



**Faculty of Engineering & Technology**  
**Electrical & Computer Engineering Department**  
**Applied Cryptography ENCS4320**  
**Hash Length Extension Attack Lab**

**Prepared by:**

Hasan Hamed      1190496

Shorooq Ngjar    1192415

**Instructor:** Dr. Ahmad Alsadeh

**Section:** 1

**Date:** 4/9/2022

## **1. Abstract**

This lab aims to learn how to verify the integrity of the request using MAC, how to generate the MAC by using a one-way hash and the key and learn the Hash Length Extension attack which allows attackers to modify the message while still being able to generate a valid MAC based on the modified message, without knowing the secret key, and to learn the importance of padding in these kinds of attacks, and finally to learn how to mitigate this attack using HMAC (a keyed hash message authentication code).

# Table of Contents

1. Abstract .....	1
2. Theory.....	1
2.1. Introduction.....	1
2.2. Padding.....	2
2.3. Hash Length Extension Attack .....	3
2.4. HMAC.....	5
3. Procedure and Results.....	6
3.1. Lab Environment.....	6
3.2. Task 1: Send Request to List Files.....	8
3.3. Task 2: Create Padding .....	12
3.4. Task 3: The Length Extension Attack .....	15
3.5. Task 4: Attack Mitigation using HMAC .....	18
4. Conclusion .....	27
5. References .....	28

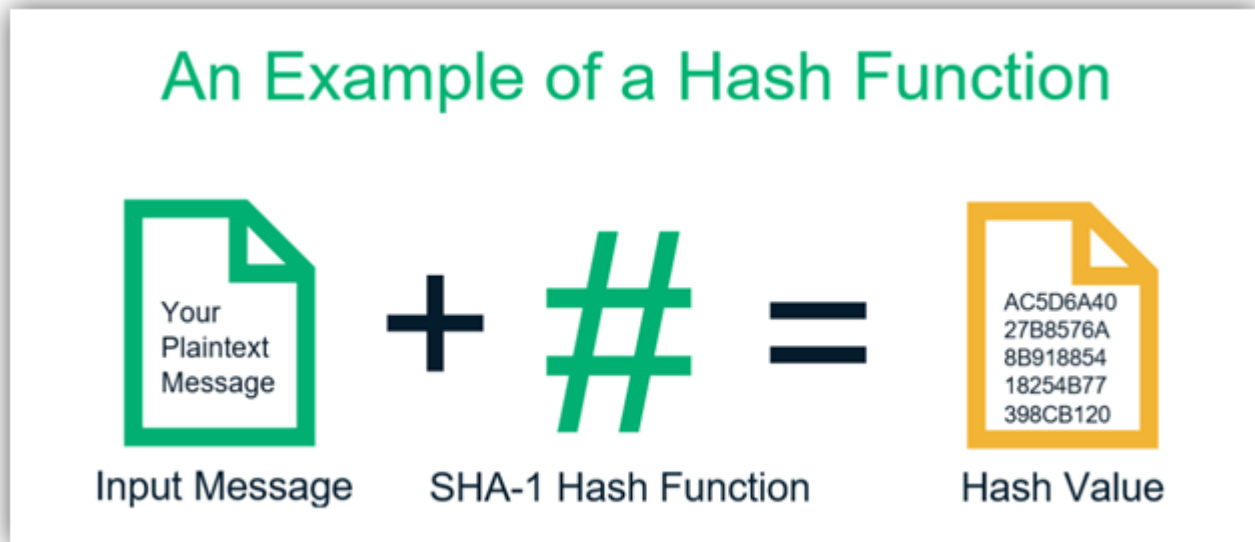
## Table of Figures

Figure 2.1: Hash Functions .....	1
Figure 2.2: SHA-256 hash values of two texts that differ by one character.....	2
Figure 3.1: Adding Entry to /etc/hosts file .....	6
Figure 3.2: Building Container Image .....	7
Figure 3.3: Starting The Container.....	8
Figure 3.4: Calculating MAC for Listing Directories.....	9
Figure 3.5: Server Response for Listing Directories.....	10
Figure 3.6: Calculating MAC for Download .....	11
Figure 3.7: Server Response for Downloading .....	12
Figure 3.8: SHA256 Padding Python Code.....	13
Figure 3.9: Shell Script to change /x to %.....	14
Figure 3.10: Padding Output .....	15
Figure 3.11: Generating MAC, Padding and Generating New MAC .....	16
Figure 3.12: Generating New Mac Code .....	16
Figure 3.13: Hash Length Extension Attack Result .....	17
Figure 3.14: Changing lab.py Function .....	18
Figure 3.15: Editing Message in Hmac.py Code .....	19
Figure 3.16: Generating a MAC for the Message Using HMAC .....	19
Figure 3.17: Server Response for HMAC Request .....	20
Figure 3.18: Editing HMAC Code for the New Request .....	21
Figure 3.19: Obtaining the New Request MAC.....	21
Figure 3.20: New Request Response .....	22
Figure 3.21: Generating SHA256 MAC.....	23
Figure 3.22: SHA256 Response Due to HMAC.....	24
Figure 3.23: Obtaining MAC and Padding .....	25
Figure 3.24: Hash Length Extension Attack Failed .....	26

## 2. Theory

### 2.1. Introduction

SHA-256, which stands for secure hash algorithm 256, is a cryptographic hashing algorithm (or function) that's used for message, file, and data integrity verification. It's part of the SHA-2 family of hash functions and uses a 256-bit key to take a piece of data and convert it into a new, unrecognizable data string of a fixed length. This string of random characters and numbers, called a hash value, is also 256 bits in size.



**Figure 2.1: Hash Functions**

Let's consider the following example. Say you write the message "Good morning" and apply a SHA-256 hash function to it. It will look like this:

*90a90a48e23dcc51ad4a821a301e3440ffeb5e986bd69d7bf347a2ba2da23bd3*, Now, say you decide to do the same with a similar message, "Good morning!" It will result in an entirely different string of hexadecimal characters of the same length.

The following graphic shows the SHA-256 hash values of two texts that differ by one character:

Text	Good morning
Hash	90a90a48e23dcc51ad4a821a301e3440ffeb5e986bd69d7bf347a2ba2da23bd3
Text	Good morning!
Hash	c9ebfb6f4b8e880908a737b8d770aa3a518fb6053b327720e8dcc79609c32858

Figure 2.2: SHA-256 hash values of two texts that differ by one character

SHA 256 ensures data integrity so that both parties can be sure that the communication is actually from the person they think it is. The recipient device creates a hash of the original message and compares it to the hash value sent by the sender. If both hash values are equal, the message has not been tampered with during transit.[1]

## 2.2. Padding

The block size of SHA-256 is 64 bytes, so a message M will be padded to the multiple of 64 bytes during the hash calculation. According to RFC 6234, paddings for SHA256 consist of one byte of `\x80`, followed by a many 0's, followed by a 64-bit (8 bytes) length field (the length is the number of bits in the M). Assume that the original message is `M = "This is a test message"`. The length of M is 22 bytes, so the padding is  $64 - 22 = 42$  bytes, including 8 bytes of the length field. The length of M in term of bits is  $22 * 8 = 176 = 0xB0$ . SHA256 will be performed in the following padded message: `"This is a test message" "\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\x00" "\x00\x00\x00"`

It should be noted that the length field uses the Big-Endian byte order, i.e., if the length of the message is `0x012345`, the length field in the padding should be: `"\x00\x00\x00\x00\x00\x01\x23\x45"`

It should be noted that in the URL, all the hexadecimal numbers in the padding need to be encoded by changing `\x` to `%`. For example, `\x80` in the padding should be replaced with `%80` in the URL above. On the server side, encoded data in the URL will be changed back to the binary numbers. See the following example:

`"\x80\x00\x00\x99"` should be encoded as `"%80%00%00%99"`.[2]

### 2.3. Hash Length Extension Attack

In cryptography and computer security, a length extension attack is a type of attack where an attacker can use  $\text{Hash}(\text{message}_1)$  and the length of  $\text{message}_1$  to calculate  $\text{Hash}(\text{message}_1 \parallel \text{message}_2)$  for an attacker-controlled  $\text{message}_2$ , without needing to know the content of  $\text{message}_1$ . Algorithms like MD5, SHA-1 and most of SHA-2 that are based on the Merkle–Damgård construction are susceptible to this kind of attack. Truncated versions of SHA-2, including SHA-384 and SHA-512/256 are not susceptible, nor is the SHA-3 algorithm.

When a Merkle–Damgård based hash is misused as a message authentication code with construction  $H(\text{secret} \parallel \text{message})$ , and  $\text{message}$  and the length of  $\text{secret}$  is known, a length extension attack allows anyone to include extra information at the end of the message and produce a valid hash without knowing the secret. Since HMAC does not use this construction, HMAC hashes are not prone to length extension attacks.

A server for delivering waffles of a specified type to a specific user at a location could be implemented to handle requests of the given format:

```
Original Data: count=10&lat=37.351&user_id=1&long=-119.827&waffle=eggo  
Original Signature: 6d5f807e23db210bc254a28be2d6759a0f5f5d99
```

The server would perform the request given (to deliver ten waffles of type `eggo` to the given location for user "1") only if the signature is valid for the user. The signature used here is a MAC, signed with a key not known to the attacker. (This example is also vulnerable to a replay attack, by sending the same request and signature a second time.)

It is possible for an attacker to modify the request, in this example switching the requested waffle from "eggo" to "liege." This can be done by taking advantage of a flexibility in the message format if duplicate content in the query string gives preference to the latter value. This flexibility does not indicate an exploit in the message format, because the message format was never designed to be cryptographically secure in the first place, without the signature algorithm to help it.

```
Desired          New          Data:          count=10&lat=37.351&user_id=1&long=-  
119.827&waffle=eggo&waffle=liege
```

In order to sign this new message, typically the attacker would need to know the key the message was signed with, and generate a new signature by generating a new MAC. However, with a length extension attack, it is possible to feed the hash (the signature given above) into the state of the hashing function, and continue

where the original request had left off, so long as you know the length of the original request. In this request, the original key's length was 14 bytes, which could be determined by trying forged requests with various assumed lengths, and checking which length results in a request that the server accepts as valid

In order to sign this new message, typically the attacker would need to know the key the message was signed with, and generate a new signature by generating a new MAC. However, with a length extension attack, it is possible to feed the hash (the signature given above) into the state of the hashing function, and continue where the original request had left off, so long as you know the length of the original request. In this request, the original key's length was 14 bytes, which could be determined by trying forged requests with various assumed lengths, and checking which length results in a request that the server accepts as valid.

