SECURE OPENID CONNECT IMPLEMENTATION USING AZURE ACTIVE DIRECTORY AND ASP .NET CORE

PETRO KOLOSOV AND DMITRIJ KUDRYASHOV

Аннотация. В данной статье мы рассмотрим проблему безопасного хранения и передачи токенов доступа между микросервисами. Веб-браузер может хранить токены доступа как в локальном хранилище, так и в файлах cookie. Мы предлагаем безопасную реализацию хранения и передачи cookie-файлов между микросервисами, используя Azure Active Directory, OpenID Connect и ASP.NET Core.

Содержание

1. Определения

- **Токен доступа (Access Token)** - Это токен, используемые для авторизации на защищенном ресурсе. Чаще всего токен доступа в JWT-формате, то есть в виде нескольких последовательных частей, разделенных точками. Каждая часть содержит base64url-закодированное значение.

Date: 26 октября 2023 г.
Key words and phrases. OpenID Connect, OIDC, Azure Active Directory, PKCE, OAuth 2.0, XSS, CSRF, ASP .NET Core .

- **Токен обновления (Refresh Token)** - Это токен, используемые для получения новых токенов доступа, когда текущий токен доступа становится недействительным или истекает срок его действия. Токены обновления выдаются клиенту сервером авторизации.
- **Владелец ресурса (Resource Owner)** - Субъект, способный предъявить токен доступа для получения доступа к защищенному ресурсу. Когда владельцем ресурса является человек, его называют конечный пользователь.
- **Сервер ресурсов (Resource Server)** - Сервер, который хранит защищенные ресурсы и может обрабатывать запросы к этим ресурсам только после проверки наличия соответствующего токена доступа. К примеру если вы реализуете Microsoft OAuth 2.0, то сервером ресурсов будет являться одна из Web API компании Microsoft.
- **Клиент (Client)** - Приложение, выполняющее запросы к защищенным ресурсам. Термин не подразумевает каких-либо конкретных характеристик реализации, клиентом может быть как сервер, так десктоп приложение и так далее.
- **Сервер авторизации (Authorization Server)** - Сервер, выдающий токен доступ клиенту после успешной аутентификации.

2. Проблема

В данной статье мы рассматриваем проблему безопасного хранения и передачи токена доступа между микросервисами. Разделяют два способа хранения токена доступа, а именно хранение в Local Storage и хранение в файлах cookie. Local Storage - это механизм веб-браузера, который позволяет веб-приложениям хранить данные локально на устройстве пользователя. Важно отметить, что локальное хранилище уязвимо для атак типа Cross-Site Scripting [1]. Cross-Site Scripting - это тип атаки, суть которого в том чтобы внедрить вредоносный JavaScript код в выдаваемую html-страницу с целью похищения данных пользователя, например токена доступа.

Различают несколько видов XSS атак:

- **Отражённый XSS (Reflected XSS)** - это тип атаки, при которой вредоносный скрипт передается веб-серверу через параметры URL или формы, а затем возвращается обратно в html-код страницы без должной фильтрации или экранирования. Если пользователь открывает страницу, то скрипт выполняется в браузере, что может привести к потере чувствительных данных, например токена доступа.
- **Хранимая XSS (Stored XSS)** - это тип атаки при которой вредоносный скрипт сохраняется на сервере, например в базе данных и отображается на веб-страницах. Скрипт выполняется в браузерах пользователей, запрашивающих страницы с вредоносным кодом.
- **XSS в DOM-модели** - это тип атаки, при вредоносный скрипт модифицирует DOM-дерево веб-страницы, выполняясь в браузере пользователя. В большинстве случаев, основан на модификации URL-строки.

