API Authorization and Role-Based Access Control (RBAC) are essential components for managing who can access specific resources or perform actions within your application. In Node.js, you can implement RBAC using middleware to check user roles and permissions.

**Steps to Implement API Authorization & RBAC in Node.js**

1. **User Authentication**: First, the user must be authenticated (e.g., through JWT, session tokens, etc.).
2. **Role-Based Access Control (RBAC)**: Once authenticated, roles are checked to determine if a user has permission to access specific routes or perform certain actions.

**Concepts for RBAC Implementation**

1. **Roles**: Define user roles (e.g., Admin, User, Manager, etc.).
2. **Permissions**: Define specific actions or resources a role can access (e.g., CREATE\_USER, VIEW\_REPORTS, etc.).
3. **Role-Permission Mapping**: A user may have one or more roles, and each role is associated with specific permissions.

**1. Database Design for Roles and Permissions**

**Roles Table**

CREATE TABLE roles (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(255) NOT NULL, -- Role name (e.g., Admin, User)

description TEXT

);

**Permissions Table**

CREATE TABLE permissions (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(255) NOT NULL, -- Permission name (e.g., CREATE\_USER, VIEW\_REPORTS)

description TEXT

);

**User Roles Table (Mapping Users to Roles)**

CREATE TABLE user\_roles (

id INT AUTO\_INCREMENT PRIMARY KEY,

user\_id INT NOT NULL, -- Foreign key to users table

role\_id INT NOT NULL, -- Foreign key to roles table

FOREIGN KEY (user\_id) REFERENCES users(id),

FOREIGN KEY (role\_id) REFERENCES roles(id),

UNIQUE (user\_id, role\_id)

);

**Role Permissions Table (Mapping Roles to Permissions)**

CREATE TABLE role\_permissions (

id INT AUTO\_INCREMENT PRIMARY KEY,

role\_id INT NOT NULL, -- Foreign key to roles table

permission\_id INT NOT NULL, -- Foreign key to permissions table

FOREIGN KEY (role\_id) REFERENCES roles(id),

FOREIGN KEY (permission\_id) REFERENCES permissions(id),

UNIQUE (role\_id, permission\_id)

);

**2. Setup Middleware for RBAC in Node.js**

Here’s a step-by-step guide to implementing the RBAC system with API authorization in a Node.js application using JWT tokens for authentication.

**Install Required Libraries**

1. jsonwebtoken – For creating and verifying JWT tokens.
2. express-jwt – For verifying JWT tokens in the request.
3. mysql2 – For connecting to MariaDB.

npm install jsonwebtoken express-jwt mysql2

**User Authentication Middleware (JWT)**

Here, the middleware will verify the JWT token, and extract the user data.

// authMiddleware.js

const jwt = require('jsonwebtoken');

const secretKey = 'your-secret-key'; // Store it securely

// Middleware to authenticate JWT token

const authenticate = (req, res, next) => {

const token = req.header('Authorization')?.replace('Bearer ', '');

if (!token) {

return res.status(401).json({ message: 'Access denied. No token provided.' });

}

try {

const decoded = jwt.verify(token, secretKey);

req.user = decoded; // Attach user info to the request

next();

} catch (error) {

return res.status(401).json({ message: 'Invalid token' });

}

};

module.exports = authenticate;

**RBAC Middleware**

The checkPermissions middleware will check if the user has the necessary permissions based on their roles.

// rbacMiddleware.js

const pool = require('./db'); // MySQL connection pool

// Middleware to check user's permissions based on roles

const checkPermissions = (requiredPermission) => {

return async (req, res, next) => {

try {

const { user\_id } = req.user; // User info decoded from JWT

// Get all roles of the user

const [userRoles] = await pool.query(

'SELECT r.name FROM user\_roles ur JOIN roles r ON ur.role\_id = r.id WHERE ur.user\_id = ?',

[user\_id]

);

if (userRoles.length === 0) {

return res.status(403).json({ message: 'No roles assigned to user.' });

}

// Check if user has the required permission in any of their roles

const roleNames = userRoles.map(role => role.name);

const [permissions] = await pool.query(

`SELECT p.name FROM role\_permissions rp

JOIN permissions p ON rp.permission\_id = p.id

JOIN roles r ON rp.role\_id = r.id

WHERE r.name IN (?) AND p.name = ?`,

[roleNames, requiredPermission]

);

if (permissions.length > 0) {

return next(); // User has the required permission

}

return res.status(403).json({ message: 'You do not have permission for this action.' });

} catch (error) {

console.error(error);

return res.status(500).json({ message: 'Internal server error' });

}

};

};

module.exports = checkPermissions;

**3. Example Routes with Role-Based Access Control**

Now, let's create some example routes that implement the RBAC system.