The message as fed into the hashing function is often padded, as many algorithms can only work on input messages whose lengths are a multiple of some given size. The content of this padding is always specified by the hash function used. The attacker must include all of these padding bits in their forged message before the internal states of their message and the original will line up. Thus, the attacker constructs a slightly different message using these padding rules:

```
New Data: count=10&lat=37.351&user_id=1&long=-119.827&waffle=eggo\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\x00\x00\x00\x00\x00
          \x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
          \x00\x00\x00\x02\x28&waffle=liege
```

This message includes all of the padding that was appended to the original message inside of the hash function before their payload (in this case, a 0x80 followed by a number of 0x00s and a message length,  $0x228 = 552 = (14+55)*8$ , which is the length of the key plus the original message, appended at the end). The attacker knows that the state behind the hashed key/message pair for the original message is identical to that of new message up to the final "&." The attacker also knows the hash digest at this point, which means he knows the internal state of the hashing function at that point. It is then trivial to initialize a hashing algorithm at that point, input the last few characters, and generate a new digest which can sign his new message without the original key.

```
New Signature: 0e41270260895979317fff3898ab85668953aaa2
```

By combining the new signature and new data into a new request, the server will see the forged request as a



valid request due to the signature being the same as it would have been generated if the password was known.[3]

## **2.4. HMAC**

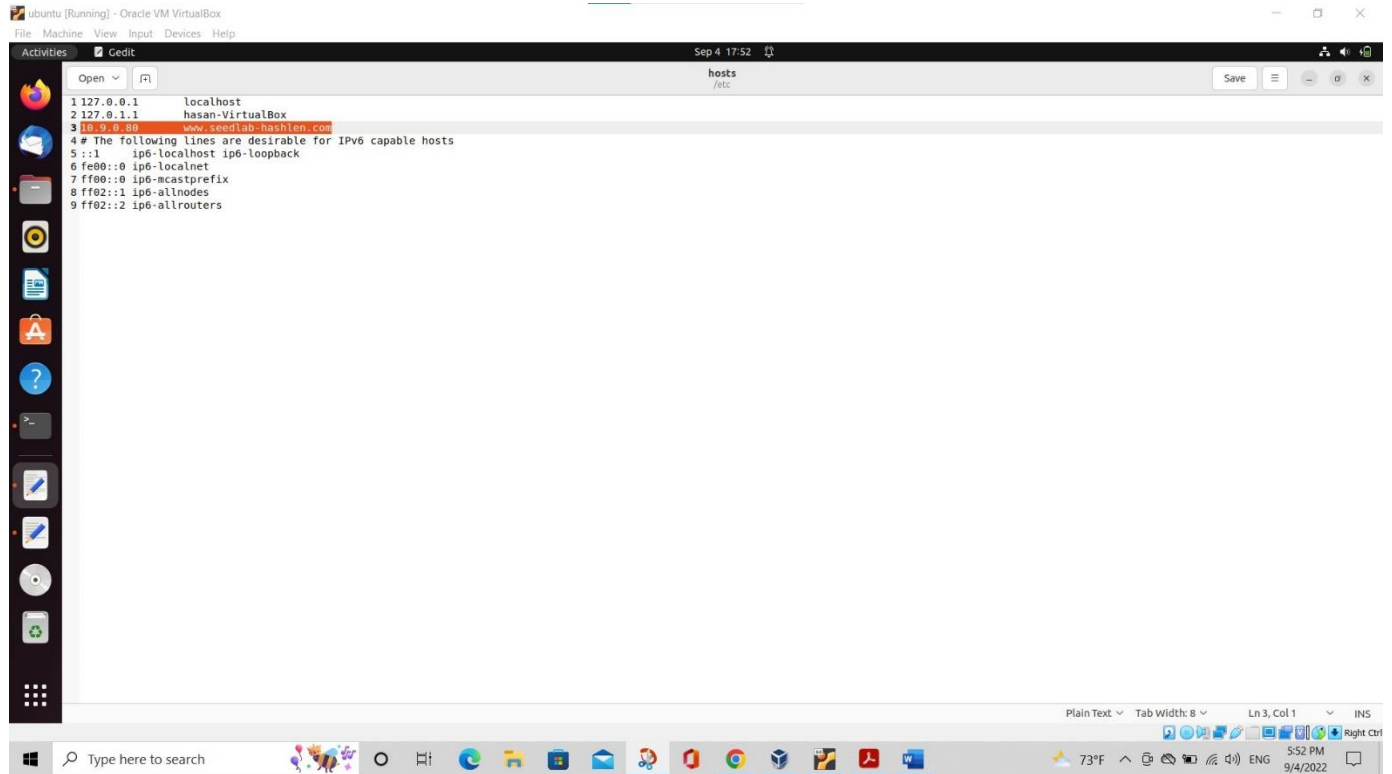
In cryptography, an HMAC (sometimes expanded as either keyed-hash message authentication code or hash-based message authentication code) is a specific type of message authentication code (MAC) involving a cryptographic hash function and a secret cryptographic key. As with any MAC, it may be used to simultaneously verify both the data integrity and authenticity of a message.

HMAC can provide authentication using a shared secret instead of using digital signatures with asymmetric cryptography. It trades off the need for a complex public key infrastructure by delegating the key exchange to the communicating parties, who are responsible for establishing and using a trusted channel to agree on the key prior to communication.[4]

### 3. Procedure and Results

#### 3.1. Lab Environment

The domain [www.seedlab-hashlen.com](http://www.seedlab-hashlen.com) was used to host the server program. In the VM, this hostname was mapped to the web server container (10.9.0.80). by adding the following entry to the /etc/hosts file as shown in Figure 3.1.

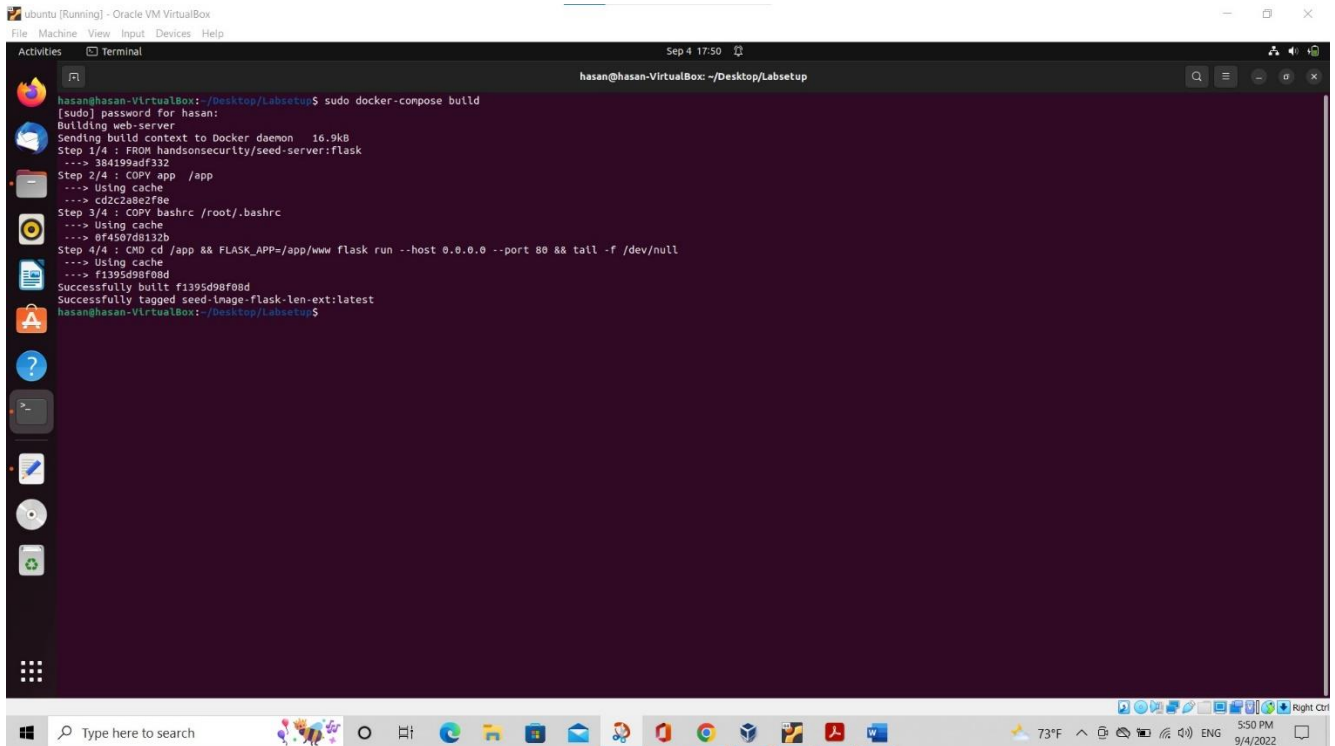


**Figure 3.1: Adding Entry to /etc/hosts file**

Then the container image was build using the following command:

```
$ docker-compose build
```

As shown in Figure 3.2.



**Figure 3.2: Building Container Image**

Then, the container was started using the following command:

```
$ docker-compose up -d
```

As shown in Figure 3.3, since it's already up it shown that it is up to date.