Другой способ хранить данные это хранить в файлах cookie. Файлы Cookie - это небольшой фрагмент данных, отправленный веб-сервером и хранимый на устройстве пользователя. Хранение токенов доступа в файлах Cookie невелирует потенциальные XSS атаки, так как достаточно установить флаг HttpOnly, запрещающий JavaScript коду чтение данных из файлов Cookie. Передача токенов доступа в запросах осуществляется с использованием JavaScript Http методов с флагом `{ withCredentials: true }`, таким образом, если файлы cookie существуют, то они передается вместе с запросом, но все еще не могут быть прочитаны используя JavaScript. Пример такого запроса следующий

```
return this.httpClient.post<TokensResponse>(
    this.baseUrl + this.sessionsRoute,
    command,
    { withCredentials: true });
```

Однако, файлы cookie уязвимы для Cross-Site Request Forgery (CSRF) атак [2]. Cross-Site Request Forgery (CSRF) - представляет собой атаку при которой злоумышленник

создает переадресацию на ресурс, где пользователь имеет активную сессию. Основной принцип Cross-Site Request Forgery (CSRF) проилюстрирован ниже

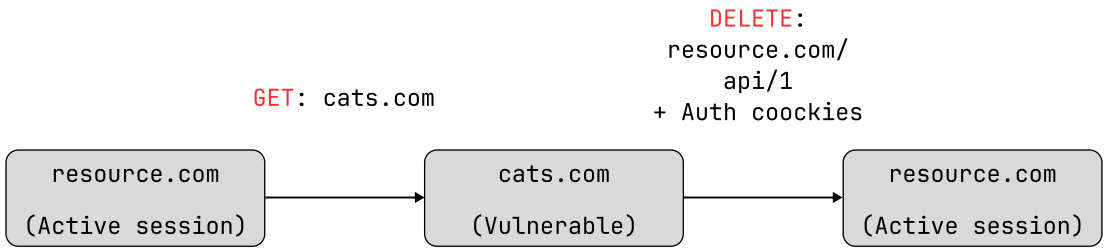

Рис. 1. CSRF attack principle diagram.

Файлы cookie имеют параметр `SameSite`, который определяет будут ли они отправлены вместе с запросом на сайт, имеющим другой домен.

Существует три возможных значения параметра `SameSite`:

- **None** - прямо указывает, что на передачу cookie-файлов не накладывается никаких ограничений.
- **Lax** - разрешает передачу cookie только безопасными HTTP-методами, которыми, согласно RFC 7231 [3], являются `GET, HEAD, OPTIONS` и `TRACE`.
- **Strict** - является самым строгим вариантом безопасности и блокирует отправку cookie с любыми запросами на сайт под другим доменом. Файлы cookie будут передаваться только в пределах того домена, с которого они и были установлены.

Таким образом, значения параметра `SameSite` такие как `Lax` и `Strict` защищают пользователя от CSRF-атаки, так как блокируют прикрепление файлов cookie к запросу, что был инициирован ресурсом злоумышленника.

Другие методы защиты от CSRF описаны в [4].

3. Знакомство с OpenID Connect

OpenId Connect — это дополнительный уровень [5, 6] идентификации поверх протокола [7] OAuth 2.0. Он позволяет клиентам проверять личность конечного пользователя на основе аутентификации, а также получать базовую информацию о профиле конечного пользователя.

OAuth 2.0 — это протокол, позволяющий стороннему приложению получить ограниченный доступ к HTTP-сервису, либо от имени владельца ресурса, организуя взаимодействие по утверждению между владельцем ресурса и HTTP-сервисом, либо путем разрешения стороннему приложению получить доступ от своего имени.

В традиционной модели аутентификации клиент-сервер, клиент запрашивает ресурс с ограниченным доступом через аутентификацию с использованием пары логин-пароль. Для того чтобы предоставить стороннему приложению доступ к ресурсам, владелец ресурса (пользователь) делится своими учетными данными (логином и паролем) со сторонним приложением. Этот подход создает ряд проблем и ограничений [7]:

- Сторонние приложения должны хранить учетные данные (логин и пароль) владельца ресурса для дальнейшего использования, бывает так, что пароль хранится в явном виде в базе данных.
- Сторонние приложения обязаны поддерживать аутентификацию с помощью пароля, несмотря на недостатки безопасности, присущие паролям.
- Сторонние приложения получают полный доступ к защищенным ресурсам владельца ресурса (пользователя), оставляя владельцев ресурсов без какой-либо возможности ограничить продолжительность доступа или подмножество доступных ресурсов (scope).
- Владельцы ресурсов (пользователи) не могут отозвать доступ к отдельному третьему лицу без отзыва доступа для всех третьих лиц, и должны отозвать доступ только путем изменения пароля.