**Route to Create Admin User**

// adminRoutes.js

const express = require('express');

const authenticate = require('./authMiddleware');

const checkPermissions = require('./rbacMiddleware');

const pool = require('./db');

const router = express.Router();

// Route to create admin user (only users with CREATE\_USER permission can access)

router.post('/create-admin', authenticate, checkPermissions('CREATE\_USER'), async (req, res) => {

const { username, password, role } = req.body;

try {

const result = await pool.query(

'INSERT INTO users (username, password, role) VALUES (?, ?, ?)',

[username, password, role]

);

res.json({ message: 'Admin user created successfully' });

} catch (error) {

console.error(error);

res.status(500).json({ message: 'Internal server error' });

}

});

module.exports = router;

**Route to View Reports (Only Admins can access)**

// reportRoutes.js

const express = require('express');

const authenticate = require('./authMiddleware');

const checkPermissions = require('./rbacMiddleware');

const router = express.Router();

// Route to view reports (only users with VIEW\_REPORTS permission can access)

router.get('/view-reports', authenticate, checkPermissions('VIEW\_REPORTS'), (req, res) => {

res.json({ message: 'Here are the reports.' });

});

module.exports = router;

**4. Combining Everything in the Main App**

In the main app file (app.js), we combine all the middleware and routes.

// app.js

const express = require('express');

const bodyParser = require('body-parser');

const adminRoutes = require('./adminRoutes');

const reportRoutes = require('./reportRoutes');

const app = express();

app.use(bodyParser.json()); // For parsing application/json

app.use('/api/admin', adminRoutes);

app.use('/api/reports', reportRoutes);

const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {

console.log(`Server running on port ${PORT}`);

});

**5. Generating JWT Token for Users**

Here’s how you can generate JWT tokens when a user logs in:

// authController.js

const jwt = require('jsonwebtoken');

const secretKey = 'your-secret-key'; // Store it securely

const loginUser = async (req, res) => {

const { username, password } = req.body;

try {

// Query the database to verify the user credentials (simplified)

const [user] = await pool.query('SELECT \* FROM users WHERE username = ? AND password = ?', [username, password]);

if (user.length === 0) {

return res.status(400).json({ message: 'Invalid credentials' });

}

// Generate JWT token

const token = jwt.sign({ user\_id: user[0].id, username: user[0].username }, secretKey, { expiresIn: '1h' });

res.json({ token });

} catch (error) {

console.error(error);

res.status(500).json({ message: 'Internal server error' });

}

};

module.exports = loginUser;

**Conclusion**

By combining the **JWT-based authentication** with **RBAC middleware**, we've created a system where:

1. **Admin users** can manage other users and roles.
2. **Roles** determine the permissions available for each user.
3. **API authorization** is controlled via middleware that checks whether a user has the necessary permissions to access a specific resource or perform an action.

This setup allows flexible, role-based access to your application and API endpoints, ensuring that only users with the appropriate permissions can execute certain actions.

To implement **JWE (JSON Web Encryption)** for user authentication and authorization in Node.js, we need to encrypt and decrypt the user's data (such as their ID, role, etc.) inside the token using public and private keys. This ensures that only the parties with the corresponding private or public keys can read or validate the encrypted data, providing better security compared to just using JWT for signing.

Here’s a step-by-step guide to implement **JWE** for user authentication, role-based access control (RBAC), and API authorization in Node.js.

**1. JWE Overview**

JWE is a specification from the **JWT (JSON Web Token)** family that encrypts the token's payload (in addition to signing it, like JWT). This ensures that only the authorized recipient with the private key can decrypt and access the information.

* **JWT** typically just signs the payload (ensuring integrity), while **JWE** encrypts the payload (ensuring confidentiality).
* We'll use **RSA** public/private keys for encryption and decryption.

**2. Libraries Required**

You will need the following libraries:

* node-jose (or jose) for handling JWE encryption and decryption.
* express for the API server.
* mysql2 for database connections.
* jsonwebtoken for JWT signing (to combine JWE and JWT features).

**Install required packages:**

npm install express jose mysql2

**3. Setup JWE (Public/Private Key Generation)**

First, generate **RSA keys** (private and public) to be used for JWE encryption and decryption.

