

Реализация системы методов авторизации через JWT-токен с асимметричным ключом шифрования

Вихляев Дмитрий Романович

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В данной статье представлена реализация серверной системы авторизации пользователей, основанной на механизме JWT с асимметричным шифрованием. Для реализации используется язык JavaScript Nodejs. Результатом исследования станет подробное описание функций и методов этапов разработки.

Ключевые слова: асимметричное шифрование JWT, генерация ключа шифрования, Base64.

Implementation of a system of authorization methods via a JWT token with an asymmetric encryption key

Vikhlyayev Dmitry Romanovich

Sholom-Aleichem Priamursky State University

Student

Abstract

This article presents the implementation of a server-based user authorization system based on the JWT mechanism with asymmetric encryption. The JavaScript Node js language is used for implementation. The result of the research will be a detailed description of the functions and methods of the development stages.

Keywords: JWT asymmetric encryption, encryption key generation, Base64.

1 Введение

1.1 Актуальность

Использование асимметричного шифрования для генерации и верификации JWT (JSON Web Token) представляет собой важный аспект обеспечения безопасности в современных системах аутентификации и авторизации. Асимметричное шифрование предполагает использование двух различных ключей: приватного, который используется для подписания токенов, и публичного, предназначенного для проверки этих подписей. Преимущества использования асимметричного шифрования заключаются в повышенной безопасности, поскольку приватный ключ не покидает сервер, а публичный ключ может безопасно распространяться среди различных сервисов. Такой подход позволяет организовать децентрализованную проверку токенов, обеспечивая масштабируемость системы. Более того,

асимметричное шифрование позволяет избежать необходимости обмена приватными ключами, что минимизирует риски компрометации безопасности.

1.2 Обзор исследований

К.Ю.Филатов описал использование токенов на примере JWT в spring boot REST api приложении [1]. С.В.Козлов, М.С.Воробьев создал аутентификацию пользователя в web-приложении с использованием стандарта JWT [2]. Г.Г.Шамян провёл сравнительный анализ методологий авторизации JWT и sessions [3]. С.Н.Девицына, П.В.Пилькевич, Удод Е.В. представили способы улучшения защищённости сервисов, использующих JWT-токены [4]. В своей работе М.В.Бабичева показал примеры Атак на jwt токены [5].

1.3 Цель исследования

Цель исследования – описать реализацию функций и методов создания JWT-токена на основе ассиметричного ключа шифрования.

2 Материалы и методы

Для реализации используется язык программирования JavaScript на основе Nodejs и встроенный модуль криптографических функций.

3 Результаты и обсуждения

В качестве алгоритмов для создания подписи чаще всего используется RSA с алгоритмом SHA-256. Чтобы сгенерировать ключи, администратор или программист должен на собственной или защищённой машине создать пару ключей приватный и публичный. Представленная на рисунке 1, функция реализует проверку наличия файлов приватного и публичного ключей в системе и, при их отсутствии, генерирует новую пару ключей RSA, сохраняет их в файлы, а также шифрует приватный ключ с использованием пароля.

В этом примере используется алгоритм шифрования RSA с размером ключа 2048 бит. При генерации ключей задаются параметры кодирования для публичного ключа, соответствующие формату `spki`, и для приватного ключа – формат `pkcs8`. Генерация ключей возвращает объект, содержащий приватный и публичный ключи, которые затем сохраняются в файлы. В случае успешной генерации и сохранения ключей также вызывается дополнительная функция, которая отвечает за шифрование приватного ключа с использованием пароля, который передаётся через переменную окружения (рис.1).

```
// Пути для хранения ключей
const privateKeyPath = path.join(__dirname, 'private.pem');
const publicKeyPath = path.join(__dirname, 'public.pem');

// Проверка наличия ключей, генерация если их нет
function ensureKeys() {
  if (!fs.existsSync(privateKeyPath) || !fs.existsSync(publicKeyPath)) {
    console.log('Ключи не найдены. Генерация новой пары ключей...');
    const { privateKey, publicKey } = crypto.generateKeyPairSync('rsa', {
      modulusLength: 2048, // Размер ключа
      publicKeyEncoding: {
        type: 'spki',
        format: 'pem',
      },
      privateKeyEncoding: {
        type: 'pkcs8',
        format: 'pem',
      },
    });

    // Сохранение ключей в файлы
    fs.writeFileSync(privateKeyPath, privateKey);
    fs.writeFileSync(publicKeyPath, publicKey);
    console.log('Ключи успешно сгенерированы и сохранены.');
```