- Взлом либо утечка данных из любого стороннего приложения приводит к потере пароля конечного пользователя и всех данных, защищенных этим паролем.

OAuth 2.0 решает эти проблемы путем введения уровня авторизации и разделения роли клиента (сервис авторизации) и владельца ресурса (пользователя). В OAuth 2.0 клиент запрашивает доступ к ресурсам, контролируемым владельцем ресурса и размещенным на сервере ресурса. Вместо того чтобы использовать учетные данные владельца ресурса для доступа к защищенным ресурсам, клиент получает токен доступа.

OAuth 2.0 with the Proof of Key Code Exchange [8] flow показан ниже

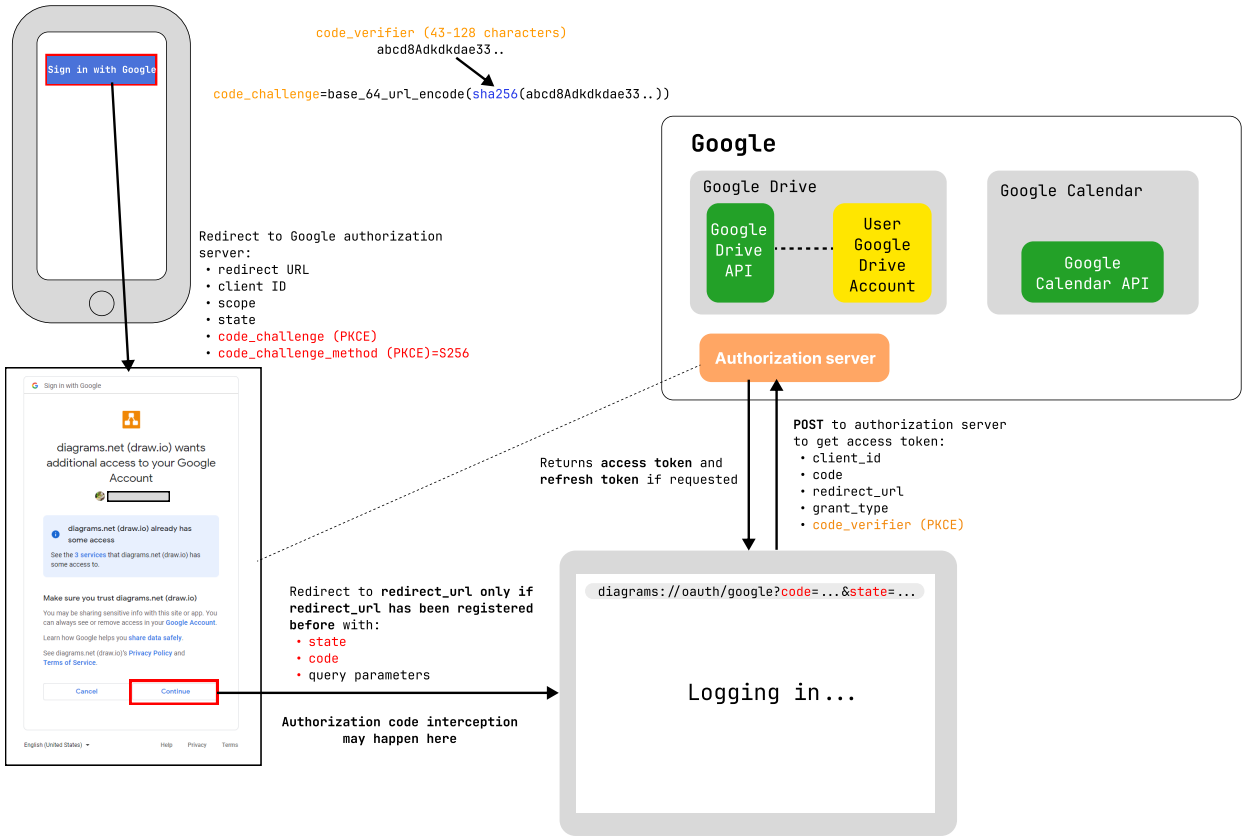

Рис. 2. OAuth 2.0 with PKCE flow diagram.

