Course: Data Integrity and Authentication

Assignment: Demonstrate and Mitigate a MAC Forgery Attack

Team Members:

Yehia Tarek - Sara Ahmed - Shadwa Ahmed

1. Overview

This project demonstrates a **length extension attack** on a naive implementation of MAC (Message Authentication Code) using **MD5** and illustrates how it can be mitigated using **HMAC**. The attacker attempts to forge a valid MAC by appending data without knowing the secret key.

---

2. File Descriptions

◈ server.py: Vulnerable MAC Verification

```python
def generate_mac(message: bytes) -> str:
    return hashlib.md5(SECRET_KEY + message).hexdigest()

```

- **MAC Calculation**: Simply hashes SECRET_KEY || message.
- **Vulnerability**: Susceptible to **length extension attacks** because MD5 processes input in blocks, and the internal state is exposed in the hash.

client.py: Simulated MAC Forgery Attack

```python
def perform_attack():
```

Constructs a forged message using MD5 padding

- Simulates an attacker guessing the key length and generating a valid-looking message:
    - Appends malicious data (e.g., &admin=true)
    - Uses original MAC as the forged MAC (since MD5 is vulnerable to length extension).
- Attempts to bypass server verification without knowing the key.

secure_server.py: Secure MAC Verification with HMAC

```python
def generate_mac(message: bytes) -> str:
    return hmac.new(SECRET_KEY, message, hashlib.md5).hexdigest()
```

- Uses HMAC which protects against length extension by applying hash functions in a nested, key-dependent manner.
- Verification uses hmac.compare_digest() to prevent timing attacks.

---

3. 🛠 Explanation of the Attack

Length Extension Attack Flow

- The attacker cannot see SECRET_KEY but can see MAC = MD5(SECRET_KEY || message).
- Because of how MD5 processes input, the attacker can:
    1. Guess the key length.
    2. Append extra data after correct padding.
    3. Use the original MAC as the base hash to extend it using MD5's properties.
    4. Fool the naive server into accepting the tampered message.

---

4. Output Observations

🔶 server.py Output

```
└─$ python3 server.py

== Server Simulation ==
Original message: amount=100&to=alice
MAC: 614d28d808af46d3702fe35fae67267c
```

--- Verifying forged message ---

MAC verification failed (as expected).

- The forged message is rejected because it doesn't include correct padding and the MAC doesn't match.

🔶 client.py Output

```
┌──(kali㉿kali)-[~/mac_forgery_demo]
└─$ python3 client.py

=== Simulated MAC Forgery ===

[+] Trying key length = 8
Forged message: b'amount=100&to=alice\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xd8\x00\x00\x00\x00\x00\x00\x00&admin=true'
Forged message (hex): 616d6f756e743d3130302674 6f3d616c69636580000000000000000000000000000000000000000000000000000
000d8000000000000002661646d696e3d74727565
Use original MAC: 614d28d808af46d3702fe35fae67267c
```

- Shows how an attacker could **construct a valid-looking message**.
- Note: The MAC is not recalculated, just reused from the original.

🔶 secure_server.py Output

```
┌──(kali㉿kali)-[~/mac_forgery_demo]
└─$ python3 secure_server.py

=== Secure Server ===
Original message: amount=100&to=alice
MAC: 616843154afc11960423deb0795b1e68

--- Verifying forged message ---
MAC verification failed ✅ (secure).
```

--- Verifying forged message ---

MAC verification failed (secure).

- The HMAC system **correctly rejects** the forged message.

---

5. Conclusion

Implementation	Secure?	Vulnerable to Length Extension?
server.py	❌ No	✅ Yes
client.py	⚠️ Attack Simulation	-
secure_server.py	✅ Yes	❌ No

- Naive use of MD5(secret || message) is insecure.
- Always use HMAC for message authentication. It's resilient against padding and length extension attacks.
- Avoid legacy hash functions like **MD5** and prefer **SHA-256** or stronger algorithms.

Keywords:

- MAC (Message Authentication Code)
- HMAC (Keyed-Hash Message Authentication Code)
- Length Extension Attack
- MD5 Vulnerability
- Python Cryptography