Single Sign-On (SSO) Authorization System Using JWT and RSA

Abstract

This document presents the design and implementation of a foundational Single Sign-On (SSO) authorization system using JSON Web Tokens (JWT) and RSA-based digital signatures. The system enables stateless authorization across services by issuing cryptographically signed tokens that can be independently verified without maintaining server-side session state. RSA digital signatures combined with SHA-256 hashing ensure token integrity and authenticity, while transport-layer encryption ensures confidentiality during communication. The project emphasizes conceptual clarity, correct cryptographic usage, and Identity and Access Management (IAM) fundamentals, serving as a baseline for more advanced IAM systems.

1. Introduction

Modern distributed systems require scalable and secure mechanisms to manage user identity and access across multiple services. Traditional session-based authentication models rely on server-side state, which introduces scalability challenges and increases operational complexity in multi-service architectures.

Single Sign-On (SSO) systems address these challenges by centralizing authentication and issuing verifiable tokens that can be reused across trusted services. This project implements a minimal yet correct SSO authorization mechanism using JWT and asymmetric cryptography, focusing on understanding how real-world IAM systems are built at a foundational level.

---

2. Purpose of the Project

The purpose of this project is to design and implement a stateless authorization mechanism that demonstrates:

- Practical use of JSON Web Tokens (JWT)
- Application of RSA digital signatures for token authenticity
- Clear separation between hashing, encryption, and signing
- Core Identity and Access Management (IAM) principles

The project is intentionally scoped to prioritize correctness and clarity over enterprise-level feature completeness.

---

3. Problem Statement

Session-based authentication mechanisms require shared state between servers, which limits scalability and complicates distributed system design. Additionally, improperly designed token systems are vulnerable to tampering, forgery, and replay attacks.

There is a need for an authorization mechanism that:

- Is stateless
- Can be verified independently by multiple services
- Ensures integrity and authenticity of authorization data
- Minimizes shared secrets between systems

---

4. Objectives

The objectives of this project are:

1. To implement a JWT-based authorization system
2. To use RSA asymmetric cryptography for token signing
3. To demonstrate secure trust establishment between services
4. To analyze security properties and limitations of the system
5. To provide a clean baseline for future IAM extensions

---

5. Background Theory

5.1 Authentication vs Authorization

Authentication is the process of verifying a user’s identity, while authorization determines what actions an authenticated user is permitted to perform. This project focuses primarily on authorization by issuing signed tokens after successful authentication.

5.2 Single Sign-On (SSO)

Single Sign-On allows users to authenticate once and access multiple services without repeated credential submission. This is achieved by establishing trust between services and a central authority that issues verifiable tokens.

---

6. JSON Web Tokens (JWT)

6.1 Overview

A JSON Web Token (JWT) is a compact, URL-safe token used to securely transmit claims between two parties. JWTs are **self-contained** and **stateless**, meaning all required authorization information is embedded within the token itself, eliminating the need for server-side session storage. These properties make JWTs particularly suitable for **distributed and scalable architectures** such as Single Sign-On (SSO) systems.

6.2 JWT Structure

In this project, authorization is performed using a JSON Web Token (JWT) that is **digitally signed using an RSA private key** with the **RS256 (RSA + SHA-256)** algorithm.

A JWT consists of three **Base64URL-encoded** components, separated by dots:

```
<Header>.<Payload>.<Signature>
```

Each component serves a distinct purpose in ensuring the security and integrity of the token.

6.2.1 Header

The header specifies metadata about the token, including the token type and the cryptographic algorithm used for signing.

Example Header:

```
{
  "alg": "RS256",
  "typ": "JWT"
}
```

Explanation:

- `alg`: Indicates the signing algorithm used. RS256 represents RSA with SHA-256.
- `typ`: Specifies that the token type is JWT.

After Base64URL encoding, the header becomes a compact string suitable for transmission.

6.2.2 Payload

The payload contains the **claims**, which represent statements about the user and the authorization context.

Common claims used in this project include:

- **Subject (`sub`)** – Unique identifier for the user
- **Issuer (`iss`)** – Entity that issued the token
- **Expiration time (`exp`)** – Time after which the token is no longer valid
- **Custom authorization data** – Such as user roles

Example Payload:

```
{
  "sub": "user_12345",
  "iss": "simple_sso_auth_server",
  "exp": 1735689600,
  "role": "user"
}
```

The payload is **Base64URL encoded but not encrypted**. This means the contents are readable by anyone in possession of the token; however, the integrity and authenticity of the payload are protected by the digital signature.

