token = (req, res) => {
  switch (req.body.grant_type) {

    case 'authorization_code':
      if (req.body.client_secret && !req.body.code_verifier) {
        handleACTokenRequest(req, res);
        break;
      }
      if (req.body.code_verifier) {
        handleACPKCETokenRequest(req, res);
        break;
      }
      res.status(400).send(JSON.stringify({
        'error': 'invalid_request',
        'error_description': 'Client secret and code verifier are exclusive' +
                             'to each other.'
      }));
      break;

    default:
      res.status(400).send(JSON.stringify({
        'error': 'invalid_request',
        'error_description': 'Grant type is invalid or missing.'
      }));
      break;
  }
};

var app = (function($){

    var baseURL = 'http://localhost/development/demos/hmac-authentication/api';
    var apiSecretKey = 'ABC123';

    var init = function(){
        $('.loggedIn').hide();
        $('#login').on('click', function(e){
            e.preventDefault();
            login();
        });
```

```
        $('#sendRequest').on('click', function(e){
            e.preventDefault();
            testRequest();
        });
    };

    var login = function() {

        var u = $('#username').val();
        var p = $('#password').val();

        $.ajax({
            type: "POST",
            url: baseURL + "/login",
            contentType: "application/json; charset=utf-8",
            dataType: "json",
            data: JSON.stringify({username: u, password: p}),
            success: function (data) {
                $('.loggedOut').hide();
                $('.loggedIn').show();
                $('.loggedIn .name').text("Hello, " + readCookie('PUBLIC-KEY') + "
");
            },
            error: function (errorMessage) {
                alert('Error logging in');
            }
        });
    };

    var testRequest = function() {
        var data = { test: 'test'}
        var timestamp = getMicrotime(true).toString();
        $.ajax({
            type: "GET",
            url: baseURL + "/test",
            contentType: "application/json; charset=utf-8",
            dataType: "json",
            beforeSend: function (request) {
                request.setRequestHeader('X-MICROTIME', timestamp);
                request.setRequestHeader('X-HASH',     getHMAC(readCookie('PUBLIC-
KEY'), timestamp));
            },
            data: JSON.stringify(data),
            success: function (data) {
                alert(data.message);
            },
            error: function (errorMessage) {
               if(errorMessage.status == 401)
                   alert('Access denied');
            }
        });
    };

    var readCookie =  function (name) {
        var nameEQ = name + "=";
        var ca = document.cookie.split(';');
        for(var i=0;i < ca.length;i++) {
```

```
            var c = ca[i];
            while (c.charAt(0)==' ') c = c.substring(1,c.length);
            if (c.indexOf(nameEQ) == 0) return c.substring(nameEQ.length,c.length);
        }
        return null;
    };

    var getHMAC = function(key, timestamp) {
        var hash = CryptoJS.HmacSHA1(key+timestamp, apiSecretKey);
        return hash.toString();
    };

    var getMicrotime = function (get_as_float) {

      var now = new Date().getTime() / 1000;
      var s = parseInt(now, 10);

      return (get_as_float) ? now : (Math.round((now - s) * 1000) / 1000) + ' ' +
s;
    };

    return {
        init:init
    };

})(jQuery);
```

ДОДАТОК Б

auth.pug

```
doctype html

html(lang="en")

head

title Granting Access

body

h3 #{client_id} is requesting access to your account.

p.

Sign in to grant #{client_id} access.

form(action='https://us-central1-testbed-195403.cloudfunctions.net/signin',
method='POST')

p.

Username

input(type='text' name='username')

p.

Password

input(type='password' name='password')

input(type='hidden' name='client_id' value=client_id)

input(type='hidden' name='redirect_url' value=redirect_url)

input(type='hidden' name='response_type' value=response_type)

input(type='submit' name='submit')

if code_challenge

input(type='hidden' name='code_challenge' value=code_challenge)

      input(type='hidden' name='code_challenge_method' value=code_challenge_method)
```