**Team Members:**

Mark Magdy – 2205040, Arsany Osama – 2205122, Rana Ashraf – 2205019

# Message Integrity Attack (MAC Forgery)

## The Authenticity Problem

### Why Message Authenticity Matters

When a message is sent over the internet, we need to ensure that:

1. The message has not been altered.
2. The message came from the expected sender.

### Ineffective Solutions

- Solution 1: The sender confirms in person. → Not practical.
- Solution 2: Sender includes a shared key. → The attacker can intercept it during transmission.

### Real Solution: Message Authentication Code (MAC)

A MAC is a cryptographic checksum created using a shared secret key and a message. It provides:

- Integrity: Detects message tampering.
- Authenticity: Ensures the message came from someone who knows the key.

General Form:

MAC = MAC_Algorithm (K, M) **Where:** K = shared secret key, M = Original Message

## MAC Verification Process

The receiver:

1. Recomputes the MAC using the shared key.
2. Compares it to the received MAC: **If (T == T'):** Message is authentic. **Else:** Message may be tampered with.

## MAC Vulnerabilities

1. Weak hash function (e.g., MD5, SHA-1).
2. Flawed implementation.
3. Length Extension Attack.
4. Known-message MACs available to attacker.

## Length Extension Attack

Vulnerable Scheme: MAC = Hash(secret_key || message)

This is insecure if the hash uses the Merkle–Damgård construction (e.g., MD5, SHA-1, SHA-256).

## Merkle–Damgård Construction

Used in many hash functions to turn **variable-length input** into a **fixed-length output.**

**Steps:**

1. Pad message to match block size.
2. Split into blocks: M1, M2, ..., Mn.
3. Apply compression function iteratively. [ H0 = IV, H1 = f(H0, M1), H2 = f(H1, M2), ... Hn = Final hash ]
4. Output final hash.

## How Length Extension Works

If attacker knows:

- MAC = Hash (K || message)
- Original message
- The hash function used like MD5 or SHA1

Then they can:
1. Guess key length (e.g., 16 bytes).
2. Reproduce the padding that the hash function applies )K || message || padding(.
3. Append extra_data.
4. Resume hashing from the internal state (without knowing the key).
5. Compute a valid MAC for the forged message (message || padding || extra_data).

## Simulation Example Using hashpumpy

```python
import hashpumpy
import hashlib

# Simulate a secret key
secret_key = b'secret!!'  # 8 bytes
key_length = len(secret_key)

# Original message
original_msg = "data=amount100"

# Compute original MAC: MD5(secret_key || message)
original_mac = hashlib.md5(secret_key + original_msg.encode()).hexdigest()
print("[+] Original MAC:", original_mac)

# Data to append
data_to_append = "&admin=true"

# Perform length extension attack
new_mac, new_message = hashpumpy.hashpump(original_mac, original_msg, data_to_append, key_length)

print("[+] New MAC:", new_mac)
print("[+] Forged message (raw bytes):", new_message)
print("[+] Forged message (printable):", repr(new_message))

is_valid = hashlib.md5(secret_key + new_message.encode()).hexdigest() == new_mac
print("[+] Is forged message valid (from server perspective)?", is_valid)
```

## Output

[+] Original MAC: 4a7d1ed414474e4033ac29ccb8653d9b
[+] Forged MAC: 51f27c8e94f64cf6175ba3dd30fd2a36
[+] Forged Message: b'data=amount100\x80...&admin=true'
[+] Is Forged MAC Valid?: True

The attacker successfully appended `&admin=true` to the original message and generated a valid MAC — all **without knowing the secret key**.

# Demonstration of the Attack

## 1. Overview of the Cryptographic Vulnerability

### 1.1 What is a Length Extension Attack?

A **Length Extension Attack** exploits the structure of certain hash functions like MD5 and SHA-1 that use the **Merkle-Damgård construction**. This attack allows an adversary to:

- Append arbitrary data to a message
- Generate a valid Message Authentication Code (MAC) *without knowing the secret key*

The vulnerability arises when MACs are computed as:

```
MAC = H(secret_key || message)
```

where H is a Merkle-Damgård hash function.

### 1.2 Why MD5 is Vulnerable

- **Merkle-Damgård Internals**: MD5 processes data in fixed-size blocks and maintains an internal state between blocks.
- **State Exposure**: The final hash value directly encodes the internal state after processing `secret_key || message`.
- **Predictable Padding**: MD5 adds padding that depends on the *total message length*, which attackers can reconstruct if they guess the secret key length.

## 2. Attack Workflow

### 2.1 Components

| **File** | **Role** |
|---|---|
| `server.py` | Insecure server using `MD5(secret + message)` |
| `client.py` | Attacker tool exploiting length extension |
| `server_hmac.py` | Secure server using HMAC-MD5 |

## 2.2 Attack Steps

1. **Intercept Legitimate Traffic**
   a. Attacker obtains a valid (message, MAC) pair:

```
=== Server Simulation ===

Original message: amount=100&to=alice

MAC: 614d28d808af46d3702fe35fae67267c
```

2. **Craft Malicious Extension**
   a. Goal: Append `&admin=true` to grant admin privileges
   b. Required knowledge:
      i. Original message
      ii. Original MAC
      iii. Secret key length (guessed as 14 bytes for "supersecretkey")
3. **Exploit Hash Padding**
   MD5 padding for a 33-byte message (14-byte secret + 19-byte original message):
4. **Generate Forged Message**

```
new_mac, new_message = hashpumpy.hashpump(
        intercepted_mac,      # Original MAC (hex string)
        intercepted_message,  # Original message (bytes)
        data_to_append,       # Data to append (bytes)
        key_length            # Secret key length
    )
```

Output:

```
Forged message: amount=100&to=alice\x80...&admin=true

Forged MAC: 97312a73075b6e1589117ce55e0a3ca6
```

5. **Bypass Verification**

Server naively recomputes MD5(secret + forged_message):

```python
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
```

## 3. Why the Attack Works

### 3.1 Merkle-Damgård Internals

1. **Initialization Vector (IV)**: MD5 starts with a fixed IV.
2. **Block Processing**: The hash function updates an internal state by applying a compression function to the current state and each message block, starting from an IV and returning the final state as the hash output.
3. **Final State Leakage**: The MAC directly represents the final state.

### 3.2 Attack Mathematics

Given:

- Original MAC = `MD5(secret || message)`
- Attacker knows `message` and `MAC`

They compute:

```
new_mac = MD5(secret || message || padding || appended_data)
```

Without knowing `secret`, by:

1. Setting initial state = `MAC` (instead of IV)
2. Processing `padding || appended_data` as new blocks