HighLoad в микросервисах

Как спроектировать качественный микросервис и не обосраться

Токены авторизации - Издание I

Автор: Владимир Желнов
Год издания: 2024

Как обмениваться токенами между микросервисами

Обмен токенами между микросервисами в микросервисной архитектуре важен для обеспечения безопасности и авторизации. Обычно используется механизм JSON Web Tokens (JWT) для передачи информации о пользователе или сервисе между микросервисами. Вот общий способ реализации:

1. **Аутентификация:**

При успешной аутентификации пользователя в одном из микросервисов выдается JWT-токен.
Токен содержит информацию о пользователе и его правах, подписанную секретным ключом микросервиса.

2. **Передача токена:**

После аутентификации, микросервис может передать токен в запросе к другим микросервисам.
Токен обычно включается в заголовок запроса, например, "Authorization: Bearer <токен>".

3. **Верификация токена:**

При получении запроса другим микросервисом, он проверяет подлинность токена, используя секретный ключ, известный только выдающему токен микросервису.
Это обеспечивает доверенность информации, предоставленной токеном.

4. **Рефреш токена (опционально):**

Для продления срока действия токена могут использоваться refresh-токены.
Refresh-токен обменяется на новый JWT-токен без необходимости повторной аутентификации.

5. **Срок действия токена:**

Токены обычно имеют ограниченный срок действия для уменьшения риска утечки информации.

Пример кода для генерации JWT-токена (используя библиотеку PyJWT в Python):

```
import jwt
from datetime import datetime, timedelta

# Заголовок токена
header = {"alg": "HS256", "typ": "JWT"}

# Полезная нагрузка токена (например, информация о пользователе)
payload = {"user_id": 123, "username": "john_doe", "exp": datetime.utcnow() + timedelta(hours=1)}

# Секретный ключ микросервиса для подписи токена
secret_key = "your_secret_key"

# Генерация токена
token = jwt.encode(payload, secret_key, algorithm="HS256")
```

Важно обеспечить безопасное хранение и передачу секретного ключа, чтобы предотвратить компрометацию токенов.

Этапы обмена токеном аутентификации между микросервисами

Обмен токенами между микросервисами в микросервисной архитектуре важен для обеспечения безопасности и авторизации. Обычно используется механизм JSON Web Tokens (JWT) для передачи информации о пользователе или сервисе между микросервисами. Вот общий способ реализации:

1. Аутентификация

При успешной аутентификации пользователя в одном из микросервисов выдается JWT-токен.
Токен содержит информацию о пользователе и его правах, подписанную секретным ключом микросервиса.

2. Передача токена

После аутентификации, микросервис может передать токен в запросе к другим микросервисам.
Токен обычно включается в заголовок запроса, например, "Authorization: Bearer <токен>".

3. Верификация токена

При получении запроса другим микросервисом, он проверяет подлинность токена, используя секретный ключ, известный только выдающему токен микросервису.
Это обеспечивает доверенность информации, предоставленной токеном.

4. Рефреш токена (опционально)

Для продления срока действия токена могут использоваться refresh-токены.
Refresh-токен обменяется на новый JWT-токен без необходимости повторной аутентификации.

5. Срок действия токена:

Токены обычно имеют ограниченный срок действия для уменьшения риска утечки информации.

Пример кода для генерации JWT-токена (используя библиотеку PyJWT в Python):

```
import jwt
from datetime import datetime, timedelta

# Заголовок токена
header = {"alg": "HS256", "typ": "JWT"}

# Полезная нагрузка токена (например, информация о пользователе)
payload = {"user_id": 123, "username": "john_doe", "exp": datetime.utcnow() + timedelta(hours=1)}

# Секретный ключ микросервиса для подписи токена
secret_key = "your_secret_key"
```

```
# Генерация токена
token = jwt.encode(payload, secret_key, algorithm="HS256")
```

Важно обеспечить безопасное хранение и передачу секретного ключа, чтобы предотвратить компрометацию токенов.

Как передавать токен между микросервисами

Передача токена между микросервисами является важным аспектом обеспечения безопасности и авторизации в микросервисной архитектуре. Вот более подробное объяснение этого процесса:

1. Получение токена при аутентификации

При успешной аутентификации пользователь получает JWT-токен от аутентификационного микросервиса. Этот токен содержит информацию о пользователе и его правах.

Примерный вид токена:
`eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c`

2. Передача токена в другие микросервисы

Другие микросервисы, которые нуждаются в информации о пользователе или его авторизации, могут получить токен.
Токен часто передается в заголовке запроса, используя схему "Bearer Token".

Пример:
`GET /api/some-endpoint`
`Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIi...`

3. Проверка подлинности токена

Микросервис, принимающий запрос, должен проверить подлинность токена. Это включает в себя дешифрацию и верификацию подписи токена, используя секретный ключ аутентификационного микросервиса.

