Alexandria National University

Faculty of Computers and Data Science

Data Integrity & Authentication Course


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

## Team Participants

Omar Magdy Abdulla 2205221

Mohamed Hany Khatab 2205053

Youssef Saeed Thabet 2205064

# Server.py

## Purpose:

**This script simulates a server using a naive MAC scheme based on MD5(secret || message). It demonstrates how such a construction is vulnerable to length extension attacks.**

server.py > main

```python
import hashlib

SECRET_KEY = b'YoussefSaeedthabet'  # Unknown to attacker

def generate_mac(message: bytes) -> str:
    # Naive MAC: hash(secret || message)
    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"hello world"
    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()
```

```python
    # --- Manual test for length extension attack ---
    # Use the exact forged message and MAC from client.py output
    forged_message = bytes.fromhex("616d6f756e743d3130302674f3d616c696365580000
    forged_mac = 'd8284d61d346af085fe32f707c2667ae'
    print("\n--- Manual verification of forged message (with padding) ---")
    print("Forged message (hex):", forged_message.hex())
    print("Forged MAC:", forged_mac)
    if verify(forged_message, forged_mac):
        print("[SUCCESS] Forged MAC is valid! (Manual test)")
    else:
        print("[FAIL] Forged MAC is not valid. (Manual test)")
```

## 🔍 Code Explanation:

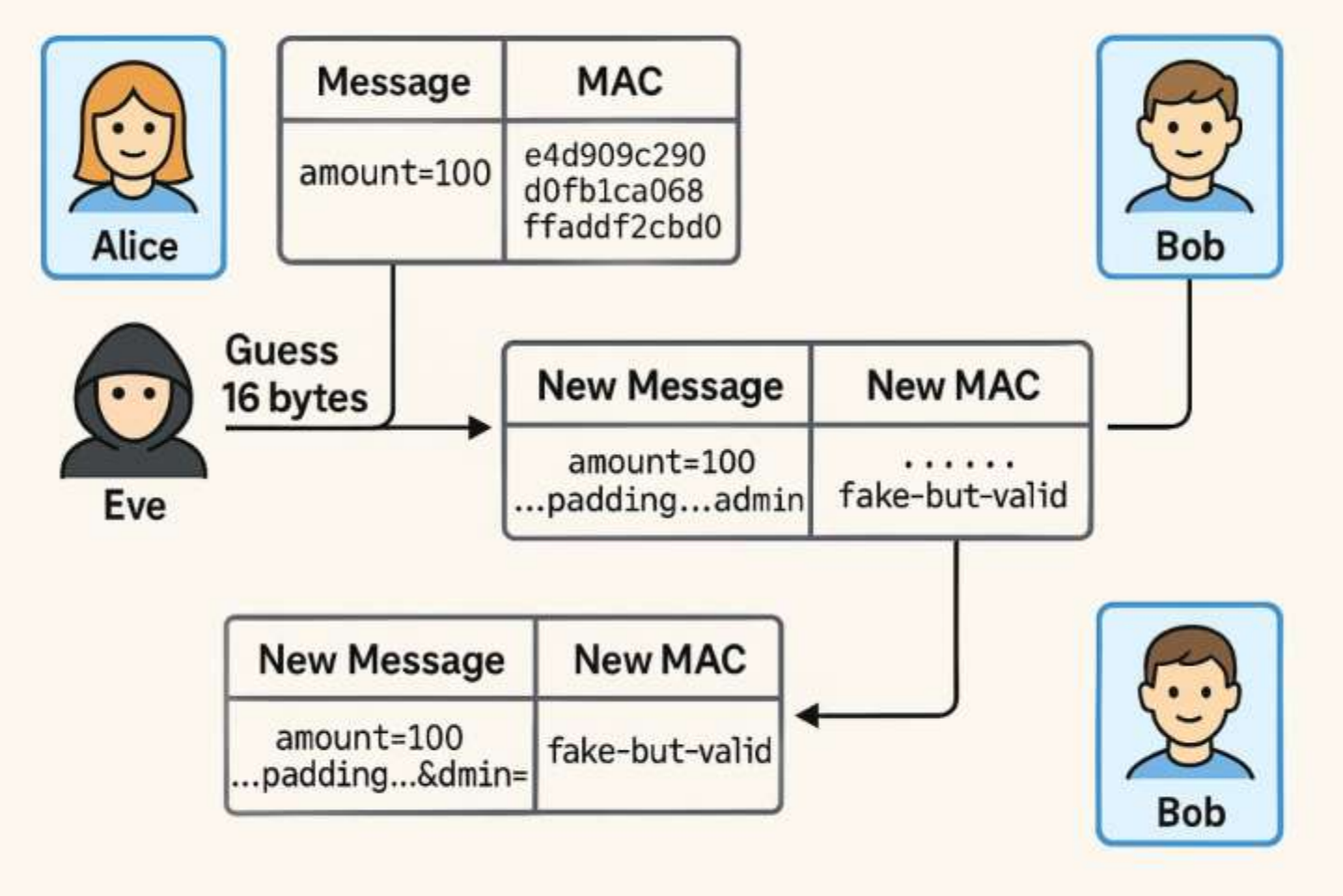
The code defines a secret key SECRET_KEY that is unknown to an attacker. The function generate_mac creates a MAC by hashing the concatenation of the secret key and the message (MD5(secret || message)). The verify function checks if the given MAC matches the expected MAC generated for the message using the secret key.

