

Authentication Security Analysis Report

Executive Summary

This report examines the security implications of using HTTP Basic Authentication in the Transaction API, identifies its vulnerabilities, and recommends more secure alternatives such as JWT (JSON Web Tokens) and OAuth2.

1. HTTP Basic Authentication Overview

HTTP Basic Authentication is a simple authentication scheme built into the HTTP protocol. It works by:

1. Client sends credentials (username:password) encoded in Base64
2. Server decodes and validates credentials
3. Client must send credentials with every request

Example Header:

Authorization: Basic YWRtaW46cGFzc3dvcmQxMjM=

2. Why Basic Authentication is Weak

2.1 Not Encrypted, Only Encoded

Problem: Base64 is an encoding scheme, not encryption. Anyone intercepting the request can easily decode the credentials.

Example:

YWRtaW46cGFzc3dvcmQxMjM= decodes to → admin:password123

Any attacker with access to network traffic can decode this instantly using simple tools or online decoders.

Risk Level: ⚠️ CRITICAL

2.2 Credentials Sent With Every Request

Problem: Unlike token-based systems, Basic Auth requires sending username and password with every single API request.

Implications:

- More opportunities for credential interception
- Increased attack surface
- If one request is compromised, credentials are exposed

Example: If you make 1000 API calls, your password travels over the network 1000 times.

Risk Level: ⚠ HIGH

2.3 No Built-in Expiration

Problem: Basic Auth credentials don't expire. Once compromised, they remain valid until manually changed.

Implications:

- Stolen credentials can be used indefinitely
- No automatic security refresh mechanism
- Difficult to revoke access for specific sessions
- Former employees or compromised accounts remain active

Risk Level: ⚠ HIGH

2.4 Vulnerable to Replay Attacks

Problem: Captured authentication headers can be replayed by attackers to gain unauthorized access.

Scenario:

1. Attacker intercepts: `Authorization: Basic YWRtaW46cGFzc3dvcmQxMjM=`
2. Attacker replays this exact header in their own requests
3. Server accepts it as valid authentication

Risk Level: ⚠ HIGH

2.5 No User Session Management

Problem: Basic Auth has no concept of sessions or login/logout.

Limitations:

- Can't track active sessions
- Can't force logout
- Can't implement "logout everywhere" feature
- No visibility into who's currently authenticated

Risk Level: ⚠ MEDIUM

2.6 Password Management Issues

Problem: Users must store passwords in applications or browsers, often in plain text.

Common Practices:

- Hardcoded in scripts
- Stored in configuration files
- Saved in browser password managers
- Embedded in mobile apps

Risk Level: ⚠ HIGH

2.7 Requires HTTPS

Problem: Basic Auth is only secure over HTTPS. Over HTTP, credentials are sent in plain text.

Real-world Issue:

- Many development environments use HTTP
- Misconfigurations can expose credentials
- Mixed content scenarios create vulnerabilities

Without HTTPS: Credentials visible to anyone on the network

Risk Level: ⚠ CRITICAL (if HTTPS not enforced)

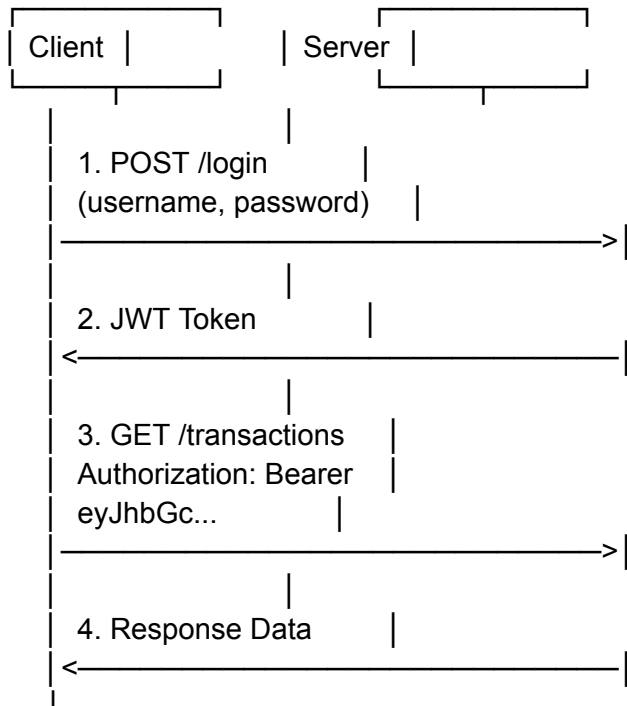
3. Secure Alternatives

3.1 JWT (JSON Web Tokens)

What is JWT?

