Alexandria National University Faculty of Computers and Data Science Cyber Security Program

Data integrity & Authentication Course

MAC Forgery Attack Report :

Background and Mitigation Using HMAC

Prepared by:

Kareem Ahmed 2205069

Adham Ahmed 2205087

Ibrahim Hassan 2205001

Background Study and Mitigation Using HMAC

I. Background Study

a. What is a MAC and Why Do We Use It?

A Message Authentication Code (MAC) is like a digital signature for data. It helps us make sure that a message hasn't been tampered with and that it actually came from the person (or system) we expect. It works by combining a secret key with the message using a cryptographic function. If someone changes the message, the MAC will no longer match, and we'll know something's wrong.

So in short, MACs are used to:

- Check data integrity (nothing was changed).
- Verify authenticity (it came from a trusted source).

---

b. What Is a Length Extension Attack?

A length extension attack is a sneaky trick that takes advantage of how some hash functions (like MD5 and SHA1) work internally. These hash functions process data in blocks, and when you hash something like hash(secret || message), they don't completely "hide" the message structure.

This means that if an attacker sees the MAC of a message, they can sometimes use it to append more data and create a new valid MAC — all without knowing the secret key! They're basically continuing the hashing process from where it left off.

This attack works especially well if the MAC is simply:

hash(secret || message)

c. Why Is hash(secret || message) a Bad Idea?

At first glance, this method seems fine — combine the secret and message, hash it, and done. But this setup is actually vulnerable, because:

- Attackers can guess the length of the secret.
- They can use the original MAC to forge a new message with extra data added.
- They can make the server accept this forged message as valid.

So even though the attacker doesn't know the secret key, they can still trick the system — which defeats the whole point of using a MAC.

II. Mitigation Using HMAC

a. How We Fix It: Switching to HMAC

To fix the problem, we replaced the old method with HMAC (Hash-based Message Authentication Code). HMAC is designed specifically to avoid these kinds of attacks.

Instead of doing just hash(secret || message), HMAC wraps the key and message in a safer structure using two layers of hashing and some padding:

HMAC(key, message) = hash((key ⊕ outer_pad) || hash((key ⊕ inner_pad) || message))

This double-wrapping makes it impossible for attackers to extend the message, because they can't fake the internal state of the hash — it's completely dependent on the secret key.

b. Showing That the Attack No Longer Works

After implementing HMAC, we ran the same attack steps — trying to forge a message by adding something like &admin=true. This time, the server rejected the message.

Here’s what changed in the code:

```
import hmac

import hashlib

SECRET_KEY = b'supersecretkey'

def generate_mac(message: bytes) -> str:

    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)
```

The attack script couldn’t generate a valid MAC anymore, which proves that HMAC successfully defends against length extension attacks.

---

c. Why HMAC Works

HMAC works because:

- It hides the internal hash state from attackers.
- Even if they see a valid MAC, they can’t use it to extend the message.
- It uses both an inner and outer hash, which adds an extra layer of security.

So with HMAC, the only way to produce a valid MAC is to know the secret key — exactly what we want in a secure authentication system.