Если токен не подлинный или срок его действия истек, микросервис может отклонить запрос.

4. Извлечение информации из токена

После успешной проверки подлинности токена, микросервис может извлечь информацию о пользователе из полезной нагрузки токена. Например, user_id, username, роли и другие данные.

Эти данные могут быть использованы для принятия решений об авторизации и выполнении действий в рамках запроса.

Пример проверки токена и извлечения данных на Python с использованием библиотеки PyJWT:

```
import jwt
from flask import request

def verify_token():
    token = request.headers.get("Authorization")
    if token and token.startswith("Bearer "):
        token = token.split(" ")[1]
        secret_key = "your_secret_key"
        try:
                decoded_data = jwt.decode(token, secret_key,
algorithms=["HS256"])
                  # Используйте decoded_data для получения
информации о пользователе
            user_id = decoded_data.get("user_id")
            return user_id
        except jwt.ExpiredSignatureError:
            # Обработка истекшего токена
            return None
        except jwt.InvalidTokenError:
            # Обработка недействительного токена
            return None
    return None
```

Важно обеспечить безопасное хранение секретного ключа и использовать HTTPS для защищенной передачи токенов между микросервисами.

Секретный ключ аутентификационного микросервиса

Секретный ключ аутентификационного микросервиса является ключевым элементом в обеспечении безопасности JWT-токенов. Этот ключ используется для подписи токена и обеспечения его подлинности. Вот более подробное объяснение:

1. Что представляет собой секретный ключ

- Секретный ключ - это секретная строка символов, которая используется для подписи JWT-токенов.
- Только тот микросервис, который знает этот ключ, может создавать и проверять токены.

2. Генерация и хранение секретного ключа

- Секретный ключ обычно генерируется с использованием криптографически стойкого генератора случайных чисел.
- Ключ должен быть строго доверенным лицам и должен храниться в безопасном месте, чтобы предотвратить его утечку.
- Обычно ключ не передается в открытом виде по сети и не хранится в репозиториях кода.

3. Использование секретного ключа для подписи токенов

При создании JWT-токена, микросервис использует секретный ключ для подписи токена. Это происходит в процессе кодирования токена.

Пример кода на Python с использованием библиотеки PyJWT:

```
import jwt
payload = {"user_id": 123, "username": "john_doe"}
secret_key = "your_secret_key"
token = jwt.encode(payload, secret_key, algorithm="HS256")
```

4. Проверка подписи токена

При получении JWT-токена, другой микросервис, который хочет проверить подлинность токена, использует тот же секретный ключ для декодирования и верификации подписи.

Если ключ совпадает с ключом, использованным для подписи токена, и подпись верна, то токен считается подлинным.

Пример проверки токена на Python:
```
import jwt
token = "your_jwt_token"
secret_key = "your_secret_key"
decoded_data = jwt.decode(token, secret_key, algorithms=["HS256"])
```

5. Обеспечение безопасности секретного ключа

- Секретный ключ должен храниться в безопасном хранилище, доступном только авторизованным лицам.
- Не следует передавать секретный ключ в открытом виде по сети, и его использование должно быть строго ограничено.

Важно помнить, что утечка секретного ключа может привести к компрометации безопасности вашей системы. Обеспечьте его безопасное хранение и передачу.

Как безопасно хранить секретные ключи микросервисов

Безопасное хранение секретных ключей микросервисов крайне важно для предотвращения потенциальных угроз безопасности. Вот несколько практических рекомендаций:

1. Используйте переменные окружения

- Секретные ключи могут быть хранены в переменных окружения. Это позволяет изолировать конфиденциальную информацию от кода и конфигурационных файлов.
- В многих облачных платформах (например, AWS, Azure, Google Cloud), есть инструменты для безопасного хранения переменных окружения.

2. Хранилища секретов

- Используйте инструменты для безопасного хранения секретов, такие как:
- AWS Secrets Manager (для Amazon Web Services).
- Azure Key Vault (для Microsoft Azure).
- Google Cloud Secret Manager (для Google Cloud Platform).
- Эти инструменты предоставляют механизмы для управления и хранения секретов, а также обеспечивают аудит и мониторинг доступа к ним.

3. Управление идентификацией и доступом (IAM)

- В облачных средах, где микросервисы могут иметь доступ к инфраструктурным службам, использование IAM позволяет строить гранулированные права доступа.
- Устанавливайте минимальные необходимые разрешения для микросервисов.

4. Локальные файлы и директории:

- Если требуется локальное хранение ключей, используйте безопасные методы, такие как файлы с ограниченными правами доступа (например, доступ только для чтения текущим пользователем).

