



Computer Engineering Department

Applied Cryptography

ENCS4320

Assignment 2

Cryptography Lab - Hash Length Extension Attack

Prepared By:

Tala Dweikat

Mazen Batrawi

1191590

1190102

Section 2

Section 2

Date: 13-2-2023

Abstract

The main aim of this homework is to get familiar with the length extension attacks and do they work, we will gain first-hand experience of how a one-way hash is used to generate a Message Authentication Code (MAC) using the message with a key. In this homework, we will launch the attack against a server program. We will forge a valid command and get the server to execute. Several tasks that guide us through the process of hash length extension attack are included. They cover the topics of Sending requests to list files, creating padding, the length extension attack and the attack Mitigation using HMAC. All these topics will be discussed with details in this report. This homework will be done using SEED Ubuntu.

Table of Contents

Introduction	1
1. Length Extension Attack	1
2. Message Authentication Code (MAC)	1
3. HMACs	2
4. Padding in Hash Functions	3
Procedure	5
1. Lab Environment	5
2. Tasks	7
a. Task 1 – Send Request to List Files	7
b. Task 2 – Create Padding	9
c. Task 3 – The Length Extension Attack	11
d. Task 4 – Attack Mitigation using HMAC	13
Conclusion	20
References	21
Appendix	22
□ Task 2 Code:	22
□ Task 3 Code:	23
□ Task 4 – First Message Code:	25
□ Task 4 – Second Message Code:	26

Table of Figures

Figure 1. Length Extension Attack [2]	1
Figure 2. How Does HMAC Work [4]	3
Figure 3. Padding in HMAC [7]	4
Figure 4. Adding Entry to the /etc/hosts file.....	5
Figure 5. Building the container image.....	6
Figure 6. Starting the container.....	6
Figure 7. Calculating MAC.....	7
Figure 8. Opening the URL in Firefox Result	8
Figure 9. Downloading MAC	8
Figure 10. Opening the URL in Firefox Download Result.....	9
Figure 11. Padding Code.....	10
Figure 12. Padding Result.....	10
Figure 13. Task 3 code.....	11
Figure 14. Task 3 original MAC + new MAC.....	12
Figure 15. Task 3 result	12
Figure 16. Editing the lab.py code	13
Figure 17. Task 4 first message code.....	14
Figure 18. Task 4 first message MAC	14
Figure 19. Task 4 first message result	15
Figure 20. Task 4 second message code	15
Figure 21. Task 4 second message MAC.....	16
Figure 22. Task 4 second message result.....	16
Figure 23. The directory files.....	17
Figure 24. Generating SHA256 MAC	17
Figure 25. SHA256 Response Due to HMAC	18
Figure 26. Getting the padding and the MAC.....	18
Figure 27. Hash Length Extension Attack Failure.....	19

Introduction

1. Length Extension Attack

A length extension attack is a form of attack in cryptography and computer security where an attacker can calculate Hash (message1 || message2) for an attacker-controlled message 2 using Hash function and the length of message 1, without needing to know the contents of message 1. The susceptible hashing operations operate by transforming an internal state using the input message. The hash digest is produced by printing the function's internal state after all of the input has been processed. The internal state can be recreated from the hash digest and then utilized to process the new input. By doing so, it is possible to extend the message and determine the hash that serves as the new message's legal signature. [1]