6.2.3 Signature

The signature ensures that the token has not been altered and verifies the authenticity of the issuer.

In this project, the signature is generated by:

1. Base64URL encoding the header
2. Base64URL encoding the payload
3. Concatenating them with a dot (`Header.Payload`)
4. Hashing the result using SHA-256
5. Signing the hash using the issuer's **RSA private key**

Conceptually:

```
RSASHA256(
  base64UrlEncode(header) + "." + base64UrlEncode(payload),
  private_key
)
```

The resulting signature is appended as the third part of the JWT.

Any modification to the header or payload will result in signature verification failure, thereby preventing token tampering or forgery without access to the private key.

Summary

The JWT structure used in this project enables:

- Stateless authorization
- Secure claim transmission
- Cryptographic verification of token integrity and issuer authenticity
- Scalable trust across multiple services without shared secrets

This design forms the cryptographic backbone of the implemented Single Sign-On authorization system.

---

7. Cryptography Used

7.1 SHA-256 Hashing

SHA-256 is a cryptographic hash function used to generate a fixed-length digest of the token data before signing. Hashing ensures integrity but does not provide authenticity on its own.

7.2 RSA Digital Signatures

RSA digital signatures are used to authenticate the issuer of the JWT. The authentication server signs tokens using its private key, while resource servers verify signatures using the corresponding public key.

This ensures:

- Token integrity
- Token authenticity
- Trust without shared secrets

7.3 Role of AES-256

AES-256 is not used within JWT creation or signing. Instead, it is used implicitly at the transport layer as part of TLS (HTTPS) to encrypt communication between clients and servers, ensuring confidentiality of credentials and tokens during transit.

