## Лабораторная работа 8. Аутоинтефикация и авторизация

Реализовать аутоинтефикацию, используя основные механизмы протоколов аутентификации.

1. **SASL**

SASL (англ. Simple Authentication and Security Layer — простой уровень аутентификации и безопасности) — это фреймворк (каркас) для предоставления аутентификации и защиты данных в протоколах на основе соединений. Он разделяет механизмы аутентификации от прикладных протоколов, в теории позволяя любому механизму аутентификации, поддерживающему SASL, быть использованным в любых прикладных протоколах, которые используют SASL. Фреймворк также предоставляет слой защиты данных. Для использования SASL протокол включает команду для идентификации и аутентификации пользователя на сервере и для опциональной защиты переговоров последующей интерактивности протокола. Если это используется в переговорах, то слой безопасности вставляется между протоколом и соединением.

В 1997 Джон Гардинер Майерс (John Gardiner Myers) написал изначальную спецификацию SASL (RFC 2222) при университете Карнеги-Меллона (Carnegie Mellon University). В 2006 году этот документ утратил силу после введения RFC 4422, под редакцией Алексея Мельникова (Alexey Melnikov) и Курта Зейлинга (Kurt Zeilenga).

Механизмы SASL реализуют серию запросов и ответов. Определенные SASL механизмы включают:

«EXTERNAL», используется, когда аутентификация отделена от передачи данных (например, когда протоколы уже используют IPsec или TLS);

«ANONYMOUS», для аутентификации гостевого доступа (RFC 4505);

«PLAIN», простой механизм передачи паролей открытым текстом. PLAIN является заменой устаревшему LOGIN ;

«OTP», механизм одноразовых паролей. OTP заменяет устаревший механизм SKEY;

«SKEY», система одноразовых паролей (устаревший);

«CRAM-MD5»;

«DIGEST-MD5»;

«NTLM»;

«GSSAPI»;

GateKeeper (& GateKeeperPassport), разработана Microsoft для MSN Chat;

«KERBEROS IV» (устаревший).

Семья механизмов GS2 поддерживает произвольные GSSAPI механизмы в SASL. Это сейчас стандартизовано в RFC 5801.

#### 2 HTTP аутентификация и SASL(Simple Authentication and Security Layer) аутентификация

*Basic* — базовая аутентификация, при которой имя пользователя и пароль передаются в заголовках *http-пакетов*. Пароль при этом не шифруется и присутствует в чистом виде в кодировке *base64*. Для данного типа аутентификации использование *SSL* является обязательным.

*Digest* — дайджест-аутентификация, при которой пароль пользователя передается в хешированном виде. По уровню конфиденциальности паролей этот тип мало чем отличается от предыдущего, так как атакующему все равно, действительно ли это настоящий пароль или только *хеш* от него: перехватив удостоверение, он все равно получает доступ к конечной точке. Для данного типа аутентификации использование *SSL* является обязательным. Но в данном случае можно перехватывать только текущую сессию, для другой сессии данный хэш не поможет, так как сервер посылает случайную строку, которая хэшируется с логином и паролем. Если говорить более точно, то можно привести в пример CRAM-MD5 или DIGEST-MD5. Напомним, что хэш MD5 является «однозначным» преобразованием строки в псевдослучайную строку, при этом обратного преобразования не существует, хотя в большинстве случаев это уже не так и MD5 не столь надежен, лучше использовать другие хэши, применяемые, например, в технологии блокчейн (SHA256 и другие). Первый протокол предполагает отправку случайной строки сервером, после чего эта строка хэшируется паролем пользователя, не будем вдаваться в подробности того, как преобразуется эта строка предварительно, но в итоге сервер должен таким же образом хэшировать отправленную им самим строку с паролем пользователя и соответственно его знать, очевидно, здесь можно перехватить сессию и при этом пароль хранится в открытом виде на сервере. Digest предполагает отправку клиенту случайной строки, но предварительно клиент хэширует вместе пароль и логин пользователя, затем полученный хэш хэширует вместе со случайной строкой сервера, затем все это вместе хэширует со своей случайной строкой, отправляет серверу полученный хэш и свою случайную строку в открытом виде, сервер проводит ту же операцию вместе со строкой полученной от клиента, при этом он может хранить хэш логина пароля, а не открытый логин пароль. Перехват от клиента хэша позволит перехватить сессию, типичная атака MITM (man in the middle, человек по середине). Все это выглядит на стороне клиента так H(H(login, pass)+server\_string)+client\_string. Здесь приведен пример, не учитывающий различные параметры протоколов, которые так же входят в хэш, например, realm, но пример приведен для понимания концепции.