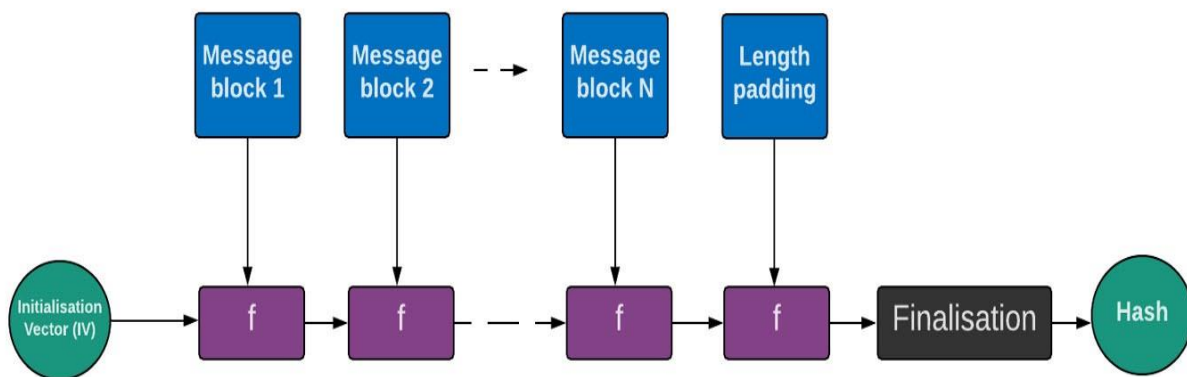


Figure 1. Length Extension Attack [2]

2. Message Authentication Code (MAC)

A session key is used in a message authentication code (MAC), a cryptographic checksum on data, to identify both unintentional and intentional modifications.

A message and a secret key that can only be known by the message's sender and intended recipient are the two inputs needed by a MAC (s). This enables the message's recipient to confirm the message's accuracy and confirm that the message's sender is in possession of the shared secret key. The recipient would be informed that the message was not from the original sender if the hash value were different if the sender didn't have the secret key. [3]

There are four different types of MACs: block cipher-based, stream cipher-based, and unconditionally secure hash function-based. In the past, using block ciphers like Data Encryption Standard (DES) was the most popular method of constructing a MAC, but hash-based MACs (HMACs), which use a secret key along with a cryptographic hash function to form a hash, have grown in popularity. [3]

MAC (Message Authentication Code) is a short piece of information derived from a secret key and a message, used to verify the authenticity and integrity of the message. The MAC is calculated using a cryptographic hash function, and the secret key is used to ensure that the message cannot be tampered with. The recipient of the message can use the secret key and the MAC to verify that the message has not been altered in transit and that it came from the claimed sender.

3. HMACs

HMAC (Hashed Message Authentication Code) is a particular kind of MAC algorithm that makes use of a secret key along with a cryptographic hash function. HMAC can be used to check a message's authenticity and data integrity. A hash function and a secret key are used in the cryptographic authentication method known as hash-based message authentication code (or HMAC).

HMAC, as opposed to methods that use signatures and asymmetric cryptography, allows authentication and the verification that data is true and legitimate. [4]

HMAC is a great file transfer data integrity-checking technology because of its effectiveness in addition to its capacity to offer data integrity and message authentication. A message of any length can be converted into a fixed-length digest using hash algorithms. As a result, even if your communications are somewhat lengthy, the message digests that go with them can still be brief, maximizing your available bandwidth. [5]

How does HMAC work?

In essence, it creates a special message authentication code that may be delivered with the message by fusing a hash algorithm (like SHA-256) with a secret key. The recipient can then verify that the

message was sent by the expected sender and was not altered in transit by using the same key in order to recompute the HMAC and compare it with the HMAC received. [4]