Рис. 1. Генерация пары ключей

Для реализации процесса шифрования приватного ключа используется алгоритм AES-256-CBC. Алгоритм требует вектор инициализации (IV). Вектор инициализации создаётся с помощью функции генерирующей 16 байтов случайных данных, что соответствует требованию алгоритма AES-256-CBC. Затем для шифрования пароля используется хеширование с алгоритмом SHA-256. Это необходимо для того, чтобы привести пароль к фиксированной длине, подходящей для использования в алгоритме AES-256, который требует 32 байта в качестве ключа. Зашифрованные данные и вектор инициализации сохраняются в объект, который включает два поля: iv, представляющее вектор инициализации в формате base64, и data, содержащее сам зашифрованный приватный ключ, также в формате base64. Зашифрованный объект записывается в файл private.pem.enc в формате JSON (рис.2,3).

```
// Функция для шифрования приватного ключа
function encryptPrivateKey(password) {
  // Чтение исходного приватного ключа
  const privateKey = fs.readFileSync('./private.pem', 'utf8');

  // Генерация случайного вектора инициализации (IV)
  const iv = crypto.randomBytes(16); // AES-256-CBC требует 16 байт IV

  // Хешируем пароль до 32 байтов для AES-256
  const key = crypto.createHash('sha256').update(password).digest();

  // Создание шифратора
  const cipher = crypto.createCipheriv('aes-256-cbc', key, iv);

  // Шифрование данных
  let encrypted = cipher.update(privateKey, 'utf8', 'base64');
  encrypted += cipher.final('base64');

  // Сохранение зашифрованного ключа в файл
  const encryptedData = {
    iv: iv.toString('base64'), // сохраняем IV в base64
    data: encrypted // зашифрованный приватный ключ
  };

  // Запись зашифрованного ключа в файл
  fs.writeFileSync('./private.pem.enc', JSON.stringify(encryptedData), 'utf8');
  console.log('Private key encrypted and saved to private.pem.enc');
}
```

Рис. 2. Шифрование приватного ключа

```
function decryptPrivateKey(password) {
  // Чтение зашифрованного файла
  const encryptedData = JSON.parse(fs.readFileSync('./private.pem.enc', 'utf8'));

  // Извлечение IV и зашифрованных данных
  const iv = Buffer.from(encryptedData.iv, 'base64');
  const encryptedKey = encryptedData.data;

  // Хешируем пароль до 32 байтов для AES-256
  const key = crypto.createHash('sha256').update(password).digest();

  // Создание расшифровщика
  const decipher = crypto.createDecipheriv('aes-256-cbc', key, iv);

  // Расшифровка данных
  let decrypted = decipher.update(encryptedKey, 'base64', 'utf8');
  decrypted += decipher.final('utf8');

  return decrypted;
}
```

Рис. 3. Расшифровка приватного ключа

Процесс создания JWT с использованием асимметричного шифрования начинается с формирования токена, который состоит из трех частей: заголовка, полезной нагрузки и подписи. Заголовок обычно содержит информацию об алгоритме шифрования, например, RS256, что указывает на использование алгоритма RSA с хешированием SHA-256. Полезная нагрузка содержит данные, которые могут быть использованы для аутентификации или авторизации пользователя, например, идентификатор пользователя или

его роль в системе. Подпись создается путем хеширования объединения заголовка и полезной нагрузки с использованием приватного ключа.

