Mahavir Education Trust's

SHAH & ANCHOR KUTCHHI ENGINEERING COLLEGE

Chembur, Mumbai - 400 088

UG Program in Information Technology

Experiment No: 03				
Date of Performance:	26/08/2024			
Date of Submission:	**30/09/2024**			
Program Formation/ Execution/ Correction (06)	**Timely Submission (01)**	**Viva (03)**	**Experiment Marks (10)**	**Teacher Signature with date**

EXPERIMENT - 03

AIM : Cryptographic Hash Functions & Applications (HMAC)

Cryptographic Hash Functions and Applications(HMAC)

HMAC (Hash-based Message Authentication Code) is a specific type of message authentication code (MAC) that involves cryptographic hash functions. Here's an explanation in points:

1. **Purpose:**
   - HMAC provides data integrity and authenticity by ensuring that a message has not been altered and comes from a legitimate source.
2. **Components:**
   - It uses a cryptographic hash function (like SHA-256) and a secret key to generate the MAC.
3. **Process:**
   - The input message is combined with the secret key.
   - The resulting message is hashed using the chosen hash function.
   - This hash value, called the HMAC, is transmitted along with the message.
4. **Keyed Hash:**
   - Unlike regular hash functions, HMAC incorporates a secret key, making it more secure against certain attacks like collision or brute force.
5. **Security:**
   - The security of HMAC depends on the strength of the underlying hash function (e.g., SHA-256) and the length of the secret key.
6. **Usage:**
   - Commonly used in APIs (like in banking or government services in India) to verify the integrity and authenticity of requests, ensuring sensitive data is protected.
7. **Advantages:**
   - Key-based security: Requires both the secret key and the message for verification.
   - Resistant to length extension attacks: Even with hash functions vulnerable to such attacks, HMAC adds an extra layer of protection.
8. **Applications:**
   - Web services: Used in secure communication, including OAuth and AWS API.
   - Encryption protocols: HMAC is used in TLS (Transport Layer Security) to secure online transactions.

1. Plaintext and Key Details:

- **Plaintext:** 1100000000111100101010
- **IV (Initialization Vector):** 11001100 (length = 8 bits)
- **Key (k):** 10000101 (length = 8 bits)
- **ipad:** 0x5C (01011100)
- **opad:** 0x36 (00110110)

2. Dividing the Plaintext into Chunks:

- Plaintext: 1100000000111100101010
- Let's break it into chunks of 8 bits each:
    - m1 = 11000000
    - m2 = 00111100
    - m3 = 101010 (6 bits, pad with two zeros to make 8 bits: 10101000)

3. Compute z0:

- Compute z0 = IV || (k XOR ipad)

  First, compute k XOR ipad

  k XOR ipad = 11011001

  IV || (k XOR ipad) = 11001100 || 11011001

  z0 = 1100110011011001

4. Compute z1:

- Compute z1 = z0 || m1:

  z1 = 1100110011011001 || 11000000

  z1 = 110011001101100111000000

5. Compute z2:

- Compute z2 = z1 || m2:

  z2 = 110011001101100111000000 || 00111100

z2 = 1100110011011001110000000111100

6. Compute z3:

- Compute z3 = z2 || m3:

  z3 = 1100110011011001110000000111100 || 10101000

  z3 = 110011001101100111000000001111001010100 0

7. Compute z4:

- Compute z4 = z3 || L where L is the length of m in bits (22 bits in total).

  L = 00010110

- Compute z4 = z3 || L:

  z4 = 1100110011011001110000000011110010101000000010110

8. Compute p:

- Compute p = IV || (k XOR opad)

  First, compute k XOR opad:
  k XOR opad = 10110011

- Concatenate IV with the result:
  css
  Copy code
  p = 11001100 || 10110011

  p = 1100110010110011

9. Compute r:

Compute r = p || z4:

r = 1100110010110011 || 1100110011011001110000000011110010101000000010110

r = 1100110010110011110011001101100111000000001111001010100000010110

10. Final Output (HMAC Tag 't'):

- To generate a binary string of size 2l for r, and given that l = 8, the size of r should be 2 * 8 = 16 bits.

  Here's a binary string of size 16 bits that you can use for r:

  Binary String (16 bits): **r = 1100110010110011**

- Now , **Hashed Value = 00001000**

  If 00001000 is the hashed value obtained from inputting the value of r into the hash function, then this hashed value represents the final HMAC tag, which is your "Final Output."

- **Final Output (t): 00001000**

  You should enter 00001000 in the "Final Output" field to complete the HMAC computation. This is the final HMAC tag derived from the given process.

SIMULATION :

A simulator for SHA-1

Plaintext (string):

test

SHA-1

Hash output(hex):

a94a8fe5ccb19ba61c4c0873d391e987982fbbd3

HMAC Construction using a "Dummy" Hash Function

HMAC construction

Plaintext: 11000000000111100101010

Next Plaintext

length of Initialization Vector (IV), l, 8

IV: 11001100

Next IV

Key, k: 10000101

Next Key

ipad: 0x5C (01011100)
opad: 0x36 (00110110)

Put your text of size 2l to get the corresponding value of hash of size l.

Your text: 1100110010110011 get hash

Hashed value: 00001000

Final Output: 00001000 Check Answer!

