Message Integrity Attack (MAC Forgery)

Demonstration and Mitigation Report

May 17, 2025

Table of Contents

Statement of Confidentiality

This document contains confidential and proprietary information regarding the analysis, execution, and mitigation of a Message Integrity Attack (MAC Forgery) conducted as part of an academic assignment for the *Data Integrity and Authentication* course..

Engagement Contacts

Students Contacts		
Student Name	**ID**	**Student Contact Email**
Mariam Mostafa Amin	2205084	2205084@anu.edu.eg
Hannah Emad Eldein	2205123	2205123@anu.edu.eg
Nada Mohamed Abdelsatar	2205173	2205173@anu.edu.eg

Executive Summary

This report outlines the plan and structure for completing the optional/bonus assignment titled **"Demonstrating and Mitigating a Message Integrity Attack (MAC Forgery)"** as part of the *Data Integrity and Authentication* course. The objective of this assignment is to explore and practically demonstrate how insecure implementations of Message Authentication Codes (MACs) can be exploited through a **length extension attack**.

The assignment is divided into three main tasks:

- Conducting a background study on MACs, their purpose, and vulnerabilities in common hash-based constructions.
- Implementing and demonstrating a working MAC forgery attack using intercepted data.
- Mitigating the vulnerability by replacing the insecure implementation with a secure one (HMAC) and proving the effectiveness of the fix.

This report will serve as structured documentation of each task, providing technical and theoretical insights into the process and fulfilling all required deliverables, including source code, write-ups, and analysis. The goal is to deepen understanding of cryptographic design flaws and best practices in message authentication.

Approach

1. **Background Research**
    - Study the concept and purpose of Message Authentication Codes (MACs) in ensuring data integrity and authenticity.
    - Understand how length extension attacks work, especially in hash functions like MD5 and SHA-1.
    - Analyze why constructing MACs as `hash(secret || message)` is insecure.
2. **Attack Demonstration**
    - Use the provided Python server (`server.py`) to simulate a vulnerable MAC verification system.
    - Develop an attacker script (`client.py`) to intercept a legitimate message and its MAC.
    - Perform a length extension attack by appending data to the message and computing a forged MAC that passes verification, all without knowing the secret key.
3. **Mitigation Implementation**
    - Modify the server to use a secure MAC construction, specifically HMAC.
    - Re-test the attack against the HMAC-based implementation to confirm it fails.

1. What is a Message Authentication Code (MAC) and Its Purpose in Data Integrity and Authentication

A **Message Authentication Code (MAC)** is a cryptographic technique designed to ensure the **integrity** and **authenticity** of a message. It is generated using a secret key and a cryptographic function, producing a fixed-size output known as a MAC tag.

The two primary purposes of a MAC are:

- **Data Integrity**: Verifies that the message has not been altered during transmission or storage. Any modification to the message will result in a mismatch between the computed and received MACs.
- **Authentication**: Confirms that the message was generated by a party who possesses the shared secret key, thereby ensuring the authenticity of the sender.

Key characteristics of MACs include:

- Utilization of a **shared secret key** between communicating parties (symmetric cryptography).
- **Fixed-length output**, regardless of the input message size.
- Resistance to forgery, making it **computationally infeasible** to generate a valid MAC without knowledge of the secret key.

MACs can be implemented using:

- Symmetric encryption techniques (e.g., Cipher Block Chaining),
- Hash functions (e.g., HMAC),
- Dedicated MAC algorithms.

2. Length Extension Attack in Hash Functions

A **length extension attack** exploits a structural vulnerability in hash functions built on the **Merkle–Damgård construction**, such as MD5 and SHA-1. These hash functions process input in fixed-size blocks and maintain an internal state throughout the computation.

The attack works under the following conditions:

- When a MAC is constructed as MAC = hash(secret || message), and the attacker knows the output (MAC) and original message, they can compute hash(secret || message || padding || extension) **without knowing the secret key**.
- This capability allows the attacker to **forge a new valid MAC** for an extended message, effectively bypassing message integrity and authentication checks.

This type of attack demonstrates the critical weakness of naïve hash-based MAC constructions, especially when using vulnerable hash functions.