JWT is a compact, self-contained token format for securely transmitting information between parties as a JSON object.

How It Works



JWT Structure

A JWT consists of three parts separated by dots:

xxxxx.yyyyy.zzzzz

1. **Header:** Algorithm and token type
2. **Payload:** Claims (user data, expiration, etc.)
3. **Signature:** Cryptographic signature

Example JWT:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxMjM0LCJpdiI6ImV4cCI6MTYzMzAyNDgwMH0.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Advantages Over Basic Auth

Feature	Basic Auth	JWT
---------	------------	-----

Credentials sent per request	✓ Yes	✗ No
Expiration support	✗ No	✓ Yes
Stateless	✓ Yes	✓ Yes
Revocation	✗ Difficult	✓ Possible
Payload data	✗ None	✓ Custom claims
Security	⚠ Low	✓ High

Key Benefits

1. **Token Expiration:** Tokens can expire after minutes/hours
2. **No Password Transmission:** Password sent only once during login
3. **Self-Contained:** Token contains user information (no database lookup needed)
4. **Cryptographically Signed:** Can't be tampered with
5. **Revocable:** Can maintain a blacklist of revoked tokens

Implementation Example

Login endpoint

```
@app.route('/login', methods=['POST'])
```

```
def login():
```

```
    username = request.json.get('username')
```

```
    password = request.json.get('password')
```

```
    if verify_credentials(username, password):
```

```
        token = jwt.encode({
```

```
            'user_id': user.id,
```

```
            'username': username,
```

```
            'exp': datetime.utcnow() + timedelta(hours=24)
```

```
        }, SECRET_KEY, algorithm='HS256')
```

```
        return {'token': token}
```

```
    return {'error': 'Invalid credentials'}, 401
```

Protected endpoint

```
@app.route('/transactions', methods=['GET'])
```

```
def get_transactions():
```

```
    token = request.headers.get('Authorization').split()[1]
```

```
    try:
```

```
        payload = jwt.decode(token, SECRET_KEY, algorithms=['HS256'])
```

```
        user_id = payload['user_id']
```

```
        # Return transactions
```

```
except jwt.ExpiredSignatureError:
    return {'error': 'Token expired'}, 401
except jwt.InvalidTokenError:
    return {'error': 'Invalid token'}, 401
```