(1) После нажатия на кнопку `Sign in with Google` происходит перенаправление на ендпоинт авторизации, на котором владелец ресурса должен ввести свои учетные

данные, а так же согласиться с некоторыми условиями. Особое внимание стоит обратить на code_challenge и code_challenge_method, первое представляет собой некоторую строку, которая будет закодирована с помощью метода кодировки указанного в параметре запроса code_challenge_method. В дальнейшем, при обмене полученного кода на токен доступа и обновления, мы должны будем предъявить значение code_verifier, которое равно ранее закондированному code_challenge.

(2) После успешной аутентификации, произойдет перенаправление на адресс `redirect_uri`.

(3) После получения авторизационного кода происходит обмен этого кода на токены доступа (access token) и обновления (refresh token). Если ранее, при переадресации владельца ресурса на авторизационный ендпоинт, мы указывали `code_challenge` и `code_challenge_method`, то теперь, при обмене кода на токены, мы обязаны передать в запросе значение `code_verifier`.

Здесь мы упоминаем такие определения, как code и state.

- **Code** - это код авторизации, который получается с помощью сервера авторизации и является посредником между клиентов и владельцем ресурса. Перед тем как сервер авторизации перенаправит владельца ресурса обратно на клиент, сервер авторизации проверяет подлинность владельца ресурса. Так образом поскольку владелец ресурса аутентифицируется только на сервере авторизации, его учетные данные никогда не передадутся клиенту.
- **State** - значение, используемое клиентом для сохранения состояния между запросом на авторизацию и обратным вызовом (callback). Сервер авторизации включает это значение при перенаправлении агента пользователя обратно клиенту. Этот параметр используется для предотвращения Cross-Site Request Forgery (CSRF) атак.

Authorization code flow with PKCE является протоколом, который представляет секрет, созданный клиентом, который может быть проверен сервером авторизации.

Этот секрет называется `code_verifier`. Клиент хеширует значение `code_verifier` и записывает его в параметр `code_challenge`. PKCE решает проблему безопасного обмена кода. Если злоумышленнику удастся заполучить авторизационный код, то у него не получится обменять его на токены доступа и обновления. Таким образом, мы гарантируем тот факт, что обмен кода на токены производит то же самое приложение, что и выполняло аутентификацию. В некотором роде, PKCE можно сравнить с цифровой подписью процесса аутентификации. При обмене кода авторизации на токены обязательно нужно указать `code_verifier`.

4. Аутентификация

Рассмотрим более практичный подход, учитывающий все рассмотренные ранее аспекты. Применяя современные фреймворки, такие как ASP.NET Core и Angular, можно реализовать следующий флоу аутентификации в соответствии с приведенной ниже схемой

