Data Integrity

Name : Mohamed Ahmed ID : 2205249
Name : Mariam Waleed ID: 2205184
Name : Amr Khaled ID: 2205220

client.py

Intercepted Data:

A legitimate message ("amount=100&to=alice") and its HMAC (intercepted_mac) are captured.

Attacker's Goal:

Append malicious data (e.g. "&admin=true") to the message without knowing the secret key used in the HMAC.

Key Assumption:

The attacker assumes the secret key is of a certain length (key_length = 14).

The attack only works if the guessed length is correct.

Attack Execution:

Uses hashpumpy.hashpump() to generate:

- *A forged message: original + padding + malicious data*
- *A forged MAC for the new message*
- *Verification:*

The forged message and MAC are passed to the server's verify() function.

If the server accepts them, the attack succeeded.

C: > Users > User > Documents > client.py > ...

```python
import hashpumpy # type: ignore
import server  # Assuming server.py has the 'verify' function

def perform_attack():
    # Step 1: Intercepted values from the server
    intercepted_message = b"amount=100&to=alice"   # Original message intercepted
    intercepted_mac = "614d28d808af46d3702fe35fae67267c"  # Original MAC from server.py output

    # Step 2: Malicious data to append
    data_to_append = b"&admin=true"  # Attack data

    # Step 3: Key length (this should match the secret key length used in the server)
    key_length = 14  # Adjust based on the server's secret key length

    # Step 4: Perform length extension attack using hashpumpy
    result = hashpumpy.hashpump(intercepted_mac, intercepted_message, data_to_append, key_length)
    forged_mac = result[0]
    forged_message = result[1]  # Already bytes, no need to encode

    # Step 5: Extract forged MAC and message

    # Print forged message and MAC
    print("=== Attacker Simulation ===")
    print("Forged message:", forged_message)
    print("Forged MAC:", forged_mac)

    # Step 6: Use server's verify function to check if the forged MAC is valid
    if server.verify(forged_message, forged_mac):
        print("MAC verified successfully (attack succeeded!)")
    else:
        print("MAC verification failed (attack failed)")

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

Steps 1 to 4

```python
def perform_attack():
    # Step 1: Intercepted values from the server
    intercepted_message = b"amount=100&to=alice"   # Original message intercepted
    intercepted_mac = "614d28d808af46d3702fe35fae67267c"  # Original MAC from server.py output

    # Step 2: Malicious data to append
    data_to_append = b"&admin=true"  # Attack data

    # Step 3: Key length (this should match the secret key length used in the server)
    key_length = 14  # Adjust based on the server's secret key length

    # Step 4: Perform length extension attack using hashpumpy
    result = hashpumpy.hashpump(intercepted_mac, intercepted_message, data_to_append, key_length)
    forged_mac = result[0]
    forged_message = result[1]  # Already bytes, no need to encode
```

Step 1: Intercept original message and MAC

- *The attacker captures a valid message and its MAC from a real request.*

•*Step 2: Prepare malicious data to append*

•*The attacker defines new data they want to add to the message.*

•*Step 3: Guess the secret key length*

•*The attacker makes an educated guess about how long the secret key is.*

•*This guess is important for the attack to succeed.*

•*Step 4: Run the length extension attack*

•*Using hashpumpy, the attacker creates:*

  - *A new forged message (original + padding + malicious data)*
  - *A new forged MAC that looks valid*

Step5 and 6

Step 5: Display forged data

The attacker prints the results of the attack:

- *forged_message: the new message (original + padding + &admin=true)*
- *forged_mac: the fake MAC that was generated using hashpumpy*

Step 6: Test if the attack worked

- *This line asks the server to verify the forged message and MAC.*
- *If the server accepts them, the attack was successful.*
- *If not, the attack failed (probably due to incorrect key length).*

```python
    # Step 5: Extract forged MAC and message

    # Print forged message and MAC
    print("=== Attacker Simulation ===")
    print("Forged message:", forged_message)
    print("Forged MAC:", forged_mac)

    # Step 6: Use server's verify function to check if the forged MAC is valid
    if server.verify(forged_message, forged_mac):
        print("MAC verified successfully (attack succeeded!)")
    else:
        print("MAC verification failed (attack failed)")

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

client_mitg.py

To demonstrate that a server using insecure MAC generation (e.g., raw SHA256(secret + message)) can be tricked into accepting a tampered message with a valid MAC, using only the original message and MAC.