In the main function, the code first generates and verifies a MAC for a legitimate message ("hello_world"), which passes verification successfully. Then, it simulates an attacker forging a new message by appending "&admin=true" to a different message but reuses the original MAC, which fails verification as expected.

At the end, the script manually tests a length extension attack. This attack exploits a weakness in the naive MAC construction: because MD5 processes data in blocks, an attacker who knows the MAC for secret || message can append additional data to the message and generate a valid MAC without knowing the secret key. The forged message with padding and the forged MAC are tested to see if verification passes. If it does, it confirms the vulnerability.

This demonstrates why using naive hash(secret || message) constructions is insecure and why HMAC, which properly combines the secret and message with additional hashing steps, should be used instead.

---




**This image illustrates a Length Extension Attack on a Message Authentication Code (MAC) using insecure hash functions like MD5 or SHA1 when not used properly.**

**Summary:**

- **Alice sends a signed message (amount=100) with its MAC.**
- **Eve intercepts it, and guesses the MAC key length (often 16 bytes).**
- **She appends data (like &admin=true) to the message using proper padding and generates a new valid MAC using the original MAC as the hash's internal state.**
- **Bob accepts the forged message as valid because the MAC still matches, even though the message was altered.**

🔒 **Lesson: Always use secure MAC algorithms like HMAC, which are resistant to length extension attacks**

---

# Client.py

## Purpose:

**This script demonstrates a length extension attack on a naive MD5-based MAC, allowing an attacker to forge a valid MAC without knowing the secret key. It proves how such MAC constructions are vulnerable to message forgery.**

