Demonstrating and Mitigating a Message Integrity Attack (MAC Forgery)

(Optional/Bonus Assignment)

Course: Data Integrity and Authentication

Weight: 10 points

Estimated Time Required: 2-3 days

Team: exactly **THREE** members

Submission Form: https://forms.office.com/r/J3gpafSnFk

Submission Deadline: Friday 16th May, 11:59 pm

Discussions will take place in the order of form responses. The first discussion slot is on Saturday, May 10th, from 2:00 to 3:00 pm for group 1 and from 4:00 to 5:00 pm for group 2. Additional discussion slots will be available on Saturday, May 17th, during lecture time.

Assignment Overview

In this assignment, you will demonstrate a **Message Authentication Code (MAC) forgery attack** against an insecure implementation of a MAC scheme and propose effective mitigations. You will analyze:

- why the attack works,
- demonstrate it practically, and
- show how stronger cryptographic practices prevent such vulnerabilities.

The attack focus will be Length Extension Attack on constructions like naive hash-based MACs (e.g., MAC = hash(secret || message)).

Assignment Tasks

1. Background Study (1–2 pages):
    a. What is a MAC and its purpose in data integrity/authentication?
    b. How does a length extension attack work in hash functions like MD5/SHA1?
    c. Why is MAC = hash(secret || message) insecure?
2. Demonstration of the Attack (Code + Documentation):
    a. Intercept a valid (message, MAC) pair.
    b. Perform a length extension attack to append new data to the message.
    c. Generate a valid MAC for the extended message **without knowing the secret key**.
    d. Demonstrate that the server accepts your forged message+MAC as valid.
3. Mitigation and Defense (Code + Explanation write-up)
    a. Modify the system to use HMAC properly.
    b. Demonstrate that your previous attack fails against this secure implementation.
    c. Explain why HMAC mitigates this attack in 1–2 pages.

Deliverables:

1. Background write-up (PDF)
2. Attack demonstration (Python code + README.md)
3. Mitigated secure implementation (Python code)
4. Mitigation write-up

Starter Code:

```
## `server.py`

import hashlib

SECRET_KEY = b'supersecretkey'  # Unknown to attacker

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

def verify(message: bytes, mac: str) -> bool:
    expected_mac = generate_mac(message)
    return mac == expected_mac

def main():
    # Example message
    message = b"amount=100&to=alice"
    mac = generate_mac(message)

    print("=== Server Simulation ===")
    print(f"Original message: {message.decode()}")
    print(f"MAC: {mac}")
    print("\n--- Verifying legitimate message ---")
    if verify(message, mac):
        print("MAC verified successfully. Message is authentic.\n")

    # Simulated attacker-forged message
    forged_message = b"amount=100&to=alice" + b"&admin=true"
    forged_mac = mac  # Attacker provides same MAC (initially)

    print("--- Verifying forged message ---")
    if verify(forged_message, forged_mac):
        print("MAC verified successfully (unexpected).")
    else:
        print("MAC verification failed (as expected).")

if __name__ == "__main__":
    main()
```

`client.py` (Attacker script)

```
import sys

def perform_attack():
    # TODO: Your attack code goes here
    # You have intercepted: message and its mac
    intercepted_message = b"amount=100&to=alice"
    intercepted_mac = "..."  # From server.py output

    # Your goal: append data and compute valid forged MAC
    data_to_append = b"&admin=true"

    forged_message = b""  # Replace with your forged message
    forged_mac = ""       # Replace with your forged MAC

    print("Forged message:", forged_message)
    print("Forged MAC:", forged_mac)

if __name__ == "__main__":
    perform_attack()
```

Next, use the **verify(message: bytes, mac: str)** method from server.py to confirm that the malicious MAC is valid, even without knowing the secret key

Modify the server.py file to mitigate the attack