*Integrated* — интегрированная аутентификация, при которой клиент и сервер обмениваются сообщениями для выяснения подлинности друг друга с помощью протоколов *NTLM* или *Kerberos*.

Этот тип аутентификации защищен от перехвата удостоверений пользователей, поэтому для него не требуется протокол *SSL*. Здесь используются доверенные центры распределения ключей.

Только при использовании данного типа аутентификации можно работать по схеме *http*, во всех остальных случаях необходимо использовать схему *https*.

Параметры требующиеся для Basic и Digest аутентификации передаются в заголовках сервера.

Например,

GET /images/image1.jpg HTTP/1.1

Authorization: Basic Zm9vOmJhcg==

Здесь в base64 передается логин и пароль пользователя, foo: bar.

Пример с Digest будет посложнее. Например, вы обратились к ресурсу и вам вернулося следующий ответ.

HTTP/1.0 401 Unauthorized

WWW-Authenticate: Digest realm="my@mу.com",

qop="auth,auth-int",

nonce="xxxxxxxxxxxxxxxxxxxxxxxxxxxxx",

opaque="yyyyyyyyyyyyyyyyyyyyyyyyyyy"

В ответ вы уже пытаетесь пройти процедуру аутентификации:

GET /dir/index.html HTTP/1.0

Host: localhost

Authorization: Digest username="my\_user",

realm="my@my.com",

nonce=" xxxxxxxxxxxxxxxxxxxxxxxxxxxxx",

uri="/dir/index.html",

qop=auth,

nc=00000001,

cnonce="0a4f113b",

response="calculated by hash function response",

opaque="yyyyyyyyyyyyyyyyyyyyyyyyyyy"

Вычисление ответа response осуществляется следующим образом.

HA1 = MD5("my\_user:my@my.com:password")

HA2 = MD5("GET:/dir/index.html")

Response = MD5( "HA1:

cmFuZG9tbHlnZW5lcmF0ZWRub25jZQ:

nc:cnonce:auth:

HA2" )

#### 3 Cookie-based авторизация

Cookie-based авторизация - это форма авторизации, которая использует куки или небольшие данные, хранящиеся в браузере пользователя, чтобы аутентифицировать его и держать его в системе. При входе в систему на компьютере пользователя создается куки и отправляется на сервер. Когда пользователь посещает сайт в будущем, сервер будет искать куки и, если она существует, позволит пользователю получить доступ к сайту, не выходя из системы.

Cookie-based авторизация работает следующим образом: при аутентификации на основе cookies уникальный идентификатор (файл cookie) создается на стороне сервера и отправляется в браузер. Этот файл cookie содержит уникальный идентификатор сеанса, который сохраняется на компьютере пользователя и используется для проверки подлинности при каждом запросе к серверу. Когда пользователь входит на сайт, сервер отправляет ему файл cookie, который содержит уникальный идентификатор сеанса. Этот идентификатор сохраняется на компьютере пользователя и используется для проверки подлинности при каждом запросе к серверу. Пользователь может оставаться авторизованным до тех пор, пока не произойдет выход из системы или пока файл cookie не будет удален.