You can generate RSA keys using OpenSSL or other tools. Example command for generating keys:

openssl genpkey -algorithm RSA -out private\_key.pem -pkeyopt rsa\_keygen\_bits:2048

openssl rsa -pubout -in private\_key.pem -out public\_key.pem

**4. Example JWE Implementation in Node.js**

**a. Database Design**

You will still need the same database tables for managing users, roles, permissions, and mappings.

1. **Users Table** - stores user info.
2. **Roles Table** - stores role definitions.
3. **Permissions Table** - stores permission definitions.
4. **User-Roles Mapping Table** - maps users to roles.
5. **Role-Permissions Mapping Table** - maps roles to permissions.

Refer to the previous response for database table design.

**b. JWE Encryption and Decryption Logic**

Here is how you can encrypt and decrypt data using JWE in Node.js.

**1. Setup JWE Utilities**

// jweUtils.js

const jose = require('jose');

const fs = require('fs');

// Load RSA public and private keys

const privateKey = fs.readFileSync('path\_to\_private\_key.pem');

const publicKey = fs.readFileSync('path\_to\_public\_key.pem');

// Encrypt data (for creating the JWE token)

const encrypt = (data) => {

const jwe = jose.JWE.encrypt(data, publicKey, {

alg: 'RSA-OAEP', // Encryption algorithm

enc: 'A256GCM', // Content encryption algorithm

});

return jwe;

};

// Decrypt data (for verifying the JWE token)

const decrypt = (jweToken) => {

const decrypted = jose.JWE.decrypt(jweToken, privateKey);

return decrypted.toString('utf8');

};

module.exports = { encrypt, decrypt };

**2. Middleware for Authentication and JWE Validation**

The following middleware will authenticate the user based on JWE token in the request, decrypt the token, and verify the user information.

// authMiddleware.js

const { decrypt } = require('./jweUtils');

const authenticate = (req, res, next) => {

const token = req.header('Authorization')?.replace('Bearer ', '');

if (!token) {

return res.status(401).json({ message: 'Access denied. No token provided.' });

}

try {

const decryptedData = decrypt(token);

const user = JSON.parse(decryptedData); // Assuming user data was JSON serialized before encryption

req.user = user; // Attach decrypted user data to the request object

next();

} catch (error) {

return res.status(401).json({ message: 'Invalid token' });

}

};

module.exports = authenticate;

**3. RBAC Middleware**

In this example, the checkPermissions middleware will check if the user has the necessary permission (based on their roles) for the requested action.

// rbacMiddleware.js

const pool = require('./db'); // MySQL connection pool

const checkPermissions = (requiredPermission) => {

return async (req, res, next) => {

try {

const { user\_id } = req.user; // User info decrypted from JWE token

// Get all roles of the user

const [userRoles] = await pool.query(

'SELECT r.name FROM user\_roles ur JOIN roles r ON ur.role\_id = r.id WHERE ur.user\_id = ?',

[user\_id]

);

if (userRoles.length === 0) {

return res.status(403).json({ message: 'No roles assigned to user.' });

}

const roleNames = userRoles.map(role => role.name);

// Check if user has the required permission in any of their roles

const [permissions] = await pool.query(

`SELECT p.name FROM role\_permissions rp

JOIN permissions p ON rp.permission\_id = p.id

JOIN roles r ON rp.role\_id = r.id

WHERE r.name IN (?) AND p.name = ?`,

[roleNames, requiredPermission]

);

if (permissions.length > 0) {

return next(); // User has the required permission

}

return res.status(403).json({ message: 'You do not have permission for this action.' });

} catch (error) {

console.error(error);

return res.status(500).json({ message: 'Internal server error' });

}

};

};

module.exports = checkPermissions;

**4. Example Route**

Here’s an example route where a user must be authenticated with a JWE token and have the appropriate role and permission to access it.