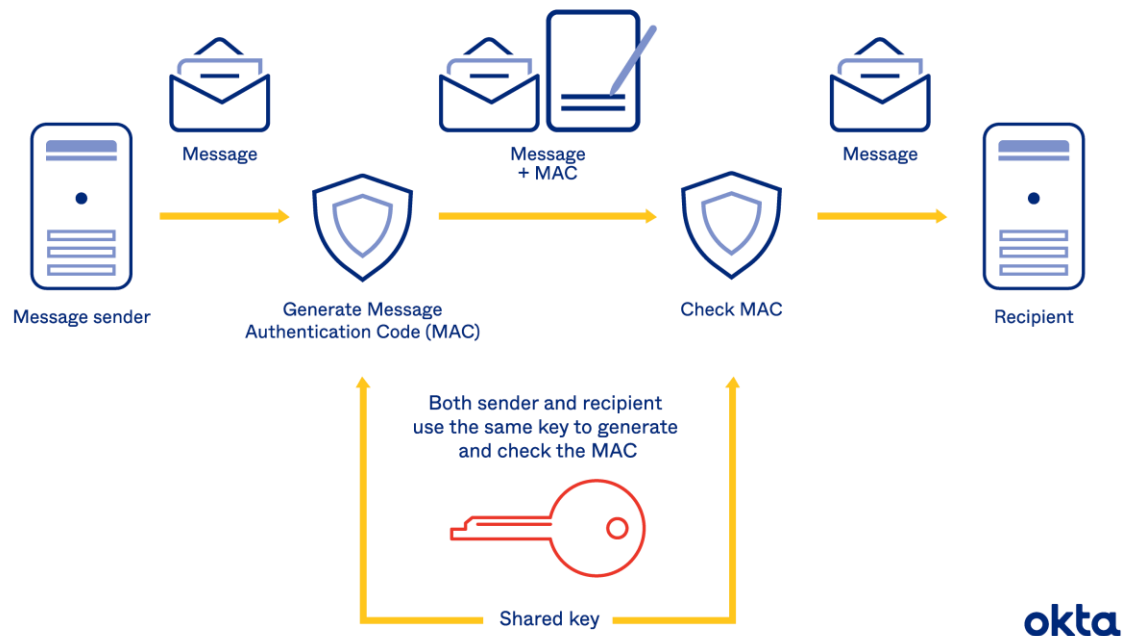


Figure 2. How Does HMAC Work [4]

4. Padding in Hash Functions

Data that may or may not be a multiple of the block size for a cipher can be extended such that it is by using padding. Many block ciphers need that the data being encrypted be an exact multiple of the block size, hence this is necessary. This is frequently used to guarantee that the message being hashed is a particular length, which is necessary for various hash algorithms to function properly. Padding is used in the context of HMAC to ensure that the input to the hash function is a multiple of the underlying hash function's block size. Through the use of padding, the HMAC calculation is able to digest the complete message without losing any data. [6]

In HMAC, there are two typical padding techniques:

1. Zero Padding: Using this technique, the message is extended by a number of 0-bits until it reaches a multiple of the block size needed by the hash function.

2. Bit Padding: This technique adds a 1-bit, a string of 0-bits, and the message's bit length to the end of the message. The Internet Engineering Task Force (IETF) specifications for HMAC specify this padding technique.