client.py > ...

```python
# Intercepted values (from server.py output)
intercepted_message = b"hello_world"
intercepted_mac = "0cd9bd4e4eb587e2481ec840948b252d"  # From server.py output

# Data attacker wants to append
append_data = b"&admin=true"

# Brute-force key length range
MIN_KEY_LEN = 8
MAX_KEY_LEN = 20

def md5_padding(msg_len):
    pad = b"\x80"
    pad += b"\x00" * ((56 - (msg_len + 1) % 64) % 64)
    pad += struct.pack("<Q", msg_len * 8)
    return pad

def parse_md5_hexdigest(h):
    return struct.unpack('<4I', bytes.fromhex(h))

def perform_attack():
    if not intercepted_mac:
        print("[!] Please set intercepted_mac from server.py output.")
        return
    for key_len in range(MIN_KEY_LEN, MAX_KEY_LEN + 1):
        orig_len = key_len + len(intercepted_message)
        padding = md5_padding(orig_len)
        forged_message = intercepted_message + padding + append_data
        state = parse_md5_hexdigest(intercepted_mac)
        total_len = orig_len + len(padding)
        m = pymd5.md5(state=state, count=total_len*8)
        m.update(append_data)
        forged_mac = m.hexdigest()
        print(f"Trying key length: {key_len}")
        print("Forged message (hex):", forged_message.hex())
        print("Forged MAC:", forged_mac)
```

```python
        if verify(forged_message, forged_mac):
            print(f"[SUCCESS] Forged MAC is valid! Key length: {key_len}")
            print("Forged message:", forged_message)
            return
        else:
            print("[FAIL] Forged MAC is not valid.")
    print("Tried all key lengths, none succeeded.")

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

## 🔍 Code Explanation:

The code starts with intercepted values: an original message and its corresponding MAC from the vulnerable server. The attacker wants to append additional data ("&admin=true") to the original message while generating a valid MAC for the extended message.

The attack **exploits the MD5 hash function's internal state and padding scheme**. The function md5_padding calculates the MD5 padding for a given message length, simulating how MD5 pads messages internally. parse_md5_hexdigest converts the intercepted MAC (hex string) into MD5's internal state format (4 integers).

The main function **perform_attack** tries different secret key lengths (from 8 to 20 bytes) because the secret key length is unknown. For each key length guess, it calculates the padding as if the message were secret_key || intercepted_message. Then it creates the forged message as the intercepted message plus padding plus the new appended data.

Using the internal MD5 state from the intercepted MAC, it initializes a custom MD5 object (pymd5.md5) that continues hashing from the previous state. It then updates this hash with the appended data to produce a forged MAC.

Finally, it uses the server's verify function to check if the forged message and forged MAC are accepted as valid. If so, the attack succeeded, revealing a valid extended message-MAC pair without knowing the secret key. If all key lengths fail, it reports failure.

This script demonstrates a practical length extension attack on insecure MAC constructions and the importance of using secure MAC algorithms like

HMAC

---

# Server_hmac.py

## Purpose:

**This script demonstrates the generation and verification of a secure message authentication code (MAC) using HMAC with MD5. It simulates verifying both legitimate and forged messages to show how HMAC prevents tampering.**

server.py | server_hmac.py

server_hmac.py > ...

```python
import hmac
import hashlib

SECRET_KEY = b'YoussefSaeedThabet'  # Unknown to attacker

def generate_mac(message: bytes) -> str:
    # Secure MAC: HMAC(secret, message)
    return hmac.new(SECRET_KEY, message, hashlib.md5).hexdigest()

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

def main():
    # Example message
    message = b"hello_world"
    mac = generate_mac(message)