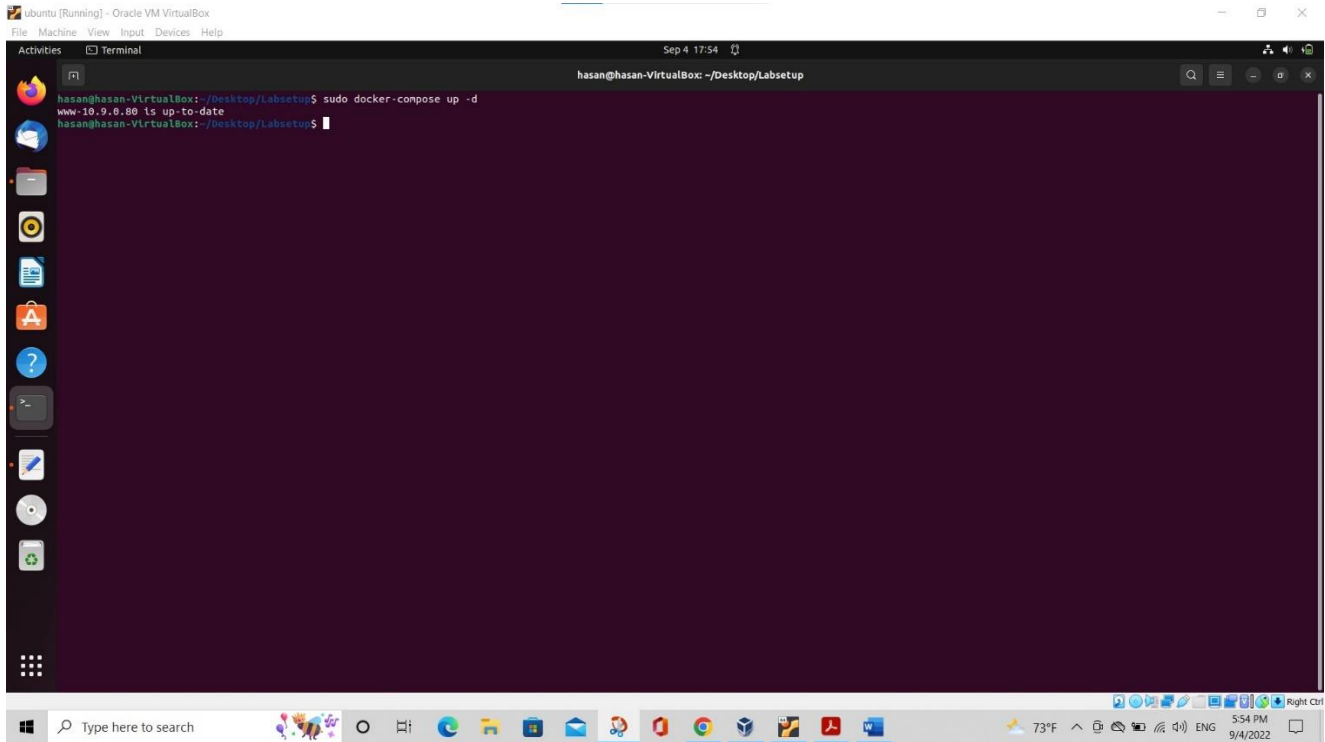


Figure 3.3: Starting The Container

### 3.2. Task 1: Send Request to List Files

The request will be sent as follows:

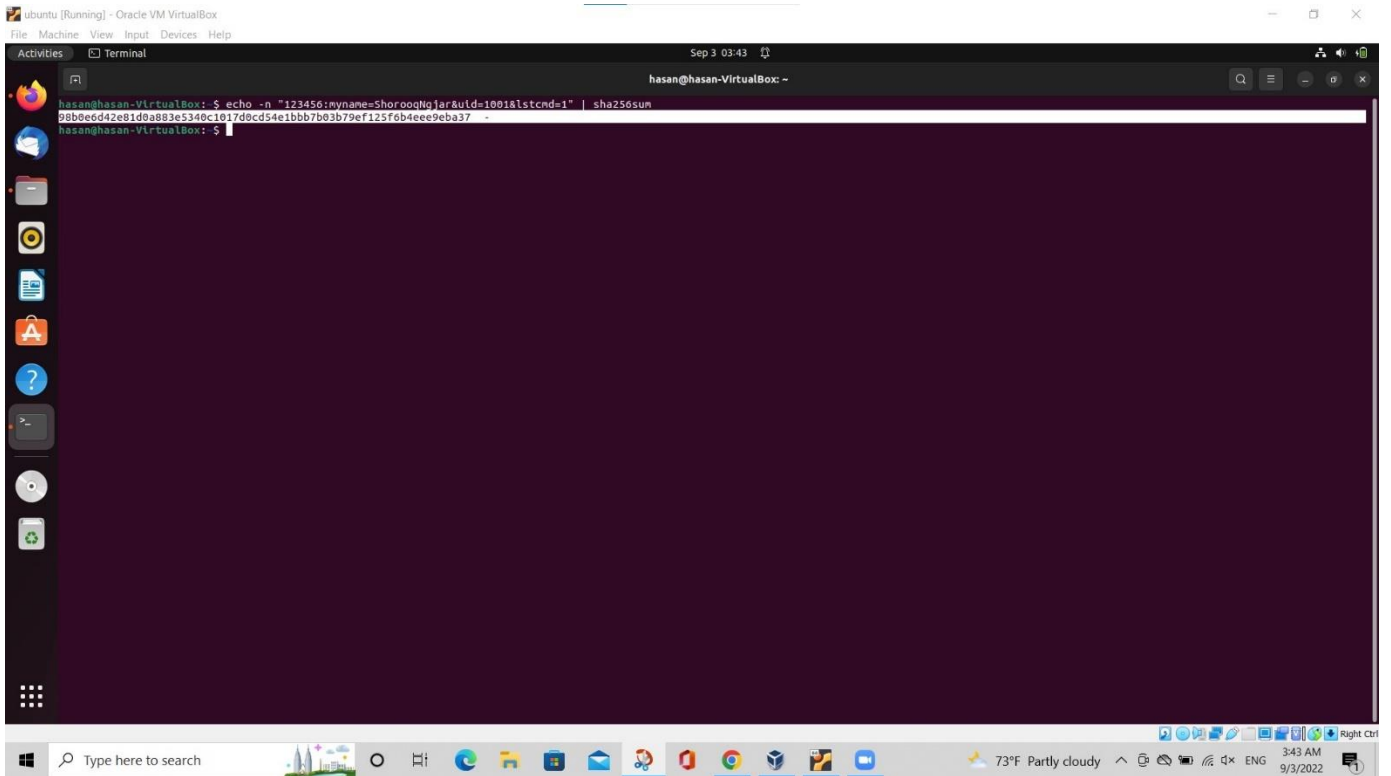
```
http://www.seedlab-hashlen.com/?myname=<name>&uid=<need-to-fill-> &lstcmd=1&mac=<need-to-calculate>
```

uid 1001 was used and its key 123456 and name Shorooq Ngjar.

The MAC will be calculated using the following command:

```
$ echo -n "123456:myname=ShorooqNgjar&uid=1001&lstcmd=1" | sha256sum
```

As shown in Figure 3.4.

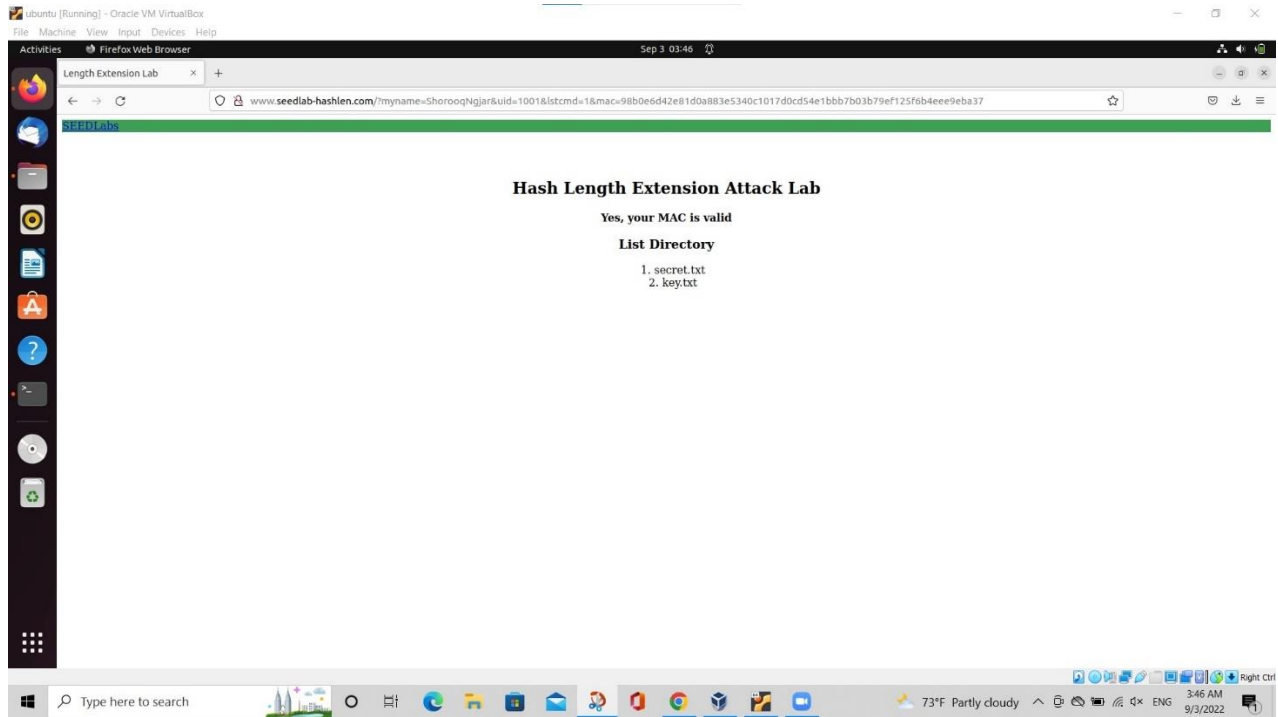


**Figure 3.4: Calculating MAC for Listing Directories**

The request was constructed as shown below:

`http://www.seedlab-hashlen.com/?myname=ShorooqNgjar&uid=1001&lstcmd=1&mac=98b0e6d42e81d0a883e5340c1017d0cd54e1bbb7b03b79ef125f6b4eee9eba37`

The request was sent to the server using Firefox and the response is shown in Figure 3.5.