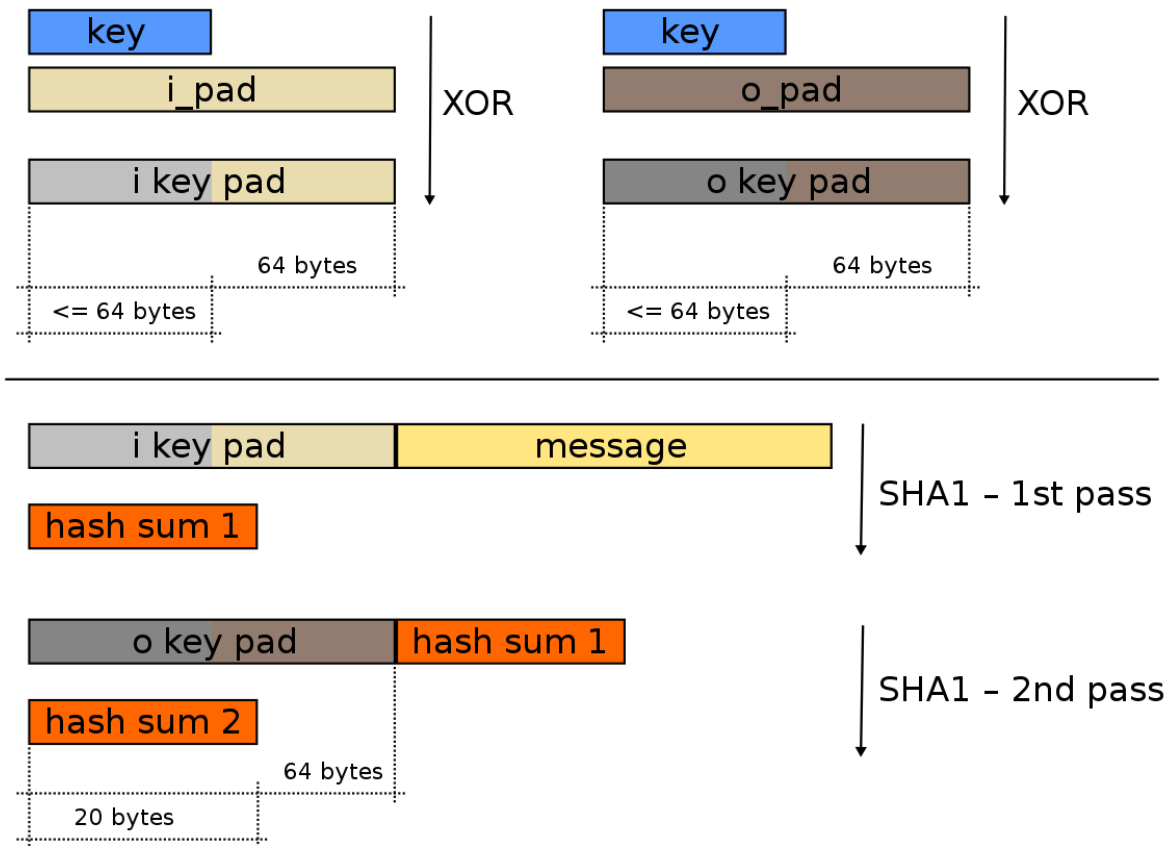


Figure 3. Padding in HMAC [7]

Procedure

1. Lab Environment

We used the domain www.seedlab-hashlen.com to host the server program. In the VM, this hostname was mapped to the web server