Lab 05
Message authenticity

Objective

In this Lab, students will learn to generate message authentication codes to let message recipients validate that the message they are receiving has not been tampered with while it was in transit. In the context of message integrity, a hash function is used to produce a fixed-length message digest from a variable-size message. The most common message digests range in length from 160 to 512 bits. Any alteration to the input message produces a dramatically different message digest. The message digests help detect unauthorized alterations and message forgeries.

Activity Outcomes:

This lab teaches you the following topics:

- Learn about Message Authentication Codes (MACs) and implementation.
- Creation of Message Digest with SHA and Binary Digests in python

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	***80***	***Medium***	***CLO-6***
2	***45***	***High***	
3	***60***	***High***	

Lab-Practice Test Activities

Lab 01-Lab 04

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	***80***	***Medium -High***	***CLO-6***

Practice -Lab Activity 1: Hash-based Message Authentication Code:

A cryptographically secure MAC is known as a Hash-based Message Authentication Code (HMAC). For a hash function to be considered crypto graphically secure, it must satisfy two properties:

■ One-way property: The one-way property refers to a hash function that makes it computationally infeasible to find a message that corresponds to a given MAC.

■ Strong collision resistance property: The strong collision resistance property refers to a hash function that makes it computationally infeasible to find two different messages that hash to the same MAC.

■ Message Integrity

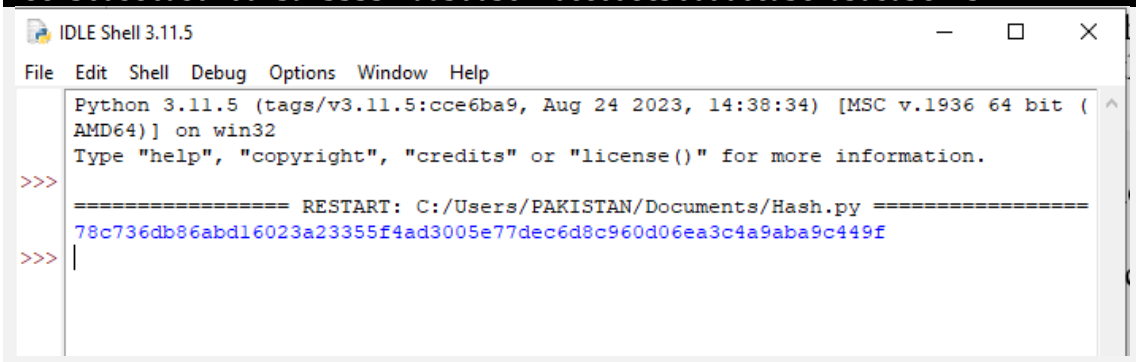
It is important to note that hash functions that are not collision resistant can be vulnerable to the birthday attack; we will explore the HMAC function and learn how to incorporate the same logic into a standard hash library. Open the Python interpreter and enter the first two lines of code to verify that you are getting the same tag:

```
import hmac, hashlib

# Calculate HMAC using SHA-256
digest = hmac.new(b"secretkey", b"our secret message", hashlib.sha256).hexdigest()

# Print the HMAC digest
print(digest)
```

```
78c736db86abd16023a23355f4ad3005e77dec6d8c960d06ea3c4a9aba9c449f
```



SHA256 has a 64-byte block length; to build an HMAC by hand, you need to build K+, ipad, and opad:

```
import binascii, hashlib

k = b"secretkey"  # Define the secret key
msg = b"our secret message"  # Define the message to be authenticated

# Padding the key to 64 bytes (512 bits)
```

```
kplus = k + b"\x00"*(64-len(k))

# Define the inner and outer pads (ipad and opad)
ipad = b"\x36" * 64
opad = b"\x5C" * 64

# XOR function to perform bitwise XOR between two byte arrays
def XOR(raw1, raw2):
   return bytes([b1 ^ b2 for b1, b2 in zip(raw1, raw2)])

# Perform the HMAC calculation using SHA-256
inner_hash = hashlib.sha256(XOR(kplus, ipad) + msg).digest()  # Inner hash
tag = hashlib.sha256(XOR(kplus, opad) + inner_hash).digest()  # Outer hash

# Print the resulting HMAC tag in hexadecimal format
print(binascii.hexlify(tag))
```

Confirm that this manual computation matches the library implementation.

Practice -Lab Activity 2: HMAC digest application to a signed message:

In next task , we will apply an HMAC digest to a signed message. Using HMAC to Sign Message The file that we are creating the message digest for is a simple text file that contains only Hello. When run, the code reads a data file and computes an HMAC signature for it:

```
import hmac
import hashlib

myKey = b'this_is_my_secret'  # Define the key
digest_maker = hmac.new(myKey, digestmod=hashlib.sha256)  # Create a new HMAC object using SHA-256

with open('test.txt', 'rb') as f:  # Open the file in binary mode
   while True:
      block = f.read(1024)  # Read the file in chunks of 1024 bytes
      if not block:  # If there's no more data, break the loop
         break
      digest_maker.update(block)  # Update the HMAC with the current block

digest = digest_maker.hexdigest()  # Get the final HMAC digest in hexadecimal format
print(digest)  # Print the resulting digest
```

The output should produce the following results:

```
c2b5ac0978608c196f6237ab3983ebd2
```