// adminRoutes.js

const express = require('express');

const authenticate = require('./authMiddleware');

const checkPermissions = require('./rbacMiddleware');

const pool = require('./db');

const router = express.Router();

// Route to create an admin user (only accessible if user has CREATE\_USER permission)

router.post('/create-admin', authenticate, checkPermissions('CREATE\_USER'), async (req, res) => {

const { username, password, role } = req.body;

try {

const result = await pool.query(

'INSERT INTO users (username, password, role) VALUES (?, ?, ?)',

[username, password, role]

);

res.json({ message: 'Admin user created successfully' });

} catch (error) {

console.error(error);

res.status(500).json({ message: 'Internal server error' });

}

});

module.exports = router;

**5. Example Route to Create and Decrypt the JWE Token on Login**

// authController.js

const { encrypt } = require('./jweUtils');

const pool = require('./db');

const loginUser = async (req, res) => {

const { username, password } = req.body;

try {

// Query the database to verify user credentials (simplified)

const [user] = await pool.query('SELECT \* FROM users WHERE username = ? AND password = ?', [username, password]);

if (user.length === 0) {

return res.status(400).json({ message: 'Invalid credentials' });

}

// Create JWT payload and encrypt it as JWE

const userPayload = {

user\_id: user[0].id,

username: user[0].username,

roles: user[0].roles, // Assuming roles are already stored in the user object

};

const jweToken = encrypt(JSON.stringify(userPayload)); // Encrypt user data into JWE token

res.json({ token: jweToken });

} catch (error) {

console.error(error);

res.status(500).json({ message: 'Internal server error' });

}

};

module.exports = loginUser;

**6. Final Setup in Express App**

// app.js

const express = require('express');

const bodyParser = require('body-parser');

const adminRoutes = require('./adminRoutes');

const authController = require('./authController');

const app = express();

app.use(bodyParser.json()); // For parsing application/json

app.use('/api/auth', authController); // Authentication route (login)

app.use('/api/admin', adminRoutes); // Admin related routes

const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {

console.log(`Server running on port ${PORT}`);

});

**7. Conclusion**

* **JWE** encrypts the token to ensure confidentiality of user information.
* The middleware checks for valid encrypted tokens and performs **RBAC** to ensure the user has the correct permissions.
* **Role-based access control (RBAC)** checks whether a user’s roles allow them to access the requested API endpoint.

This setup ensures that your API is both secure (using JWE encryption) and flexible (with RBAC for access control).

To generate a **public and private key pair** in memory for a specific **JTI (JWT ID) or unique ID**, and store it for **X hours** while keeping the **old one** available for **X-10 minutes** to validate previous keys, you can follow these steps:

**Overview of Steps:**

1. **Generate Public/Private Key Pair**: Use RSA key generation to create a public/private key pair.
2. **Store Keys in Memory**: Store the generated keys in memory (using a **Map** or **Redis** for persistence, if necessary).
3. **Expiration of Keys**: Set up expiration times for the keys, keeping the old key for validation.
4. **Validate with Previous Keys**: Allow validation against previous keys for up to **X-10 minutes**.

We will implement this in **Node.js**, using **RSA** keys and **memory-based storage** with a time-to-live (TTL) for the keys.

**1. Key Generation and Storage in Memory (with Expiration)**

We will use a **Map** in Node.js to store the keys temporarily. The map will store the **JTI** as the key, and the **value** will contain both the public and private keys with expiration timestamps.

**2. Steps in Code**

**a. Generate RSA Keys**

You can generate RSA keys using the **crypto** module in Node.js. We’ll use RSA keys for encryption and decryption.

**b. Store Keys in Memory with Expiration Logic**

We will use a Map to store the keys and their expiration times. We will also handle expiration using **setTimeout** to automatically remove keys after the defined TTL.

**Code Implementation**

const crypto = require('crypto');

// Store the keys in memory with their expiration times

const keysStore = new Map();

// Helper function to generate RSA keys

const generateKeyPair = () => {

return new Promise((resolve, reject) => {

crypto.generateKeyPair('rsa', {

modulusLength: 2048,

publicKeyEncoding: {

type: 'spki',

format: 'pem'

},

privateKeyEncoding: {

type: 'pkcs8',

format: 'pem'

}

}, (err, publicKey, privateKey) => {

if (err) {

reject(err);

} else {

resolve({ publicKey, privateKey });

}

});

});

};

// Function to generate and store keys with expiration times

const storeKeysWithExpiration = async (jti, ttl = 3600, previousKeyTtl = 600) => {

// Generate the key pair

const { publicKey, privateKey } = await generateKeyPair();

// Set the current expiration time for this key

const expirationTime = Date.now() + ttl \* 1000;

const previousExpirationTime = Date.now() + previousKeyTtl \* 1000;

// Store the key pair in memory with expiration times

keysStore.set(jti, {

publicKey,

privateKey,

expirationTime

});