container (10.9.0.80). This has been achieved by adding the entry: 10.9.0.80 www.seedlab-hashlen.com to the /etc/hosts file. Figure 4 shows the process.

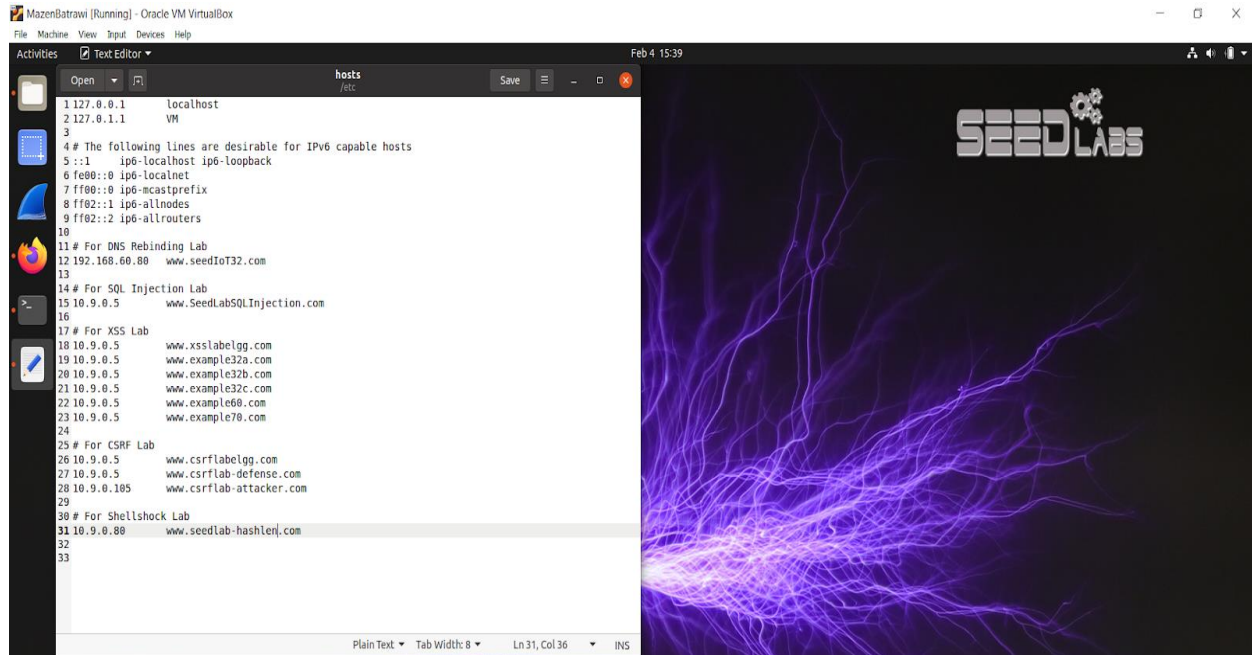
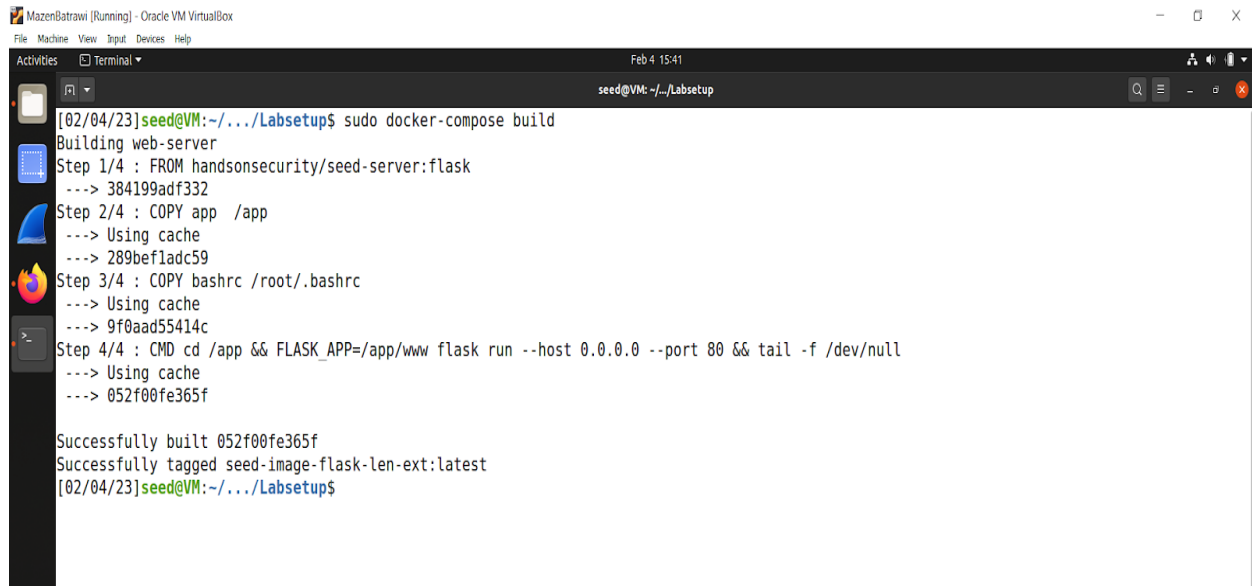


