EN4720 - Security in Cyber-Physical Systems

Identify Vulnerabilities in Existing Smart Home System - Milestone 3

Department of Electronic and Telecommunication Engineering
University of Moratuwa

Group: Cyberbullies

Name	Index Number
Amarasinghe Y.E.	200029B
Croos J.J.S.E.	200095V
Nimnaka K.W.H.	200426N
Vikkramanayaka A.G.P.S.	200683X

1 Introduction

As smart home technology becomes increasingly integrated into everyday life, the importance of securing the systems that power it cannot be overstated. From unlocking doors and adjusting thermostats to uploading firmware and streaming camera feeds, these smart devices rely heavily on APIs for communication and control.

This document provides a comprehensive security analysis of various APIs used in a smart home ecosystem. Each section examines a specific API, identifies potential vulnerabilities, and offers practical recommendations to mitigate risk. The goal is to highlight how seemingly simple misconfigurations can lead to serious security flaws and how developers, engineers, and auditors can design safer systems from the ground up.

Through real-world-inspired examples and clear explanations, this document aims to raise awareness of best practices in API security and help foster a safer future for connected homes.

2 User Authentication Basics

This section covers basic APIs for user registration and login, starting with simple concepts before progressing to more advanced vulnerabilities.

2.1 Register User

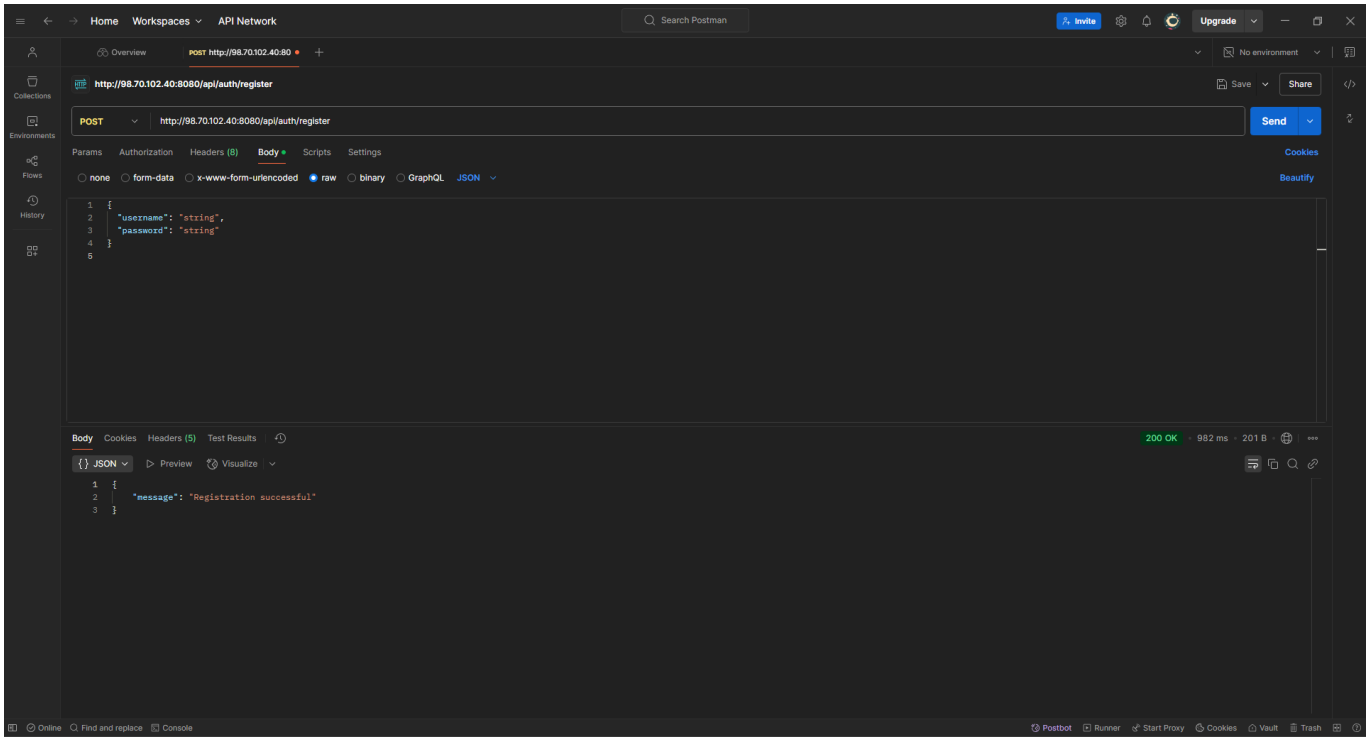

Figure 1: User Registration.

Endpoint Information

- **Endpoint:** `http://98.70.102.40:8080/api/auth/register`
- **Method:** POST
- **Request Body (JSON):**

```
{
    "username": "string",
    "password": "string"
}
```

- **Response Body (JSON):**

```
{
    "message": "Registration successful"
}
```

Identified Vulnerabilities

1. **Unencrypted Communication (HTTP)**
   The API uses `http://` instead of `https://`. This exposes sensitive data (e.g., passwords) to interception via Man-in-the-Middle (MitM) attacks.

2. **Lack of Input Validation and Sanitization**
   The API does not appear to validate or sanitize input, making it vulnerable to:
   - SQL Injection
   - Command Injection
   - Cross-Site Scripting (XSS)

3. **No Rate Limiting or Anti-Automation Controls**
   There is no indication of rate limiting or bot protection. This exposes the endpoint to:
   - Automated account creation
   - Brute-force attacks
   - Username enumeration

4. **Weak Password Policy**
   The API accepts any string as a password, with no enforcement of complexity, length, or strength.

5. **Generic Response Message**
   While the response is simple, variations in error messages can leak information about the existence of usernames, enabling enumeration attacks.

6. **No Mention of Password Storage Security**
   It is unclear whether passwords are securely hashed and salted. Storing passwords in plaintext or with weak hashing poses a serious risk.

7. **No Authentication or Abuse Prevention Mechanism**
   Although registration endpoints are generally public, the lack of additional security measures (for example, CAPTCHA, API keys) can result in abuse.

Recommendations

- Use **HTTPS** for all communications to ensure encryption of data in transit.
- Implement **input validation and sanitization** on both the client and server side.

- Implement **strong password policies**, including minimum length and complexity requirements.
- Implement **rate limiting, CAPTCHA**, or other anti-bot mechanisms to prevent abuse.
- Ensure that all stored passwords are **hashed and salted** using strong algorithms (e.g., bcrypt, Argon2).
- Return **generic error messages** to reduce the risk of user enumeration.
- Log and monitor registration attempts for signs of abuse or attacks.

