Data Integrity

BOUNS ASSIGNMENT

Supervised by :

Dr.Maged Abdelaty

Mahmoud Zaghloula 2205055
Abdelrahman Hisham 2205032
Yehia Ahmed Tawfik 2205126

Mitigation and Defense Against Length Extension Attacks

Introduction

Length extension attacks exploit vulnerabilities in hash functions like **MD5** and **SHA-1** when used in insecure **Message Authentication Code (MAC)** constructions, such as MAC = hash(secret || message). In the provided server.py, this insecurity allows an attacker, using client.py, to forge a message by appending data, such as "&admin=true", and compute a valid **MAC** without knowing the **secret key**. To mitigate this threat, the system must adopt **HMAC (Hash-based Message Authentication Code)**, a **secure construction** designed to prevent such attacks. This document details the implementation of **HMAC** in server1.py, demonstrates the failure of the **length extension attack** against this secure system, and explains why **HMAC** ensures robust **message integrity** and **authentication**.

Secure Implementation with HMAC

HMAC Overview

HMAC is a standardized **cryptographic mechanism** that securely combines a **secret key** with a message to produce a **MAC** resistant to **length extension attacks**. Unlike the insecure hash(secret || message), **HMAC** uses a **dual-key, double-hash** construction:

$$\mathrm{HMAC}(K, m) = H((K \oplus opad) \,||\, H((K \oplus ipad) \,||\, m))$$

Here, **H** is the **hash function** (e.g., MD5 in the code), **K** is the **secret key**, **ipad** and **opad** are constant **padding values**, and ⊕ denotes XOR. This design prevents attackers from extending the message without the **secret key**, addressing the vulnerabilities of **MD5**'s **Merkle-Damgård** structure.

Implementation in server1.py

The insecure implementation in server.py uses:

```
import hashlib

SECRET_KEY = b'supersecretkey'

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

This is vulnerable because **MD5** exposes its **internal state**, allowing an attacker to append data and compute a new **MAC**.
In contrast, server1.py implements **HMAC** using Python's hmac module:

```
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)
```

Key improvements include:

- **Secure HMAC**: The hmac.new function applies **MD5** twice with **inner** and **outer keys**, preventing **message extension**.
- **Timing Attack Mitigation**: hmac.compare_digest ensures **constant-time comparison**, reducing risks of **timing-based attacks**.

Demonstration of Attack Failure

Attack in client.py

The **length extension attack** in client.py targets the insecure **MAC** in server.py. The attacker intercepts a message ("amount=100&to=alice") and its **MAC** ("616843154afc11960423deb0795b1e68"), then uses hashpumpy to append "&admin=true":

```
import hashpumpy
intercepted_message = b"amount=100&to=alice"
intercepted_mac = "616843154afc11960423deb0795b1e68"
data_to_append = b"&admin=true"
secret_length = 14
new_mac, new_message = hashpumpy.hashpump(
  intercepted_mac, intercepted_message.decode(),
  data_to_append.decode(), secret_length
)
```

When verified in server.py, the forged **MAC** is accepted:

```
Attack successful! Forged MAC is accepted by the server.
```

Failure Against server1.py

When tested against server1.py, the attack fails. The verify function recomputes the **HMAC** for the extended message ("amount=100&to=alice&admin=true") and compares it with the forged **MAC**. Since **HMAC** requires the **secret key** for the **outer hash**, the attacker's **MAC** is invalid:

```
--- Verifying forged message ---
MAC verification failed (as expected).
```

This demonstrates HMAC's effectiveness, as the attacker cannot manipulate the hash state without the secret key.

Why HMAC Mitigates Length Extension Attacks

HMAC's Security Design

HMAC prevents **length extension attacks** through its **dual-key, double-hash** construction:

- **Inner Hash**: Processes the message with an **inner key** (K ⊕ ipad), producing H((K ⊕ ipad) || m).
- Outer Hash: Encapsulates the inner hash with an outer key (K ⊕ opad), yielding H((K ⊕ opad) || inner hash). The attacker cannot extend the message because the outer hash requires the secret key, which is unavailable.

Technical Analysis

In server.py, the attack succeeds because **MD5**'s **hash output** exposes its **internal state**, enabling hashpumpy to append data. **HMAC** in server1.py neutralizes this:

- **Key Dependency**: The **outer hash** depends on the **secret key**, blocking forgery.
- **State Protection**: The **inner hash** is not exposed, preventing **state manipulation**.
- **Robustness**: HMAC is secure even with MD5, relying on the keying mechanism rather than the **hash function**'s collision resistance.

Practical Benefits

HMAC in server1.py offers:

- **Security**: Ensures **message integrity** and **authentication**.
- **Efficiency**: Requires only two **hash operations**, ideal for **performance-sensitive** systems.
- **Compatibility**: Works with **MD5**, **SHA-1**, and **SHA-256**, supporting **legacy systems**.
- **Adoption**: Used in **TLS**, **OAuth**, and **IPsec**, ensuring **interoperability**.

Conclusion

The insecure **MAC** in server.py is vulnerable to **length extension attacks**, as shown in client.py. By adopting **HMAC** in server1.py, the system prevents forgery, as the attack fails due to **HMAC**'s **secure design**. **HMAC** ensures robust **integrity** and **authentication**, making it the ideal solution for **secure MAC implementations**.