Message Authentication Codes and Length Extension Attacks

2. Mitigation Write-Up

Name :

Ahmed Mohammed Adel Ibrahim - 2205096
Abdelrahman Ayman Saad Abdelhalim - 2205033
Ahmed Mohammed Bekhit - 2205136

COURSE : Data Integrity and Authentication

Doctor : Maged Abdelaty

References : RFC 2104-Cryptanalysis of MD5 and SHA-1- Glenn Askins

A. Introduction & Problem Recap :

In the attack demonstration, we exploited a naive MAC construction:

MAC = MD5(secret || message)

that is vulnerable to a Length Extension Attack because Merkle-Damgard hash function reveal their internal state, allowing an attacker to forge **MAC(secret || message || padding || extension)** without knowing the secret.

1.1 Consequences of the Vulnerability

Data Integrity Compromised: Attackers can append malicious parameters (**&admin=true**) to valid messages.

Authentication Broken: Forged messages pass verification on insecure servers.

B. Secure Solution: HMAC :

The recommended mitigation is the **HMAC** construction, defined in **RFC 2104**:

HMAC(K, M) = H((K ⊕ opad) || H((K ⊕ ipad) || M)):

- H is a secure hash function
- K is the secret key, padded to the hash block size.
- ipad (0x36) and opad (0x5c) are fixed constants.

2.1 Why HMAC Prevents Extension

1. **Double Hashing**: Internal HMAC state H((K ⊕ ipad) || M) is hidden under an outer hash with opad.
2. **Key Separation**: The key is mixed in two contexts, preventing continuation of the inner hash without knowing K.

C. Implementation in Code :

In secure_server.py, we replaced the insecure MD5-based MAC with HMAC-SHA256:

```
import argparse

import base64

import hmac

import hashlib

SECRET_KEY = b'supersecretkey'  # Unknown to attacker

def generate_hmac(message: bytes) -> str:

    return hmac.new(SECRET_KEY, message, hashlib.sha256).hexdigest()

def verify(message: bytes, mac: str) -> bool:

    return hmac.compare_digest(generate_hmac(message), mac)
```

This change ensures message authentication and integrity using a cryptographically robust pattern.

D. Attack Failure Verification :

We re-ran the same length extension attack from client.py against secure_server.py. The forged message and MAC that bypassed the MD5 server now produce:

```
=== Secure HMAC-SHA256 Server ===

Message: in=true' HMAC:

b'amount=100&to=alice\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x08\x01\x00\x00\x00\x00\x00\x00&adm

97312a73075b6e1589117ce55e0a3ca6

HMAC verification failed.
```

E. Conclusion :

In this project, we have demonstrated the practical risks of using naive hash-based MAC constructions by mounting a successful length extension attack against an MD5-based server. **By switching to the standardized HMAC construction with SHA-256, we restored message integrity and authentication, as shown by the rejection of forged messages in our automated tests**.

Key takeaways:

- **Never** implement MACs as hash(key || message) with Merkle-Damgård hashes.
- **Always** use vetted HMAC implementations from cryptographic libraries to avoid subtle vulnerabilities.
- Integrate automated tests into development pipelines to catch regressions quickly.

-Thank You-