- *The attack exploits weaknesses in hash functions like MD5 or SHA1 when used improperly in MAC construction.*
- *Proper use of HMAC (Hash-based Message Authentication Code) prevents such attacks.*
- *The attacker must guess the secret key length correctly for the attack to work*

C: > Users > User > Documents > client_mitg.py > ...

```python
import hashpumpy # type: ignore
import server_mitg  # Assuming server.py has the 'verify' function

def perform_attack():
    # Step 1: Intercepted values from the server
    intercepted_message = b"amount=100&to=alice"   # Original message intercepted
    intercepted_mac = "a86f897948d15c923c1f77133e805c707ca4fa752e3960efde47d618425027d5"  # Original MAC from server_mitg.py output

    # Step 2: Malicious data to append
    data_to_append = b"&admin=true"  # Attack data

    # Step 3: Key length (this should match the secret key length used in the server)
    key_length = 14  # Adjust based on the server's secret key length

    # Step 4: Perform length extension attack using hashpumpy
    result = hashpumpy.hashpump(intercepted_mac, intercepted_message, data_to_append, key_length)
    forged_mac = result[0]
    forged_message = result[1]  # Already bytes, no need to encode

    # Step 5: Extract forged MAC and message

    # Print forged message and MAC
    print("=== Attacker Simulation ===")
    print("Forged message:", forged_message)
    print("Forged MAC:", forged_mac)

    # Step 6: Use server's verify function to check if the forged MAC is valid
    if server_mitg.verify(forged_message, forged_mac):
        print("MAC verified successfully (attack succeeded!)")
    else:
        print("MAC verification failed (attack failed)")

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

Activate

Steps 1 to 4

- *Step 1: Intercepted values from the server*
- *The attacker captures the original message (amount=100&to=alice) and its corresponding MAC from the server.*
- *These are necessary to craft a forged message.*

Step 2: Malicious data to append

- *The attacker defines extra data (&admin=true) they want to add to the original message to escalate privileges.*

Step 3: Key length

- *The attacker guesses the length of the secret key used by the server in the MAC.*
- *This guess must be correct for the attack to work.*
- *Step 4: Perform length extension attack using hashpumpy*
- *The attacker uses hashpumpy.hashpump to generate:*
    - *A forged MAC valid for the extended message.*
    - *A forged message consisting of the original message plus padding and the malicious appended data*
- *Step 5: Print forged message and MAC*
- *The forged message and forged MAC are printed for inspection.*
- *This shows the attacker's modified message and its new valid MAC.*
- *Step 6: Verify forged MAC using the server's function*
- *The forged message and MAC are sent to the server's verify() function.*
- *If verification passes, the attack succeeded; otherwise, it failed*

server.py

- ***Secret key:***

A 14-byte secret key (b'supersecretkey') is defined but unknown to attackers.

- ***MAC generation (generate_mac)****:*

The MAC is computed by hashing the concatenation of the secret key and the message

- ***MAC verification (verify):***

The server recalculates the MAC for the received message and compares it to the provided MAC.

f they match, the message is considered authentic.

- ***Simulation in main():***

The server generates a MAC for an original message (b"amount=100&to=alice").

It verifies this original message successfully.

Then, a forged message is created by appending &admin=true to the original message.

The attacker naïvely reuses the original MAC to verify the forged message.

Verification fails as expected, showing that the server rejects this tampered message if no length extension attack is done.

: > Users > User > Documents > server.py > ...

```python
import hashlib

SECRET_KEY = b'supersecretkey'  #14 byte Unknown to attacker
print(len(b'supersecretkey'))
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}")
    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()
```

server_mitg.py

- *Secret key:*

A 14-byte secret key (b'supersecretkey') is used to sign messages.

The key is unknown to the attacker.

- *MAC generation (generate_mac):*

The MAC is generated using Python's hmac module with SHA-256:

<hmac.new(SECRET_KEY, message, hashlib.sha256).hexdigest() >

This method securely combines the key and message to prevent vulnerabilities.

- *MAC verification (verify):*

Recomputes the MAC on the received message and compares it with the provided MAC.

Returns True if they match, indicating authenticity.

- *Simulation in main():*

Generates a MAC for the original message "amount=100&to=alice".

Verifies the original message successfully.

Attempts to verify a forged message with appended data "&admin=true" but using the original MAC.

Verification fails, showing the system correctly detects tampering.

C: > Users > User > Documents > server_mitg.py > ...