Для реализации cookie-based авторизации на Python можно использовать библиотеку Requests. Для этого нужно отправить POST-запрос на страницу входа, передав в теле запроса учетные данные пользователя (логин и пароль). Затем необходимо сохранить полученный файл cookie и использовать его для последующих запросов к сайту. Пример кода для аутентификации на сайте с помощью библиотеки Requests:

import requests

# Адрес страницы входа

login\_url = 'https://example.com/login'

# Учетные данные пользователя

payload = {

'username': 'user',

'password': 'pass'

}

# Отправляем POST-запрос на страницу входа

session = requests.Session()

response = session.post(login\_url, data=payload)

# Сохраняем файл cookie

cookie = session.cookies.get\_dict()

# Используем файл cookie для последующих запросов к сайту

response = session.get('https://example.com/profile', cookies=cookie)

#### 4 JWT токен авторизация

JWT (JSON Web Token) - это открытый стандарт для создания токенов доступа, основанный на формате JSON. Он используется для передачи информации между клиентом и сервером в зашифрованном виде. JWT состоит из трех частей: заголовка, полезной нагрузки и подписи. Заголовок содержит информацию о типе токена и используемом алгоритме шифрования. Полезная нагрузка содержит данные, которые нужно передать между клиентом и сервером. Подпись гарантирует целостность данных.

Примеры использования JWT-токенов включают в себя авторизацию пользователей на сайтах, API-интерфейсы и мобильные приложения.

При работе с JWT-токенами происходят следующие шаги:

1. Пользователь отправляет запрос на сервер для авторизации.

2. Сервер проверяет учетные данные пользователя и создает JWT-токен.

3. Сервер отправляет JWT-токен обратно пользователю.

4. Пользователь сохраняет JWT-токен в локальном хранилище (например, в localStorage).

5. При каждом последующем запросе к серверу пользователь отправляет JWT-токен

Для создания JWT-токена в Python можно использовать библиотеку PyJWT

Вот пример кода, который создает JWT-токен:

import jwt

# Определение заголовка и полезной нагрузки токена

header = {'alg': 'HS256'}

payload = {'sub': '1234567890', 'name': 'John Doe', 'iat': 1516239022}

# Определение секретного ключа для подписи токена

secret\_key = 'mysecretkey'

# Создание JWT-токена с помощью PyJWT

jwt\_token = jwt.encode(payload, secret\_key, algorithm='HS256', headers=header)

print('JWT Token:', jwt\_token)

В этом примере мы определяем заголовок и полезную нагрузку токена в виде словарей Python. Затем мы определяем секретный ключ, который будет использоваться для подписи токена. Мы используем функцию jwt.encode() из библиотеки PyJWT для создания JWT-токена. Функция принимает на вход полезную нагрузку, секретный ключ и алгоритм хэширования (в данном случае HS256). Мы также передаем заголовок токена через параметр headers.После выполнения кода выше будет напечатано значение JWT-токена.Обратите внимание, что при создании JWT-токенов необходимо использовать безопасные методы генерации случайных чисел для генерации секретных ключей.

5. HTTP Authorization

Заголовок HTTP-запроса Authorization используется для передачи учетных данных, которые аутентифицируют пользовательский агент на сервере.

Этот заголовок может использоваться для различных схем аутентификации, таких как Basic и Bearer.

В заголовке Authorization обычно указывается тип схемы аутентификации (например, Basic или Bearer) и учетные данные пользователя (например, имя пользователя и пароль или JWT-токен).

Например, в случае с JWT-токеном заголовок Authorization может выглядеть следующим образом:

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0I

#### 6 Авторизация Auth 2.0

Является открытым фреймворком (протоколом) авторизации, позволяющим получить сторонним приложениям ограниченный доступ к ресурсам HTTP-сервиса, позволяет предоставить права на использование некоторого ресурса (например, API какого-либо сервиса). При этом в общем случае нельзя определить, кто в настоящий момент пользуется правами.

Схематично сценарий протокола OAuth 2.0 представлен на рисунке. Он описывает взаимодействие между упомянутыми выше четырьмя сторонамии включает следующие шаги.

1. Клиент запрашивает авторизацию у владельца ресурса. Запрос авторизации может быть направлен владельцу ресурса напрямую, как показано на рисунке, или косвенно через сервер авторизации. Предпочтительным является второй вариант.

