

Team Members:

Mark Magdy – 2205040, Arsany Osama – 2205122, Rana Ashraf -2205019

Mitigation and Defense

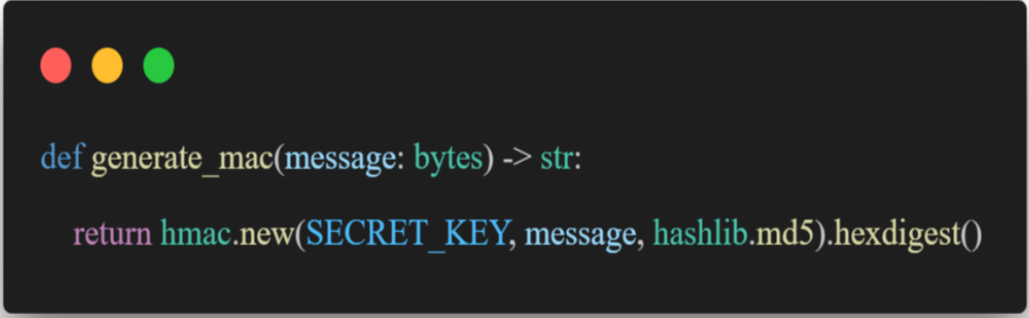
Defense Strategies

Use HMAC (Hash-based Message Authentication Code)

HMAC is a construction used for message authentication, which combines a cryptographic hash function (like MD5, SHA-256, etc.) with a secret key. The goal of HMAC is to verify both the integrity and the authenticity of a message.

HMAC uses a nested structure that prevents recovery of the hash state, even if the attacker knows the MAC.

Implementation:

A code block with a dark background and light-colored text. It contains two lines of Python code. The first line is a function definition: `def generate_mac(message: bytes) -> str:`. The second line is a return statement: `return hmac.new(SECRET_KEY, message, hashlib.md5).hexdigest()`. The code is color-coded: `def` is blue, `generate_mac` is green, `message` is blue, `bytes` is green, `->` is blue, `str` is green, `return` is blue, `hmac` is green, `new` is green, `SECRET_KEY` is blue, `message` is blue, `hashlib` is green, `md5` is green, and `hexdigest` is green.

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

Why HMAC Mitigates Length Extension Attacks?

1. HMAC Structure

HMAC does not directly compute $H(\text{secret} \parallel \text{message})$. Instead, it uses a **two-layer hashing process** with cryptographic padding and key derivation. The HMAC algorithm is defined as:

$$\text{HMAC}(K,m)=H((K\oplus\text{opad}) \parallel H((K\oplus\text{ipad}) \parallel m))$$

Where:

K: Secret key	H: Cryptographic hash function
m: Message	\parallel : Concatenation
ipad: Inner padding (0x36 repeated)	opad: Outer padding (0x5C repeated)

Key Security Properties:

- **Key Masking:** The secret key is XORed with constants (ipad and opad) for the inner and outer layers. This ensures the key is never directly exposed in either hash computation.
- **Nested Hashing:** The inner hash output is encrypted again with the outer key-derived value, breaking the linear processing of Merkle-Damgård hashes.

2. Prevention of State Reuse

In a length extension attack, the attacker leverages the final internal state of the hash function (exposed by the original MAC) to compute a valid MAC for an extended message. HMAC thwarts this by:

1. **Isolating the Inner Hash:** The inner hash $H(K\oplus\text{ipad} \parallel m)$ processes the message with a modified key. Even if an attacker recovers this hash output, they cannot reuse it to extend the message because the outer layer requires $K\oplus\text{opad}$, which is unknown.
2. **Destroying the Hash State:** The outer hash encrypts the inner hash result with a different key-derived value ($K\oplus\text{opad}$). This ensures no intermediate state from the inner hash is exposed or reusable.

3. Cryptographic Padding

The use of ipad and opad ensures:

- **Key Obfuscation:** The XOR operations mask the secret key in both layers. Even if an attacker knows the original MAC, they cannot derive $K \oplus \text{ipad}$ or $K \oplus \text{opad}$ without the key.
- **Input Randomization:** The padding constants break predictable message structures, making it impossible for an attacker to guess how the secret key is integrated into the hash inputs.

Demonstrate Attack Failing Against HMAC Implementation

1. Attacker Code (client.py)

Attempts the same length extension attack as before:

```
new_mac, new_message = hashpumpy.hashpump(  
    intercepted_mac,    # Original MAC (hex string)  
    intercepted_message, # Original message (bytes)  
    data_to_append,    # Data to append (bytes)  
    key_length         # Secret key length  
)
```

2. Attack Execution

1. Server Generates Legitimate MAC:

=== Server Simulation (HMAC) ===

Original message: amount=100&to=alice

MAC: 616843154afc11960423deb0795b1e68

2. Attacker Forges Message and MAC:



```
forged_message=b'amount=100&to=alice\x80\x00\x00\x00&admin=true'
```



```
forged_mac = "97312a73075b6e1589117ce55e0a3ca6"
```

3. Server Verification returns False:



```
if verify(forged_message, forged_mac):  
    print("MAC verified successfully (unexpected).")  
else:  
    print("MAC verification failed (as expected).")
```



```
--- Verifying forged message ---
```

```
MAC verification failed (as expected).
```

Conclusion

This assignment explored the critical role of Message Authentication Codes (MACs) in ensuring data integrity and authenticity.

The study revealed that naive MAC constructions like $MAC = \text{hash}(\text{secret} || \text{message})$ are vulnerable to **length extension attacks** due to the Merkle-Damgård structure of hash function. Attackers exploit this by appending malicious data to a message and forging a valid MAC without knowing the secret key, compromising system security.

The practical demonstration illustrated how an attacker could successfully extend a message and generate a forged MAC that the vulnerable server accepts. However, transitioning to **HMAC (Hash-based MAC)** mitigated this risk entirely. By using a hashing structure with key masking (ipad and opad), The secure HMAC implementation reject forged messages, proving its efficacy.