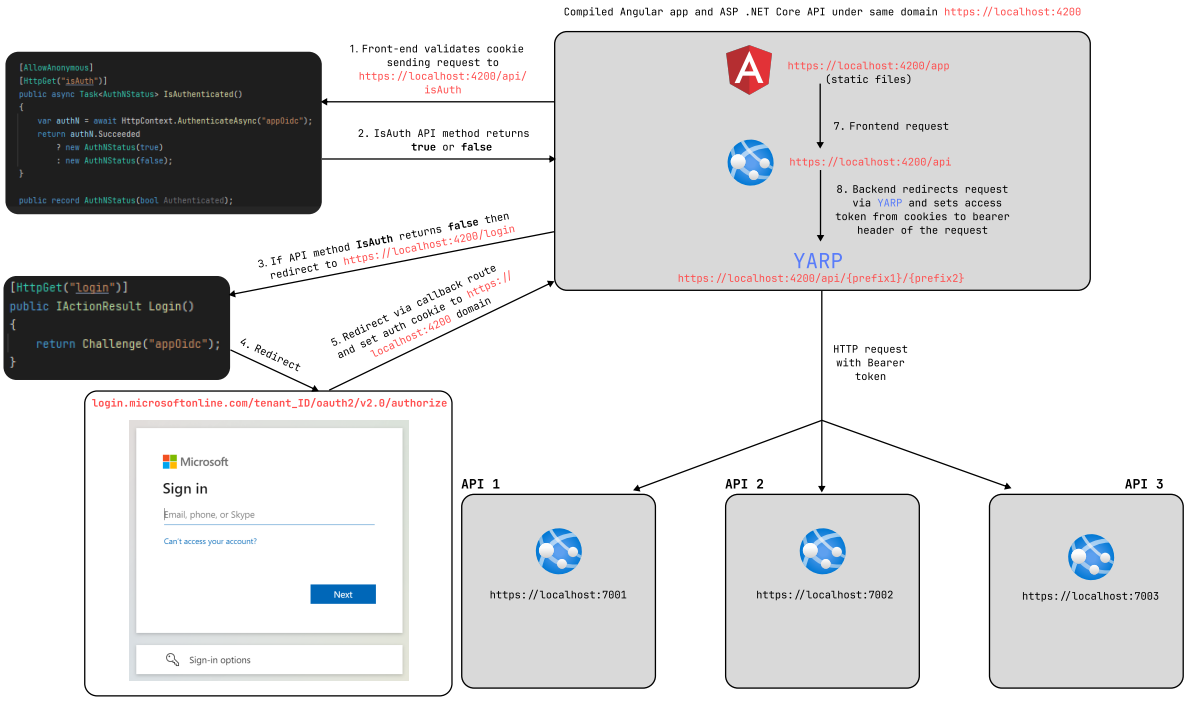

Рис. 3. Authentication flow diagram.

Таким образом, весь процесс аутентификации можно описать в виде восьми этапов, таких как

(1) Скомпилированное Angular-приложение отправляет запрос к ендпоинту ASP.NET Core API для проверки текущего состояния аутентификации. Angular-приложение представляет собой набор предварительно скомпилированных пакетов, которые раздаются через тот же ASP.NET Core API на ендпоинте `/app`, что позволяет отказаться от кросс-доменных запросов и надежно хранить токены в файлах cookie.

(2) Ендпоинт ASP.NET Core API отвечает либо со HTTP статус кодом `200 (OK)` или `401 (Unauthorized)`

(3) Если на предыдущем шаге получен статусный код `401 (Unauthorized)`, то браузер перенаправляется на ендпоинт `/login` API ASP.NET Core, в ином случае пользователь получает доступ к защищенным ресурсам

(4) Метод Login ASP.NET Core приложения перенаправляет браузер на сайт авторизации Azure AD `login.microsoftonline.com/tenant/oauth2/v2.0/authorize`, где пользователь вводит свои учетные данные. Важно уточнить, что для получения ID-токена нам необходимо поместить параметр `openid` в область видимости (scope)

```
serviceCollection
.AddAuthentication(options => {...})
.AddCookie(CookieAuthenticationDefaults.AuthScheme,
    options => {...})
.AddOpenIdConnect(AuthConstants.AppOidc, options =>
{
    ...
    options.Scope.Add("openid");
});
```

(5) После успешной аутентификации на стороне Azure AD, браузер перенаправляется уже с установленными файлами cookie на `callback_url`, который определен при регистрации приложения в Azure AD. В этот момент в флоу включается `TickerStore` [9, 10], который нужен для управления сессиями пользователей. Каждая сессия хранится в базе данных как сущность `UserSessionEntity`.

```
public class UserSessionEntity
{
    public Guid Id { get; set; }
    public DateTimeOffset CreatedAt { get; set; }
    public DateTimeOffset ExpiresAt { get; set; }
    public DateTimeOffset UpdatedAt { get; set; }
    public DateTimeOffset DateOfLastAccess { get; set; }
    public byte[] Value { get; set; }
}
```

Свойство Value типа `byte[]` содержит сериализованный `AuthenticationTicket` [11] объект, содержащий всю необходимую информацию, такую как ID-токен, токен доступа (access token) и токен обновления (refresh token). Класс `TickerStore` реализует интерфейс `ITickerStore`, который предлагает 4 метода: `StoreAsync`, `RenewAsync`, `RetrieveAsync`, `RemoveAsync`.