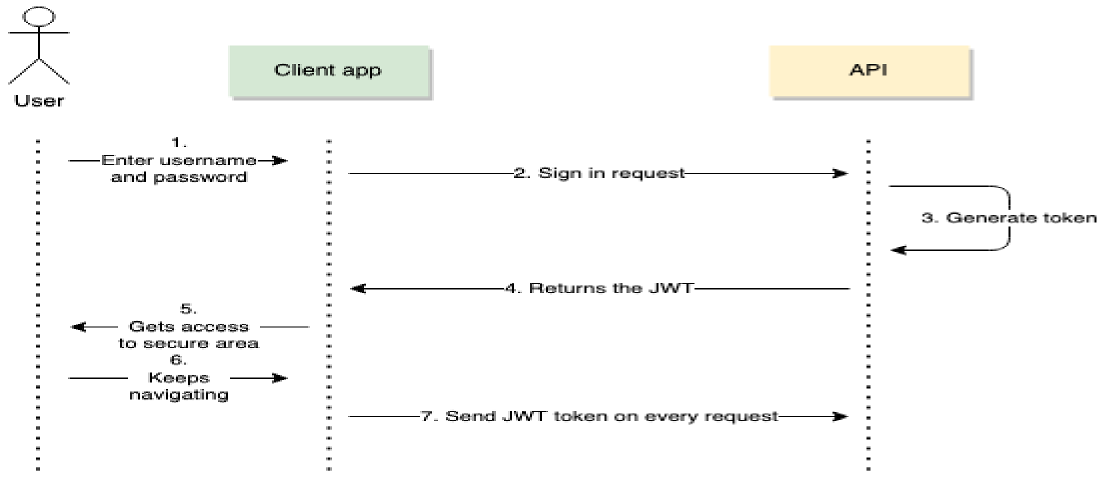
8. System Architecture

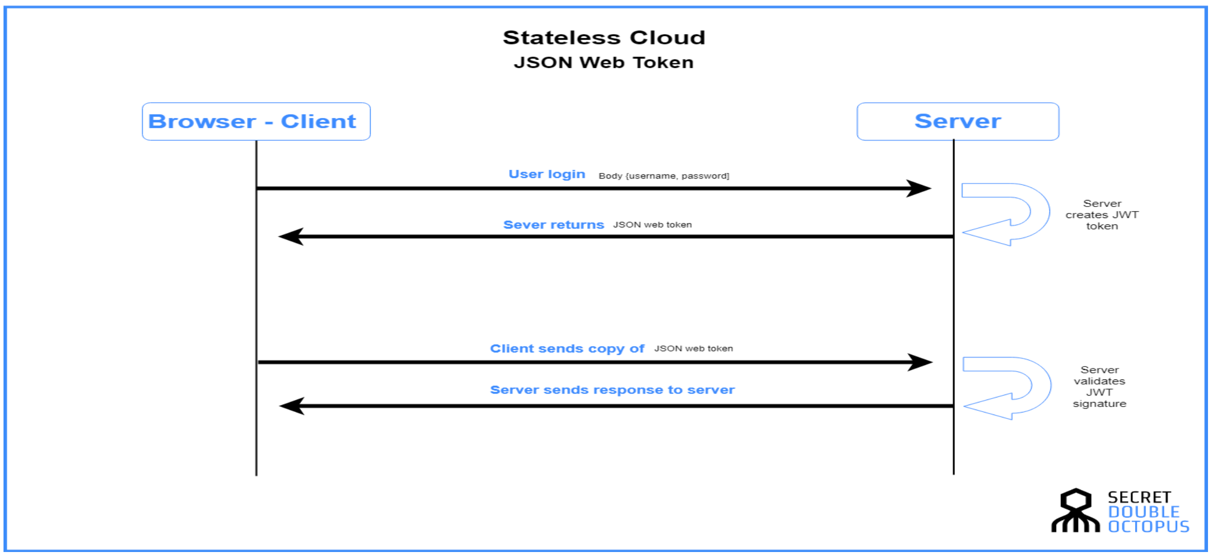
8.1 Components

- **Client** – Requests access to protected resources
- **Authentication Server** – Issues signed JWTs
- **Resource Server** – Verifies JWT signatures and claims

8.2 Trust Model

Resource servers trust tokens issued by the authentication server by verifying digital signatures using the public key. No shared secrets or session state are require


User
Client app
API
1. Enter username and password
2. Sign in request
3. Generate token
4. Returns the JWT
5. Gets access to secure area
6. Keeps navigating
7. Send JWT token on every request


Stateless Cloud
JSON Web Token
Browser - Client
Server
User login Body {username, password}
Sever returns JSON web token
Server creates JWT token
Client sends copy of JSON web token
Server sends response to server
Server validates JWT signature
SECRET DOUBLE OCTOPUS

9. Authorization Flow

1. User authenticates with the authentication server
2. Authentication server validates credentials
3. JWT is generated with relevant claims
4. JWT is signed using RSA private key
5. Token is returned to the client
6. Client includes token in API requests
7. Resource server verifies signature using public key
8. Claims are validated (expiry, issuer)
9. Access is granted or denied

---

10. Security Analysis

10.1 Threats Addressed

- Token tampering
- Token forgery
- Unauthorized token issuance
- Replay attacks (limited via expiration)

10.2 Threats Not Addressed

- Token theft due to client compromise
- Private key leakage
- Advanced phishing or XSS attacks
- Token revocation after issuance

These are acknowledged limitations of the foundational design.

---

11. Limitations

- No refresh token mechanism
- No token revocation strategy
- No key rotation
- Single issuer model
- No OAuth 2.0 or OpenID Connect compliance
- No audit logging

These limitations are intentional and addressed in Repository 2

Repository 2->https://github.com/RameshChettiar0806/secure-sso-trust-audit

---

12. Future Enhancements

Planned extensions include:

- Refresh tokens
- Key rotation using JWKS
- Role-Based Access Control (RBAC)
- OAuth 2.0 / OpenID Connect integration
- Audit logging
- Cloud IAM integration
- Improved threat modeling

Several of the enhancements listed above are implemented or explored in Repository 2, which extends this foundational SSO model toward real-world IAM systems

Repository 2=>https://github.com/RameshChettiar0806/secure-sso-trust-audit

---

13. Learning Outcomes

Through this project, the following learning outcomes were achieved:

- Clear understanding of JWT internals
- Correct use of RSA digital signatures
- Proper distinction between hashing and encryption
- Foundational IAM architecture knowledge
- Ability to reason about system limitations

---

14. Conclusion

This project demonstrates a foundational Single Sign-On authorization system using JWT and RSA. By focusing on cryptographic correctness and architectural clarity, it provides a strong baseline for understanding modern IAM systems and serves as a stepping stone toward more advanced implementations.

This repository establishes the cryptographic and architectural foundation, while advanced IAM mechanisms such as refresh tokens, key rotation, and OAuth alignment are explored in a separate, dedicated repository, Repository 2.

Repository 2=>https://github.com/RameshChettiar0806/secure-sso-trust-audit