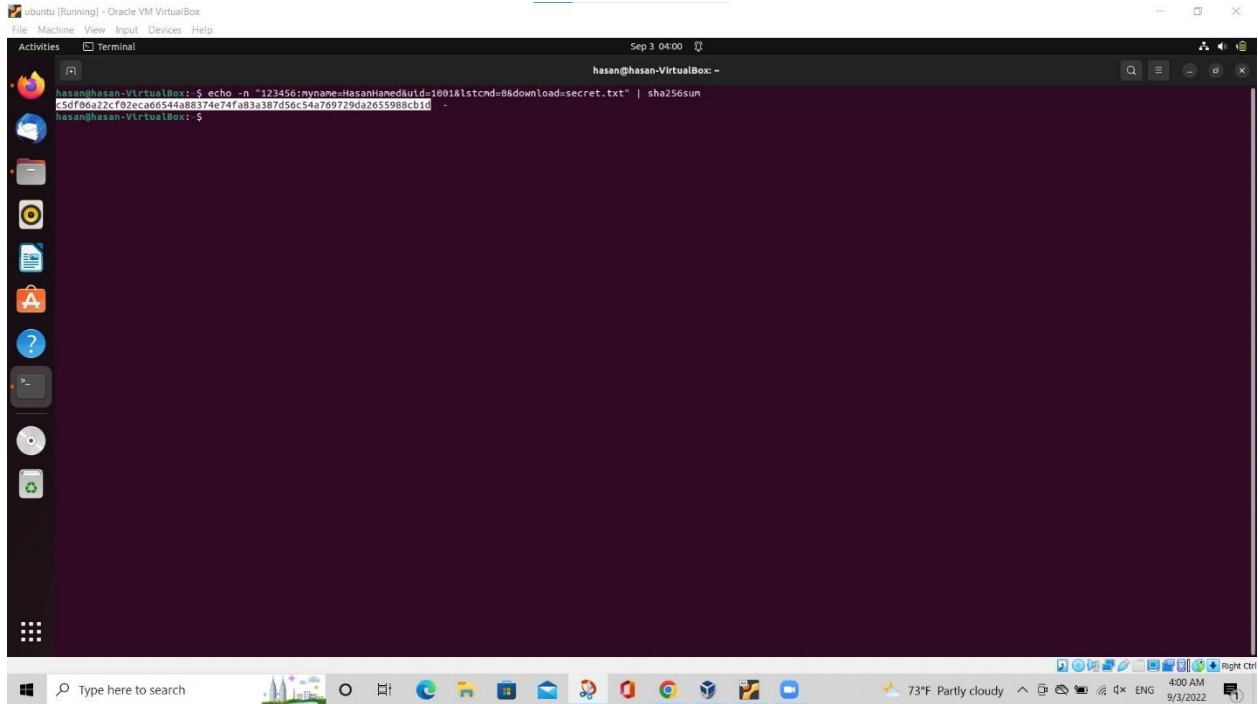
**Figure 3.5: Server Response for Listing Directories**

Now, sending a download command to the server.

First the MAC was obtained using the following command:

```
$ echo -n "123456:myname=HasanHamed&uid=1001&lstcmd=0&download=secret.txt" | sha256sum
```

As shown in Figure 3.6.

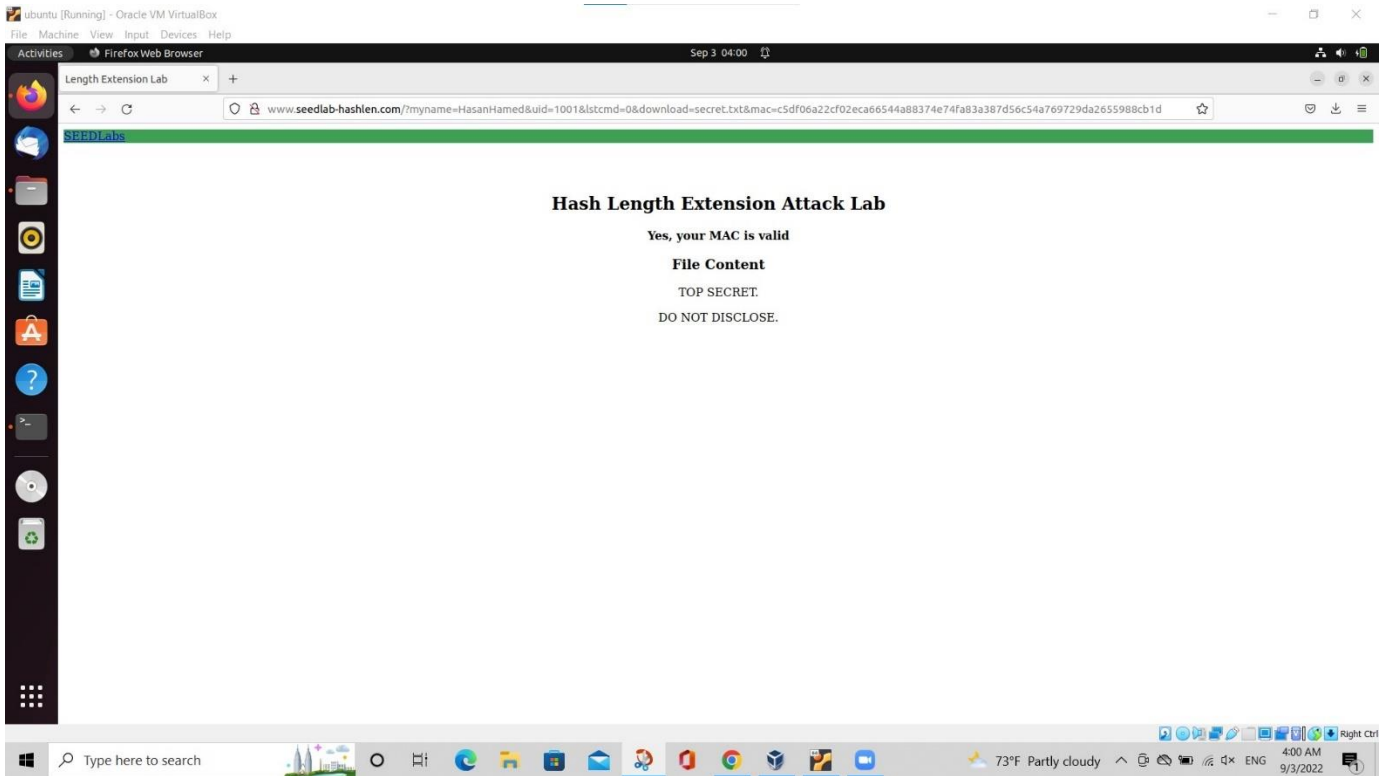


**Figure 3.6: Calculating MAC for Download**

The request was constructed as shown below:

[http://www.seedlab-hashlen.com/?myname=HasanHamed&uid=1001&lscmd=1&download=secret.txt  
&mac=c5df06a22cf02eca66544a88374e74fa83a387d56c54a769729da2655988cb1d](http://www.seedlab-hashlen.com/?myname=HasanHamed&uid=1001&lscmd=1&download=secret.txt&mac=c5df06a22cf02eca66544a88374e74fa83a387d56c54a769729da2655988cb1d)

The request was sent to the server using Firefox and the response is shown in Figure 3.7.



**Figure 3.7: Server Response for Downloading**

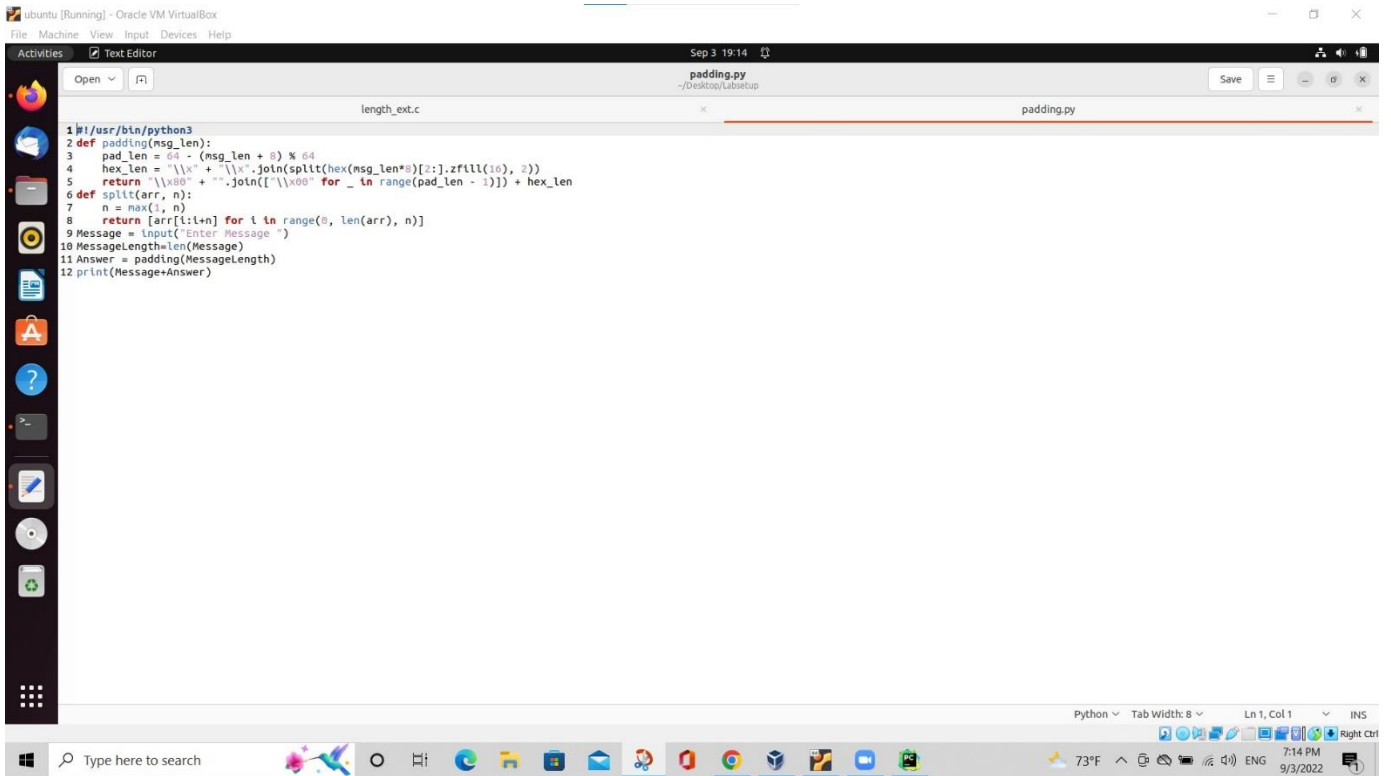
As shown in both requests since both MAC's for both requests were valid, the server responded and sent the request response which was a list request in the first part and a download request in the second one.

### **3.3. Task 2: Create Padding**

A python program was created for padding messages according to the algorithm:

padding for SHA256 consist of one byte of  $\backslashx80$ , followed by a many 0's, followed by a 64-bit (8 bytes) length field (the length is the number of bits in the M). as shown in Figure 3.8.