5. Шифрование данных в покое:

- Если секретные ключи необходимо хранить в репозитории кода или в каком-то хранилище, используйте механизмы шифрования, чтобы защитить данные от несанкционированного доступа.

6. Контейнеризация:

- Если вы используете контейнеры, избегайте включения секретных ключей в образ контейнера. Лучше передавайте их через переменные окружения или механизмы управления секретами, предоставляемые оркестратором контейнеров (например, Kubernetes Secrets).

7. Регулярная смена ключей:
- Периодически изменяйте секретные ключи, чтобы уменьшить риск компрометации.

Важно также уделять внимание правам доступа к системам, хранящим секретные ключи, чтобы уменьшить возможность несанкционированного доступа. Всегда придерживайтесь принципа "не храни секреты в коде".

Обмен данными между микросервисами через HTTP

Предположим, у вас есть два микросервиса: один для работы с пользователями и другой для работы с постами. Давайте рассмотрим пример использования HTTP-запросов для получения данных о пользователях из микросервиса пользователей в микросервисе постов.

Данный пример является лишь примером, и не является гарантированным качественным выбором для использования его в продакнш системах.

Микросервис пользователей

Предположим, что ваш микросервис пользователей предоставляет API для получения данных о пользователях. Примерный код на Python с использованием Flask:

```
from flask import Flask, jsonify, request

app = Flask(__name__)

# Пример эндпоинта, возвращающего данные о пользователе по его ID
@app.route('/users/<int:user_id>', methods=['GET'])
def get_user(user_id):
    # Проверка наличия токена в заголовках запроса
    auth_token = request.headers.get('Authorization')
    if not auth_token:
            return jsonify({'error': 'Authorization token is missing'}), 401
```

```
    # Логика проверки и обработки токена
    # (в реальном приложении следует использовать стороннюю библиотеку или сервис для аутентификации)

    # Логика получения данных о пользователе из базы данных или другого источника
    user_data = {
        'user_id': user_id,
        'username': 'john_doe',
        'email': 'john.doe@example.com'
        # Другие данные о пользователе
    }
    return jsonify(user_data)

if __name__ == '__main__':
    app.run(port=5001)
```

Микросервис постов

Теперь, предположим, что у вас есть микросервис постов, который хочет получить данные о пользователе для каждого поста. Вы можете использовать HTTP-запрос для обращения к микросервису пользователей:

```
import requests

# URL микросервиса пользователей
USERS_SERVICE_URL = 'http://users-service:5001/users/'

# Пример эндпоинта, возвращающего посты с данными о пользователях
@app.route('/posts', methods=['GET'])
def get_posts_with_user_data():
    # Проверка наличия токена в заголовках запроса
    auth_token = request.headers.get('Authorization')
    if not auth_token:
        return jsonify({'error': 'Authorization token is missing'}), 401
```

```
    # Логика получения данных о постах из базы данных или
другого источника
    posts = [
        {
            'post_id': 1,
            'content': 'Lorem ipsum',
            'user_id': 123
        },
        {
            'post_id': 2,
            'content': 'Dolor sit amet',
            'user_id': 456
        }
        # Другие посты
    ]

    # Запрос данных о пользователях для каждого поста
    for post in posts:
        user_id = post['user_id']
                                              user_response     =
requests.get(f'{USERS_SERVICE_URL}{user_id}',
headers={'Authorization': auth_token})
        user_data = user_response.json()
        post['user_data'] = user_data

    return jsonify(posts)

if __name__ == '__main__':
    app.run(port=5002)
```

В этом примере микросервис постов делает HTTP-запросы к микросервису пользователей, передавая идентификатор пользователя из данных о посте. После этого он включает данные о пользователе в каждый пост и возвращает их в виде ответа.

Обратите внимание, что в реальной системе вы также должны учитывать обработку ошибок, обеспечение безопасности и эффективную обработку запросов.

Разделение секретных ключей

В традиционных системах аутентификации и авторизации с использованием токенов, каждый микросервис обычно имеет свой уникальный секретный ключ для подписи и верификации токенов. Разделение секретных ключей для каждого микросервиса увеличивает безопасность системы. Вот почему:

1. Принцип минимальных привилегий

Каждый микросервис должен иметь только те права, которые ему действительно необходимы. Если один микросервис получит доступ к секретному ключу другого, это может повлечь за собой серьезные безопасностные риски.

2. Изоляция

Использование разных ключей обеспечивает изоляцию. Компрометация одного ключа не должна автоматически означать компрометацию всех микросервисов.

3. Управление и обслуживание

Раздельные ключи облегчают управление и обслуживание. В случае необходимости изменения ключа, это можно сделать для одного микросервиса, не затрагивая другие.

4. Аудит и мониторинг

Использование различных ключей упрощает аудит и мониторинг. Вы можете легко определить, какой микросервис создал или проверил тот или иной токен.