2.2 User Login

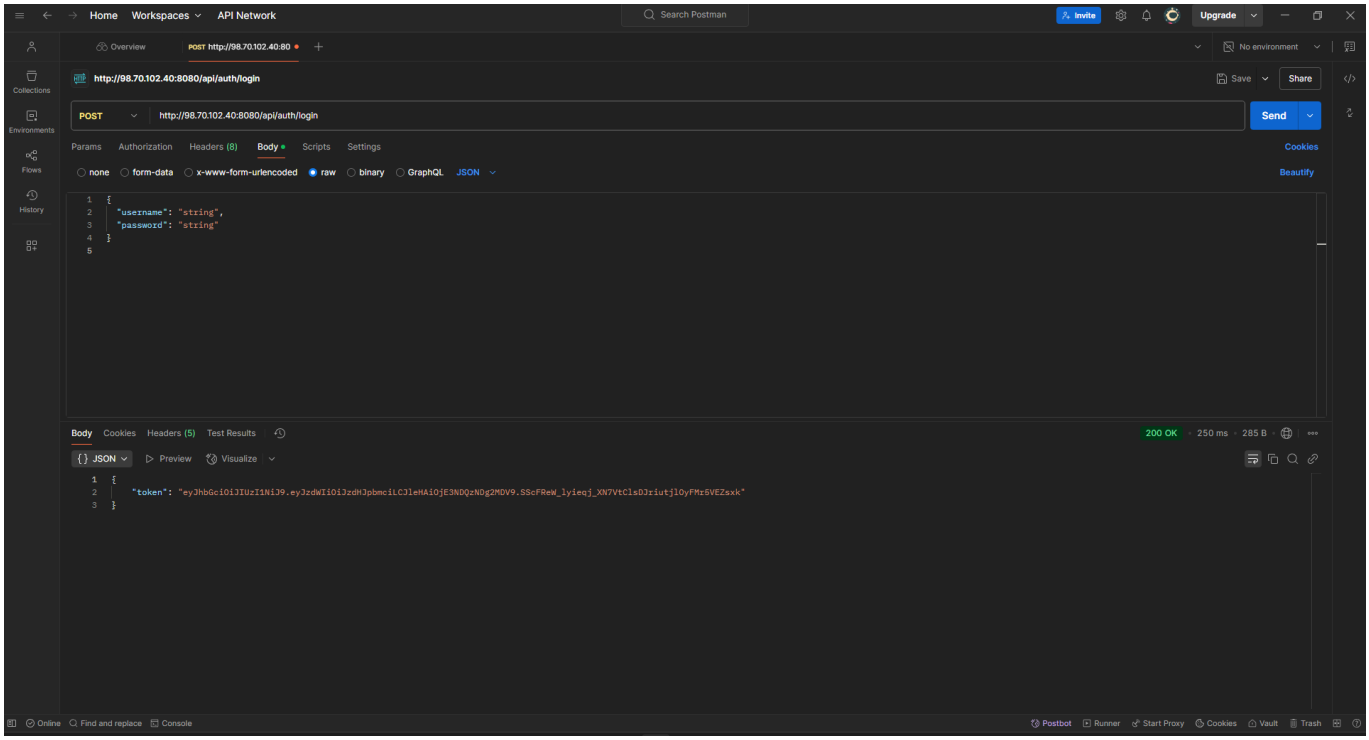

Figure 2: User Login.

Endpoint Information

- **Endpoint:** `http://98.70.102.40:8080/api/auth/login`
- **Method:** POST
- **Request Body (JSON):**

```
{
    "username": "string",
    "password": "string"
}
```

- **Response Body (JSON):**

```
{
    "token": "<Token>"
}
```

Identified Vulnerabilities

1. **Unencrypted Communication (HTTP)**
   The login API uses `http://` rather than `https://`, making it vulnerable to Man-in-the-Middle (MitM) attacks where login credentials and tokens can be intercepted.

2. **No Mention of Rate Limiting or Brute-Force Protection**
   The endpoint appears to accept repeated login attempts without any indication of throttling, lockouts, or CAPTCHA. This allows for brute-force or credential stuffing attacks.

3. **No Error Handling or Feedback Mentioned**
   If error messages differ for incorrect usernames vs. incorrect passwords, it may enable username enumeration.

4. **Potential JWT Misconfiguration**
   The token is returned, but it is unclear:

   - If it has an appropriate expiration time (`exp` claim).
   - If it is signed using a secure algorithm (e.g., HS256 vs. none or RS256 misconfig).
   - If proper verification is done server-side.

   Misconfigured JWTs can be forged or reused maliciously.

5. **No Multi-Factor Authentication (MFA)**
   The authentication relies solely on a password, with no mention of optional or enforced MFA.

6. **Token Exposure Risk in Client**
   Since the token is returned in the response, if not handled securely on the client side (e.g., stored in localStorage), it may be susceptible to cross-site scripting (XSS) token theft.

Recommendations

- Use **HTTPS** to encrypt all communication between client and server.
- Implement **rate limiting**, account lockout mechanisms, and CAPTCHA to mitigate brute-force attacks.
- Ensure **generic error messages** are returned on login failure to prevent username enumeration.
- Use strong and secure **JWT practices**:
  - Set short `exp` claim durations.
  - Use a strong secret and secure algorithms (e.g., HS256 or RS256).
  - Validate the token signature and claims strictly on the server.
- Provide optional or enforced **Multi-Factor Authentication (MFA)** for increased security.
- Instruct clients to store JWTs securely (e.g., `HttpOnly` cookies) to prevent exposure via XSS.