When to Use JWT

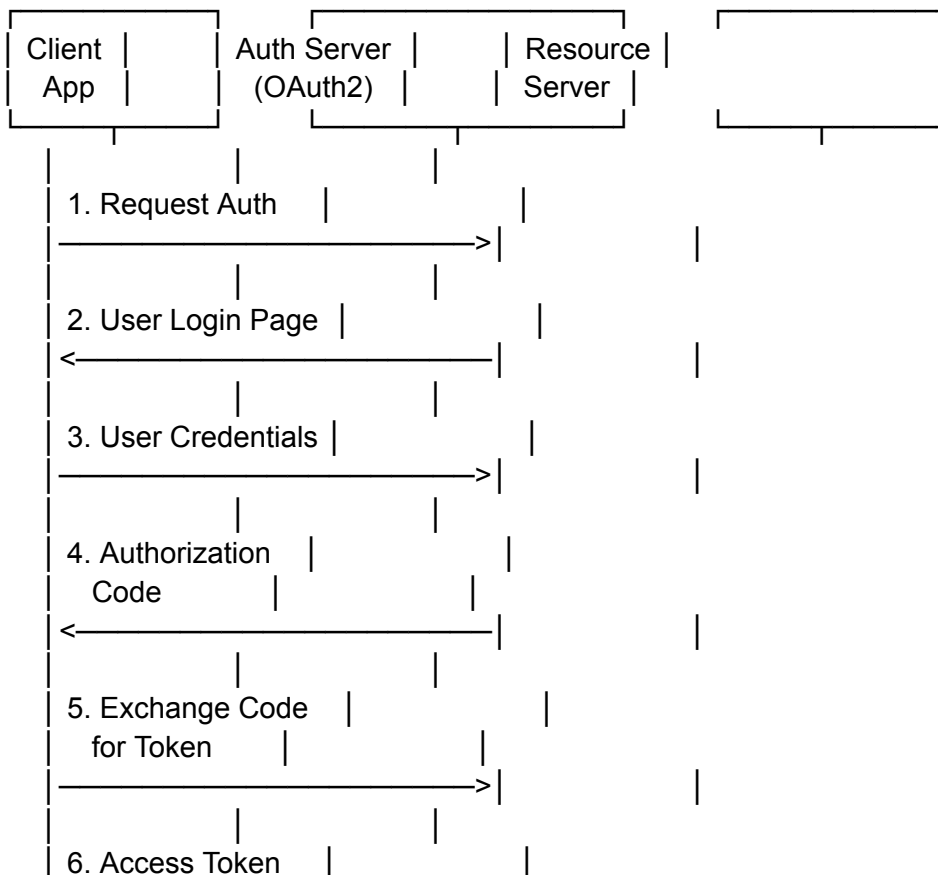
- ☒ Microservices architectures
- ☒ Mobile applications
- ☒ Single Page Applications (SPAs)
- ☒ APIs with stateless authentication
- ☒ Systems requiring token expiration

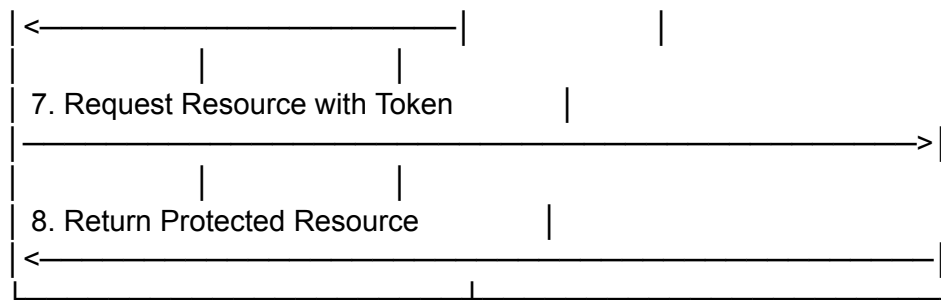
3.2 OAuth2

What is OAuth2?

OAuth2 is an authorization framework that enables applications to obtain limited access to user accounts without exposing passwords. It's commonly used for "Sign in with Google/Facebook/GitHub" functionality.

How It Works





Key Components

1. **Resource Owner:** The user
2. **Client:** The application requesting access
3. **Authorization Server:** Issues access tokens (e.g., Google, GitHub)
4. **Resource Server:** Hosts protected resources (e.g., your API)

OAuth2 Grant Types

1. Authorization Code (Most Secure)

- Used by web applications
- User redirected to auth server
- Returns authorization code
- Code exchanged for access token

2. Client Credentials





- Used for machine-to-machine communication
- No user involvement
- Application authenticates directly

3. Refresh Token

- Used to obtain new access tokens
- Long-lived token for getting short-lived access tokens

Advantages Over Basic Auth

Feature	Basic Auth	OAuth2
Password exposure	⚠ Always sent	✅ Never sent
Token expiration	❌ No	✅ Yes
Granular permissions	❌ All or nothing	✅ Scopes
Third-party integration	❌ No	✅ Yes

Revocation	 Change password	 Revoke tokens
User consent	 No	 Yes

Key Benefits

1. **Delegated Access:** Users don't share passwords with third-party apps
2. **Scoped Permissions:** Limit what applications can access (read-only, etc.)
3. **Token Refresh:** Short-lived access tokens with refresh capability
4. **Centralized Security:** Authentication handled by specialized servers
5. **Better User Experience:** "Sign in with..." buttons