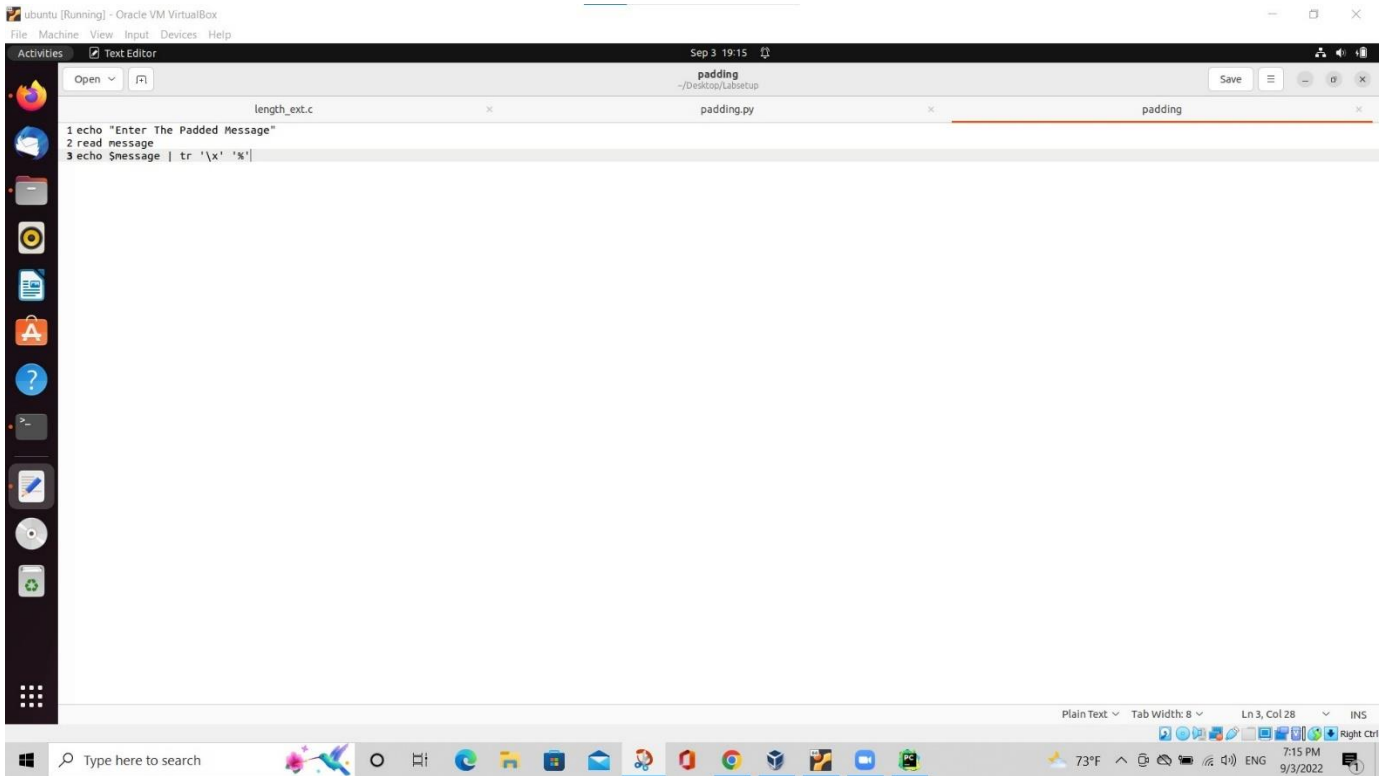


The image shows a screenshot of a text editor window titled 'padding.py' in an Ubuntu environment. The code defines a 'padding' function that takes a message length and returns a hex string of zeros. It also defines a 'split' function to split an array into chunks of size 'n'. The main logic prompts the user for a message, calculates its length, and pads it to a total length of 64 bytes. The code is as follows:

```
1 #!/usr/bin/python3
2 def padding(msg_len):
3     pad_len = 64 - (msg_len + 8) % 64
4     hex_len = "\\x" * pad_len
5     return "\\x00" * pad_len
6 def split(arr, n):
7     n = max(1, n)
8     return [arr[i:i+n] for i in range(0, len(arr), n)]
9 Message = input("Enter Message ")
10 MessageLength=len(Message)
11 Answer = padding(MessageLength)
12 print(Message+Answer)
```

**Figure 3.8: SHA256 Padding Python Code**

It should be noted that in the URL, all the hexadecimal numbers in the padding need to be encoded by changing `\x` to `%`. And because of that a shell script was created to change them as shown in Figure 3.9.



**Figure 3.9: Shell Script to change /x to %**

Using both codes, the padding for the message was created:

**123456:myname=Hasan&uid=1001&lstcmd=1**

As shown in Figure 3.10.







### 3.5. Task 4: Attack Mitigation using HMAC

In the tasks so far, it was observed the damage caused when a developer computes a MAC in an insecure way by concatenating the key and the message. In this task, the mistake made by the developer will be fixed. The standard way to calculate MACs is to use HMAC. The server program's verify mac() function should be modified and Python's hmac module will be used to calculate the MAC. The function resides in lab.py. Given a key and message (both of type string), the HMAC can be computed as shown below:

```
real_mac = hmac.new(bytearray(key.encode('utf-8')), msg=message.encode('utf-8', 'surrogateescape'),  
digestmod=hashlib.sha256).hexdigest()
```

The function was changed in the file lab.py as shown in Figure 3.14.

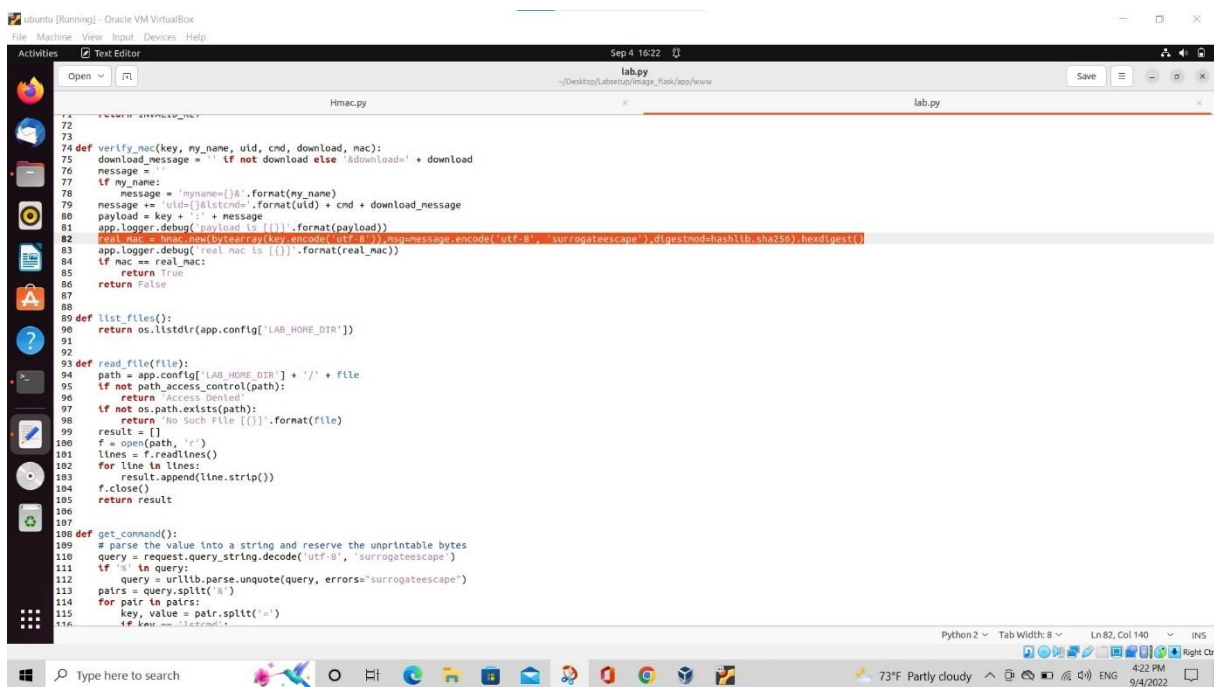


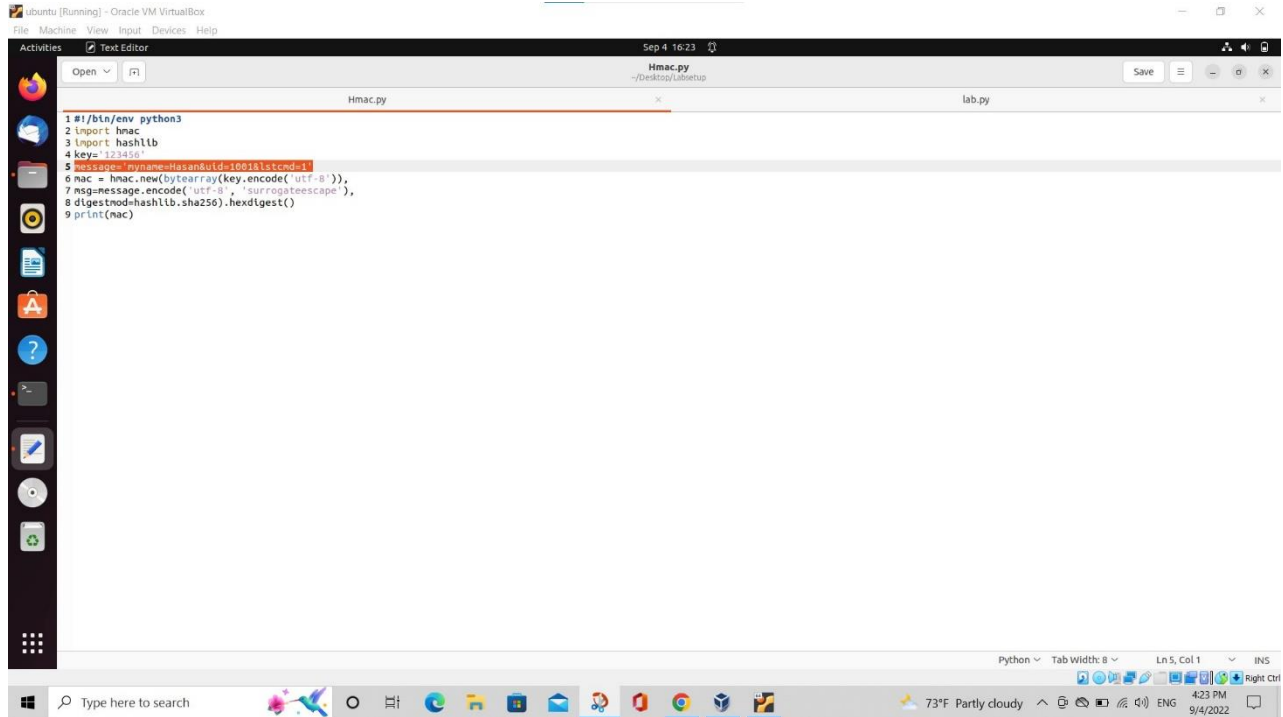
Figure 3.14: Changing lab.py Function

After making the changes, all the containers were stopped using docker-compose down command and then were rebuilt using docker-compose build and all the containers were started again using docker-compose up -d by following the steps in section 3.1. after that the changes took places.