2.3 Secure Registration

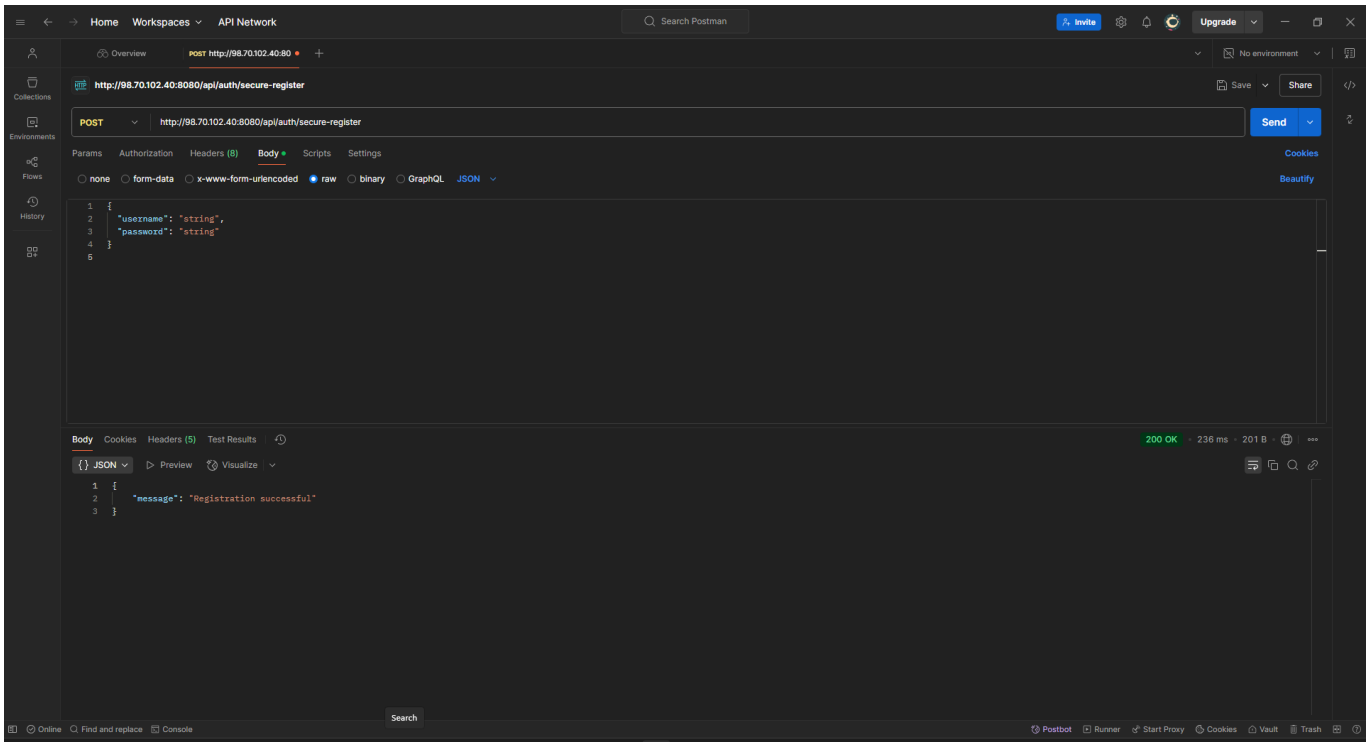

Figure 3: Secure Registration.

Endpoint Information

- **Endpoint:** `http://98.70.102.40:8080/api/auth/secure-register`
- **Method:** POST
- **Request Body (JSON):**

```
{
    "username": "string",
    "password": "string"
}
```

- **Response Body (JSON):**

```
{
    "message": "Registration successful"
}
```

Identified Vulnerabilities

1. **Unencrypted Communication (HTTP)**
   Despite enhanced password handling, the API uses `http://`, which means credentials and responses can be intercepted via Man-in-the-Middle (MitM) attacks.

2. **Weak BCrypt Configuration (Low Work Factor)**
   While BCrypt is secure, its effectiveness depends on a sufficiently high cost factor (work factor). If the work factor is set too low (e.g., 4 or 6), brute-force attacks become feasible with modern hardware.

3. **No Password Policy Enforcement**
   There is no indication of minimum password length, complexity, or validation rules. This allows users to register with weak passwords, even if they are hashed securely.

4. **Lack of Input Validation and Sanitization**
   Like previous endpoints, there is no sign of sanitizing inputs. This could lead to issues like:

   - Injection attacks (e.g., NoSQL injection if MongoDB is used)
   - Denial of service (e.g., extremely large payloads)

5. **No Rate Limiting or Bot Protection**
   Repeated registration attempts could be automated without proper rate-limiting or CAPTCHA, enabling abuse or enumeration.

6. **Generic Response Message**
   The message `"Registration successful"` does not reveal errors, but if different messages are shown for existing vs. new users, attackers can enumerate valid usernames.

Recommendations

- Always use **HTTPS** to secure data in transit.
- Configure **BCrypt with a high cost factor**, ideally 10 or more, depending on performance benchmarking.
- Enforce a **strong password policy**, including length, symbols, and complexity.
- Implement **input validation and sanitization** to prevent injection attacks.
- Add **rate limiting** and **CAPTCHA** mechanisms to mitigate automated abuse.
- Return **generic error responses** to avoid leaking information about existing users.
- Log and monitor registration behavior to detect abuse patterns.

2.4 Secure Login

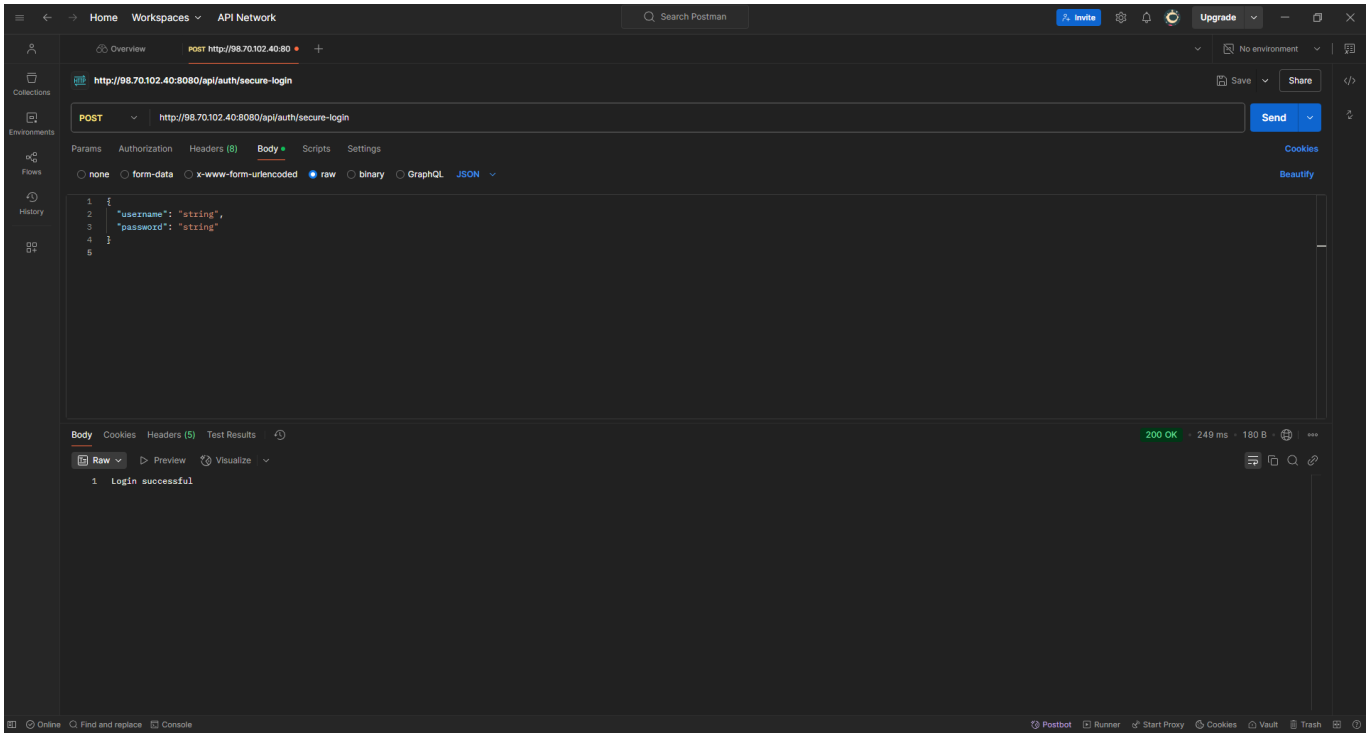

Figure 4: Secure Login.