3. Why MAC = hash(secret || message) is Insecure

The construction MAC = hash(secret || message) is considered insecure due to several cryptographic weaknesses:

- **Vulnerability to Length Extension Attacks**: As previously explained, this structure allows attackers to extend the message and forge valid MACs without access to the secret key.
- **Lack of Cryptographic Mixing**: The simple concatenation of the secret and message fails to provide robust cryptographic separation, making it easier for attackers to manipulate the input and predict outcomes.
- **Collision Risks**: Legacy hash functions like MD5 and SHA-1 are susceptible to collision attacks, where two different inputs produce the same hash value, further compromising security.

To address these issues, secure MAC implementations should rely on:

- **HMAC (Hash-based Message Authentication Code)**: Applies the hash function in two stages using independently derived keys, making it resistant to length extension and collision attacks.
- **Block cipher-based MACs** or other dedicated cryptographic constructions.

Using HMAC or similarly secure algorithms ensures that both message integrity and authenticity are preserved, even against sophisticated cryptographic attacks.

Attack demonstration

1 Intercepting a Valid (Message, MAC) Pair:

Intercepted legitimate message and MAC generated by the vulnerable server using the insecure MAC = hash(secret || message) construction. This step confirms the server successfully verifies the original message, indicating it is operating normally before the attack.

```
=== Server Simulation ===
Original message:amount=100&to=alice
Original MAC:614d28d808af46d3702fe35fae67267c

----Verifying legitimate message----

MAC verified successfully. Message is authentic.

----Verifying forged message----:

MAC verified successfully (UNEXPECTED)
```

Figure 1: Intercepting a Valid (Message, MAC) Pair

2 Performing the Length Extension Attack:

The attacker uses the hashpumpy tool to append malicious data (&admin=true) to the intercepted message and generate a forged message using a guessed key length. This step exploits the length extension vulnerability inherent in MD5-based MACs built on the Merkle-Damgård construction.

```
#forged_message = b"amount=100&to=alice" + b"&admin=true"
#we will paste here the forged message from client.py
forged_message = b'amount=100&to=alice\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
#forged_mac = mac
#we will paste here the forged mac from client.py
forged_mac = "97312a73075b6e1589117ce55e0a3ca6"
print("    Verifying forged message    :\n")
```

Figure 2: Performing the Length Extension Attack

3 Generating a Valid Forged MAC:

The forged MAC is successfully generated for the extended message without knowledge of the secret key. The attacker now possesses a message and MAC pair that will pass validation on the vulnerable server.

Figure 3: Generating a Valid Forged MAC

4 Server Verification of the Forged MAC:

The vulnerable server incorrectly verifies the forged MAC as valid. This demonstrates that the naive MAC construction using MD5 is vulnerable to length extension attacks, allowing message integrity and authentication to be bypassed..