CORRECT !

HASHING - SEED UBUNTU

1. Generate a Private Key:

Command : **openssl genpkey -algorithm RSA -out private_key.pem -aes256**

```
[08/26/24]seed@VM:~$ openssl genpkey -algorithm RSA -out private_key.pem -aes256
.......+++++
.........................+++++
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
```

2. Extract the Public Key:

Next, extract the public key from the private key:

Command : **openssl rsa -pubout -in private_key.pem -out public_key.pem**

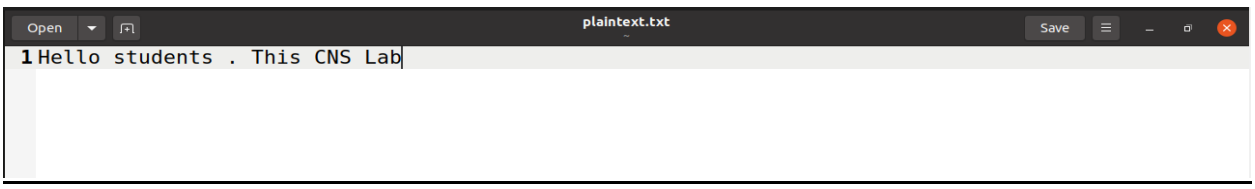
```
[08/26/24]seed@VM:~$ openssl rsa -pubout -in private_key.pem -out public_key.pem
Enter pass phrase for private_key.pem:
writing RSA key
```

3. Create or Obtain the Plaintext File:

Create or obtain the message you want to sign. For this example, create a file named plaintext.txt:

Command : **touch plaintext.txt**

```
[08/26/24]seed@VM:~$ touch plaintext.txt
[08/26/24]seed@VM:~$
```



4. Sign the Message Using the Private Key :

Generate a digital signature for plaintext.txt.This command creates a SHA-256 hash of plaintxt.txt, signs it with your private key, and saves the signature in signature.bin.

Command : **openssl dgst -sha256 -sign private_key.pem -out signature.bin plaintext.txt**

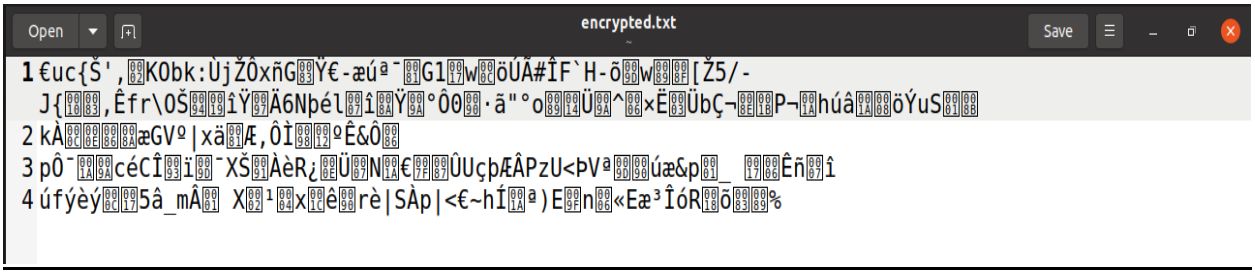
```
[08/26/24]seed@VM:~$ openssl dgst -sha256 -sign private_key.pem -out signature.b
in plaintext.txt
Enter pass phrase for private_key.pem:
```

5. Encrypt the Message Using the Public Key

Encrypt the plaintxt.txt file using the public key:

Command : **openssl rsautl -encrypt -inkey public_key.pem -pubin -in plaintext.txt -out encrypted.bin**

```
[08/26/24]seed@VM:~$ openssl rsautl -encrypt -inkey public_key.pem -pubin -in pl
aintext.txt -out encrypted.bin
[08/26/24]seed@VM:~$
```



6. Decrypt the Message Using the Private Key

Decrypt the encrypted.bin file using the private key. This decrypts the encrypted.bin file using the private key and saves the decrypted content in decrypted.txt.

Command : **openssl rsautl -decrypt -inkey private_key.pem -in encrypted.bin -out decrypted.txt**

```
[08/26/24]seed@VM:~$  openssl rsautl -decrypt -inkey private_key.pem -in encrypt
ed.bin -out decrypted.txt
Enter pass phrase for private_key.pem:
```

7. Verify the Signature Using the Public Key

Finally, verify that the signature matches the original plaintxt.txt file: **Verified OK**

Command : **openssl rsa -pubout -in private_key.pem -out public_key.pem**

```
[08/26/24]seed@VM:~$ openssl rsa -pubout -in private_key.pem -out public_key.pem
Enter pass phrase for private_key.pem:
writing RSA key
```

Command : **openssl dgst -sha256 -verify public_key.pem -signature signature.bin plaintxt.txt**

```
[08/26/24]seed@VM:~$ openssl dgst -sha256 -verify public_key.pem -signature sign
ature.bin file_to_sign.txt
Verified OK
```

CONCLUSION :

In summary, HMAC (Hash-based Message Authentication Code) is a secure method for verifying the integrity and authenticity of messages by combining a cryptographic hash function with a secret key. It is widely used in web services and encryption protocols due to its strong resistance to tampering and attacks, ensuring data remains secure during transmission.