Figure 4. Adding Entry to the /etc/hosts file

In order to build the container image, the following command is used;

Command: \$ docker-compose build # Build the container image

Building the container process is shown in figure 5 below:

A screenshot of a terminal window titled 'MazenBatrawi [Running] - Oracle VM VirtualBox'. The terminal shows the command 'sudo docker-compose build' being executed. The output indicates the building of a 'web-server' image. It shows four steps: 1/4 FROM handsongsecurity/seed-server:flask, 2/4 COPY app /app, 3/4 COPY bashrc /root/.bashrc, and 4/4 CMD cd /app && FLASK_APP=/app/www flask run --host 0.0.0.0 --port 80 && tail -f /dev/null. Each step shows a cache hit. The final output is 'Successfully built 052f00fe365f' and 'Successfully tagged seed-image-flask-len-ext:latest'. The prompt returns to the user's shell.

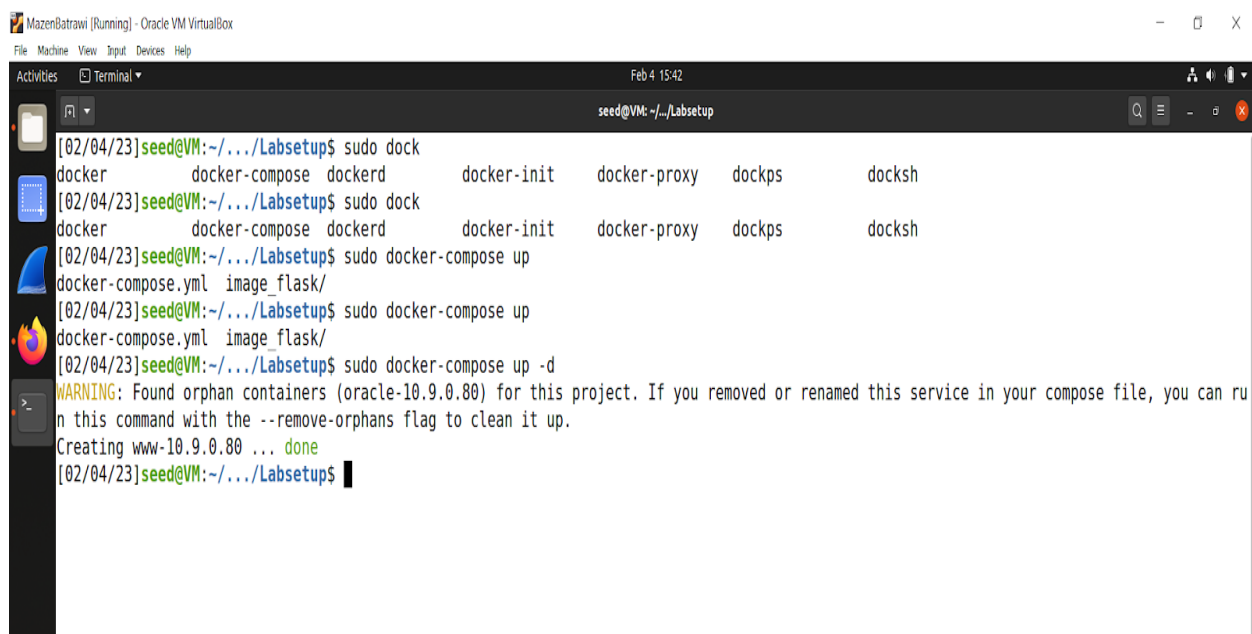
```
[02/04/23]seed@VM:~/../Labsetup$ sudo docker-compose build
Building web-server
Step 1/4 : FROM handsongsecurity/seed-server:flask
----> 384199adf332
Step 2/4 : COPY app /app
----> Using cache
----> 289bef1adc59
Step 3/4 : COPY bashrc /root/.bashrc
----> Using cache
----> 9f0aad55414c
Step 4/4 : CMD cd /app && FLASK_APP=/app/www flask run --host 0.0.0.0 --port 80 && tail -f /dev/null
----> Using cache
----> 052f00fe365f

Successfully built 052f00fe365f
Successfully tagged seed-image-flask-len-ext:latest
[02/04/23]seed@VM:~/../Labsetup$
```

Figure 5. Building the container image

After building the container, in order to start it, the following command is used;

\$ docker-compose up # Start the container

A screenshot of a terminal window titled 'MazenBatrawi [Running] - Oracle VM VirtualBox'. The terminal shows the command 'sudo docker-compose up' being executed. The output lists the services being started: docker, docker-compose, dockerd, docker-init, docker-proxy, dockps, and docksh. It then shows the command 'sudo docker-compose up' being executed again, followed by 'docker-compose.yml image_flask/'. The output shows 'WARNING: Found orphan containers (oracle-10.9.0.80) for this project. If you removed or renamed this service in your compose file, you can run this command with the --remove-orphans flag to clean it up.' and 'Creating www-10.9.0.80 ... done'. The prompt returns to the user's shell.

```
[02/04/23]seed@VM:~/../Labsetup$ sudo dock
docker      docker-compose dockerd      docker-init  docker-proxy  dockps      docksh
[02/04/23]seed@VM:~/../Labsetup$ sudo dock
docker      docker-compose dockerd      docker-init  docker-proxy  dockps      docksh
[02/04/23]seed@VM:~/../Labsetup$ sudo docker-compose up
docker-compose.yml image_flask/
[02/04/23]seed@VM:~/../Labsetup$ sudo docker-compose up
docker-compose.yml image_flask/
[02/04/23]seed@VM:~/../Labsetup$ sudo docker-compose up -d
WARNING: Found orphan containers (oracle-10.9.0.80) for this project. If you removed or renamed this service in your compose file, you can run
this command with the --remove-orphans flag to clean it up.
Creating www-10.9.0.80 ... done
[02/04/23]seed@VM:~/../Labsetup$
```