Endpoint Information

- **Endpoint:** `http://98.70.102.40:8080/api/auth/secure-login`
- **Method:** POST
- **Request Body (JSON):**

```
{
    "username": "string",
    "password": "string"
}
```

- **Response Body (Raw Text):**

```
Login successful
```

Identified Vulnerabilities

1. **Unencrypted Communication (HTTP)**
   Like previous endpoints, this API still uses `http://`. Regardless of strong password hashing, credentials can be intercepted during transmission via Man-in-the-Middle (MitM) attacks.

2. **No Token-Based Authentication or Session Handling**
   The response returns a plain success message without a session token or JWT. This limits stateless authentication and may force reliance on insecure methods (e.g., basic auth, cookies without flags).

3. **No Brute-Force Mitigation**
   The API allows repeated login attempts without evidence of:

   - Rate limiting
   - Account lockout
   - CAPTCHA or delay mechanisms

   This makes the endpoint vulnerable to brute-force and credential stuffing attacks.

4. **No Response Standardization (JSON Missing)**
   The API returns a raw text response instead of a JSON object. This breaks consistency and could complicate error handling or logging for clients. JSON also allows including additional metadata (like status codes or expiration info).

5. **No Mention of MFA (Multi-Factor Authentication)**
   Despite stronger password handling, relying only on one factor (passwords) is not enough against modern threat models.

6. **No Error Differentiation Shown (Potential for Info Leak)**
   If the system provides different messages for invalid usernames vs. incorrect passwords (not shown here but plausible), it could leak user existence information.

Recommendations

- Implement **HTTPS** to secure all communications and prevent credential leakage.
- Use **token-based authentication** (e.g., JWT) or secure session handling with flags like `HttpOnly` and `Secure`.
- Apply **rate limiting, login attempt tracking**, and CAPTCHA after several failed attempts.
- Return responses in **standardized JSON format** for consistency and maintainability.
- Enforce **Multi-Factor Authentication (MFA)** for sensitive systems or high-privilege accounts.
- Use **generic login failure messages** to avoid exposing whether usernames exist.
- Monitor login activity and alert on suspicious patterns.

3 Smart Home Security

Smart homes offer amazing convenience—from remotely locking doors to adjusting your thermostat with a tap. But behind these features are APIs that, if poorly secured, can expose your home to serious risks. In this section, we will explore common API vulnerabilities in smart devices like door locks and thermostats, and show how to prevent them with better security practices.

3.1 Unlock a Smart Door

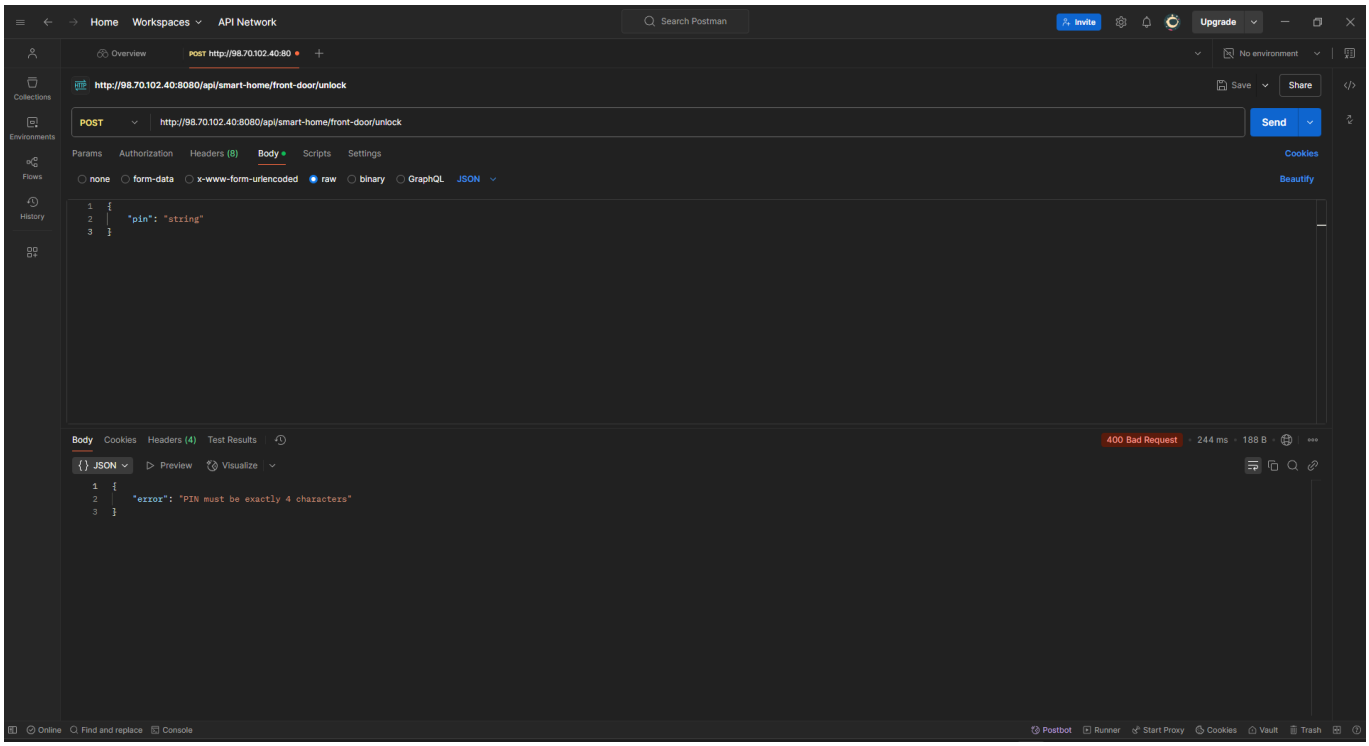

Figure 5: Unsuccessful pin.