// Store the previous key (this will be used for validation for the next X-10 minutes)

if (keysStore.size > 1) {

const previousJti = Array.from(keysStore.keys())[keysStore.size - 2]; // Get the previous JTI

const previousKey = keysStore.get(previousJti);

keysStore.set(previousJti, {

...previousKey,

expirationTime: previousExpirationTime // Update the expiration time of the previous key

});

}

// Set a timeout to automatically remove expired keys

setTimeout(() => {

keysStore.delete(jti);

}, ttl \* 1000);

// Set a timeout to automatically remove the previous key after X-10 minutes

if (keysStore.size > 1) {

const previousJti = Array.from(keysStore.keys())[keysStore.size - 2];

setTimeout(() => {

keysStore.delete(previousJti);

}, previousKeyTtl \* 1000);

}

console.log(`Keys for JTI ${jti} stored with expiration in ${ttl} seconds`);

};

// Function to get keys from memory (based on JTI)

const getKeys = (jti) => {

return keysStore.get(jti);

};

// Example Usage

const exampleUsage = async () => {

const jti1 = 'unique-jti-1';

const jti2 = 'unique-jti-2';

// Store keys for JTI1 with a TTL of 1 hour (3600 seconds) and previous key TTL of 10 minutes (600 seconds)

await storeKeysWithExpiration(jti1, 3600, 600);

// Store keys for JTI2 with the same TTL values

await storeKeysWithExpiration(jti2, 3600, 600);

// Retrieve keys by JTI (e.g., for validation)

const keysForJTI1 = getKeys(jti1);

console.log('Keys for JTI 1:', keysForJTI1);

const keysForJTI2 = getKeys(jti2);

console.log('Keys for JTI 2:', keysForJTI2);

};

exampleUsage();

**3. Key Management Details**

* **Key Generation**: The function generateKeyPair generates an RSA key pair using the crypto.generateKeyPair method.
* **Key Storage**: The storeKeysWithExpiration function stores the generated key pair along with the expiration timestamps in the keysStore map. The current key will expire after **X hours (3600 seconds)** and the previous key will expire after **X-10 minutes (600 seconds)**.
* **Expiration**: The setTimeout function is used to automatically delete the keys from memory once their expiration times are reached.
* **Access to Keys**: The getKeys function retrieves the keys from memory based on the JTI, and it can be used for validating subsequent requests.

**4. JWE Token Handling**

Once you have stored the keys in memory, you can generate a **JWE** token with the public key for encryption and use the private key to decrypt the data as required.

For example, when creating a JWE, you could use the public key corresponding to the jti for encryption and store the resulting token.

**JWE Token Generation Example**

const { JWE } = require('jose'); // Using the jose library to create JWE

const createJweToken = (jti, payload) => {

const keys = getKeys(jti);

if (!keys) {

throw new Error('No keys found for this JTI');

}

// Encrypt the payload with the public key

const encryptedToken = JWE.encrypt(JSON.stringify(payload), keys.publicKey, {

alg: 'RSA-OAEP', // Encryption algorithm

enc: 'A256GCM', // Content encryption algorithm

});

return encryptedToken;

};

**JWE Token Validation Example**

When validating the JWE token, you can decrypt it using the stored private key:

const validateJweToken = (jti, encryptedToken) => {

const keys = getKeys(jti);

if (!keys) {

throw new Error('No keys found for this JTI');

}

// Decrypt the token with the private key

const decryptedToken = JWE.decrypt(encryptedToken, keys.privateKey);

const payload = JSON.parse(decryptedToken.toString());

return payload;

};

**5. Conclusion**

* You now have a mechanism to generate and store RSA public and private key pairs in memory with expiration times.
* The current key will expire after the defined TTL (e.g., 1 hour), and the previous key will expire 10 minutes after that to allow validation of previous tokens.
* You can use these keys for **JWE** encryption and decryption to securely authenticate users.

This approach is flexible and scalable for scenarios where you need to manage short-lived encryption keys and validate them during a transition period (previous keys).

Yes, you can extract the \*\*JTI (JWT ID)\*\* from a \*\*JWE (JSON Web Encryption)\*\* token before decrypting it. The \*\*JTI\*\* is part of the \*\*JWT claims\*\* section and can be found inside the \*\*protected header\*\* of the \*\*JWE\*\* token.

To extract the \*\*JTI\*\* (or any claim) from a \*\*JWE\*\* token without decrypting it, you can access the \*\*JWE protected header\*\*. The protected header contains metadata about the encryption process, including information such as the algorithm used and the key ID (if applicable).

