Length Extension Attack Demonstration

This project demonstrates a **length extension attack** on an insecure **Message Authentication Code (MAC)** implementation and shows how **HMAC** mitigates this vulnerability. It includes three Python scripts: client.py, server.py, and server1.py.

Purpose

- **Show the vulnerability**: Illustrate how an attacker can exploit a weak MAC construction (hash(secret || message)) to forge messages.
- **Demonstrate the fix**: Show how using **HMAC** prevents the attack, ensuring **message integrity** and **authentication**.

Prerequisites

- **Python 3.x** installed.
- The hashpumpy library for the attack simulation. Install it using:

```
pip install hashpumpy
```

Project Structure

- client.py: Simulates an attacker performing a **length extension attack** to forge a message and MAC.
- server.py: Implements an insecure MAC using hashlib.md5(secret + message), vulnerable to the attack.
- server1.py: Implements a secure MAC using hmac with MD5, resistant to the attack.

How to Run

1. **Clone or download** the project files to your local machine.
2. **Install dependencies**:

```
pip install hashpumpy
```

3. **Run the scripts** in the following order:

Step 1: Run the insecure server simulation

```
python server.py
```

   - This simulates a server generating and verifying a MAC for the message "amount=100&to=alice".

- It also tests a forged message ("amount=100&to=alice&admin=true") with the original MAC, which fails (as expected).

Output:

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

--- Verifying legitimate message ---
MAC verified successfully. Message is authentic.

--- Verifying forged message ---
MAC verification failed (as expected).
```

Step 2: Run the attack

```
python client.py
```

- This performs a **length extension attack**, appending "&admin=true" to the message and computing a new valid MAC.
- It verifies the forged message and MAC using *server.py*'s verify function, which succeeds, showing the vulnerability.

Output:

```
Forged message: [extended message with padding and &admin=true]
Forged MAC: [new MAC]
Attack successful! Forged MAC is accepted by the server.
```

Step 3: Run the secure server simulation

```
python server1.py
```

- This uses **HMAC** to generate and verify the MAC for "amount=100&to=alice".
- It tests the same forged message from *client.py*, which fails, showing **HMAC**'s resistance to the attack.

Output:

```
=== Server Simulation with HMAC ===
Original message: amount=100&to=alice
MAC: [HMAC value]

--- Verifying legitimate message ---
MAC verified successfully. Message is authentic.

--- Verifying forged message ---
MAC verification failed (as expected).
```

How It Works

- **Insecure MAC (server.py)**: Uses hashlib.md5(secret + message) to create a MAC. This is vulnerable because **MD5** exposes its internal state, allowing an attacker to append data and compute a new valid MAC without the secret key.
- **Attack (client.py)**: Uses hashpumpy to perform a **length extension attack**, forging a message by appending "&admin=true" and generating a valid MAC for *server.py*.
- **Secure HMAC (server1.py)**: Uses hmac.new(secret, message, hashlib.md5) to create a secure MAC. **HMAC** applies the hash function twice with inner and outer keys, preventing the attack because the secret key is required to forge a valid MAC.

Key Files

- client.py:
  - Performs the **length extension attack**.
  - Key function: hashpumpy.hashpump to forge the message and MAC.
- server.py:
  - Simulates an insecure server with a vulnerable MAC.
  - Key function: generate_mac using hashlib.md5.
- server1.py:
  - Simulates a secure server with **HMAC**.
  - Key function: generate_mac using hmac.new.

Notes

- The **secret key** in both servers is "supersecretkey" (14 bytes), assumed known for the attack's length parameter.
- The attack in *client.py* only works against *server.py*, not *server1.py*, demonstrating **HMAC**'s security.
- Ensure hashpumpy is installed, as it's critical for *client.py*.