Preventing MAC Forgery Using HMAC

Introduction

Message Authentication Codes (MACs) are essential in ensuring data integrity and authenticity in communication. However, their security heavily depends on how they are constructed. This write-up discusses how using HMAC (Hash-based Message Authentication Code) mitigates vulnerabilities like **length extension attacks**, which affect naive constructions such as `MAC = hash(secret || message)`

Why `hash(secret || message)` Is Insecure

Hash functions like **MD5** and **SHA-1** are vulnerable because of how they are designed:

- They process input in fixed-size blocks (e.g., 512 bits for MD5).
- They automatically add padding at the end of the input.
- They maintain an internal state that can be continued, if someone guesses the length of the original input.

So what can an attacker do?

If the attacker intercepts:

- A valid message like: `amount=100&to=alice`
- And its MAC: `hash(secret || message)`

Then, without knowing the secret key, they can:

1. Guess the length of the key (e.g., 14 bytes).
2. Use tools like `hashpumpy` or a custom MD5 class to:
    a. Reconstruct the internal state of the hash from the known MAC.
    b. Append extra data like `&admin=true`.
    c. Generate a valid MAC for the forged message.

Example: Simulated insecure server:

- secret = b'supersecretkey'
- message = b'amount=100&to=alice'
- mac = hashlib.md5(secret + message).hexdigest()

Attacker builds:

- forged_message = b'amount=100&to=alice...[padding]...&admin=true'
- forged_mac = new_mac_generated_by_pymd5_or_hashpumpy

The vulnerable server **accepts** the forged message and MAC:

- verify(forged_message, forged_mac) → True

This confirms that the system is **vulnerable** to a length extension attack.

Mitigation: Using HMAC

To defend against this attack, we replace the insecure MAC function with:

- import hmac, hashlib
- mac = hmac.new(secret, message, hashlib.md5).hexdigest()

HMAC stands for **Keyed-Hash Message Authentication Code** and is defined as:

- HMAC(K, m) = H((K ⊕ opad) || H((K ⊕ ipad) || m))

Why HMAC Is Secure

1. ***Double Hashing with Key Mixing***

- The key is mixed with constants (`ipad`, `opad`) and used both **i**nside and outside the hash.
- An attacker cannot "continue" the inner hash because the structure is nested and requires the key.

2. ***Internal State Isolation***

- The internal hash state is not exposed. Even if an attacker knows the HMAC output, they cannot resume hashing to append new data.
- HMAC prevents all padding and continuation tricks used in length extension attacks.

3. Example: Secure HMAC Construction

- secret = b'supersecretkey'
- message = b'amount=100&to=alice'
- mac = hmac.new(secret, message, hashlib.md5).hexdigest()

If an attacker tries to forge a message using tools like pymd5 or hashpumpy, the result is:

➢ verify(forged_message, forged_mac) → False

HMAC protects the system completely from this type of forgery.

Result of the Mitigation

After switching to HMAC:

- Forged messages fail verification.
- The server detects tampering.
- The system is now resistant to length extension attacks.

References

- RFC 2104 – HMAC Spec
- OWASP – Message Authentication Code (MAC)
- Crypto StackExchange – Why HMAC Prevents Length Extension