Data integrity and Authentication

Mitigation Write-Up: Securing MACs With HMAC

Submitted by:

Mohamed Abdelrahman Awad | ID:2205114
Mohamed Ahmed Ramdan |ID:2205043
Omar Ahmed Hemaid | ID:2205213

Submitted to:

Dr. Maged Abdelaty

May,2025

Attack Demonstration

a. Intercepted valid message and MAC

This simulates a server generating a MAC:

```
import hashlib

SECRET_KEY = b'supersecretkey'  # unknown to attacker

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

message = b"amount=100&to=alice"
mac = generate_mac(message)

print("Original message:", message)
print("Original MAC:", mac)
```

Explanation:

- The server uses MD5 to compute a MAC.
- The attacker only sees the `message` and `mac`, not the `SECRET_KEY`.

b. Length Extension Attack Code

The attacker uses the original MAC to forge a longer message and valid MAC:

```
import struct
from Crypto.Hash import MD5

def md5_pad(msg_len_bits):
    msg_len_bits += 64
    pad_len = (56 - (msg_len_bits // 8) % 64) % 64
    padding = b'\x80' + b'\x00' * (pad_len - 1)
    padding += struct.pack('<Q', msg_len_bits)
    return padding

# Known by attacker
intercepted_message = b"amount=100&to=alice"
original_mac = "614d28d808af46d3702fe35fae67267c"
data_to_append = b"&admin=true"
secret_len_guess = 16  # attacker guesses
```

```
# Simulate the internal MD5 state from original MAC
message_bits = (secret_len_guess + len(intercepted_message)) * 8
pad = md5_pad(message_bits)

# Resume hashing using internal state
h = MD5.new()
h._md5 = struct.unpack("<4I", bytes.fromhex(original_mac)) + (0,)  # state + count
h.update(data_to_append)
forged_mac = h.hexdigest()

# Final forged message
forged_message = intercepted_message + pad + data_to_append

print("Forged Message:", forged_message)
print("Forged MAC:", forged_mac)
```

Explanation:

- Padding is calculated exactly like MD5 does.
- The attacker reconstructs the state from the MAC.
- New data is hashed as a continuation.
- Result is a new message and forged MAC that the server will accept.

---

c. Forged message accepted by insecure server

Insecure server code:

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

# Try verifying the forged message
if verify(forged_message, forged_mac):
    print("❌ Forgery succeeded!")
else:
    print("✅ Forgery blocked!")
```

Explanation:

- The server re-hashes the secret + message.
- Since the forged message includes correct padding, the MAC matches.
- Attacker successfully tricks the server.

Mitigation and Defense

a. Use HMAC instead of raw hash

HMAC is a secure standard for MACs:

```python
import hmac
import hashlib

SECRET_KEY = b'supersecretkey'

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

def verify_hmac(message: bytes, mac: str) -> bool:
    return hmac.compare_digest(mac, generate_hmac(message))
```

Explanation:

- hmac.new() handles key padding and inner/outer hashes.
- Prevents attackers from using the internal state.

b. Attack fails against HMAC

Trying the previous attack:

```python
if verify_hmac(forged_message, forged_mac):
    print("❌ Forgery succeeded!")
else:
    print("✅ Forgery blocked!")
```

Explanation:

- HMAC structure doesn’t allow continuing the hash externally.
- Forgery is completely blocked.

c. Why HMAC defends against length extension

HMAC works as: `HMAC(K, m) = H((K ⊕ opad) || H((K ⊕ ipad) || m))`

Key Benefits:

- The key is used inside two layers of hashing.
- Internal state can’t be reused for new data.
- Padding is handled securely.

This makes HMAC **resistant to length extension attacks**.

Recommendations & Suggestions

1. Always Use HMAC for Message Authentication

- Never use `hash(secret || message)` directly.
- Always rely on secure, standard MAC constructions like **HMAC**, which are proven resistant to attacks like **length extension** and **collision attacks**.

2. Avoid Weak Hash Functions like MD5 and SHA1

- Both MD5 and SHA1 are **cryptographically broken** and vulnerable to various attacks.
- Use stronger alternatives like:
  - **SHA-256** (preferred)
  - **SHA-3** (even more modern)

3. Apply Key Separation for Different Purposes

- Never reuse the same secret key across multiple algorithms (e.g., don't use the same key for encryption and MAC).
- Instead, derive separate keys for each task using a **key derivation function (KDF)**.

4. Validate Message Length and Structure

- On the server, always **parse and validate** the structure of incoming messages (e.g., only allow known keys like `amount`, `to`, etc.).
- This helps prevent malicious parameter injection (e.g., `&admin=true`).

5. Use HTTPS to Protect MACs and Messages

- Ensure all communication between client and server is over **TLS/HTTPS**.
- Prevents attackers from intercepting (message, MAC) pairs in the first place.

6. Use Established Libraries

- Avoid implementing your own MACs or cryptographic primitives from scratch.
- Use well-vetted libraries like `hmac`, `cryptography`, or `PyNaCl`.

7. Log and Monitor Authentication Failures

- Log failed MAC/HMAC verifications.
- Multiple failures may indicate an active attack attempt and should trigger alerts.