    print("=== Secure Server Simulation (HMAC) ===")
    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()
```

## 🔍 Code Explanation:

**The program uses the hmac module with a secret key and MD5 hashing to produce a secure MAC for messages. The generate_mac function creates the MAC by applying HMAC to the secret key and message, ensuring integrity and authenticity. The verify function compares the provided MAC to the expected one using a timing-safe comparison to prevent side-channel attacks.**

**In the main routine, a legitimate message "hello_world" is authenticated successfully. Then, an attacker-simulated forged message ("amount=100&to=alice&admin=true") is tested with the original MAC, which correctly fails verification because the MAC does not match the modified message. This confirms that HMAC protects against forgery and message tampering without knowledge of the secret key.**

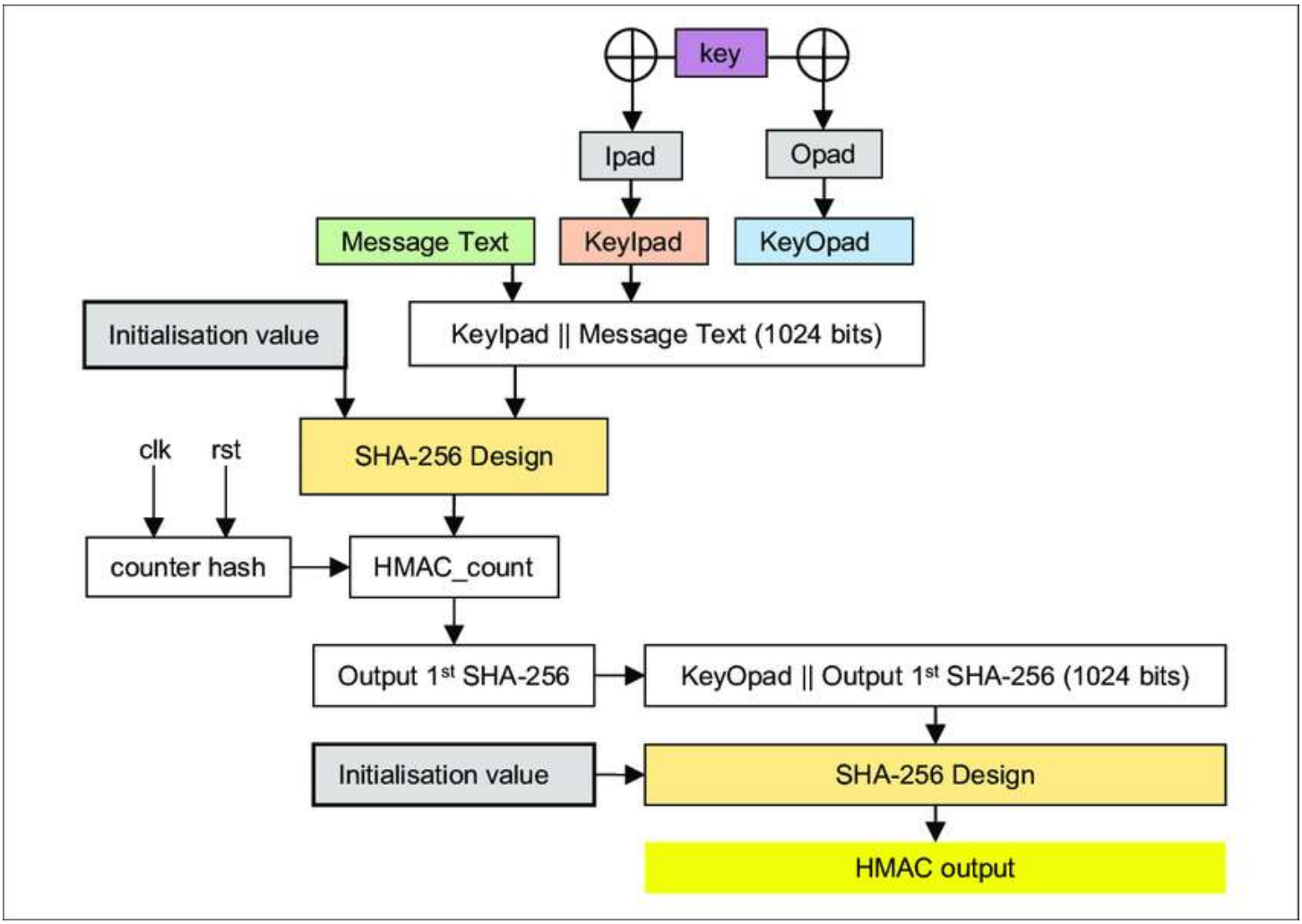
# Pymid.py

## Purpose:

**This code implements the MD5 hash function from scratch in Python. It allows incremental hashing of data, handling padding and length encoding according to the MD5 standard, and producing the final digest or hex digest.**

## 🔍 Code Explanation:

- **Defines MD5's core functions, constants, and bit rotations.**
- **Processes data in 64-byte blocks, updating internal state (A, B, C, D).**
- **Handles padding and appends message length as per MD5 spec.**
- **Produces 16-byte digest or hex string digest.**
- **Supports incremental updates and state restoration for hash continuation.**




## HMAC Process

- **Key is XORed with ipad and opad to produce KeyIpad and KeyOpad.**
- **KeyIpad is concatenated with the message and fed into the first SHA-256 hash.**
- **The output of the first SHA-256 is concatenated with KeyOpad and passed into a second SHA-256 hash.**
- **The final SHA-256 output is the HMAC result, with initialization and counter logic supporting the process.**