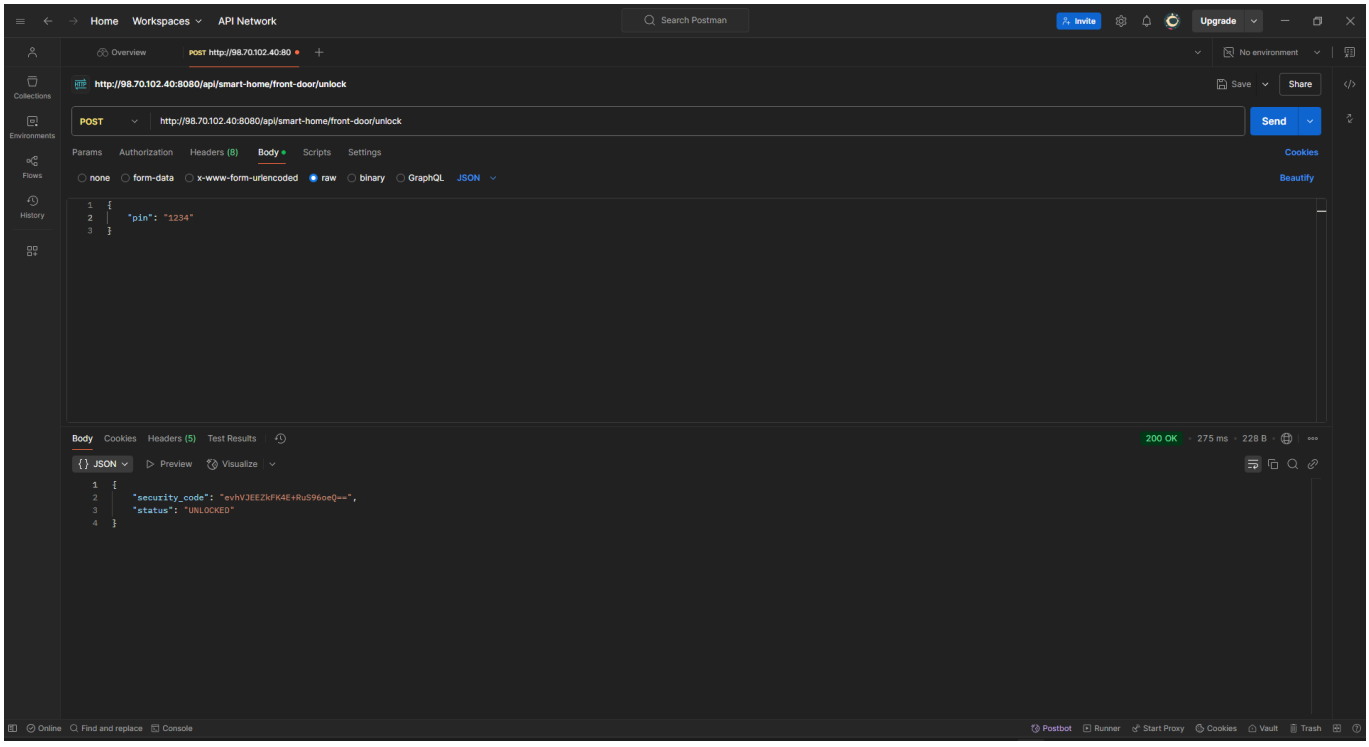

Figure 6: Successful Brute-Force Attack.

Endpoint Information

- **Endpoint:** `http://98.70.102.40:8080/api/smart-home/front-door/unlock`
- **Method:** POST

- **Request Body (JSON):**

```
{
    "pin": "1234"
}
```

- **Response Body (JSON):**

```
{
    "security_code": "<encrypted-value>",
    "status": "UNLOCKED"
}
```

Identified Vulnerabilities

1. **Hardcoded or Default PIN**
   The system unlocks with a generic PIN such as `1234`. This may indicate that a default value is being accepted or that user-specific PIN validation is occurring.

2. **Lack of Authentication or Authorization**
   No access token, API key, or user identity is required to access this endpoint. Anyone who discovers the endpoint can attempt to unlock the door.

3. **No Rate Limiting or Brute Force Protection**
   There is no indication of limits on the number of PIN entry attempts. Attackers could brute-force all 4-digit PIN combinations in a short time.

4. **Insecure Communication (HTTP)**
   The API uses `http://`, allowing sensitive data (PINs, security codes) to be intercepted during transmission.

5. **Overexposed Response Data**
   The response includes a `security_code`, which could potentially be misused or analyzed if improperly encrypted.

6. **Insufficient Input Validation Feedback**
   While the system enforces a 4-digit length check, the error messages are too descriptive. Overly specific feedback can help attackers craft a valid input.

Recommendations

- Remove or randomize default PINs; enforce per-user setup during device initialization.
- Require proper authentication (e.g., bearer tokens or device binding) before allowing unlock attempts.
- Implement brute-force protection: rate limiting, lockouts, and alerts.
- Use HTTPS to encrypt API traffic and protect sensitive input and response data.
- Return minimal error information to avoid aiding attackers.
- Encrypt and validate any `security_code` values securely, and ensure they do not leak useful information.