Then, a MAC was generated for the following message:

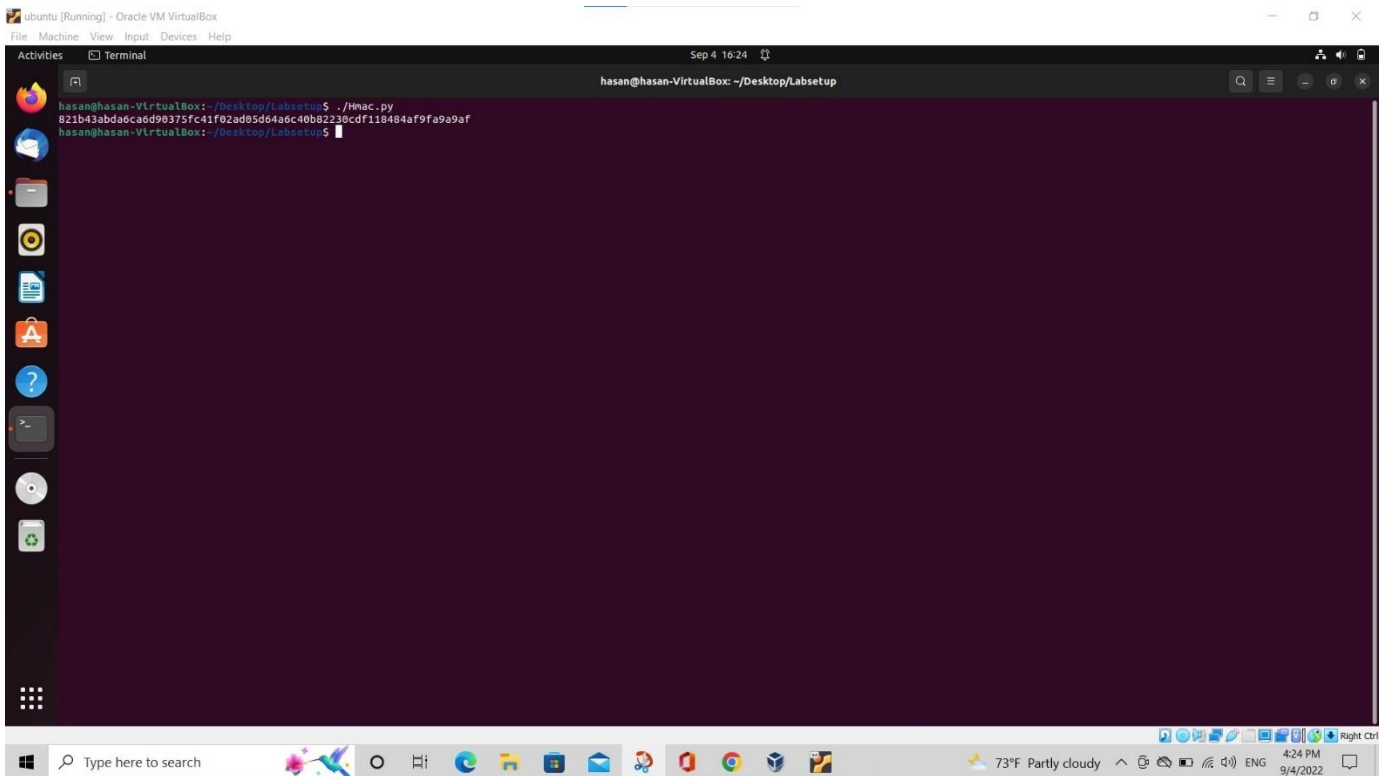
```
myname=Hasan&uid=1001&lstcmd=1
```

By editing the message in the Hmac.py code shown in Figure 3.15.



**Figure 3.15: Editing Message in Hmac.py Code**

The MAC was generated after executing the code as shown in Figure 3.16.

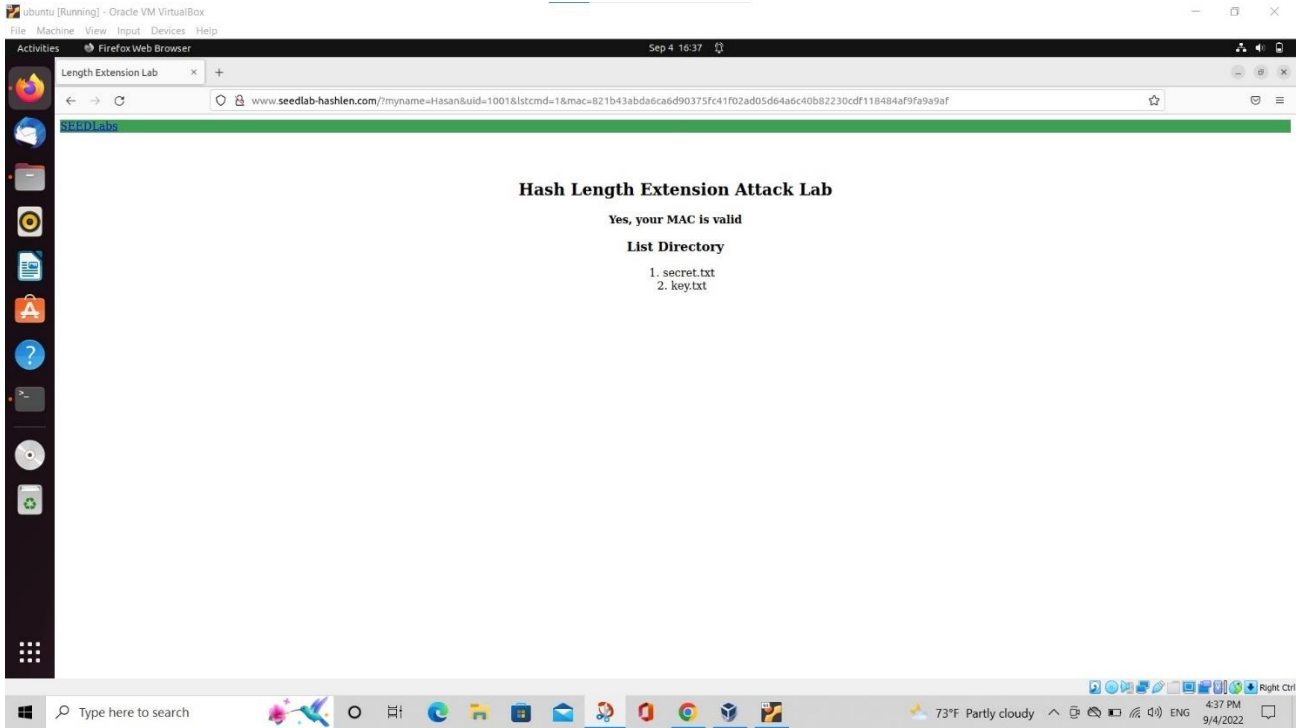


**Figure 3.16: Generating a MAC for the Message Using HMAC**

Then a request was constructed as shown below:

<http://www.seedlab-hashlen.com/?myname=Hasan&uid=1001&lstcmd=1&mac=821b43abda6ca6d90375fc41f02ad05d64a6c40b82230cdf118484af9fa9a9af>

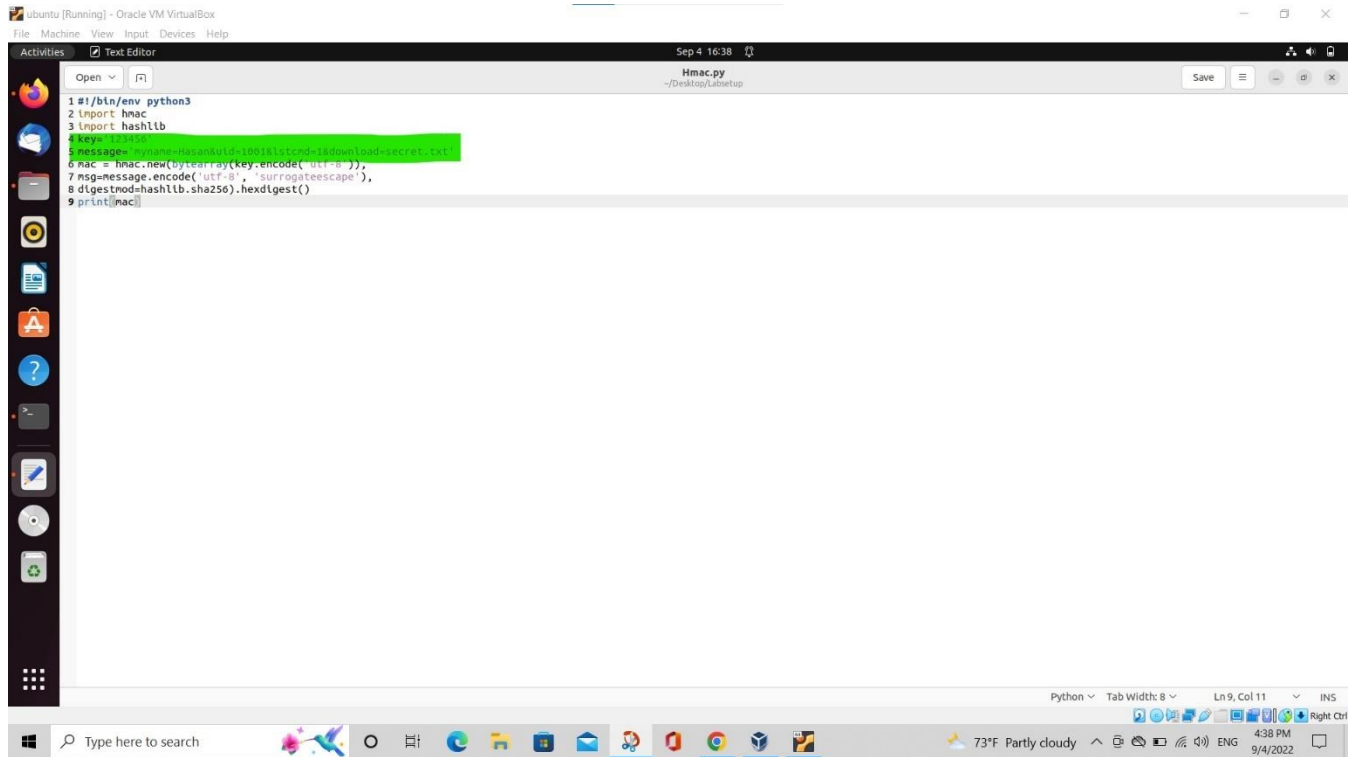
And the response got is shown in Figure 3.17.