```python
import hmac
import hashlib

SECRET_KEY = b'supersecretkey'  # 14 byte Unknown to attacker

def generate_mac(message: bytes) -> str:
    # Using HMAC with SHA-256
    return hmac.new(SECRET_KEY, message, hashlib.sha256).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).")
```

server_mitg.py

C: > Users > User > Documents > server_mitg.py > ...

```python
if __name__ == "__main__":
    main()
import hmac
import hashlib

SECRET_KEY = b'supersecretkey'  # 14 byte Unknown to attacker

def generate_mac(message: bytes) -> str:
    # Using HMAC with SHA-256
    return hmac.new(SECRET_KEY, message, hashlib.sha256).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).")
```

This Python script simulates the generation and verification of a Message Authentication Code (MAC) using HMAC with SHA-256, which is a secure method to verify message integrity and authenticity.

Key points:

Secret key (SECRET_KEY) is used to generate the MAC and is unknown to any attacker.

MAC generation is done securely with hmac.new() using SHA-256.

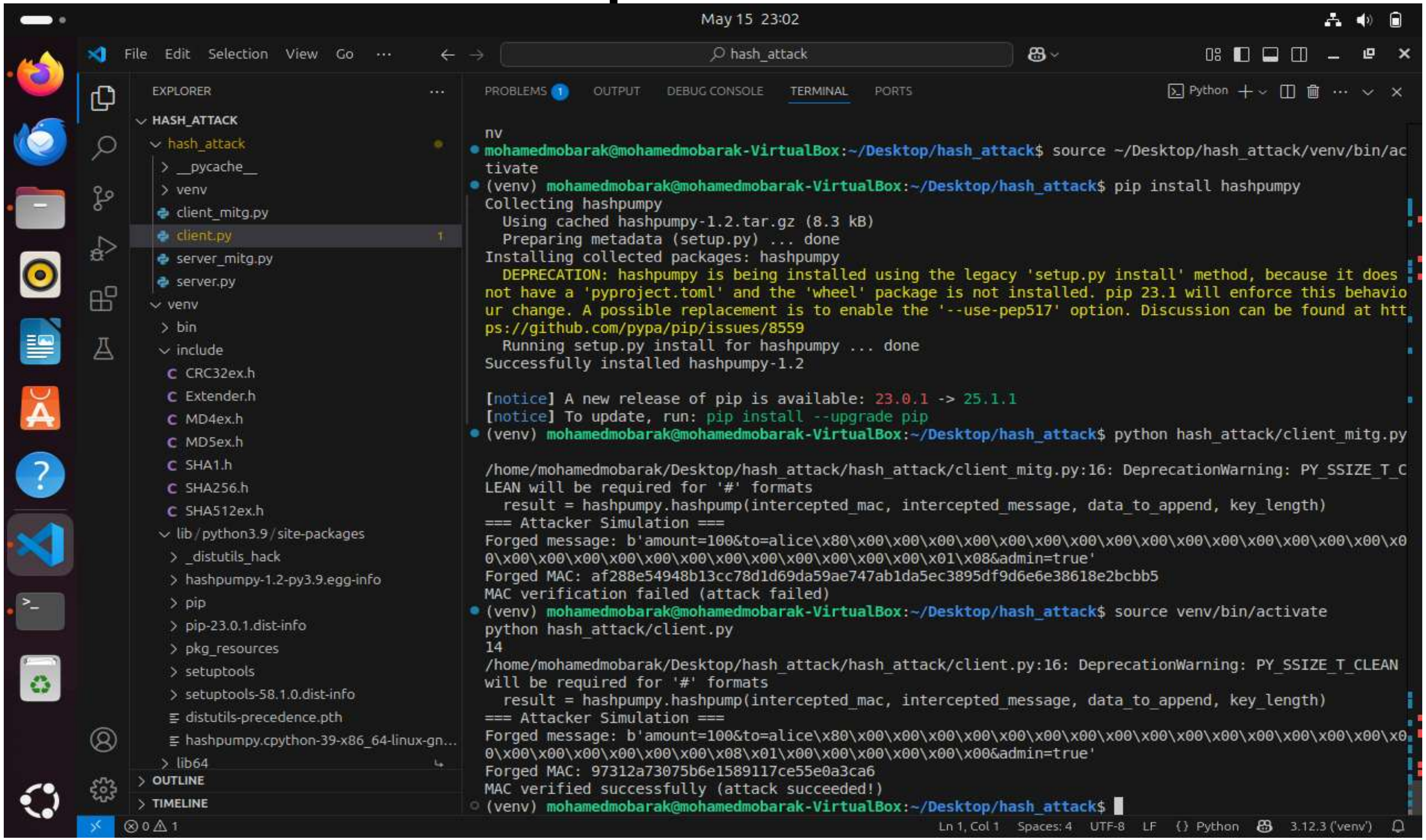
The verify function recalculates the MAC for a given message and checks if it matches the provided MAC.

The script tests two cases:

Legitimate message: The original message and MAC match, so verification succeeds.

Forged message: The message is altered by appending &admin=true, but the attacker uses the original MAC. Verification fails, demonstrating the security of HMAC.

output


May 15 23:02
File Edit Selection View Go
hash_attack
EXPLORER
HASH_ATTACK
hash_attack
__pycache__
venv
client_mitg.py
client.py
server_mitg.py
server.py
venv
bin
include
CRC32ex.h
Extender.h
MD4ex.h
MD5ex.h
SHA1.h
SHA256.h
SHA512ex.h
lib / python3.9 / site-packages
_distutils_hack
hashpumpy-1.2-py3.9.egg-info
pip
pip-23.0.1.dist-info
pkg_resources
setuptools
setuptools-58.1.0.dist-info
distutils-precedence.pth
hashpumpy.cpython-39-x86_64-linux-gn...
lib64
OUTLINE
TIMELINE
PROBLEMS 1
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
Python
nv
mohamedmobarak@mohamedmobarak-VirtualBox:~/Desktop/hash_attack$ source ~/Desktop/hash_attack/venv/bin/activate