На специальном сервере авторизации (auth-server), размещается зашифрованный публичный ключ и пароль для расшифровки. Для создания нового токена функция извлекает пароль для дешифрования приватного ключа из переменной окружения. Затем с использованием этого пароля вызывается метод (рис.3), который восстанавливает приватный ключ. Этот ключ используется для подписи токена. Затем формируется заголовок токена, который кодируется в формат base64url. После этого, в зависимости от наличия параметра expiresIn, добавляется время истечения срока действия токена (exp) в payload. Если параметр expiresIn указан, токен будет действовать указанное количество секунд после текущего времени, которое также вычисляется с помощью `Math.floor(Date.now() / 1000)` (преобразование текущего времени в секунды). Время создания токена (iat) добавляется в payload, чтобы указать момент его выпуска. Создаётся подпись для токена. Для этого заголовок и payload конкатенируются через точку, а результат передаётся в функцию sign (рис. 4).

```
function createToken(payload, expiresIn) {
  const password = process.env.KEY_PASSWORD;
  const privateKey = decryptPrivateKey(password);
  //const { privateKey } = loadKeys();
  const header = {
    alg: 'RS256',
    typ: 'JWT',
  };

  const now = Math.floor(Date.now() / 1000);
  payload.iat = now;
  if (expiresIn) {
    payload.exp = now + expiresIn;
  }

  const encodedHeader = base64urlEncode(JSON.stringify(header));
  const encodedPayload = base64urlEncode(JSON.stringify(payload));
  const signature = sign(`${encodedHeader}.${encodedPayload}`, privateKey);

  return `${encodedHeader}.${encodedPayload}.${signature}`;
}
```

Рис. 4. Генерация JWT-токена

Функция sign предназначена для создания подписи с приватного ключа, указав формат вывода в base64. Это позволяет получить подписанное значение в формате строки, пригодном для передачи и использования в JWT. После создания подписи выполняются дополнительные операции с заменой символов, что необходимо для соответствия стандарту base64url, используемому в JWT, где эти символы должны быть изменены для корректной работы с URL.

Функция base64urlEncode выполняет кодирование данных в формат base64url с помощью строчковых методов JavaScript (рис. 5).

```
function sign(data, privateKey) {  
  return crypto  
    .createSign('RSA-SHA256')  
    .update(data)  
    .sign(privateKey, 'base64')  
    .replace(/=+$/, '')  
    .replace(/\+/g, '-')  
    .replace(/\//g, '_');  
}  
  
function base64urlEncode(data) {  
  return Buffer.from(data)  
    .toString('base64')  
    .replace(/=+$/, '')  
    .replace(/\+/g, '-')  
    .replace(/\//g, '_');  
}
```

Рис. 5. Кодирование Base64URL

Для верификации токена другой сервис, предоставляющий защищённые данные клиенту использует публичный ключ для проверки подписи, что подтверждает подлинность данных и их целостность.

Проверяется валидность JWT-токена, куда включены проверка подписи, срока действия и структуры токена. Сначала происходит загрузка публичного ключа. Токен разделяется на три части: заголовок, полезная нагрузка (payload) и подпись. Затем каждая из частей токена декодируются с помощью функции `base64urlDecode`. Заголовок и полезная нагрузка преобразуются в объекты JSON. Важно, что в процессе проверки алгоритма токена проверяется, что он использует алгоритм RS256. Если алгоритм отличается, возвращается ошибка с сообщением 'Unsupported algorithm'. После этого производится верификация подписи с использованием публичного ключа. Проверяется срок действия токена. Если в полезной нагрузке имеется параметр `exp` (время истечения срока действия токена), то вычисляется текущее время в секундах и сравнивается с этим значением. Если токен истёк, возвращается ошибка 'Token expired' (рис. 6).

```
function verifyToken(token) {
  const { publicKey } = loadKeys();
  const parts = token.split('.');
  if (parts.length !== 3) {
    return { valid: false, error: 'Malformed token' };
  }

  const [encodedHeader, encodedPayload, signature] = parts;

  const header = JSON.parse(base64urlDecode(encodedHeader));
  if (header.alg !== 'RS256') {
    return { valid: false, error: 'Unsupported algorithm' };
  }

  const payload = JSON.parse(base64urlDecode(encodedPayload));

  // Проверка подписи
  const isValidSignature = verify(`${encodedHeader}.${encodedPayload}`, signature, publicKey);
  if (!isValidSignature) {
    return { valid: false, error: 'Invalid signature' };
  }

  const now = Math.floor(Date.now() / 1000);
  if (payload.exp && now > payload.exp) {
    return { valid: false, error: 'Token expired' };
  }

  return { valid: true, payload };
}
```