**Figure 3.17: Server Response for HMAC Request**

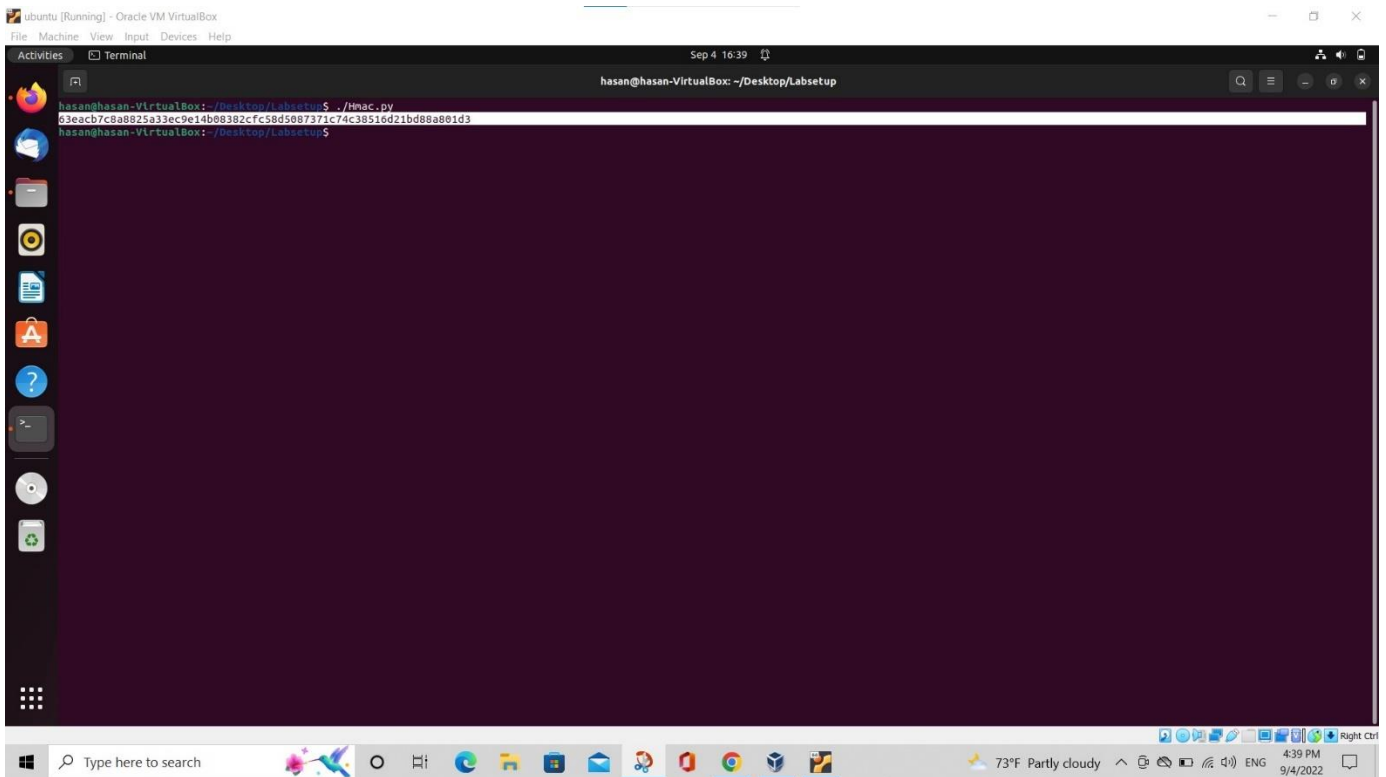
Then, a MAC for a request for download was generated as shown in Figure 3.18, the code was first edited.





**Figure 3.18: Editing HMAC Code for the New Request**

Then, the code was executed and the new MAC was obtained as shown in Figure 3.19.



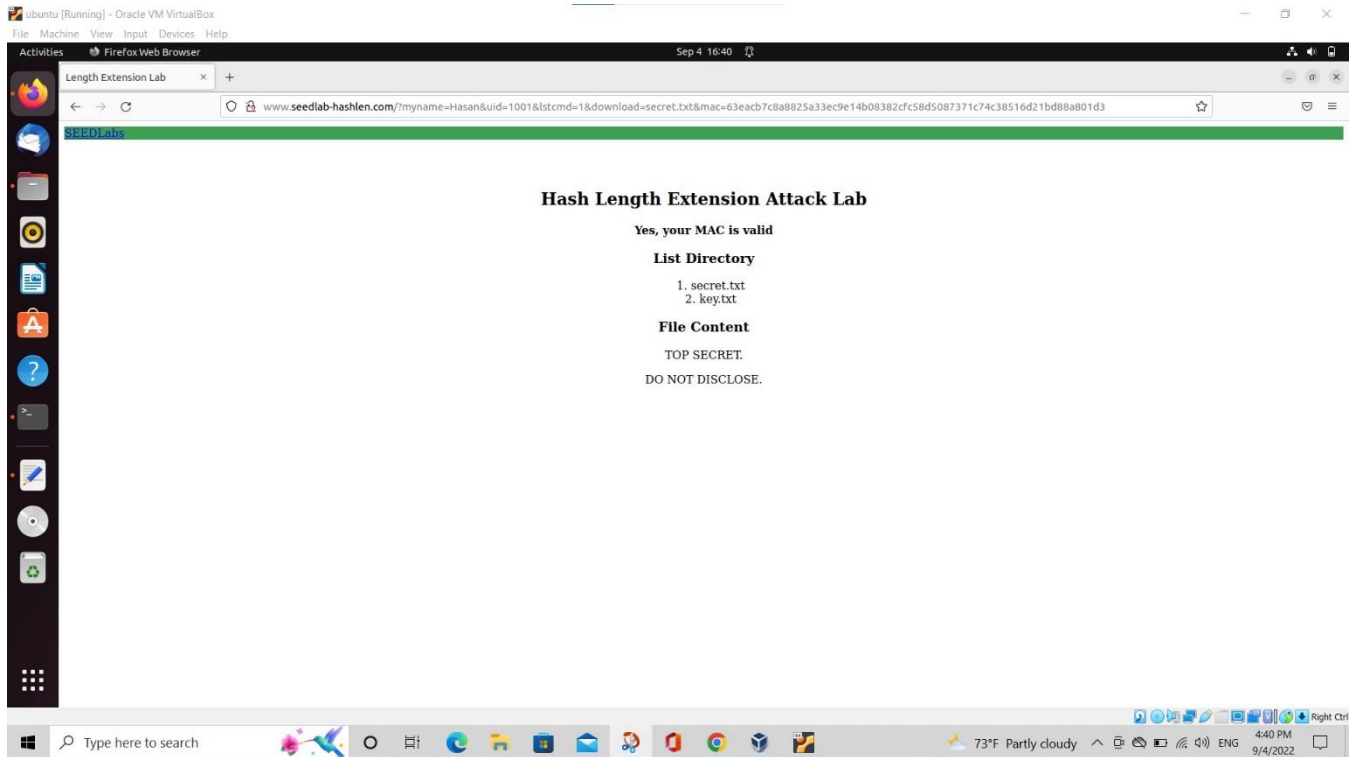
**Figure 3.19: Obtaining the New Request MAC**

Then a request was constructed as shown below:

<http://www.seedlab->

[hashlen.com/?myname=Hasan&uid=1001&lstcmd=1&download=secret.txt&mac=63each7c8a8825a33ec9e14b08382cfc58d5087371c74c38516d21bd88a801d3](http://www.seedlab-hashlen.com/?myname=Hasan&uid=1001&lstcmd=1&download=secret.txt&mac=63each7c8a8825a33ec9e14b08382cfc58d5087371c74c38516d21bd88a801d3)

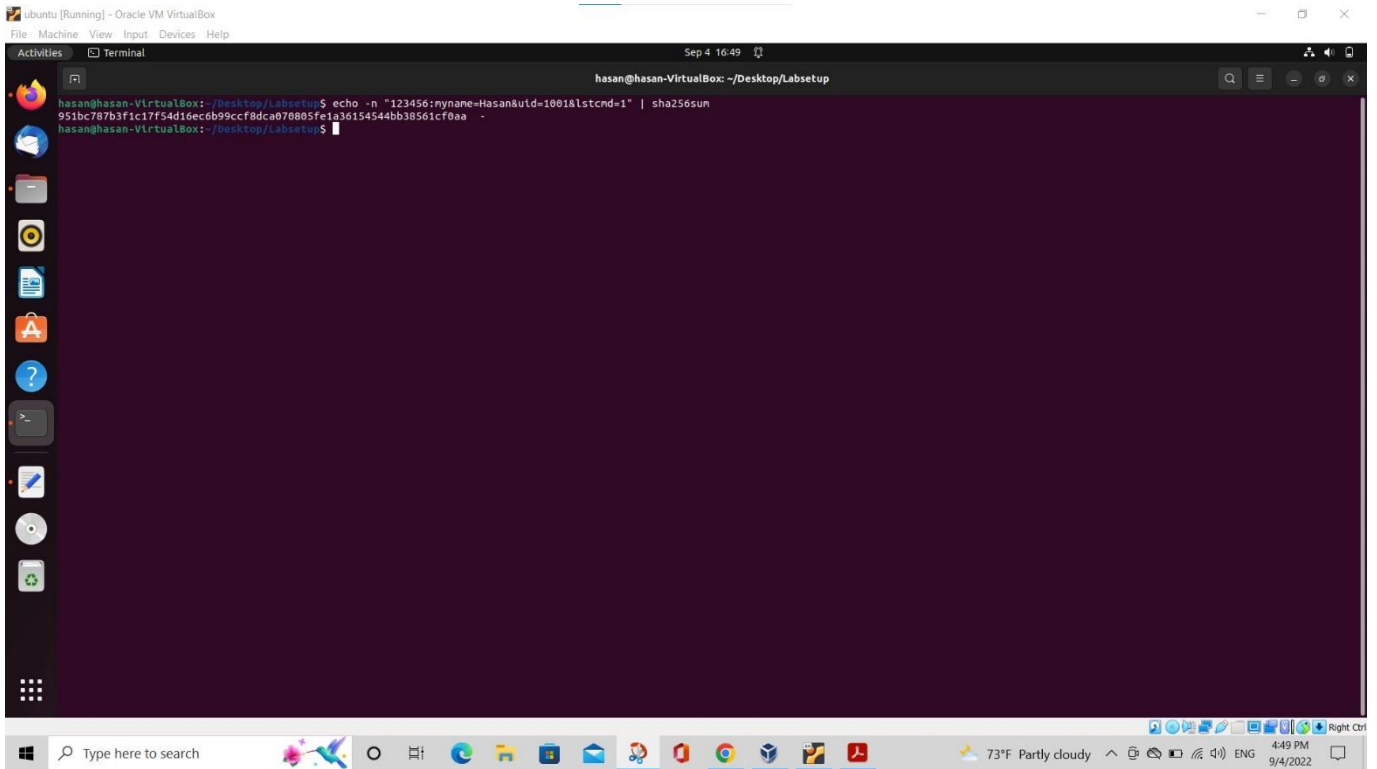
And the response after sending the request to the server was got as shown in Figure 3.20.