(venv) mohamedmobarak@mohamedmobarak-VirtualBox:~/Desktop/hash_attack$ pip install hashpumpy
Collecting hashpumpy
Using cached hashpumpy-1.2.tar.gz (8.3 kB)
Preparing metadata (setup.py) ... done
Installing collected packages: hashpumpy
DEPRECATION: hashpumpy is being installed using the legacy 'setup.py install' method, because it does not have a 'pyproject.toml' and the 'wheel' package is not installed. pip 23.1 will enforce this behaviour change. A possible replacement is to enable the '--use-pep517' option. Discussion can be found at https://github.com/pypa/pip/issues/8559
Running setup.py install for hashpumpy ... done
Successfully installed hashpumpy-1.2
[notice] A new release of pip is available: 23.0.1 -> 25.1.1
[notice] To update, run: pip install --upgrade pip
(venv) mohamedmobarak@mohamedmobarak-VirtualBox:~/Desktop/hash_attack$ python hash_attack/client_mitg.py
/home/mohamedmobarak/Desktop/hash_attack/hash_attack/client_mitg.py:16: DeprecationWarning: PY_SSIZE_T_CLEAN will be required for '#' formats
result = hashpumpy.hashpump(intercepted_mac, intercepted_message, data_to_append, key_length)
=== Attacker Simulation ===
Forged message: 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\x00\x00\x00\x00\x00\x00\x00\x01\x08&admin=true'
Forged MAC: af288e54948b13cc78d1d69da59ae747ab1da5ec3895df9d6e6e38618e2bcbb5
MAC verification failed (attack failed)
(venv) mohamedmobarak@mohamedmobarak-VirtualBox:~/Desktop/hash_attack$ source venv/bin/activate
python hash_attack/client.py
14
/home/mohamedmobarak/Desktop/hash_attack/hash_attack/client.py:16: DeprecationWarning: PY_SSIZE_T_CLEAN will be required for '#' formats
result = hashpumpy.hashpump(intercepted_mac, intercepted_message, data_to_append, key_length)
=== Attacker Simulation ===
Forged message: 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\x00\x00\x00\x08\x01\x00\x00\x00\x00\x00\x00&admin=true'
Forged MAC: 97312a73075b6e1589117ce55e0a3ca6
MAC verified successfully (attack succeeded!)
(venv) mohamedmobarak@mohamedmobarak-VirtualBox:~/Desktop/hash_attack$
Ln 1, Col 1 Spaces: 4 UTF-8 LF {} Python 3.12.3 ('venv')