Building and Securing a REST API

Project Report

Course: Enterprise Web Development

Project: SMS Mobile Money Transaction API

Repository: github.com/FabriceMbarushimana/Ewdgroup4-Building-and-Securing-a-REST-API

Table of Contents

1. Introduction to API Security
2. API Endpoints Documentation
3. DSA Comparison Results
4. Reflection on Basic Auth Limitations
5. Conclusion

---

1. Introduction to API Security

1.1 What is API Security?

API security refers to the practices and protocols used to protect Application Programming Interfaces (APIs) from malicious attacks and unauthorized access. As APIs serve as the communication layer between different software systems, securing them is critical to protecting sensitive data and maintaining system integrity.

1.2 Why API Security Matters

- **Data Protection:** APIs often handle sensitive information such as financial transactions, personal data, and authentication credentials
- **Prevent Unauthorized Access:** Without proper security, attackers can access, modify, or delete data
- **Maintain Service Availability:** Security measures help prevent denial-of-service attacks
- **Compliance Requirements:** Many industries require specific security standards (GDPR, PCI-DSS)

1.3 Common API Security Threats

Threat	Description
Injection Attacks	Malicious code inserted through API inputs
Broken Authentication	Weak or improperly implemented authentication
Data Exposure	Sensitive data transmitted without encryption
Rate Limiting Absence	APIs vulnerable to brute force attacks
Insufficient Logging	Inability to detect and respond to breaches

1.4 Security Implemented in This Project

This project implements **HTTP Basic Authentication** to secure all API endpoints. Every request must include valid credentials encoded in Base64 format in the Authorization header.

```
Authorization: Basic base64(username:password)
```

---

2. API Endpoints Documentation

2.1 Base URL

```
http://localhost:8000
```

2.2 Authentication

All endpoints require Basic Authentication with valid credentials:

- `admin:password123`
- `user:user123`
- `test:test123`

2.3 Endpoints Summary

Method	Endpoint	Description	Request Body
GET	/transactions	Retrieve all transactions	None
GET	/transactions/{id}	Retrieve single transaction	None
POST	/transactions	Create new transaction	JSON
PUT	/transactions/{id}	Update existing transaction	JSON
DELETE	/transactions/{id}	Delete a transaction	None

2.4 Response Examples

GET /transactions - Success (200 OK)

```
{
  "success": true,
  "count": 20,
  "data": [
    {
      "id": 1,
      "type": "received",
      "amount": "2000",
      "sender": "Jane Smith",
      "receiver": "You",
      "balance": "2000",
      "fee": "0"
    }
  ]
}
```

GET /transactions/{id} - Not Found (404)

```
{
  "success": false,
  "error": "Transaction with ID 999 not found"
}
```

POST /transactions - Required Fields

- `type` (string): Transaction type (received, payment, transfer, deposit)
- `amount` (string): Transaction amount
- `sender` (string): Sender name
- `receiver` (string): Receiver name

2.5 Error Codes

Code	Status	Description
200	OK	Request successful
201	Created	Resource created successfully
400	Bad Request	Invalid request or missing fields
401	Unauthorized	Invalid or missing credentials
404	Not Found	Resource not found
500	Internal Server Error	Server error

3. DSA Comparison Results

3.1 Overview

This project implements and compares two search algorithms for finding transactions by ID:

1. **Linear Search** - Sequential search through the list
2. **Dictionary Lookup** - Hash-based direct access

3.2 Algorithm Implementations

Linear Search - O(n)

```
def linear_search(transactions, transaction_id):
    for transaction in transactions:
        if transaction['id'] == transaction_id:
            return transaction
    return None
```

- **Time Complexity:** O(n) - Must check each element
- **Space Complexity:** O(1) - No additional space needed
- **Best Case:** O(1) - Element found at first position
- **Worst Case:** O(n) - Element at last position or not found

Dictionary Lookup - O(1)

```
def dict_search(transaction_dict, transaction_id):
    return transaction_dict.get(transaction_id)
```

- **Time Complexity:** O(1) - Direct hash-based access
- **Space Complexity:** O(n) - Requires dictionary creation
- **Best Case:** O(1)
- **Worst Case:** O(1) average, O(n) in rare hash collision scenarios

3.3 Performance Comparison

Metric	Linear Search	Dictionary Lookup
Time Complexity	O(n)	O(1)
Space Complexity	O(1)	O(n)
Setup Time	None	O(n) for dict creation
Average Search Time	~0.00001s	~0.000001s
Speedup Factor	1x (baseline)	~10x faster

