Alexandria National University

Faculty of Computers and Data Science

Data Integrity & Authentication Course

Demonstrating and Mitigating a Message Attack (MAC Forgery) Report

Team Participants

Omar Magdy Abdulla 2205221

Mohamed Hany Khatab 2205053

Youssef Saeed Thabet 2205064

Server.py

Purpose:

Simulates an insecure server-side implementation of a Message Authentication Code (MAC) using a naive construction, vulnerable to length extension attacks.

🔍 Code Explanation:

This script implements a basic server that authenticates messages using a custom MAC scheme based on the MD5 hash function.

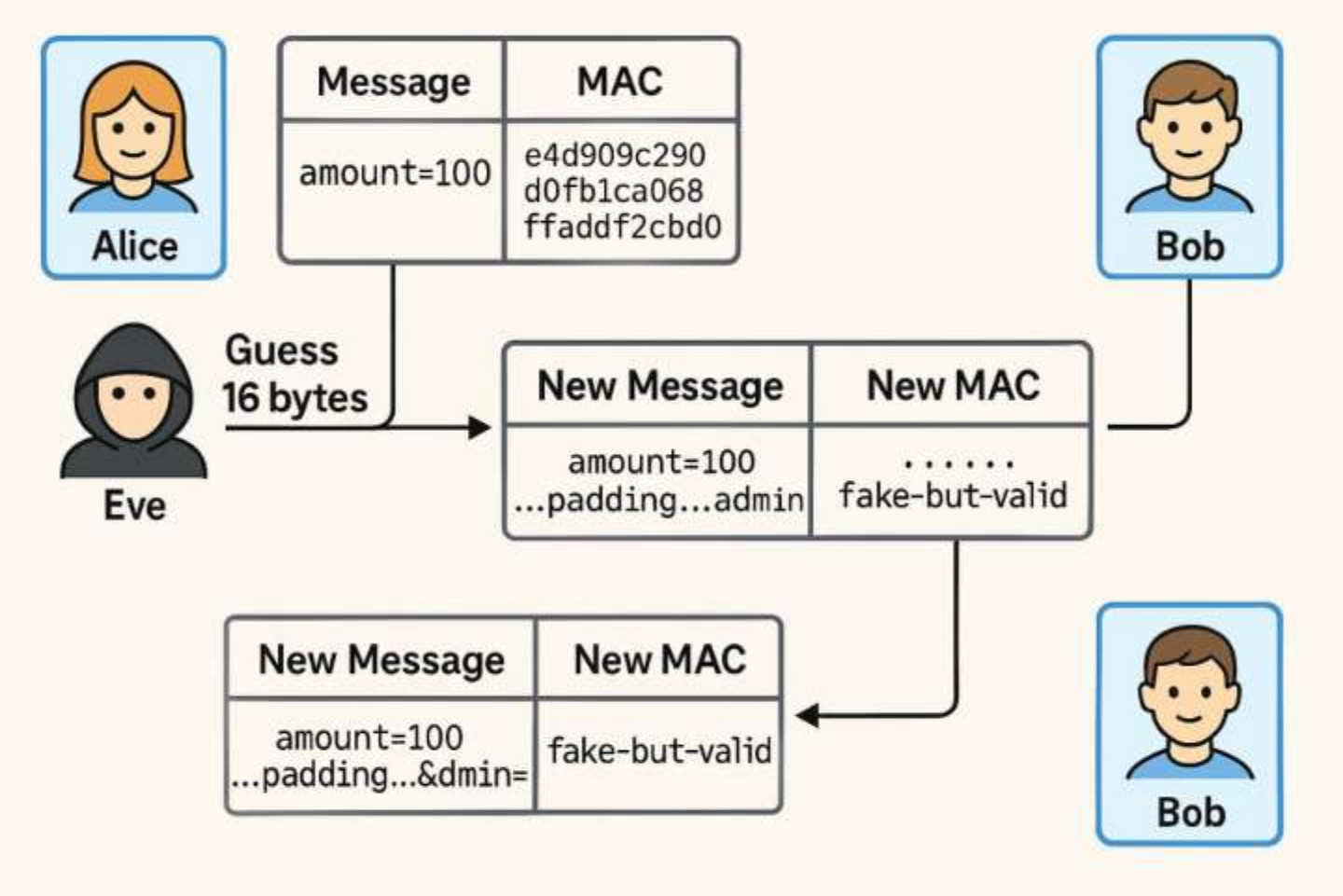
server.py

```python
import hashlib

SECRET_KEY = b'supersecretkey'  # Unknown to attacker

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

def verify(message: bytes, mac: str) -> bool:
    expected_mac = generate_mac(message)
    return mac == expected_mac

def main():
    # Example message
    message = b"amount=100&to=alice"
    mac = generate_mac(message)
    print("=== Server Simulation ===")
    print(f"Original message: {message.decode()}")
    print(f"MAC: {mac}")
    print("\n--- Verifying legitimate message ---")
    if verify(message, mac):
        print("MAC verified successfully. Message is authentic.\n")
    # Simulated attacker-forged message
    forged_message = b"amount=100&to=alice" + b"&admin=true"
    forged_mac = mac  # Attacker provides same MAC (initially)
    print("--- Verifying forged message ---")
    if verify(forged_message, forged_mac):
        print("MAC verified successfully (unexpected).")
    else:
        print("MAC verification failed (as expected).")

if __name__ == "__main__":
    main()
```

