# Report

## Demonstrating and Mitigating a Message Integrity Attack (MAC Forgery)

## Team Members


| | |
|---|---|
| **Hossam Ahmed Aldesouky** | **2205097** |
| **Omar Hossam Ahmed** | **2205150** |
| **Tarek Gamal Elkelany** | **2205029** |

**Introduction to MAC (Message Authentication Code)**

# What is a MAC?

A **Message Authentication Code (MAC)** is a short piece of information used to **verify the integrity and authenticity** of a message. It ensures that the message has not been altered and that it originates from a trusted source.

# Purpose of a MAC:

- **Data Integrity:** Ensures the message has not been tampered with during transmission.
- **Authentication:** Verifies the message was generated by someone who possesses the shared secret key.
- **Non-repudiation (to some extent):** Prevents the sender from denying their involvement (when combined with other methods).

# How MACs Work:

A MAC is generated by computing a cryptographic function over the **message** and a **secret key**.

This MAC is then sent along with the message.

The receiver recalculates the MAC using the same key and compares it with the received MAC. If they match, the message is accepted.

# Insecure MAC Construction

## Naive Construction Example:

MAC = hash(secret || message)

## Why it's Insecure:

Relies on appending the secret to the message before hashing.

Vulnerable to attacks like *Length Extension* due to the way hash functions process data.

## Common Mistake:

Misusing general-purpose hash functions like MD5 or SHA1 directly for authentication

# Understanding Length Extension Attacks

## **Overview**:

- Exploits the internal structure of hash functions based on the Merkle–Damgård construction.

- Allows attacker to compute

hash(secret || message || extension) from hash(secret || message).

## **Example**:

- Intercepted message: amount=100&to=alice

- Attacker appends: &admin=true

- Computes valid MAC for extended message without secret.

**Hash Functions Affected**: MD5, SHA1

# Attack Scenario - Step-by-Step

- **Step 1**: Intercept a valid (message, MAC) pair.

- **Step 2**: Attacker crafts new message:

  message || extra_data.

- **Step 3**: Exploit hash function to compute new valid MAC.

- **Step 4**: Send forged

  (new_message, forged_mac) to server.

## **Goal**:

Trick server into accepting a tampered message.

## The Vulnerable Server Code (server.py)

```python
import hashlib
SECRET_KEY = b'supersecretkey'

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
```

- **Problem**:

  Secret is prepended to message in a naive way.

- **Vulnerability**:

  Exposes the hash's internal state through MAC.

# Attacker’s Script (client.py)

- **Intercepted Data**:

```
intercepted_message = b"amount=100&to=alice"
intercepted_mac = "..." # Obtained from the server
```

- **Tools Used**:
  - hashpumpy – Automates length extension for MD5/SHA1
  - Alternatively: Manual crafting using padding and guessed key length

- **Forged Message & MAC**:

forged_message = intercepted_message + padding + b"&admin=true"

forged_mac = hash(secret || message || extension)

# Successful Forgery Demo

- **Before Forgery**:
  - Message: amount=100&to=alice
  - MAC: abcd1234...
- **After Forgery**:
  - Forged Message: amount=100&to=alice&admin=true
  - Forged MAC:

    abcd1234... (matches new message)
- **Server Output**:

  Accepts the modified message as authentic

# Why the Attack Works

- **Root Cause**:

MD5 and SHA1 allow continuation of hashing due to block-based processing.

- **No Secret Validation**:

Server only sees final MAC, not internal processing.

- **MAC = hash(secret || message)** fails to hide secret length and internal state.

# Proper Mitigation – Using HMAC

- **Secure Construction**:

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

- **Why HMAC is Secure**:

  - Inner and outer key padding.
  - Defends against length extension
  - Cryptographic security proof under certain assumptions.