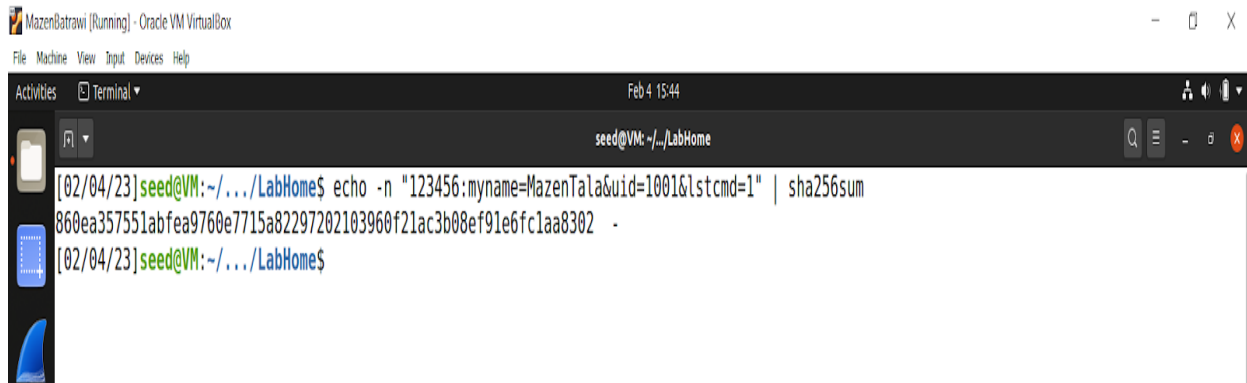
Figure 6. Starting the container

2. Tasks

a. Task 1 – Send Request to List Files

In this task, we have sent a benign request to the server in order to see how the server responds to the request.

We have used a uid of 1001, a key of 123456 and our names Mazen and Tala were used, too. The MAC address was calculated. Figure 7 that is shown below shows the process;



```
Mazen@atrawi [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal Feb 4 15:44
seed@VM: ~/../LabHome
[02/04/23]seed@VM:~/../LabHome$ echo -n "123456:myname=MazenTala&uid=1001&lstdcmd=1" | sha256sum
860ea357551abfea9760e7715a82297202103960f21ac3b08ef91e6fc1aa8302 -
[02/04/23]seed@VM:~/../LabHome$
```

Figure 7. Calculating MAC

The output of the command we got is as follows;

860ea357551abfea9760e7715a82297202103960f21ac3b08ef91e6fc1aa8302

The URL that we have generated for the output command that is shown above is;

URL:

<http://www.seedlab-hashlen.com/?myname=MazenTala&uid=1001&lstdcmd=1&mac=860ea357551abfea9760e7715a82297202103960f21ac3b08ef91e6fc1aa8302>

We accessed the URL using the Firefox. Figure 8 shows the result we got;

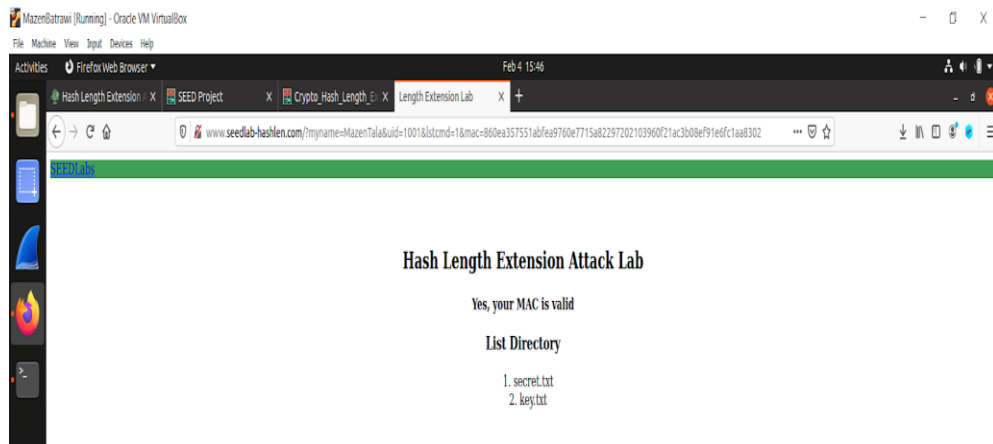


Figure 8. Opening the URL in Firefox Result

As shown in Figure 8 above, the MAC for the request sent was valid. The server responded and sent a list request as the request response.