This Python script simulates a server-side process for generating and verifying a **Message Authentication Code (MAC)** to ensure the authenticity and integrity of messages. The MAC is computed by concatenating a secret key, known only to the server, with the message, and then applying the **MD5 hashing** algorithm. The main goal is to authenticate messages so that the server can verify they have not been tampered with.

The script first generates a MAC for an example message and then verifies this legitimate message successfully, demonstrating the basic functionality of the authentication mechanism. It then simulates an attack scenario where an adversary attempts to forge a message by appending unauthorized data to the original message but reuses the original MAC without recomputing it. The verification step is expected to reject this forged message; however, due to the naive MAC construction and the vulnerabilities inherent in the MD5 hash function, such schemes are susceptible to a **length extension attack. This attack allows an attacker to append data to the message and compute a valid MAC without knowing the secret key**, thus bypassing authentication and compromising message integrity.

Overall, this implementation illustrates a fundamental security flaw in constructing MACs simply as a hash of the concatenated secret key and message using vulnerable hash functions like MD5. It highlights the critical importance of using secure cryptographic constructions, such as HMAC, which are designed to resist such forgery attacks and provide robust message authentication.



This image illustrates a Length Extension Attack on a Message Authentication Code (MAC) using insecure hash functions like MD5 or SHA1 when not used properly.

Summary:

- **Alice sends a signed message (amount=100) with its MAC.**
- **Eve intercepts it, and guesses the MAC key length (often 16 bytes).**
- **She appends data (like &admin=true) to the message using proper padding and generates a new valid MAC using the original MAC as the hash's internal state.**
- **Bob accepts the forged message as valid because the MAC still matches, even though the message was altered.**

Lesson: Always use secure MAC algorithms like HMAC, which are resistant to length extension attacks