Рис. 6. Проверка подлинности JWT-токена

Функция `verify` отвечает за проверку подписи токена. Первоначально строка подписи декодируется из формата `base64url` в стандарт `base64`. В JWT формат `base64url` используется для безопасной передачи данных в URL, и для этого необходимо заменить символы `-` на `+` и `_` на `/`. Также добавляется недостающий символ `=` для выравнивания длины строки до кратности 4, что необходимо для корректного декодирования `base64`.

Для самой верификации подписи используется метод, который проверяет корректность подписи, используя публичный ключ и закодированную подпись в формате `base64`.

Функция `base64urlDecode` выполняет декодирование строки из формата `base64url` в `base64` (рис. 7).

```
function verify(data, signature, publicKey) {
  const decodedSignature = signature
    .replace(/-/g, '+')
    .replace(/_/g, '/')
    + '='.repeat((4 - signature.length % 4) % 4);

  return crypto
    .createVerify('RSA-SHA256')
    .update(data)
    .verify(publicKey, decodedSignature, 'base64');
}

function base64urlDecode(data) {
  const padded = data + '='.repeat((4 - data.length % 4) % 4);
  return Buffer.from(padded.replace(/-/g, '+').replace(/_/g, '/'), 'base64').toString();
}
```

Рис. 7. Декодирование JWT-токена

4 Выводы

В результате исследования было рассмотрено проектирование методов для создания среды авторизации пользователей через JWT-токен с использованием пары ассиметричных ключей. Использование асимметричного шифрования для создания и проверки JWT-токенов является основным элементом обеспечения безопасности в распределенных системах, предоставляющих доступ к защищенным данным и ресурсам.

Библиографический список

1. Филатов К.Ю. Использование токенов на примере JWT в spring boot REST api приложении // Информационные технологии моделирования и управления. 2022. Т. 128. № 2. С. 153-157.
2. Козлов С.В., Воробьев М.С. Реализация аутентификации пользователя в web-приложении с использованием стандарта JWT // Системы компьютерной математики и их приложения. 2022. № 23. С. 362-366.
3. Шамян Г.Г. Сравнительный анализ методологий авторизации JWT и sessions // Студенческий вестник. 2022. № 19-16 (211). С. 15-17.
4. Девыцина С.Н., Пилькевич П.В., Удод Е.В. Способы улучшения защищённости сервисов, использующих JWT-токены // Экономика. Информатика. 2023. Т. 50. № 1. С. 144-151.
5. Бабичева М.В. Атаки на jwt токены // В сборнике: Донецкие чтения 2022: образование, наука, инновации, культура и вызовы современности. Материалы VII Международной научной конференции, посвящённой 85-летию Донецкого национального университета. Под общей редакцией С.В. Беспаловой. Донецк, 2022. С. 176-178.
6. Попов В.С. Авторизация пользователя angular веб приложения с использованием JWT-токена // В сборнике: Научное сообщество студентов XXI столетия. технические науки. сборник статей по материалам LXXXIX студенческой международной научно-практической конференции. Новосибирск, 2020. С. 133-138.
7. Колесников А.О. Идентификация пользователей клиент-серверных приложений с помощью JWT-токена // В сборнике: EurasiaScience. Сборник статей XXXVI международной научно-практической конференции. Москва, 2021. С. 42-43.
8. Абдурайимов Л.Н. Аутентификация в REST-приложениях с использованием spring security и JWT // Информационно-компьютерные технологии в экономике, образовании и социальной сфере. 2021. № 4 (34). С. 5-13.