Here's how you can extract the \*\*JTI\*\* from the \*\*JWE\*\* token before decryption using the \*\*`jose`\*\* library:

### \*\*Steps to Extract the JTI from JWE\*\*

1. \*\*Parse the JWE Token\*\*: The \*\*JWE\*\* token is composed of \*\*5 parts\*\* (header, encrypted key, initialization vector, ciphertext, and authentication tag). You can parse the header to get information like \*\*JTI\*\*.

2. \*\*Access the Protected Header\*\*: The \*\*JTI\*\* claim is typically stored inside the JWT claims in the protected header.

### \*\*Code Example Using `jose` Library\*\*:

Here’s how you can extract the \*\*JTI\*\* from the JWE header before decrypting:

```js

const { JWE } = require('jose');

// Function to extract JTI from JWE token

const extractJtiFromJwe = (encryptedToken) => {

// Parse the JWE token

const parsedJwe = JWE.decode(encryptedToken);

// The JTI will be in the protected header

const protectedHeader = parsedJwe.protected;

// JTI is stored in the claims of the protected header

const jti = protectedHeader.jti;

return jti;

};

// Example usage

const encryptedToken = "your-jwe-token-here"; // Replace with actual JWE token

const jti = extractJtiFromJwe(encryptedToken);

console.log("Extracted JTI:", jti);

```

### \*\*Explanation:\*\*

- \*\*`JWE.decode(encryptedToken)`\*\*: The `decode` method is used to decode the \*\*JWE\*\* token without actually decrypting it. It will return the full content of the \*\*JWE\*\* token, including the \*\*protected header\*\*.

- \*\*`protectedHeader.jti`\*\*: The `jti` (JWT ID) is usually part of the \*\*protected header\*\* of the \*\*JWE\*\* token. By extracting it, you can verify which specific key (identified by the JTI) to use for decryption.

- \*\*Extracting JTI\*\*: After decoding the JWE token, you can access the `jti` in the protected header, which you can use for key lookup or logging.

### \*\*Key Points:\*\*

- The \*\*JTI\*\* (JWT ID) is typically set as a claim in the \*\*JWT\*\* part of the \*\*JWE token\*\*. It’s used to uniquely identify the token.

- You \*\*don’t need to decrypt\*\* the token to extract the \*\*JTI\*\*. You can simply decode the token’s header to retrieve this information.

- Extracting the \*\*JTI\*\* from the header is useful for tracking and validating tokens, especially when you need to match the token to a specific \*\*RSA key\*\* stored in memory or database.

### \*\*Decryption after JTI Extraction:\*\*

Once you've extracted the \*\*JTI\*\* from the header, you can use it to fetch the corresponding public/private key pair from your key store (like Redis or in-memory cache). After retrieving the correct key, you can proceed to decrypt the \*\*JWE\*\* payload.

```js

// Decrypt JWE after extracting the JTI

const decryptJweToken = (encryptedToken, privateKey) => {

// Decrypt the JWE token using the private key

const decryptedToken = JWE.decrypt(encryptedToken, privateKey);

const payload = JSON.parse(decryptedToken.toString());

return payload;

};

// Example usage:

const privateKey = "your-private-key-here"; // Replace with the appropriate private key

const decryptedPayload = decryptJweToken(encryptedToken, privateKey);

console.log("Decrypted Payload:", decryptedPayload);

```

### \*\*Conclusion:\*\*

You can easily extract the \*\*JTI\*\* from the \*\*JWE\*\* token by decoding the header without needing to decrypt the payload. This allows you to verify the \*\*JTI\*\* and identify which private/public key pair to use for decryption, making the process more efficient, especially in systems where key rotation or validation of previous keys is necessary.

To implement a pool of 5 RSA key pairs with unique jti IDs, and to handle the JWE (JSON Web Encryption) token generation and validation process based on these keys with overlapping TTL (Time to Live), you can follow the outlined approach below.

**Overview of Steps:**

1. **Generate 5 RSA Key Pairs** every **X minutes** and assign each key pair a **unique jti** (Key ID). These key pairs will be stored in memory (or Redis) with their expiration times.
2. **Key Management**: Store each RSA key pair in a map, keyed by jti. Ensure the keys have a TTL, and every **X - 10 minutes**, generate new keys, ensuring an overlap of 10 minutes.
3. **JWE Token Generation**: When generating a JWE token, use a specific jti for that particular RSA key pair. This jti will be embedded in the token's header.
4. **Token Validation**: On receiving a request with a JWE token, extract the jti from the token's claims, retrieve the corresponding RSA key from the cache, and validate the token. If the key is expired or not found, return a **401 Unauthorized** response.