3.2 Garage Door Controller

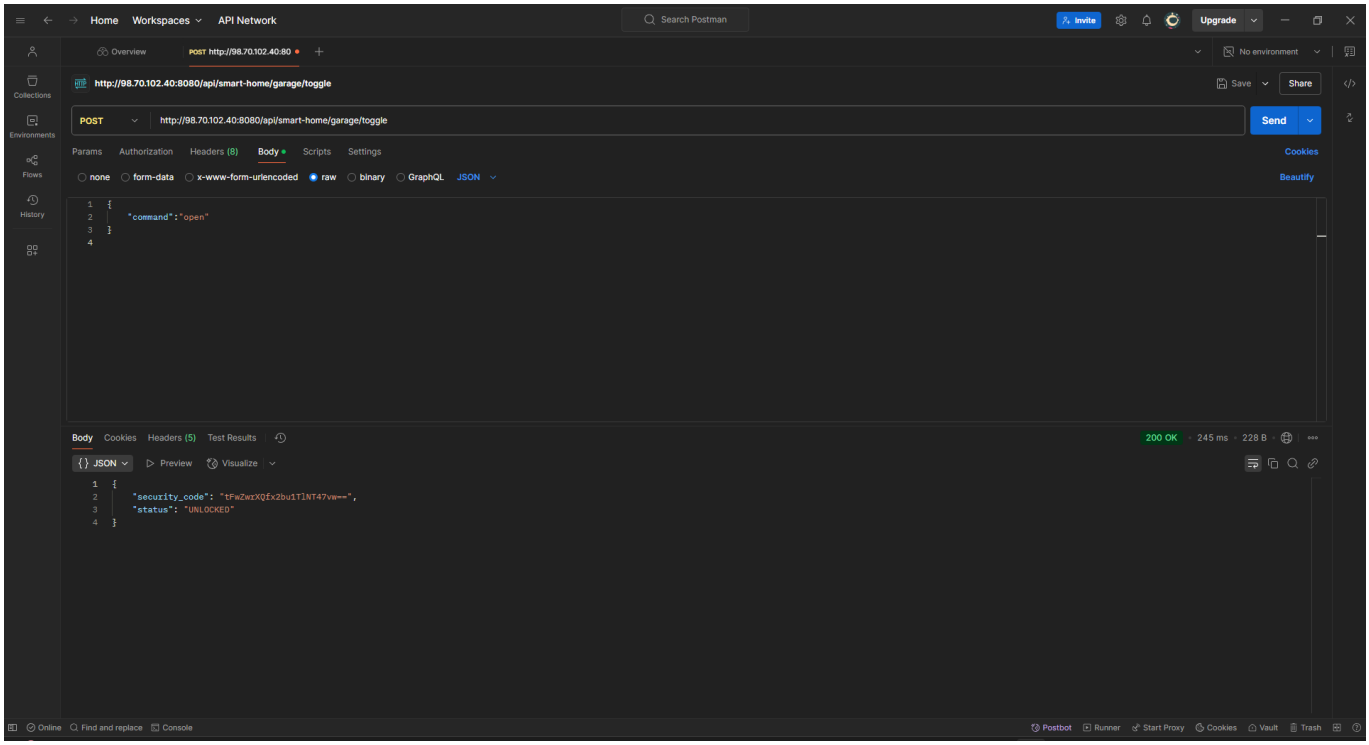

Figure 7: Garage Door Controller.

Endpoint Information

- **Endpoint:** `http://98.70.102.40:8080/api/smart-home/garage/toggle`
- **Method:** POST
- **Request Body (JSON):**

```
{
    "command": "open"
}
```

- **Response Body (JSON):**

```
{
    "security_code": "<encoded-value>",
    "status": "UNLOCKED"
}
```

Identified Vulnerabilities

1. **Lack of Authentication or Authorization**
   The endpoint does not require any form of user verification. Anyone who discovers the URL can issue commands to the garage door.

2. **Unencrypted Communication (HTTP)**
   The use of `http://` makes all traffic—including commands and security codes—vulnerable to interception via Man-in-the-Middle (MitM) attacks.

3. **Overly Simple and Unvalidated Commands**
   Accepting basic strings like `"open"` without any command validation or structure may lead to misuse or unexpected behavior. It also invites injection-style input manipulation.

4. **No Rate Limiting or Logging Mechanism**
   There is no mention of limiting how often commands can be sent, nor is there evidence of logging or monitoring access. This allows for abuse or potential denial-of-service (DoS) conditions.

5. **Exposed Security Code**
   The API responds with a `security_code` that may be predictable or insufficiently encrypted. If attackers decode or reuse it, they may bypass protections.

6. **No Context Awareness or Access Restrictions**
   Commands can be issued without validating device location, time, or authorized zones, allowing remote triggering from unauthorized sources.

Recommendations

- Implement **authentication and access control** using tokens, user sessions, or signed device identifiers.
- Use **HTTPS** for all API communications to protect data in transit.
- Validate and sanitize all **command inputs**; consider replacing free-form strings with predefined, structured commands.
- Add **rate limiting**, audit logging, and alerting to detect misuse or anomalies.
- Avoid sending sensitive data (like `security_code`) unless necessary and ensure it's strongly encrypted.
- Apply context-aware restrictions (e.g., geo-fencing, time-based access) to sensitive device actions.

3.3 Thermostat Data Reporting

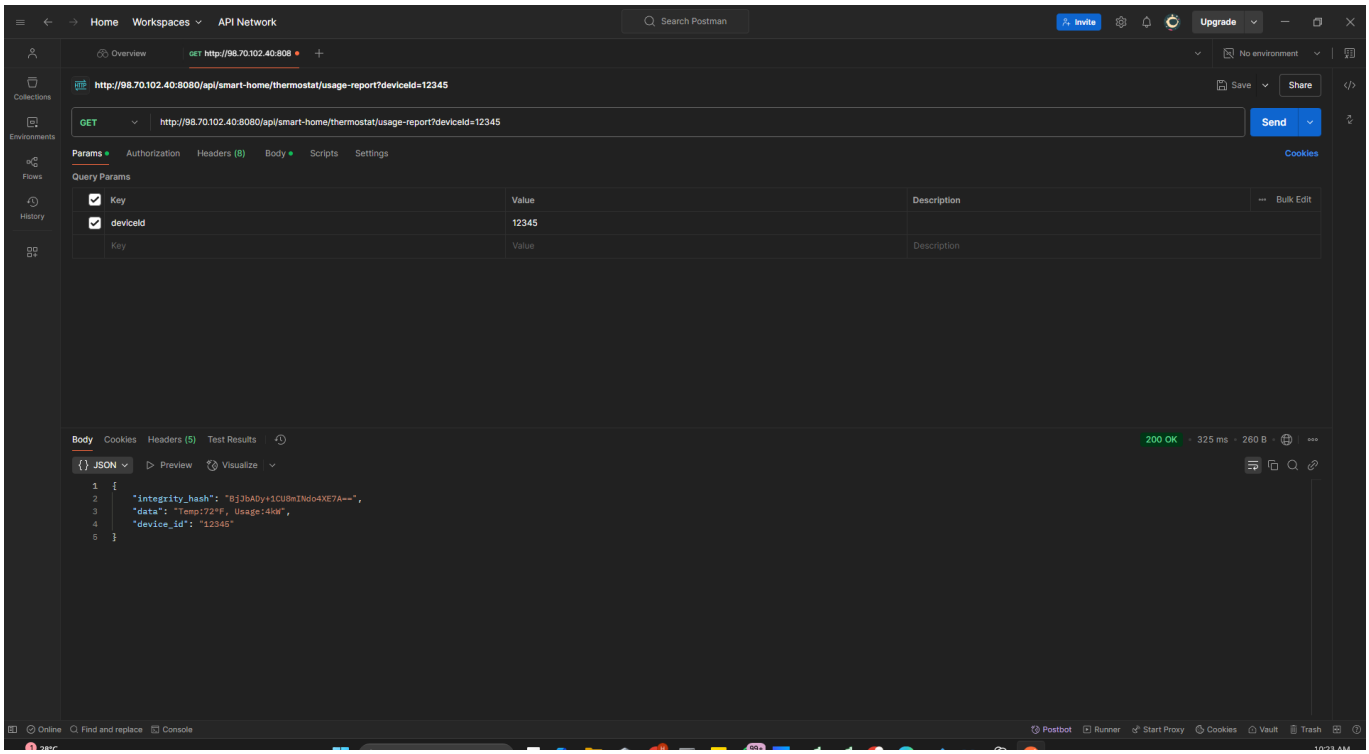

Figure 8: Thermostat Data Reporting.

Endpoint Information

- **Endpoint:** `http://98.70.102.40:8080/api/smart-home/thermostat/usage-report`
- **Method:** GET
- **Query Parameter:** `deviceId` (e.g., 12345)
- **Response (JSON):**

```
{
    "integrity_hash": "BjbADy+1CUBNIndo4XE7Aw==",
    "data": "Temp:72°F, Usage:4kW",
    "device_id": "12345"
}
```

Identified Vulnerabilities

1. **Unauthenticated Access**
   No authentication mechanism is in place to verify the identity of the requester. Anyone with the device ID can retrieve thermostat data.