3.4 Benchmark Results

> **Testing with 20 transactions, searching for IDs 1-20:**
>
> Linear Search Average Time: `0.00000850 seconds`
>
> Dictionary Lookup Average Time: `0.00000085 seconds`
>
> **Winner: Dictionary Lookup (~10x faster)**

3.5 When to Use Each Method

Use Case	Recommended Method
Single search, small dataset	Linear Search
Multiple searches	Dictionary Lookup
Memory-constrained environment	Linear Search
Large dataset with frequent lookups	Dictionary Lookup
Data changes frequently	Linear Search
Read-heavy operations	Dictionary Lookup

4. Reflection on Basic Auth Limitations

4.1 How Basic Authentication Works

Basic Authentication is a simple authentication scheme built into the HTTP protocol:

1. Client sends request with Authorization header
2. Credentials are encoded in Base64: `base64(username:password)`
3. Server decodes and validates credentials
4. Access granted or denied (401 Unauthorized)

```
Authorization: Basic YWRtaW46cGFzc3dvcmQxMjM=
                     ↓ (decoded)
               admin:password123
```

4.2 Advantages of Basic Auth

Advantage	Description
Simplicity	Easy to implement with no external dependencies
Universal Support	Supported by all HTTP clients and browsers
Stateless	No session management required
No Token Expiry	Credentials work until changed

4.3 Security Limitations

4.3.1 Credentials Transmitted in Every Request

- Credentials are sent with **every single request**
- Increases exposure to interception attacks
- Base64 encoding is **NOT encryption** - easily decoded

4.3.2 No Built-in Encryption

- Credentials are only Base64 encoded (reversible)
- Without HTTPS, credentials are transmitted in plain text
- Vulnerable to Man-in-the-Middle (MITM) attacks

```
# Anyone can decode Base64 credentials
import base64
encoded = "YWRtaW46cGFzc3dvcmQxMjM="
decoded = base64.b64decode(encoded).decode('utf-8')
print(decoded)  # Output: admin:password123
```

4.3.3 No Session Management

- Cannot invalidate sessions without changing passwords
- No logout functionality
- Difficult to implement session timeouts

4.3.4 Vulnerable to Brute Force

- No built-in rate limiting
- Attackers can try unlimited password combinations
- No account lockout mechanism

4.3.5 No Granular Permissions

- Basic Auth only verifies identity
- No role-based access control (RBAC)
- All authenticated users have same permissions

4.4 Comparison with Modern Alternatives

Feature	Basic Auth	OAuth 2.0	JWT
Complexity	Low	High	Medium
Token Expiry	No	Yes	Yes
Refresh Tokens	No	Yes	Optional
Stateless	Yes	No	Yes
Third-party Auth	No	Yes	No
Revocation	Difficult	Easy	Difficult

4.5 Recommendations for Production

To address Basic Auth limitations, consider implementing:

1. **HTTPS (TLS/SSL):** Always use HTTPS to encrypt credentials in transit
2. **Rate Limiting:** Limit login attempts to prevent brute force attacks
3. **Token-Based Auth:** Use JWT or OAuth 2.0 for better security
4. **Password Hashing:** Store passwords using bcrypt or Argon2
5. **Multi-Factor Authentication (MFA):** Add additional verification layer
6. **API Keys:** Use rotating API keys for service-to-service communication

4.6 Why We Used Basic Auth

Despite its limitations, Basic Auth was chosen for this project because:

- **Educational Purpose:** Demonstrates fundamental authentication concepts
- **Simplicity:** No external dependencies or complex setup
- **Focus on Core Concepts:** Allows focus on REST API design and DSA
- **Python Standard Library:** Works with http.server module without additional packages

---

5. Conclusion

5.1 Project Summary

This project successfully implemented a REST API for managing SMS mobile money transactions with the following achievements:

- **Full CRUD Operations:** Complete Create, Read, Update, Delete functionality
- **Basic Authentication:** Secure access control for all endpoints
- **XML Data Parsing:** Conversion of SMS data to JSON format
- **DSA Implementation:** Comparison of Linear Search vs Dictionary Lookup
- **Documentation:** Comprehensive API documentation and testing scripts

5.2 Key Learnings

1. **API Design:** Understanding RESTful principles and HTTP methods
2. **Authentication:** Implementing and understanding Basic Auth limitations
3. **Data Structures:** Practical comparison of search algorithms
4. **Security Awareness:** Recognizing vulnerabilities and mitigation strategies

5.3 Future Improvements

- Implement JWT-based authentication
- Add HTTPS support
- Implement rate limiting
- Add role-based access control
- Create a frontend interface
- Add database persistence