**Implementation:**

1. **RSA Key Generation**
2. **Storing RSA Key Pairs with Unique jti**
3. **TTL and Overlap Management**
4. **JWE Token Generation and Validation**

**Step-by-Step Code:**

Let's implement the RSA key pair generation, storage, JWE creation, and validation, including the logic for the 10-minute overlap:

const crypto = require('crypto');

const { JWE } = require('jose');

const redis = require('redis');

// In-memory store or Redis store for key pairs

let keysStore = new Map(); // Using in-memory store

// If you want to use Redis instead, initialize a Redis client

// const redisClient = redis.createClient();

// Function to generate RSA Key pair

const generateKeyPair = () => {

return new Promise((resolve, reject) => {

crypto.generateKeyPair('rsa', {

modulusLength: 2048,

publicKeyEncoding: {

type: 'spki',

format: 'pem'

},

privateKeyEncoding: {

type: 'pkcs8',

format: 'pem'

}

}, (err, publicKey, privateKey) => {

if (err) {

reject(err);

} else {

resolve({ publicKey, privateKey });

}

});

});

};

// Function to generate and store 5 RSA keys with unique jti and TTL

const storeKeysWithExpiration = async (ttl = 3600) => {

const keyPairs = [];

const keyIds = [];

// Generate 5 RSA key pairs with unique jti

for (let i = 0; i < 5; i++) {

const { publicKey, privateKey } = await generateKeyPair();

const jti = `jti-${Date.now()}-${i}`; // Unique jti for each key

keyPairs.push({ jti, publicKey, privateKey });

keyIds.push(jti);

// Store each key pair with its expiration time

keysStore.set(jti, {

publicKey,

privateKey,

expirationTime: Date.now() + ttl \* 1000 // TTL in ms

});

}

// Set timeout to expire keys after TTL

setTimeout(() => {

keyIds.forEach(jti => {

keysStore.delete(jti); // Remove expired keys

console.log(`Key ${jti} expired and removed from store.`);

});

}, ttl \* 1000);

// Generate next set of keys after X-10 minutes

setTimeout(() => {

storeKeysWithExpiration(ttl); // Generate new keys for the next batch

}, (ttl - 600) \* 1000); // Generate new keys after X-10 minutes

console.log('Key pairs with unique jti stored and ready to use:', keyIds);

};

// Function to retrieve key pair by jti

const getKeyByJti = (jti) => {

return keysStore.get(jti);

};

// JWE Token Creation with `jti` as the key id

const createJweToken = (jti, payload) => {

const key = getKeyByJti(jti);

if (!key) {

throw new Error('No keys found for this JTI');

}

// Encrypt the payload with the public key

const encryptedToken = JWE.encrypt(JSON.stringify(payload), key.publicKey, {

alg: 'RSA-OAEP',

enc: 'A256GCM',

kid: jti // jti as the key id

});

return encryptedToken;

};

// JWE Token Validation (Decrypt using private key)

const validateJweToken = (jti, encryptedToken) => {

const key = getKeyByJti(jti);

if (!key) {

throw new Error('No keys found for this JTI');

}

// Decrypt the token using the private key

const decryptedToken = JWE.decrypt(encryptedToken, key.privateKey);

const payload = JSON.parse(decryptedToken.toString());

return payload;

};

// Example usage of creating and validating JWE token

const exampleUsage = async () => {

// Generate 5 keys for the first batch and store them

await storeKeysWithExpiration(3600); // 1 hour TTL

// Simulate creating a JWE token with a specific jti

setTimeout(async () => {

const jti = 'jti-1690407770000-1'; // Example jti

const payload = { user: 'John Doe', role: 'admin' };

// Create a JWE token using the jti

const encryptedToken = createJweToken(jti, payload);

console.log('Encrypted JWE Token:', encryptedToken);

// Simulate a request that validates the JWE token

setTimeout(() => {

try {

const decryptedPayload = validateJweToken(jti, encryptedToken);

console.log('Decrypted Payload:', decryptedPayload);

} catch (error) {

console.log('Error:', error.message);

}

}, 2000); // Validate the token after 2 seconds

}, 1000); // Wait for key pair generation

};

exampleUsage();