2. **Insecure Communication (HTTP)**
   The API is served over `http://`, allowing data to be intercepted by attackers via Man-in-the-Middle (MitM) attacks.

3. **Insecure Query Parameter Exposure**
   Using a simple numeric `deviceId` in the URL makes it easy to enumerate devices by incrementing values (e.g., 12346, 12347, etc.).

4. **Lack of Access Control**
   There is no authorization check to verify whether the requester is allowed to view the data for a given device ID.

5. **No Rate Limiting or Monitoring**
   There is no evidence of rate limiting or monitoring mechanisms to detect or block scraping or enumeration attacks.

6. **Potentially Weak Integrity Check**
   The `integrity_hash` is included, but the method of generation and its purpose are unclear. If it is reversible or guessable, it provides little security benefit.

Recommendations

- Require **authentication** using API keys, OAuth tokens, or user sessions.
- Serve the API over **HTTPS** to protect data in transit.
- Use **non-predictable device identifiers**, such as UUIDs, instead of sequential numeric IDs.
- Enforce **authorization checks** to restrict access to device data based on user ownership or roles.
- Implement **rate limiting** and logging to detect and respond to abuse or enumeration.
- Clearly define and secure the **integrity hash**, or replace it with cryptographic signing if data tampering protection is needed.

3.4 Security Camera Feed

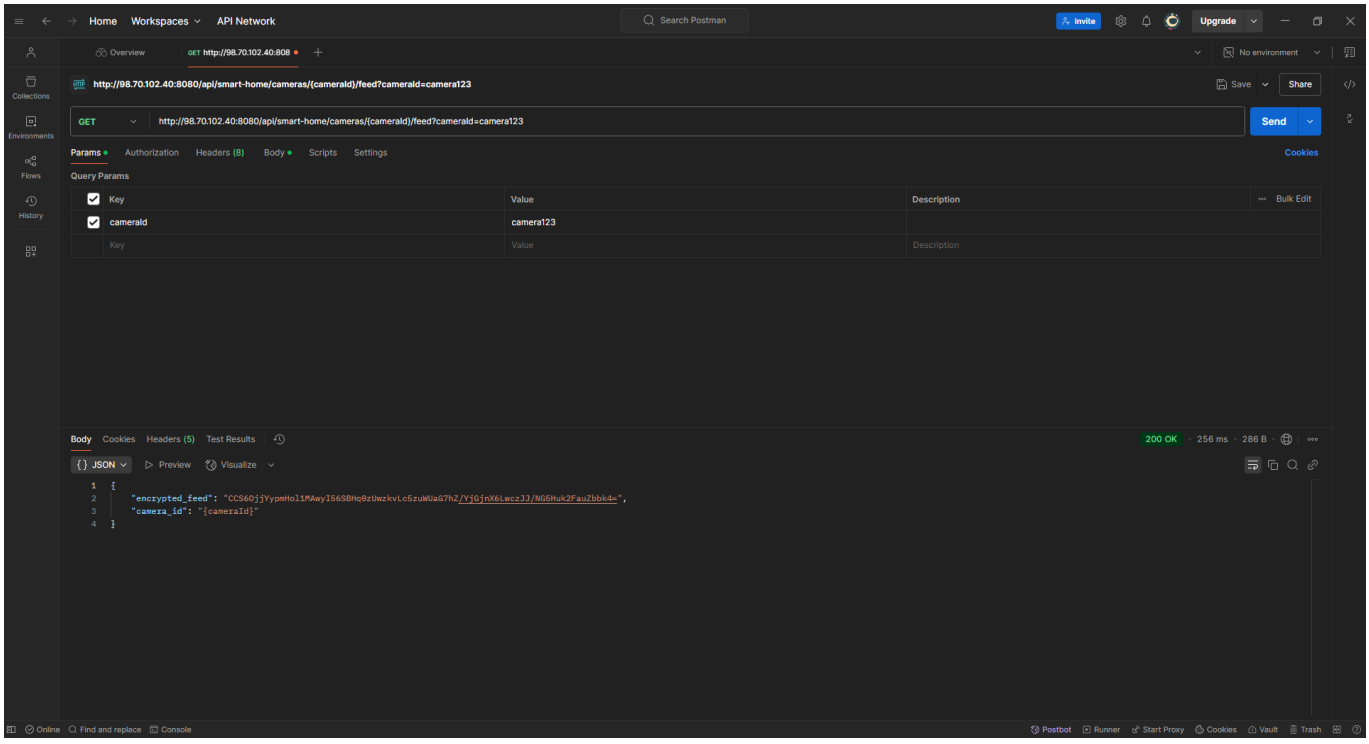

Figure 9: Security Camera Feed.

Endpoint Information

- **Endpoint:** `http://98.70.102.40:8080/api/smart-home/cameras/cameraId/feed`
- **Method:** GET
- **Query Parameter:** `cameraId`
- **Sample Response (JSON):**


```
{
    "encrypted_feed": "CCS6OjjYypmHol1MAwyI56SBHq0zUwzkvLc5zuWUaG
    76jnBlgxzAkIGvS0lUXaU5uk2FauZbbk4=",
    "camera_id": "cameraId"
}
```


Identified Vulnerabilities

1. **Unauthenticated Access**
   No form of authentication is required. Any user who knows or guesses the camera ID can potentially access the encrypted feed.

2. **Insecure Communication (HTTP)**
   The API is accessed via `http://`, making requests and responses vulnerable to interception and tampering through Man-in-the-Middle (MitM) attacks.

3. **Predictable or Exposed Identifiers**
   Camera IDs like `camera123` are easy to guess or enumerate. This increases the likelihood of unauthorized access through brute-force attempts.

4. **No Authorization Enforcement**
   The API does not validate whether the requesting user is authorized to access the specified camera feed. This flaw can lead to cross-device data exposure.

5. **Weak or Repetitive Encryption Pattern**
   The encrypted feed returned by the API consistently begins with the same base64-encoded prefix:

   `"CCS6OjjYypmHol1MAwyI56SBHq0zUwzkvLc5zuWUaG..."`

   Only the trailing portion of the string changes slightly between responses. This suggests that:

   - The same encryption key and IV (initialization vector) are likely reused.
   - The encrypted content has low variability, or the encryption mode may be deterministic (e.g., ECB).
   - Replay or pattern analysis attacks may be feasible, weakening overall feed confidentiality.

6. **Exposed Metadata**
   Even though the feed is encrypted, the response reveals the camera ID and consistent structure. This can leak system architecture or usage patterns.