Practice -Lab Activity 3: Message Digest with SHA

MD5 is one of the most common algorithms used for hashing, but over the years MD5 hashes have proven to have a number of weaknesses such as collisions and are vulnerable to length extension attacks. The SHA family of algorithms offer stronger options and should be used instead, but many of these algorithms are also susceptible to the length extension attack;

The new() function on the hmac object takes three arguments. The first is the secret passphrase or key; this will be needed by both the sender and receiver.
The second value is an initial message. If the message content that needs to be authenticated is small, such as a timestamp or HTTP POST, the entire body of the message can be passed to the function; if not, you will need to use the update() method. The last argument is the digest module to be used. The efault is hashlib.md5. The following example will create a test.txt file that contains Hello and then checks if the hash is using SHA256:

```
import hmac
import hashlib

myKey = b'this_is_my_secret'
digest_maker = hmac.new(myKey, b'', hashlib.sha256)

# Write 'Hello' to the file
with open("chapter7/test.txt", "wb") as hello_file:
    hello_file.write(b'Hello')

# Calculate HMAC for the file content
with open('chapter7/test.txt', 'rb') as f:
    while True:
        block = f.read(5)
        if not block:
            break
        digest_maker.update(block)

# Get the final digest in hexadecimal format
digest = digest_maker.hexdigest()

# Print the resulting HMAC digest
print(digest)
```

```
import hmac
import hashlib
myKey = b'this_is_my_secret'
digest_maker = hmac.new(myKey, b'', hashlib.sha256,)
with open("chapter7/test.txt", "wb") as hello_file:
```

```
  hello_file.write(b'Hello')
  hello_file.close()
# the test.txt file contains the bytes 'Hello'
with open('chapter7/test.txt', 'rb') as f:
  while True:
    block = f.read(5)
    if not block:
      break
    digest_maker.update(block)
    digest = digest_maker.hexdigest()
print(digest)
```

Code output:

```
6833cebacb9495c1cccba617d4b5f3aefda3dc03fcb3f8d070d61a09a4084a02
```

Lab Activity 4: Binary Digests----GRADED TASK

The previous examples used the hexdigest() method to produce a printable digest. The hexdigest() is a different representation of the value calculated by the digest() method, which is a binary value that may include unprintable characters, including NUL. Some web services such as Amazon S3 and Google checkout use the Base64-encoded version of the binary digest instead of the hexdigest().

To see the difference, I am using the same Hello text file from the previous example:

```
import base64
import hmac
import hashlib
myKey = b'this_is_my_secret'
with open('test.txt', 'rb') as f:
  body = f.read(5)
hash = hmac.new(myKey, body, hashlib.sha256,)
digest = hash.digest()
print(base64.encodebytes(digest))
```

The output generated should resemble the following:

```
b'aDPOusuUlcHMy6YX1LXzrv2j3AP8s/jQcNYaCaQISgI=\n'
```

HMAC authentication should be used for any public network service, and any time data is stored where security is important. For example, when sending data through a pipe or socket, that data should be signed, and then the signature should be tested before the data is used.

The first step is to establish a function to calculate a digest for a string, and a simple class to be instantiated and passed through a communication channel:

```
import hashlib
import hmac

def make_digest(message):
    """Return a digest for the message."""
    myKey = b'this_is_my_secret'
    hash = hmac.new(myKey, message, hashlib.sha3_256)
    return hash.hexdigest().encode('utf-8')

# You must encode your message before it is hashed.
message = b'This is a test of the emergency broadcast system; it is only a test.'
rd = make_digest(message)
print(rd)
```

The preceding code listing should produce a digest for our intended message. Your output should look identical to the output shown in Figure 7.1.

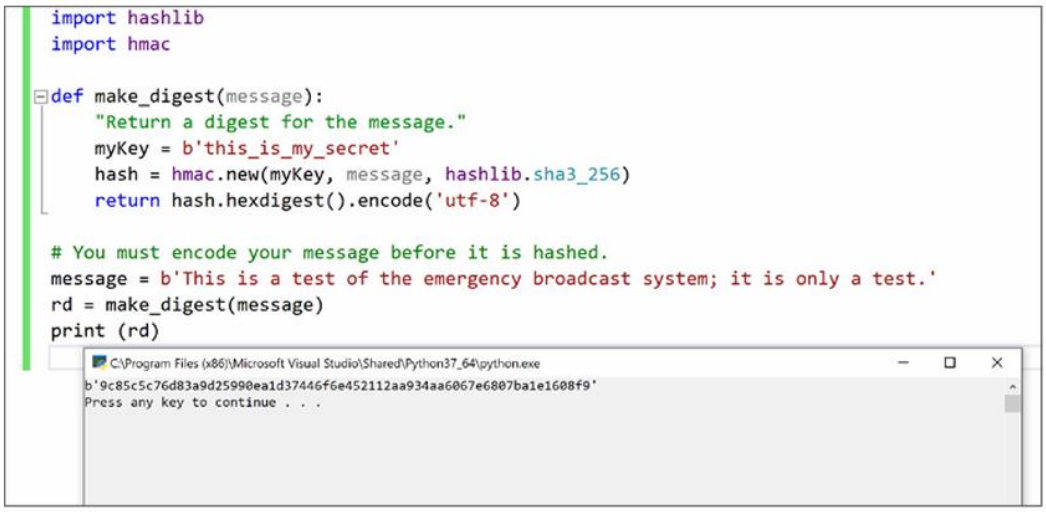


Figure 7.1: Binary digests