**How the Code Works:**

1. **Key Generation**:
   * Every **X minutes**, the storeKeysWithExpiration() function generates 5 RSA key pairs. Each key pair is assigned a unique jti ID.
   * These key pairs are stored in the keysStore (a Map in memory), where each entry is keyed by jti. This ensures that each key pair has a unique identifier.
2. **Key Expiration**:
   * Each key pair has an expiration time set when it is stored. The keys are removed from the keysStore after their TTL expires using setTimeout.
   * The next set of keys is generated **X-10 minutes** after the previous set, ensuring an overlap of key pairs (10 minutes).
3. **JWE Token Creation**:
   * When creating a JWE token, the public key corresponding to the jti is used to encrypt the payload. The jti is embedded in the kid (Key ID) field in the JWE header.
4. **JWE Token Validation**:
   * When a request with a JWE token is received, the jti is extracted from the token's header. The corresponding private key is then used to decrypt the token.
   * If the key is not found in the store (or has expired), an error is thrown, and a **401 Unauthorized** response can be sent.
5. **Key Rotation**:
   * Every **X - 10 minutes**, the function generates a new set of 5 RSA key pairs. After **X minutes**, a new set of 5 key pairs is generated, ensuring a 10-minute overlap of key pairs.

**Conclusion:**

* The system ensures that the RSA key pairs are rotated every **X minutes**, with an overlap of **10 minutes**.
* The **jti** ID is used as a unique identifier for each key pair, and it is embedded in the **JWE token**.
* When validating the token, the correct key pair is retrieved based on the jti and used for decryption.
* If the key pair is not found (or has expired), a **401 Unauthorized** error is thrown.

Yes, you can extract the jti (or any other claim) directly from the JWE token itself before attempting to decrypt it.

In the JWE standard, the jti (JWT ID) claim is typically included in the **header** of the token, and you can access it without needing to decrypt the payload.

**Steps to extract jti from a JWE token:**

1. **JWE Structure**:
   * JWE consists of 5 parts:
     1. **Header**
     2. **Encrypted Key**
     3. **Initialization Vector**
     4. **Ciphertext**
     5. **Authentication Tag**

The jti is part of the **Header**.

1. **Extract jti from JWE header**:
   * Before decrypting the JWE token, you can parse the header and extract the jti field.

**Example Implementation:**

const { JWE } = require('jose'); // jose library for JWE handling

// Function to extract the JTI from JWE token (without decrypting)

const extractJtiFromJwe = (encryptedToken) => {

const { protected: protectedHeader } = JWE.decode(encryptedToken); // Decode the JWE token

const header = JSON.parse(Buffer.from(protectedHeader, 'base64url').toString('utf8'));

return header.jti; // Extract the JTI from the header

};

// Function to decrypt the JWE token after extracting JTI

const validateJweToken = (encryptedToken) => {

const jti = extractJtiFromJwe(encryptedToken); // Extract the jti from the header

console.log('Extracted JTI:', jti);

const key = getKeyByJti(jti); // Retrieve the key pair associated with this jti

if (!key) {

throw new Error('No keys found for this JTI');

}

// Decrypt the token using the private key associated with the jti

const decryptedToken = JWE.decrypt(encryptedToken, key.privateKey);

const payload = JSON.parse(decryptedToken.toString());

return payload;

};

// Example usage

const encryptedToken = 'your-encrypted-jwe-token-here';

try {

const decryptedPayload = validateJweToken(encryptedToken);

console.log('Decrypted Payload:', decryptedPayload);

} catch (error) {

console.log('Error:', error.message);

}

**Key Points:**

1. **Extracting the jti**:
   * The JWE.decode function from the jose library allows you to extract the **protected header** without decrypting the entire token.
   * The protected field is base64url-encoded, so you need to decode it to get the JSON header, which contains the jti claim.
2. **Decrypting the JWE**:
   * Once you've extracted the jti, you can fetch the corresponding RSA key pair from your cache (or Redis) using that jti as the key ID.
   * After retrieving the correct private key, you can decrypt the JWE token.

**Why Use the Header?**

* The JWE **header** contains metadata about the encryption and is accessible without needing to decrypt the ciphertext. This metadata can include:
  + alg (algorithm used)
  + enc (encryption method)
  + kid (key ID)
  + jti (JWT ID, which is unique for each key pair)

By extracting the jti from the header, you can map it to the correct key pair and use that key pair to decrypt the rest of the JWE token.

**Conclusion:**

You **can** indeed extract the jti from the JWE token itself by decoding the protected header. This enables you to validate the key used for encryption and subsequent decryption without having to decrypt the full payload upfront.