Implementation Example

```
from flask import Flask, redirect, request
from authlib.integrations.flask_client import OAuth
```

```
app = Flask(__name__)
oauth = OAuth(app)
```

```
# Configure OAuth provider
```

```
google = oauth.register(
    name='google',
    client_id='YOUR_CLIENT_ID',
    client_secret='YOUR_CLIENT_SECRET',
    authorize_url='https://accounts.google.com/o/oauth2/auth',
    access_token_url='https://accounts.google.com/o/oauth2/token',
    client_kwargs={'scope': 'openid email profile'})
```

```
# Login route
```

```
@app.route('/login')
def login():
    redirect_uri = 'http://localhost:8000/callback'
    return google.authorize_redirect(redirect_uri)
```

```
# Callback route
```

```
@app.route('/callback')
def callback():
    token = google.authorize_access_token()
    user_info = google.get('userinfo').json()







    # Create session or JWT token
    # Return access token to client
    return {'access_token': token['access_token']}
```

```
# Protected route
```





















```
@app.route('/transactions')
def transactions():
    token = request.headers.get('Authorization')
    # Verify token with OAuth provider
    # Return protected data
```

When to Use OAuth2

-  Third-party application integration
-  "Sign in with..." functionality
-  Multi-tenant applications
-  APIs accessed by multiple clients
-  When you need granular permissions
-  Enterprise applications

4. Comparison Summary

Aspect	Basic Auth	JWT	OAuth2
Complexity	Simple	Moderate	Complex
Setup Time	Minutes	Hours	Days
Security Level	 Low	 High	 Very High
Password Exposure	 Every request	 Login only	 Never
Token Expiration	 No	 Yes	 Yes
Revocation	 Difficult	 Possible	 Easy
Scalability	 Good	 Excellent	 Excellent
Third-party Access	 No	 Limited	 Yes
Best For	Internal tools	APIs, SPAs	Enterprise, SSO

5. Recommendations

For the Transaction API

Short-term (Minimal Changes)

1. **Enforce HTTPS:** Never allow HTTP connections
2. **Strong Passwords:** Require complex passwords with minimum length
3. **Rate Limiting:** Prevent brute force attacks
4. **IP Whitelisting:** Restrict access to known IPs

Medium-term (Recommended)

Implement JWT Authentication:

- Add `/login` endpoint that returns JWT token
- Replace Basic Auth with Bearer token validation
- Set token expiration (e.g., 24 hours)
- Implement token refresh mechanism

Benefits:

- Significantly improved security
- Token expiration prevents long-term compromise
- Passwords only sent during login
- Moderate implementation effort

Long-term (Enterprise-grade)

Migrate to OAuth2:

- Set up OAuth2 authorization server
- Implement different grant types
- Add scope-based permissions
- Enable third-party integrations

Benefits:

- Industry-standard security
- Support for multiple clients
- Granular access control
- Scalable for future growth

6. Conclusion




HTTP Basic Authentication, while simple to implement, presents significant security risks including:

- Credentials sent with every request
- No built-in expiration mechanism
- Vulnerability to replay attacks

- Base64 encoding provides no real security

For the Transaction API, upgrading to JWT authentication would provide substantial security improvements with moderate implementation effort. For enterprise or multi-tenant scenarios, OAuth2 would be the gold standard.

Action Items

1.  **Immediate:** Document Basic Auth risks for stakeholders
 2.  **High Priority:** Implement JWT authentication within 2-4 weeks
 3.  **Future Planning:** Evaluate OAuth2 for long-term roadmap
-

References

- RFC 7617: The 'Basic' HTTP Authentication Scheme
 - RFC 7519: JSON Web Token (JWT)
 - RFC 6749: The OAuth 2.0 Authorization Framework
 - OWASP Authentication Cheat Sheet
 - JWT.io - JWT Debugger and Documentation
-

Report Prepared For: Class Assignment

Date: October 2025

Topic: API Authentication Security Analysis