2) Клиент получает разрешение на доступ (grant), структуру данных, представляющую авторизацию владельца ресурса, выраженную с использованием одного из четырех типов разрешений: код авторизации (authorization code), неявное разрешение (implicit), пароль владельца ресурса (resource owner password credentials) и учетные данные клиента (client credentials), так же есть еще Device authorization для доступа устройств (если нет веб-браузера, RFC 8628) . Тип разрешения на доступ зависит от метода, используемого клиентом для запроса авторизации, и типов разрешений, поддерживаемых сервером авторизации. Типы разрешений, поддерживаемые сервером авторизации, определяются при его разработке, исходя из его прикладных целей и задач. Настоящий стандарт регламентирует использование в качестве типа разрешения код авторизации.

3) Клиент запрашивает токен доступа посредством аутентификации на сервере авторизации и предоставления разрешения на доступ.

4) Сервер авторизации аутентифицирует клиента, проверяет разрешение на доступ и, если оно действительно, выдает токен доступа.

5) Клиент запрашивает защищенный ресурс на сервере ресурсов и аутентифицируется, представляя токен доступа.

6) Сервер ресурсов проверяет токен доступа и, если он действителен, обслуживает запрос.

Клиент (имеется в виду приложение получающее доступ, может быть клиент-серверным)

Владелец ресурса

1) Запрос авторизации

2) Разрешение на доступ (грант авторизации)

Сервер авторизации

3) Грант авторизации

4) Токен авторизации

Сервер ресурсов

5) Токен авторизации

6) Разрешение на доступ от сервера ресурсов

Стандарт OAuth 2.0 определяет следующие четыре роли:

• владелец ресурса — сущность, обладающая правом на выдачу доступа к защищенным ресурсам. В случае, если владелец является человеком, его называют конечным пользователем;

• сервер ресурсов — сервер, содержащий защищаемые ресурсы и обладающий возможностью получения и формирования ответа на запросы к защищаемым ресурсам посредством использования маркера доступа;

• клиент — приложение, осуществляющее доступ к защищенным ресурсам от имени Владельца. Термин "клиент" явно не определяет какое-либо конкретное исполнение (будь то сервер, персональный компьютер или мобильное приложение); Приложением может являться браузер, толстый клиент или мобильное приложение. Клиент может быть public и confedential. Public не может безопасно хранить свои учётные данные. Этот клиент работает на устройстве владельца ресурса, например, это браузерные или мобильные приложения. Confedential может безопасно хранить свои учетные данные, например бэкенд приложения.

• сервер авторизации — сервер, осуществляющий выпуск маркеров доступа для клиентских приложений после успешной аутентификации и авторизации Владельца ресурсов.

Протокол OAuth обладает возможностью аутентификации не только Пользователя, но и клиентского приложения, осуществляющего доступ к ресурсам.

Рассмотрим пять способов получения доступа (grant) в auth 2.0, начиная с самого простого.

Client credentials grant flow

Предполагает самый простой способ получения прав доступа к ресурсам. Клиент отправляет на сервер авторизации client id и client secret, на что возвращается access token с которым клиент может обратиться к серверу ресурсов. Данный способ используется для доступа к собственным ресурсам или предоставление доступа к серверу ресурсов согласованному с сервером авторизации, когда клиент совпадает с владельцем ресурсов. Допускается только для защищенных клиентов (confedential, клиент, который может безопасно хранить свои учётные данные, к такому типу клиентов относят web-приложения, имеющие backend.).