**Figure 3.20: New Request Response**

Now a new request using SHA256 will be sent.

First the MAC will be generated as shown in Figure 3.21.

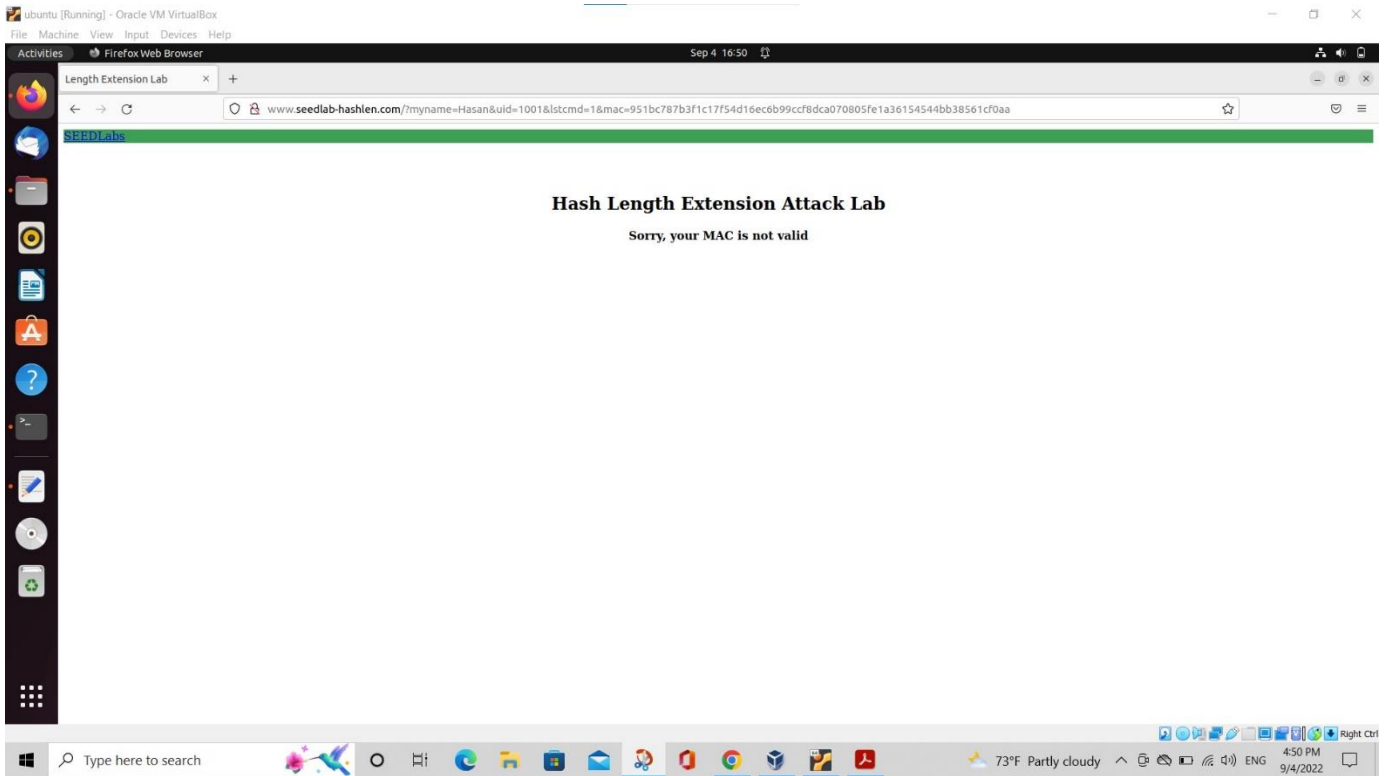


**Figure 3.21: Generating SHA256 MAC**

Then, a new request was made as shown below:

<http://www.seedlab-hashlen.com/?myname=Hasan&uid=1001&lstcmd=1&mac=951bc787b3f1c17f54d16ec6b99ccf8dca070805fe1a36154544bb38561cf0aa>

The request was sent using Firefox and the response is as shown in Figure 3.22.

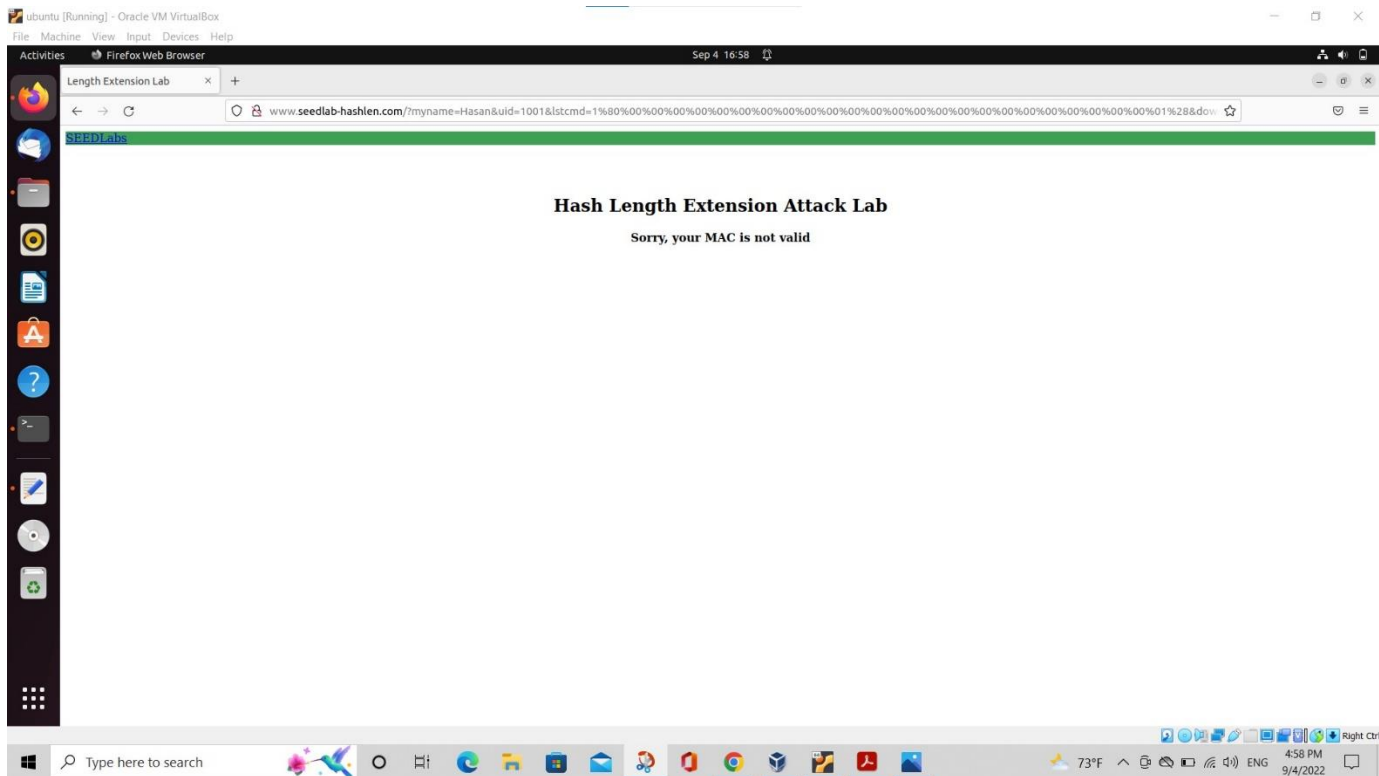


**Figure 3.22: SHA256 Response Due to HMAC**

As shown in Figure 3.22, the server response tells that the MAC is not valid since HMAC is used.

Now the Hash Length Extension attack will be tested. The original MAC and padding were obtained as shown in Figure 3.23.





**Figure 3.24: Hash Length Extension Attack Failed**

As shown in Figure 3.24 that Hash Length Extension attack was failed due to using HMAC since it's secure against those attack and checks the integrity.

## **4. Conclusion**

In conclusion, we understand the MAC and how it give the integrity for messages, and we understand how to generate the MAC's using one way hash such as SHA256, and we understand how to launch Hash Extension attacks for SHA256 using padding and original MAC and a new MAC to generate a new request without knowing the key, and finally HMAC was understood and how it's stronger and more secure for those kind of attacks.

## 5. References

[1] <https://sectigostore.com/blog/sha-256-algorithm-explained-by-a-cyber-security-consultant/>

Accessed 4 September 2022

[2] Hash Length Extension Attack Lab PDF by SEED Labs

Accessed 4 September 2022

[3] [https://en.wikipedia.org/wiki/Length\\_extension\\_attack](https://en.wikipedia.org/wiki/Length_extension_attack)

Accessed 4 September 2022

[4] <https://en.wikipedia.org/wiki/HMAC>

Accessed 4 September 2022