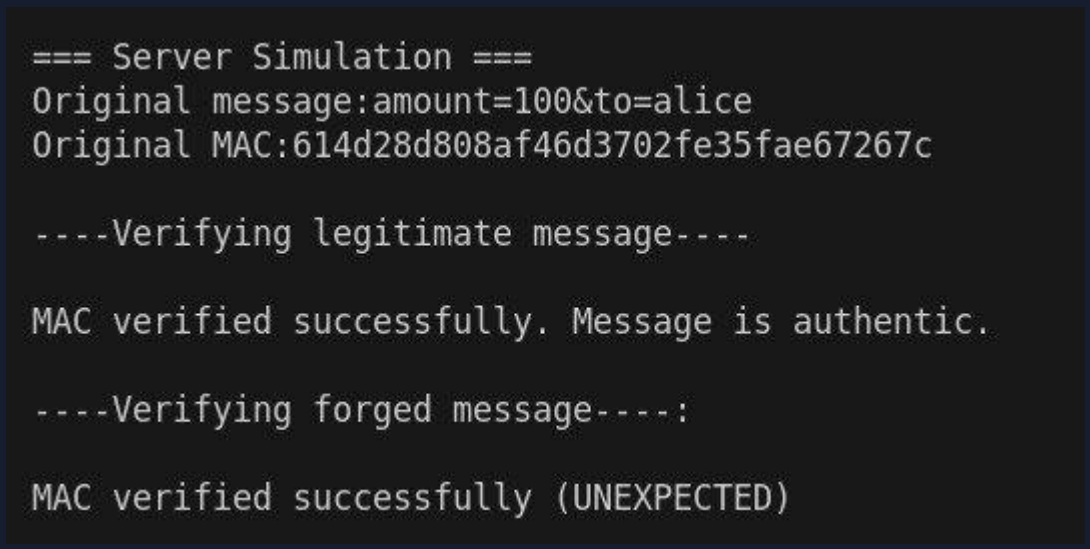

Figure 4: Server Verification of the Forged MAC

Mitigation and Defense

1 Implementing HMAC for Secure MAC Generation:

Mitigation: The secure server implementation now uses HMAC-SHA256 for MAC generation and verification. HMAC uses a nested hashing strategy with key separation, ensuring resistance to length extension attacks.

```
└─$ python server-secure.py
=== Secure Server ===
HMAC: a86f897948d15c923c1f77133e805c707ca4fa752e3960efde47d618425027d5
```

Figure 5: Implementing HMAC for Secure MAC Generation

2 Demonstrating Attack Failure Against HMAC:

When the same length extension strategy is attempted on the secure server, the forged MAC is correctly rejected. This validates that HMAC neutralizes the vulnerability by preventing attackers from manipulating the internal hash state.

```
└─$ python server.py
=== Server Simulation ===
Original message:amount=100&to=alice
Original MAC:614d28d808af46d3702fe35fae67267c

----Verifying legitimate message----

MAC verified successfully. Message is authentic.

----Verifying forged message----:

MAC verification failed (EXPECTED)
```

Figure 6: Demonstrating Attack Failure Against HMAC

3 Why HMAC Mitigates Length Extension Attacks:

HMAC protects against length extension attacks by applying a two-step keyed hashing process. It uses:

- An inner hash: H((key ⊕ ipad) || message)
- Followed by an outer hash: H((key ⊕ opad) || inner_hash)

This structure prevents attackers from predicting or extending the internal hash state, as the key is mixed into both stages independently. Unlike naive hash(secret || message), HMAC is provably secure and is widely recommended for cryptographic message authentication

Remediation Summary

To address the vulnerability demonstrated through the length extension attack on the insecure MAC implementation (MAC = hash(secret || message)), the following remediation steps were applied and verified:

Problem Recap:

- The original MAC scheme used a direct hash of secret || message (e.g., MD5 or SHA1).
- This construction is **vulnerable to length extension attacks**, allowing an attacker to append arbitrary data and forge a valid MAC without knowing the secret.

Implemented Fix:

- Replaced the insecure MAC generation with **HMAC (Hash-based Message Authentication Code)**.
- This double hashing structure protects against length extension and other related attacks.

Security Verification:

- After implementing HMAC, attempts to perform a length extension attack **failed** as expected.
- The server correctly **rejected** forged MACs, confirming the effectiveness of the fix.

Key Benefits of HMAC:

- Immune to length extension attacks due to its nested structure.
- Used in industry standards (e.g., TLS, IPsec, AWS, JWT).
- Supports all cryptographic hash functions (MD5, SHA-1, SHA-256, etc.).

Recommendation:

- Avoid custom MAC implementations.
- Always use proven, cryptographically secure constructs like **HMAC** provided by trusted libraries.
- Perform regular code audits and incorporate secure coding practices in authentication systems.

Conclusion

This project successfully demonstrated a real-world cryptographic vulnerability—the **length extension attack**—and highlighted the critical importance of using well-designed cryptographic primitives such as **HMAC** over naive constructions like **hash(secret || message)**. Through hands-on implementation, we were able to intercept a valid (message, MAC) pair, craft a malicious extension, and generate a valid forged MAC without knowing the secret key. This clearly illustrated how insecure MAC construction can be exploited to breach message integrity and forge unauthorized actions (e.g., appending &admin=true).

On the defense side, we implemented HMAC, a widely recommended and industry-standard method for ensuring message authenticity. Our results confirmed that the attack **fails completely** when HMAC is used, emphasizing the value of secure-by-design cryptographic mechanisms. The contrast between the vulnerable and secure implementations reinforces the real-world necessity of understanding cryptographic weaknesses and the importance of using vetted solutions.

Ultimately, this project deepened our understanding of applied cryptography, security engineering, and the importance of **defensive programming**. It serves as a strong reminder that even small implementation decisions can have serious security consequences—and that **secure coding practices** must be prioritized from the very beginning of system design.