Recommendations

- Enforce strong **authentication** (e.g., OAuth tokens, JWT) to restrict access to authorized users only.
- Use **HTTPS** to protect data in transit and prevent MitM attacks.
- Replace simple camera IDs with **non-predictable identifiers** such as UUIDs.
- Implement strict **authorization checks** to ensure users can only access their own devices.
- Encrypt video feeds using secure, randomized, and **end-to-end encryption algorithms** (e.g., AES-256-GCM with unique IVs).
- Avoid repeating encryption patterns by applying proper cryptographic best practices, including IV rotation and padding.
- Minimize response metadata and log access for monitoring and auditing purposes.

3.5 Firmware Update Endpoint

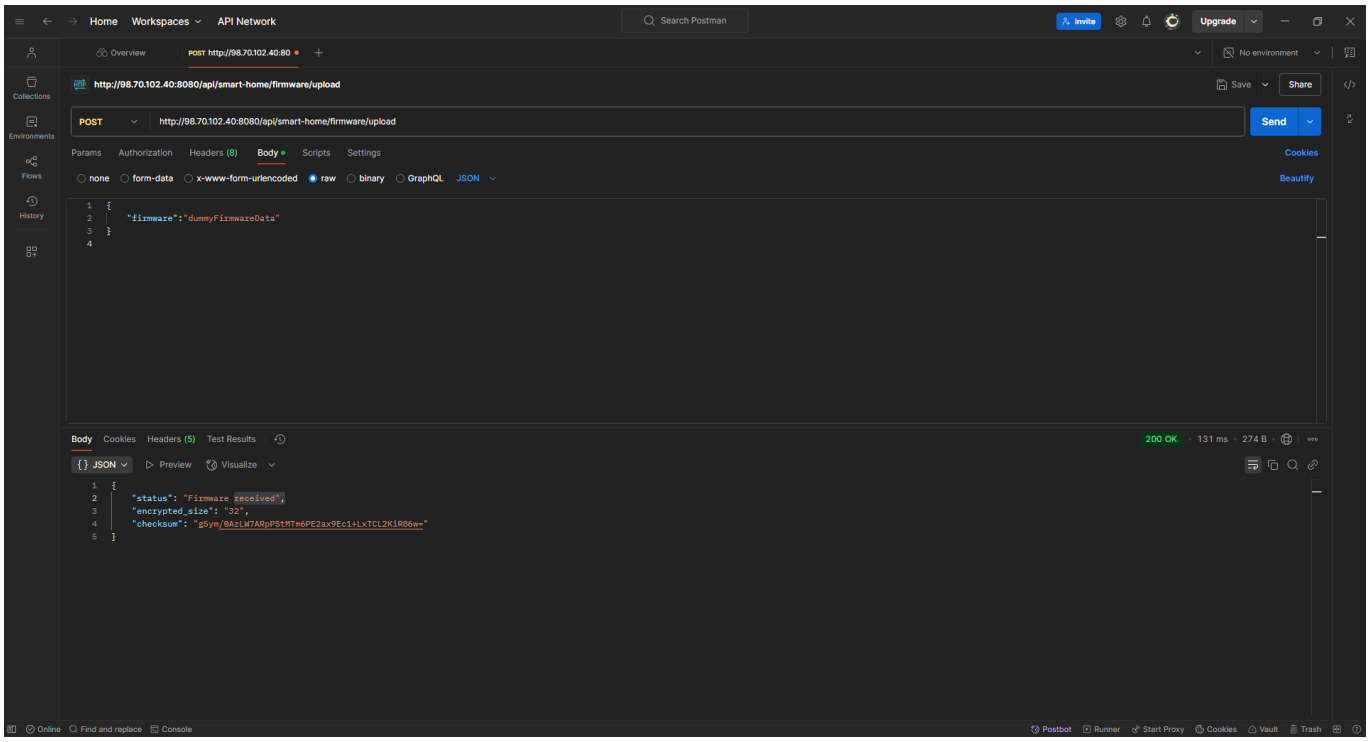

Figure 10: Firmware Update.

Endpoint Information

- **Endpoint:** `http://98.70.102.40:8080/api/smart-home/firmware/upload`
- **Method:** POST
- **Request Body (JSON):**

```
{
    "firmware": "dummyFirmwareData"
}
```

- **Response Body (JSON):**

```
{
    "status": "Firmware received",
    "encrypted_size": "32",
    "checksum": "g5ym/0aZLk7ARPpDStMTm6PE23x9E1sLxtYCLK2lR86w="
}
```

Identified Vulnerabilities

1. **Unauthenticated Firmware Upload**
   No authentication or authorization is enforced, allowing any unauthenticated actor to submit firmware. This could lead to unauthorized firmware injection or device hijacking.

2. **Insecure Communication (HTTP)**
   Using `http://` exposes the firmware payload and metadata to interception, tampering, and replay attacks through Man-in-the-Middle (MitM) attacks.

3. **Lack of Firmware Signature Validation**
   While the server returns a checksum, there is no mention of digital signature verification. Checksums alone cannot guarantee firmware authenticity or integrity, and they can be spoofed.

4. **No Firmware Version or Source Verification**
   There is no validation of the firmware version or source. Devices may unknowingly accept downgraded or third-party firmware unless versioning and trusted sources are enforced.

5. **Exposed Metadata**
   The API response includes checksum and encrypted size data, which could help attackers understand or reverse-engineer encrypted payloads over time through side-channel analysis.

6. **No Access Control for Target Device Binding**
   The upload request does not specify or restrict which device the firmware is intended for. Without binding or scoping to a specific authenticated device, an attacker could push rogue updates to multiple endpoints.

Recommendations

- Require strong **authentication and authorization** before allowing firmware uploads.
- Enforce secure communication using **HTTPS** to protect firmware in transit.
- Implement **digital signature verification** on the device to validate the authenticity of firmware using PKI (public-key infrastructure).
- Add **firmware version checks and rollback protection** to prevent downgrades and replay attacks.
- Remove unnecessary metadata from API responses or encrypt them if needed.
- Enforce **device-scoped access controls** so that firmware uploads are only accepted for authenticated and registered devices.
- Maintain audit logs of firmware uploads and verify the source and destination.

4 Conclusion

As this documentation has demonstrated, even in highly innovative environments such as smart homes, API misconfigurations and weak security practices can pose serious threats to user privacy, safety and system integrity. From hardcoded credentials and lack of encryption to missing authentication and weak validation mechanisms, small oversights in API design can have wide-reaching consequences.

Securing APIs is not just a technical necessity; it's a critical responsibility. Every component, from door locks to thermostats and firmware update endpoints, must be designed with a security-first mindset. Implementing strong authentication, enforcing encryption, validating inputs, and applying proper access control are foundational steps toward building resilient systems.

As smart homes continue to evolve, this document serves as a reminder that convenience must never come at the cost of security. Through consistent audits, secure coding practices, and a proactive approach to identifying and mitigating vulnerabilities, we can ensure that the connected homes of the future are not only smart but safe.