Примеры реальных сервисов аутентификации, таких как Auth0 или Firebase Authentication, предоставляют уникальные секретные ключи для каждого клиента или проекта.

Важно также следить за безопасностью хранения секретных ключей и использовать средства и методы, предназначенные для безопасного хранения, такие как инструменты управления секретами в облачных сервисах.

Проверка валидности токена из одного микросервиса в другом

Микросервис постов обычно не должен знать секретный ключ микросервиса пользователей для валидации токена. По сути, микросервис постов, который хочет проверить токен, может использовать открытый ключ (публичный ключ) микросервиса пользователей для проверки подписи токена.

Процесс может выглядеть следующим образом:

1. **Генерация токена:**
   Микросервис пользователей, при выдаче токена, подписывает его с использованием своего секретного ключа.

2. **Проверка токена в микросервисе постов:**
   Микросервис постов, получив токен, может запросить у микросервиса пользователей его открытый ключ (публичный ключ).

3. **Проверка подписи с использованием открытого ключа:**
   Микросервис постов использует открытый ключ микросервиса пользователей для проверки подписи токена.

Если подпись верна, это подтверждает, что токен был подписан с использованием секретного ключа микросервиса пользователей и не был изменен после этого.

Этот подход основан на асимметричной криптографии, где пара ключей состоит из открытого ключа (используемого для проверки подписи) и секретного ключа (используемого для создания подписи). Открытый ключ распространяется открыто, а секретный ключ хранится в безопасности.

Такой метод обеспечивает безопасность, поскольку микросервис постов не имеет доступа к секретному ключу, и проверка токена может происходить без раскрытия конфиденциальной информации.

В реальной реализации это может зависеть от используемых библиотек и языка программирования.

Ниже приведен пример использования библиотеки PyJWT на языке Python для создания и проверки JWT с использованием асимметричных ключей:

Микросервис пользователей:

```
import jwt

# Пример функции для генерации JWT
def generate_jwt(user_id):
    private_key = open('path/to/private_key.pem', 'r').read()
       token = jwt.encode({'user_id': user_id}, private_key,
algorithm='RS256')
    return token

# В реальном приложении открытый ключ следует распространять
публично
# Здесь приведен только пример использования
public_key = open('path/to/public_key.pem', 'r').read()

# Пример функции для проверки JWT
def validate_jwt(token):
    try:
                decoded_data = jwt.decode(token, public_key,
algorithms='RS256')
        return decoded_data
    except jwt.ExpiredSignatureError:
        # Обработка истекшего токена
        return None
    except jwt.InvalidTokenError:
        # Обработка недействительного токена
        return None
```

Микросервис постов:

```
import requests
import jwt

# URL микросервиса пользователей
USERS_SERVICE_URL = 'http://users-service:5001/'

# Пример функции для получения открытого ключа микросервиса
пользователей
def get_public_key():
```

```
    response = requests.get(USERS_SERVICE_URL + 'public-key')
    public_key = response.text
    return public_key

# Пример функции для проверки токена
def validate_token(token):
    public_key = get_public_key()
    try:
                decoded_data = jwt.decode(token, public_key,
algorithms='RS256')
        return decoded_data
    except jwt.ExpiredSignatureError:
        # Обработка истекшего токена
        return None
    except jwt.InvalidTokenError:
        # Обработка недействительного токена
        return None
```

Важно отметить, что в реальной среде нужно обеспечивать безопасное хранение и передачу ключей, а также регулярно обновлять и перевыпускать ключи для поддержания безопасности.

Изменение секретного ключа в микросервисе

Если поменять секретный ключ в микросервисе постов, это повлияет на возможность валидации и верификации токенов, подписанных предыдущим ключом. Все ранее выданные токены, подписанные старым ключом, станут недействительными при использовании нового ключа.

Когда микросервис постов попытается проверить токен с использованием нового ключа, подпись не совпадет, и валидация токена не пройдет.

Это означает, что пользователи, которые были авторизованы с использованием токенов, подписанных старым ключом, должны будут повторно аутентифицироваться и получить новый токен с использованием нового ключа.

При изменении секретного ключа обычно советуется выполнить следующие шаги:

1. Уведомление пользователей

Пользователям нужно предоставить информацию о необходимости повторной аутентификации из-за изменения ключа.

2. Плавный переход

Постепенно перевыпускайте токены с использованием нового ключа при следующих запросах пользователя, чтобы снизить влияние на опыт пользователей.

3. Обновление всех токенов:

В процессе перехода, удостоверьтесь, что все токены, хранящиеся на стороне клиента (например, в куках или локальном хранилище), также обновляются с использованием нового ключа.

Важно также оценить потенциальные риски и последствия изменения секретного ключа, особенно в зависимости от конкретной архитектуры и потребностей вашей системы.