Сервер ресурсов

Сервер авторизаци

Клиент

Client id, client secret

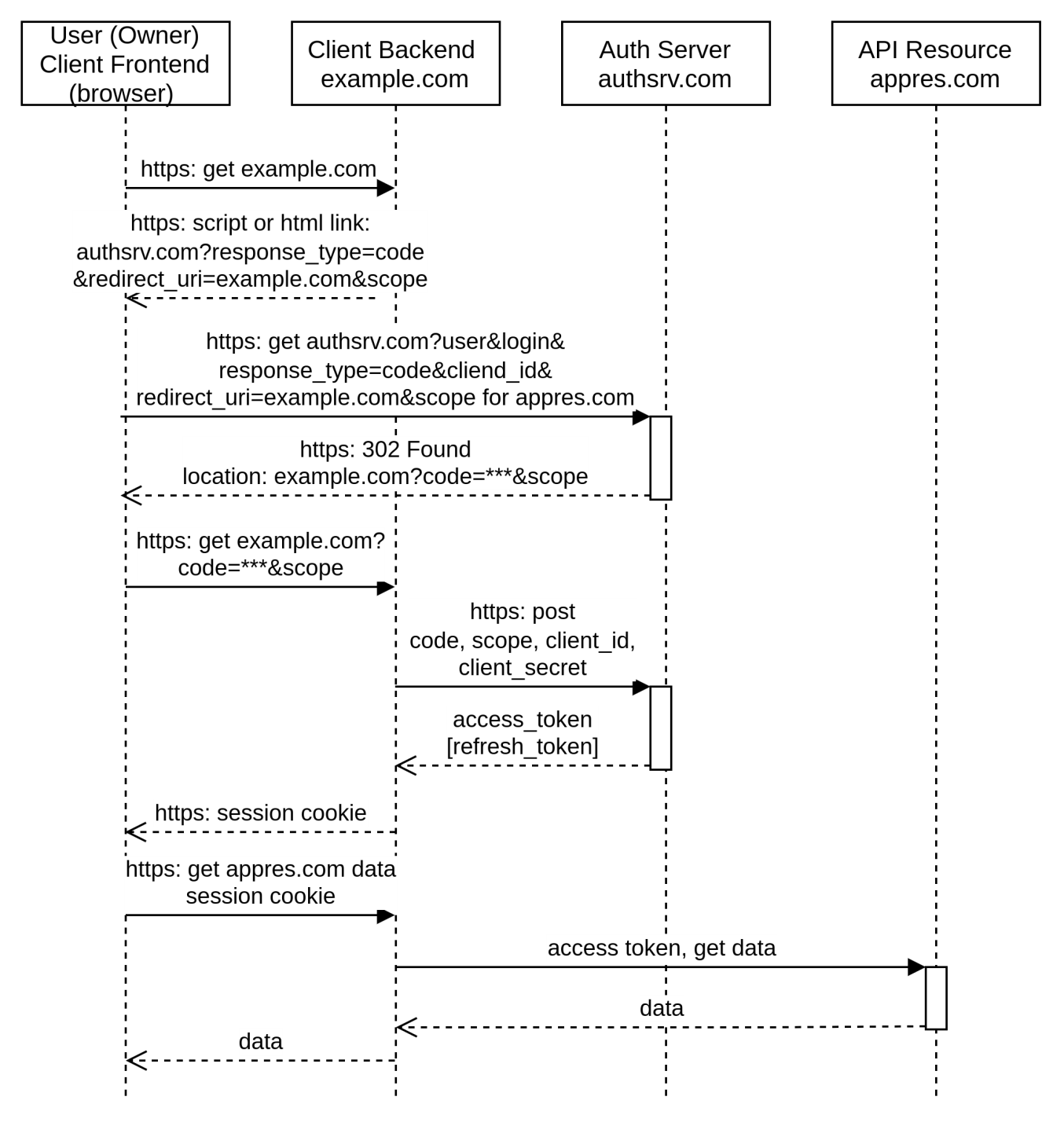
Access token

Access token

Рисунок – Схема AUTH 2.0 взаимодействия где клиент является владельцем ресурсов

Здесь не указано как и в других примерах взаимодействие сервера и авторизации и сервера ресурсов, но очевидно оно должно быть для согласования режима доступа передаваемого в scope. Кроме того, сервер ресурсов и сервер авторизации могут быть одним и тем же. Аuth 2.0 не накладывает органичения на роли указанных взаимодействующих частей.