# Verifying Attack Failure on HMAC

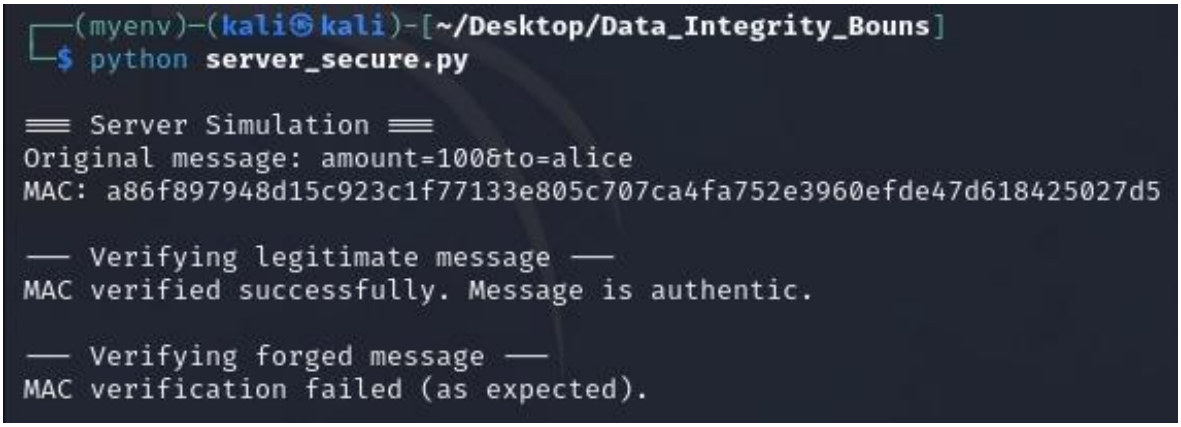
- **Re-run attack against updated server**

- **Result**:

  - Forged message is rejected
    - MAC mismatch (attacker cannot generate valid HMAC without secret)

**Screenshot**:

Show failed verification on server output



# Brute Force + Length Extension on Vulnerable MD5 MAC

## Scenario:

- Server uses weak MAC:
  MAC = MD5(secret || message)
- Vulnerable to brute-force and length extension attacks.

## Attack Process:

1. Intercept original message and MAC:
   amount=100&to=alice

2. Brute-force the secret key from a wordlist using:
   **hashlib.md5(key.encode() + message).hexdigest()**

3. On key discovery (e.g., supersecretkey), use hashpumpy to append &admin=true and forge a valid MAC.

## Result:

- Secret key successfully recovered via brute-force.
- Forged message and MAC accepted by the vulnerable server.
- Attack fully successful due to insecure MAC construction.

# Brute Force and Length Extension Attack

## Attack Attempt Summary:

- Weak key discovered from wordlist: supersecretkey
- Appended malicious data: &admin=true
- Used hashpumpy to forge new message and MAC
- Sent forged message to server for verification

## Server Response:

- MAC verification failed
- Attack unsuccessful — Server rejected the forged message

## Result :

- Even if the key is discovered through brute-force, the attack may still fail if:
- The server uses proper MAC validation
- Insecure MAC constructions (like MD5(secret || message)) are replaced with secure ones such as **HMAC**

# Conclusion

- Naive MAC constructions like MD5(secret || message) are **vulnerable** to length extension and brute-force attacks, allowing attackers to forge valid MACs without knowing the secret.

- Length extension attacks exploit the internal workings of hash functions (e.g., MD5, SHA1), enabling message tampering with valid MACs.

- Brute force can reveal weak secret keys if predictable or short keys are used, further compromising message integrity.

- **HMAC** provides a secure alternative by using inner and outer key padding, preventing length extension and resisting brute-force forgery.

- Implementing **HMAC** and using strong, unpredictable keys effectively **mitigates these attacks**, ensuring message authenticity and integrity.

- Demonstrations confirmed:
  - Attacks succeed against vulnerable MACs.
  - Attacks fail when HMAC is properly implemented.