MicroJWT: A Lightweight and Secure JWT Implementation for MicroPython

Arman Ghobadi

June 14, 2025

Contents

0.1 Introduction . . . 3
0.2 Design Principles . . . 3
0.3 Architecture Overview . . . 3
0.4 Core Functionalities . . . 4
0.4.1 Token Creation . . . 4
0.4.2 Token Verification . . . 5
0.4.3 Token Encryption . . . 5
0.4.4 Token Revocation and Refresh . . . 5
0.5 Code Example . . . 5
0.6 Optimizations for Embedded Systems . . . 6
0.7 Security Considerations . . . 6
0.8 Conclusion . . . 6

MicroJWT is a production-ready JSON Web Token (JWT) implementation tailored for resource-constrained embedded systems using MicroPython. This article explores the architecture, design principles, and implementation details of MicroJWT, focusing on its lightweight yet secure approach to token generation, verification, encryption, and revocation. Optimized for low-memory environments, MicroJWT supports the HS256 algorithm, incorporates AES-256-CBC encryption, and includes features like token revocation and constant-time signature verification to mitigate security risks. We present the system's logic through detailed explanations and visualize its architecture with diagrams, providing a comprehensive understanding of its functionality and suitability for embedded applications.

0.1 Introduction

JSON Web Tokens (JWTs) are widely used for secure authentication and authorization in distributed systems. However, implementing JWTs in resource-constrained embedded environments, such as those running MicroPython, poses significant challenges due to limited memory and computational resources. MicroJWT addresses these challenges by providing a lightweight, secure, and feature-rich JWT implementation tailored for MicroPython-based devices.

This article details the architecture of MicroJWT, its design principles, and the logic behind its core functionalities, including token creation, verification, encryption, and revocation. We also include diagrams to illustrate the systems workflow and discuss its optimizations for embedded systems.

0.2 Design Principles

The design of MicroJWT is guided by the following principles:

- **Lightweight Implementation**: Minimize memory and computational overhead to suit MicroPythons constraints.
- **Security**: Incorporate robust cryptographic mechanisms, including HMAC-SHA256, AES-256-CBC encryption, and constant-time comparisons to prevent timing attacks.
- **Modularity**: Provide a modular architecture that supports core JWT features while allowing extensibility.
- **Reliability**: Ensure consistent behavior with comprehensive error handling and logging.

0.3 Architecture Overview

MicroJWT is structured around three main components:

1. **SimpleLogger**: A lightweight logging system for debugging and monitoring.
2. **Cryptographic Utilities**: Custom implementations of HMAC and PBKDF2 for key derivation and signing.
3. **MicroJWT Core**: The main class handling token creation, verification, encryption, and revocation.

The `SimpleLogger` provides configurable logging levels (INFO, WARNING, ERROR) to track operations without excessive resource use. The cryptographic utilities include a custom HMAC implementation (following RFC 2104) and PBKDF2 for secure key derivation. The `MicroJWT` class orchestrates token operations, leveraging these utilities for secure and efficient processing.

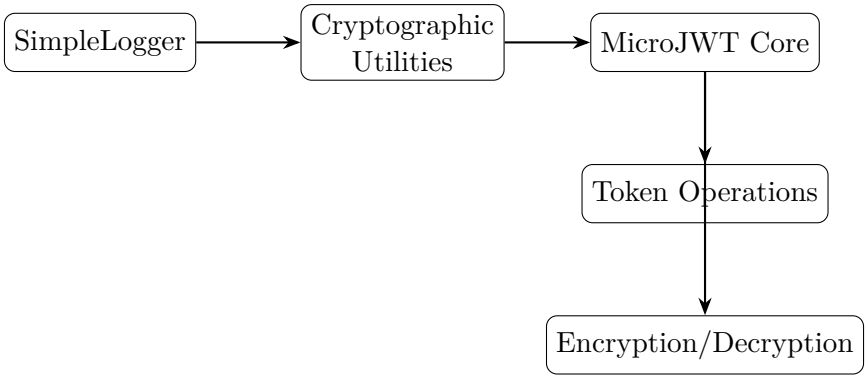

Figure 1: MicroJWT Architecture Overview

0.4 Core Functionalities

0.4.1 Token Creation

The `create_token` method generates a JWT comprising a header, payload, and signature, optionally encrypted with AES-256-CBC. The logic includes:

- **Input Validation**: Ensures username, role, and additional claims are valid.
- **Header Construction**: Defines the algorithm (HS256) and token type (JWT).
- **Payload Construction**: Includes standard claims (`sub`, `role`, `iat`, `exp`, `jti`) and optional audience or custom claims.
- **Signature Generation**: Uses HMAC-SHA256 with a derived signing key.
- **Encryption (Optional)**: Applies AES-256-CBC encryption for enhanced security.

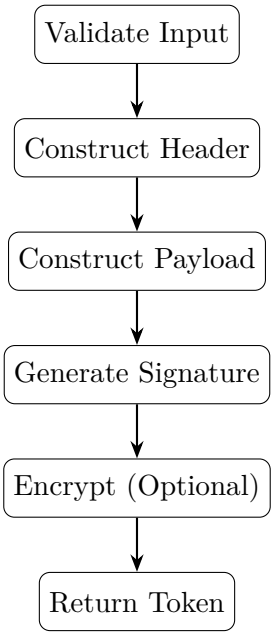

Figure 2: Token Creation Flow

0.4.2 Token Verification

The `verify_token` method validates a token by checking its signature, expiration, revocation status, and audience. It includes:

- **Decryption (Optional)**: Decrypts the token if encrypted.
- **Parsing**: Splits the token into header, payload, and signature.
- **Algorithm Check**: Ensures the header specifies HS256.
- **Revocation Check**: Verifies the tokens `jti` is not revoked.
- **Expiration Check**: Confirms the token is not expired.
- **Audience Validation**: Matches the audience if specified.
- **Signature Verification**: Uses constant-time comparison to prevent timing attacks.

0.4.3 Token Encryption

MicroJWT supports AES-256-CBC encryption for tokens, using a derived encryption key from PBKDF2. The process involves:

- Generating a random 16-byte IV.
- Padding the token to a multiple of 16 bytes.
- Encrypting with AES-CBC and combining IV with ciphertext.

0.4.4 Token Revocation and Refresh

The `revoke_token` method adds a tokens `jti` to an in-memory revocation list, while `clear_revoked_tokens` manages the list. The `refresh_token` method verifies an existing token and issues a new one with updated expiration, preserving other claims.

0.5 Code Example

Below is an example of using MicroJWT to create and verify a token:

```
from jwt import MicroJWT

# Initialize MicroJWT
jwt = MicroJWT(secret_key="your-32-byte-secret-key-here", ttl=3600)

# Create a token
token = jwt.create_token(username="user1", role="admin", encrypt=True)

# Verify the token
payload = jwt.verify_token(token, encrypted=True)
if payload:
    print(f"Verified token for user: {payload['sub']}")
else:
    print("Token verification failed")
```

Listing 1: MicroJWT Usage Example

0.6 Optimizations for Embedded Systems

MicroJWT is optimized for MicroPythons constraints:

- **Minimal Dependencies**: Uses only `hashlib`, `ubinascii`, `json`, `time`, `os`, `binascii`, and `ucryptolib`.
- **Constant-Time Operations**: Prevents timing attacks in signature verification.
- **Memory Efficiency**: Uses in-memory revocation lists and avoids external storage.
- **Configurable Logging**: Allows silent mode to reduce resource usage.

0.7 Security Considerations

MicroJWT incorporates several security features:

- **Key Derivation**: Uses PBKDF2 with 1000 iterations for secure key generation.
- **Constant-Time Comparison**: Mitigates timing attacks during verification.
- **Random Salts and IVs**: Enhances token uniqueness and encryption security.
- **Error Handling**: Comprehensive logging and custom exceptions prevent silent failures.

0.8 Conclusion

MicroJWT provides a robust and efficient JWT implementation for MicroPython, balancing security and resource constraints. Its modular architecture, support for encryption, and optimizations make it suitable for embedded systems requiring secure authentication. Future enhancements could include support for additional algorithms (e.g., HS512) or persistent revocation storage, depending on application needs.

lmodern