Далее рассмотрим Authorization code flow, самая распространенная схема и одна из защищенных, за счет отправки дополнительно между клиентом и сервером авторизации client\_id и client\_secret, но и она может быть подвергнута атакам, связанным с неправильным использованием redirect\_uri. На рисунке приводится схема взаимодействия с учтом того, что клиент представляет собой веб приложение с фронт и бэк частью.

 Рисунок – Схема взаимодействия получения прав доступа Authorization code flow

На первом шаге клиент перенаправляет resource owner с помощью user-agent на страницу аутентификации Authorization server. В URI он указывает client ID и redirection URI. Redirection URI используется для понимания, куда вернуть resource owner после того, как авторизация пройдёт успешно (resource owner выдаст разрешение на scope, запрашиваемый клиентом).

Взаимодействуя с сервером авторизации через user-agent, resource owner проходит аутентификацию на сервере авторизации.

Resource owner проверяет права, которые запрашивает клиент на consent screen и разрешает их выдачу.

Resource owner возвращается клиенту с помощью user-agent обратно на URI, который был указан как redirection URI. В качестве query-параметра будет добавлен authorization code — строка, подтверждающая то, что resource owner выдал необходимые права сервису.

С этим authorization code клиент отправляется на сервер авторизации, чтобы получить в ответ access token (ну и refresh token, если требуется).

Сервер авторизации валидирует authorization code, убеждаясь, что токен корректный и выдаёт клиенту access token (и опционально refresh token). С его помощью клиент сможет получить доступ к заветному ресурсу.

Еще одним способом доступа является implicit grant flow (неявный доступ), но он не рекомендуется. В данном случае вместо кода сразу возвращается аксес и рефреш токен, это небезопасно.

В другом варианте, который уже запрещен пароль владельца ресурса (resource owner password credentials) пароль и логин передается через клиента на сервер авторизации, что, совсем небезопасно потому, что пароль и логин пользователя становятся известны клиентскому приложению.

Появились схемы, которые усиливают защиту доступа по авторскому коду.

**Authorization Code Flow with Proof Key for Code Exchange (PKCE).** Это усовершенствованный вариант Authorization code grant flow. В нем происходит генерация еще двух случайных кодов: Code Verifier и Code Challenge. Авторизация реализуется с посылкой Code Challenge, после введения пользователем логина и пароля авторизационным сервером отдается Authorization Code, далее идет обращение за аксес токеном с только что выданным Authorization Code и с Code Verifier. В результате Identity-провайдер (Авторизационный сервер) валидирует на основании трех кодов.

Если злоумышленник получит один код из системы, он ничего не сможет сделать с ним — нужны все три кода. Примерами, где можно задействовать такой flow, являются Single Page Applications: Angular приложения, React JS приложения, и нативные приложения под Android/iOS.

Полный процесс описанный в rfc7637:

A. Клиент создает и записывает секрет с именем «code\_verifier» и получает преобразованную версию "t(code\_verifier)" (или «code\_challenge»), которая отправляется по протоколу OAuth 2.0 в запросе авторизации вместе с методом преобразования "t\_m".

B. Конечная точка авторизации отвечает как обычно, отправляя код авторизации, но записывает "t(code\_verifier)" и метод преобразования "t\_m".

C. Затем клиент отправляет код авторизации как обычно в точку токена авторизации, но включает сгенерированный секрет «code\_verifier» из шага (А).

D. Сервер авторизации преобразует «code\_verifier» и сравнивает его в "t(code\_verifier)" из (B). Доступ запрещен, если они не равны.

Злоумышленник, перехватывающий код авторизации в точке (B), не может обменять его на токен доступа, так как не владеет секретом "code\_verifier".

То есть перед запросом авторизации клиенту надо сформровать code\_verifier и t(code\_verifier) = code\_challenge (на бэке приложения) и выслать на аторизирующий сервер вместе с методом преобразования t. Затем сервер авторизации вышлет code (запомнив code\_challenge). Приложение-клиент (бэк приложения) далее должно выслать на авторизационный сервер code и code\_verifier. Авторизационный сервер проверит code и t(code\_verifier) сравнив с code\_challenge. Обратно высылает access\_token.