- Метод `StoreAsync` выполняется сразу после аутентификации на сервере авторизации, он сохраняет сессию пользователя в базе данных.
- Метод `RenewAsync` в нашем случае используется фоновым сервисом для обновления пользовательских сессий.
- Метод `RetrieveAsync` выполняется каждый раз, когда запрос отправляется на ендпоинт, помеченный `[Authorize]` атрибутом.

- Метод `RemoveAsync` выполняется по истечении срока действия cookie-файлов, а также используется тем же `RefreshBackgroundService` для удаления сессий, которые не использовались в течение длительного времени.

Пример реализации `TicketStore` можно найти на [10]. Пример инъекции зависимостей (DI) `TicketStore` можно найти на [12]. На этом шаге происходит настройка аутентификационных cookie.

(6) Шаг 1 здесь повторяется, но теперь HTTP-запрос обязательно должен быть с кодом состояния `200 (OK)`.

(7) Скомпилированное фронтенд-приложение Angular отправляет запрос к ASP.NET Core API с cookie-файлами. Микросервис, принимающий запрос, используя библиотеку `YARP` [13], вынимает из cookie файлов токен доступа и кладет его в заголовок, после чего перенаправляет запрос на один из микросервисов. Конфигурация `YARP` осуществляется в соответствии с [14, 15].

(8) Если на предыдущем шаге был получен статусный код `401 (Unauthorized)`, то шаг 1 повторяется.

5. Токен обновления

Реализация обновления токенов пользователя крайне проста. Нам нужно создать фоновый сервис [16], который будет каждые несколько минут выбирать из базы данных те сессии, которые скоро истекут, затем фоновый сервис должен десериализовать объект `AuthenticationTicket` [11] у каждой найденой сессии, из десериализованного объекта сервис возьмет токен обновления и отправит запрос на сервер авторизации с целью получения новых токенов доступа, обновления и ID-токена. Новые токены заменяют старые в объекте `AuthenticationTicket`, после чего объект нужно заного сериализовать и установить уже сериализованный объект в свойство `Value`. Кроме того, в ответе сервера авторизации в поле `ExpiresAt` будет число, которое говорит о том через какое время (в секундах) истечет токен доступа, фоновый сервис должен

обновить свойство ExpiresAt у объекта `UserSessionEntity`UserSessionEntity, добавив к текущему времени секунды, полученные из ответа запроса.

Помимо обновления пользовательских сессий, фоновый сервис отвечает за удалений сессий, которые долго не использовались. Каждые несколько минут выбираются сессии, их свойства `DateOfLastAccess` сравнивается с текущем временем, в случае если разница между двумя датами больше 3 суток - сессия удаляется. Каждый раз когда пользователь совершает действие на сайте, свойство `DateOfLastAccess` обновляется.

Реализация фонового сервиса может быть выполнена в соответствии с приведенными ссылками [17, 18].

6. Вывод

В данной статье мы рассмотрели проблему безопасного хранения и передачи токена доступа между микросервисами. Особое внимание было уделено возможным уязвимостям при передаче токена доступа, таким как как Cross-Site Scripting (XSS) и Cross-Site Request Forgery (CSRF).

Для устранения данных уязвимостей необходимо хранить авторизационные токены в файлах cookie, с обязательными настройками `HttpOnly` и `SameSite` так, что значения `SameSite` должны быть `Lax` или `Strict`, таким образом файлы куки передаются либо на безопасные `HTTP` методы, либо не передаются вовсе.

Аутентификация должна быть реализована с использованием протокола OIDC [5, 6], а Authorization code flow с помощью PKCE [8]. Более подробно основной принцип работы протокола OIDC описан в главе 2.

Аутентификация пользователя происходит по протоколу OpenID Connect [5, 6] через Authorization code flow with PKCE [8]. Подробнее принцип работы протокола OpenID Connect и Authorization code flow with PKCE изложен во главе 2.