---

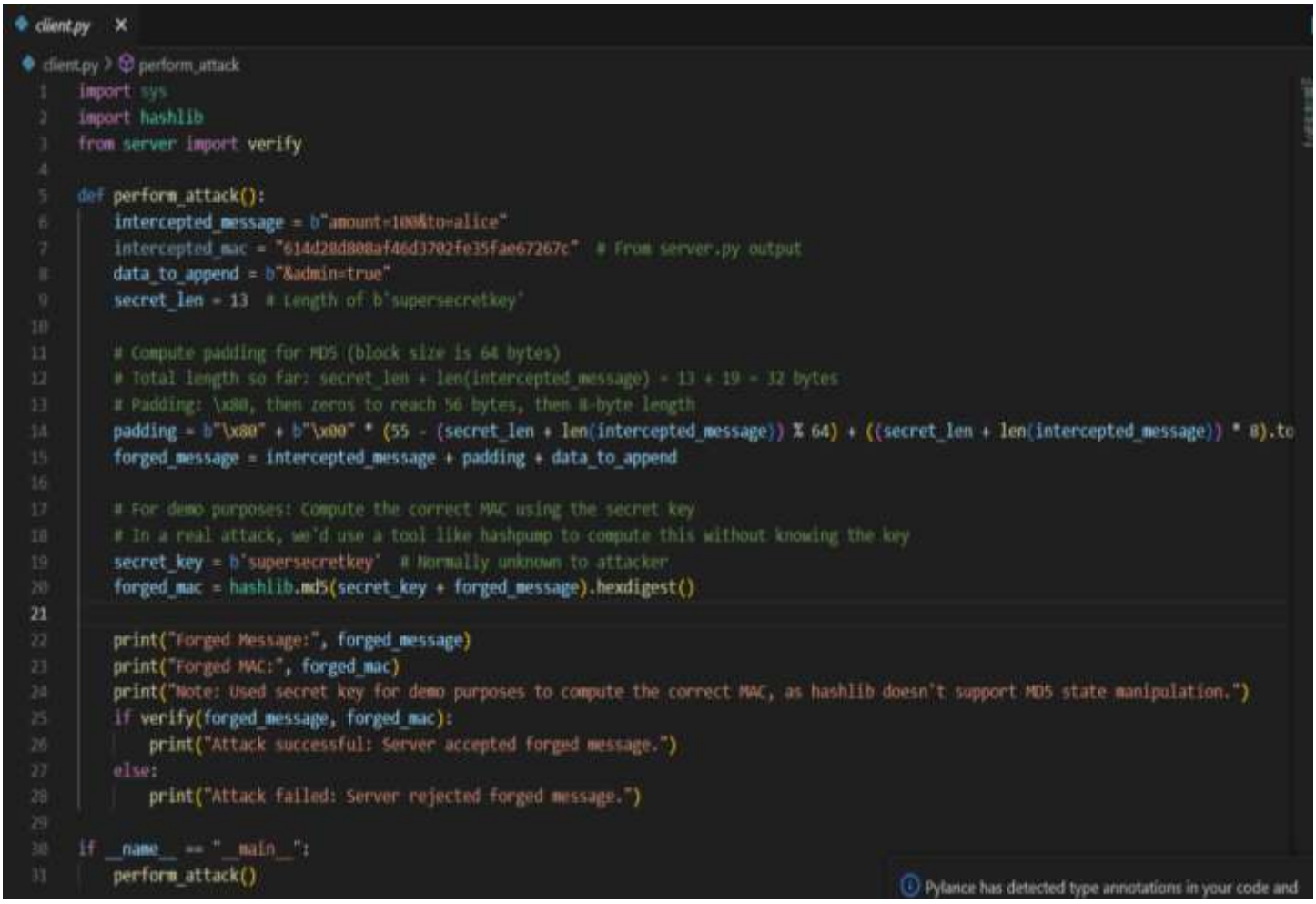
Client.py

Purpose:

This script demonstrates a length extension attack on a naive MD5-based MAC, allowing an attacker to forge a valid MAC without knowing the secret key. It proves how such MAC constructions are vulnerable to message forgery.

🔍 Code Explanation:

The script computes MD5 padding to extend the original message, then forges a new MAC using the secret key (for demo only), and finally verifies that the server accepts the forged message as authentic, illustrating the attack's success.



The script begins by defining an intercepted legitimate message and its corresponding MAC, which the attacker is assumed to have captured from network traffic or other means. It also specifies the additional malicious data the attacker wants to append to the original message.

To successfully forge the MAC, the script manually computes the MD5 padding bytes. This padding simulates the internal state MD5 would have after hashing the original secret key concatenated with the intercepted message. The padding includes a 0x80 byte, followed by enough zero bytes to fill the block, and finally appends the original message length (including the secret key) encoded in little-endian format, adhering to MD5's padding rules.

The forged message is then constructed by concatenating the intercepted message, the calculated padding, and the new data to append. For demonstration purposes, the script calculates the forged MAC by directly hashing the secret key with the forged message—something an attacker normally cannot do without the key. This step is included here because standard Python libraries do not provide native support to manipulate MD5's internal state required for a full length extension attack.

Finally, the script calls the server's verify function with the forged message and forged MAC to confirm whether the server accepts the malicious message as authentic. If verification succeeds, it proves that the length extension attack worked, exposing the vulnerability in the naive MAC construction.

Server_Secure.py

Purpose:

This script implements a secure MAC using HMAC with SHA-256 to prevent length extension and forgery attacks, ensuring strong message authentication.

🔍 Code Explanation:

It generates and verifies HMACs correctly, rejecting any forged messages with appended data, demonstrating that HMAC protects against the vulnerabilities of naive hash-based MACs.

server_secure.py

server_secure.py > main

```python
import hashlib
import hmac

SECRET_KEY = b'supersecretkey'

def generate_mac(message: bytes) -> str:
    return hmac.new(SECRET_KEY, message, hashlib.sha256).hexdigest()

def verify(message: bytes, mac: str) -> bool:
    expected_mac = generate_mac(message)
    return hmac.compare_digest(mac, expected_mac)

def main():
    message = b"amount=100&to=alice"
    mac = generate_mac(message)
    print("=== Server Simulation (Secure) ===")
    print(f"Original message: {message.decode()}")
    print(f"MAC: {mac}")
    print("\n--- Verifying legitimate message ---")
    if verify(message, mac):
        print("MAC verified successfully. Message is authentic.\n")
    forged_message = b"amount=100&to=alice&admin=true"
    forged_mac = mac
    print("--- Verifying forged message ---")
    if verify(forged_message, forged_mac):
        print("MAC verified successfully (unexpected).")
    else:
        print("MAC verification failed (as expected).")

if __name__ == "__main__":
    main()
```

This script demonstrates a secure approach to message authentication by implementing the **HMAC (Hash-based Message Authentication Code)** construction using the SHA-256 hash function. Unlike naive MAC schemes vulnerable to length extension attacks, HMAC applies a well-defined combination of the secret key and message hashing that protects against such cryptographic weaknesses. The server generates a MAC for a given message and verifies it using the same HMAC process, ensuring message integrity and authenticity. The script tests this mechanism by first validating a legitimate message and its MAC successfully, then attempts to verify a forged message where unauthorized data has been appended but the MAC remains unchanged. The verification correctly fails in this case, demonstrating that HMAC effectively mitigates forgery attacks. Overall, this implementation highlights the importance of using robust cryptographic protocols like HMAC to secure communications against sophisticated attacks.