Данный вид авторизации использует два вида токенов, access и refresh токен. Позволяет удобно реализовать концепцию микросервисной архитектуры, где будет существовать отдельный сервер авторизации. Данный сервер выдает клиенту или клиентскому приложению access токен и refresh токен (хотя рефреш токен опционален). Добавление refresh токена позволяет миновать процедуру запроса пароля пользователя и ввода его вручную. Обычно предполагается создание авторизирующего сервера, который на логин и пароль выдает данные токены. Первый выдается на короткое время и служит для того, чтобы аутентифицироваться на ресурсных серверах, когда его время заканчивается, то используется уже refresh токен для его обновления. Обычно в такие токены включают сведения о пользователе, например его идентификатор и прочую информацию, чтобы не искать ее в базе. Refresh токен как уже было сказано нужен для того, чтобы обновление jwt токена было реализовано автоматически, по прошествии времени. Очевидно, что тут нужно использовать асинхронные методы обмена. При этом если злоумышленник получил каким-то образом access токен, то через какое-то время он станет невалидным, так как будет выпущен новый токен и старый уже не будет приниматься, в этом случае доступ может быть заморожен. При получении злоумышлинником refresh токена, он может обновить токен, но опять же у легального пользователя устареет его access токен и в то же время его refresh токен и ему придется авторизоваться, так же можно получив от того же пользователя устаревший рефреш токен можно заморозить доступ на всех устройствах. Для дополнительной защиты часто используют так называемый finger print, уникальную информацию с устройства пользователя или клиента, дополнительно проверяя и ее.

Хотя refresh токен не обязательно делать невалидным после обновления, но в большинстве случаев отправляют и новый refresh и access токен и время жизни и после истечения этого времени, аксес и рефреш токен обновляются, либо по отправке рефреш токена, либо при заходе со страницы авторизации (хотя рефреш токен можно оставлять прежним, а можно как уже было сказано вовсе не использовать). Таким образом, например, если аксес токен стал невалидным по истечении времени, то пользователя или клиент пользователя отправят страницу авторизации или будет выслан рефреш токен. Если, например, злоумышленник перехватил рефреш токен, то у легального пользователя рефреш токен станет невалидным и его отправят на страницу авторизации с требованием ввести логин и пароль.

#### Open ID Connect

Протокол OpenID Connect (OIDC)

OIDC – семейство протоколов, являющихся расширением протоколов OAuth 2.0, позволяющих расширить их функционал путем более точного описания процесса аутентификации владельца ресурса и возможности клиенту получить информацию о нем. Это этапы (1) – (4) протокола OAuth 2.0.

Схематично протокол OpenID Connect можно описать следующим образом

1) Клиент отправляет серверу авторизации запрос аутентификации.

2) Сервер авторизации аутентифицирует конечного пользователя и получает согласие пользователя на доступ клиента к запрошенному ресурсу.

3) Сервер авторизации отвечает клиенту ID токеном и (опционально) токеном доступа.

4) Клиент может отправить серверу авторизации запрос информации о пользователе по токену доступа.

5) Сервер авторизации возвращает клиенту информацию о конечном пользователе.

Сервер авторизации OpenID Connect поддерживает три сценария аутентификации, реализующие этот сценарий: с генерацией кода авторизации (Authorization Code Flow), неявный сценарий (Implicit Flow), гибридный сценарий (Hybrid Flow). Передача сообщений между клиентом и сервером авторизации (на конечных точках авторизации, токена и UserInfo) должна производиться с использованием протокола TLS.

**Задание.**

1. Реализовать HTTP Authorization Digest
2. Реализовать CRAM-MD5
3. Реализовать Authorization Basic и Cookie-based
4. Реализовать Authorization Bearer
5. Реализовать Аuthorization Code flow Auth2.0
6. Реализовать Client credentials flow Auth2.0
7. Реализовать implicit flow Auth2.0
8. Реализовать проверку фингерпринта при авторизации устройств
9. Реализовать Authorization Code Flow with Proof Key for Code Exchange
10. Реализовать выдачу access и refresh и их обновление со временем