Так же в работе был предложен механизм аутентификации/авторизации, основанный на ASP.NET Web API и Angular фронтенд-приложении под единым доменом. Таким образом исключается необходимость передачи файлов cookie на ресурсы под другим

доменом. Передача токена доступа на микросервисы происходит по средствам Reverse Proxy YARP [13], таким образом что токен доступа автоматически подставляется в заголовок запроса.

Кроме этого, в работе был предложен механизм обновления токена доступа через реализацию `TicketStore` [9] и `HostedService` [16]. Таким образом, `TicketStore` отвечает за проверку каждого запроса на истечение срока действия токена доступа. В случае истечения срока действия токена доступа, токен доступа обновляется с помощью запроса к эндпоинту сервера авторизации. Также задача `TicketStore` состоит в сохранении токена доступа и токена обновления, которые являются частью `AuthenticationTicket` [11 Hosted Service необходим для фонового обновления истекающих токенов доступа, чтобы поддерживать сессии пользователей активными.

В статье мы решили проблему безопасного хранения токена доступа и передачи его между микросервисами, а так же предложили решения выявленным уязвимостям вида Cross-Site Scripting (XSS) и Cross-Site Request Forgery (CSRF).

Список литературы

[1] Kevin Spett. Cross-site scripting. *SPI Labs*, 1(1):20, 2005.

[2] Mohd Shadab Siddiqui and Deepanker Verma. Cross site request forgery: A common web application weakness. In *2011 IEEE 3rd International Conference on Communication Software and Networks*, pages 538–543. IEEE, 2011.

[3] Roy Fielding and Julian Reschke. RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): semantics and content. https://www.rfc-editor.org/rfc/rfc7231, 2014.

[4] OWASP Cheat Sheet Series. Cross-Site Request Forgery Prevention Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html, 2023.

[5] Prabath Siriwardena and Prabath Siriwardena. OpenID Connect (OIDC). https://link.springer.com/chapter/10.1007/978-1-4842-2050-4_6, 2020.

[6] Nat Sakimura, John Bradley, Mike Jones, and E Jay. OpenID connect dynamic client registration 1.0. https://openid.net/specs/openid-connect-registration-1_0-final.html, 2014.

[7] Dick Hardt. The OAuth 2.0 authorization framework. https://www.rfc-editor.org/rfc/rfc6749, 2012.

[8] J Bradley and N Agarwal. RFC 7636: Proof Key for Code Exchange by OAuth Public Clients. https://www.rfc-editor.org/rfc/rfc7636, 2015.

[9] Microsoft. ITicketStore Interface. https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.authentication.cookies.iticketstore?view=aspnetcore-7.0, 2023.

[10] Dmitrij Kudryashov. Ticket store implementation. https://gist.github.com/Ketteiteki/7eff8e3bb35d8d2877bd74404dcef129, 2023.

[11] Microsoft. AuthenticationTicket Class. https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.authentication.authenticationticket?view=aspnetcore-7.0, 2023.

[12] Dmitrij Kudryashov. Ticket store dependency injection. https://gist.github.com/Ketteiteki/3046261b345955f6e3d4e164774bfed1, 2023.

[13] Microsoft. YARP: Yet Another Reverse Proxy. https://microsoft.github.io/reverse-proxy, 2021.

[14] Dmitrij Kudryashov. YARP dependency injection. https://gist.github.com/Ketteiteki/d1077303ef1c11d70b286f08cfd44824, 2023.

[15] Dmitrij Kudryashov. YARP section app settings. https://gist.github.com/Ketteiteki/45d76a8409b3bb2fd9030fc8f117ca38, 2023.

[16] Microsoft. Background tasks with hosted services in ASP.NET Core. https://learn.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services?view=aspnetcore-7.0&tabs=visual-studio, 2023.

[17] Dmitrij Kudryashov. Background service implementation. https://gist.github.com/Ketteiteki/560afca832c6143a4b847c907439c44c, 2023.

[18] Dmitrij Kudryashov. Background service configuration. https://gist.github.com/Ketteiteki/735d2a3f0bb5e54cbe41895df74b8504, 2023.

Version: `Local-0.1.0`

Email address: kolosovp94@gmail.com

URL: https://kolosovpetro.github.io

Email address: kudryashov.kd@gmail.com