For the second part, in order to download the secret.txt file and send a download command to the server, the MAC was calculated, first.

We have used a uid of 1001, a key of 123456 and our names Mazen and Tala were used, too.

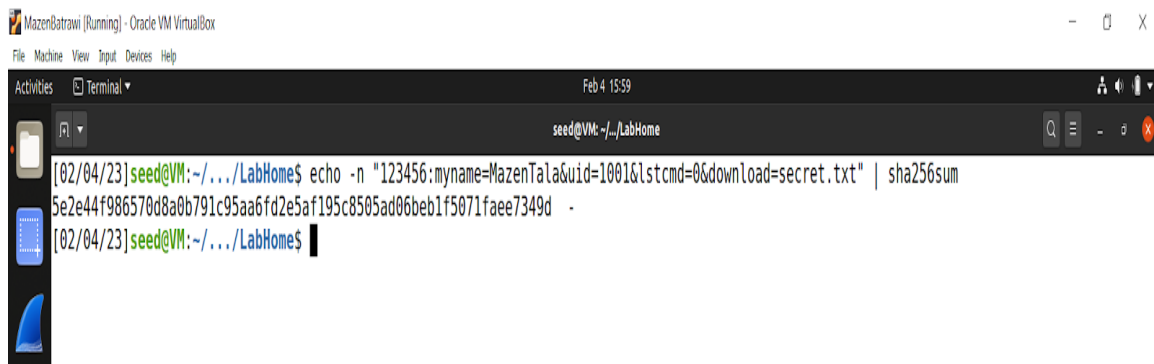


Figure 9. Downloading MAC

The output of the command we got is as follows;

5e2e44f986570d8a0b791c95aa6fd2e5af195c8505ad06beb1f5071faee7349d

The URL that we have generated for the output command that is shown above is;

URL: <http://www.seedlab-hashlen.com/?myname=MazenTala&uid=1001&lscmd=0&download=secret.txt&mac=5e2e44f986570d8a0b791c95aa6fd2e5af195c8505ad06beb1f5071face7349d>

We accessed the URL using the Firefox. Figure 10 shows the result we got;

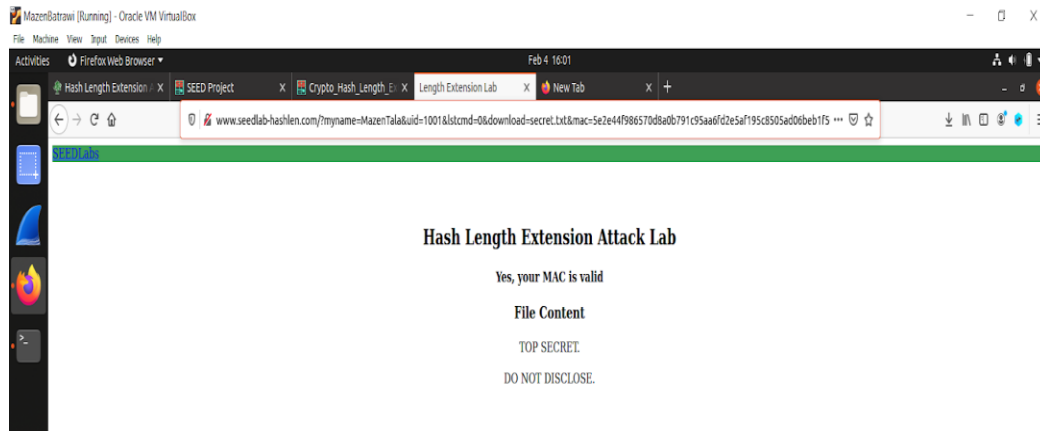


Figure 10. Opening the URL in Firefox Download Result

As shown in Figure 10 above, the MAC for the request sent was valid. The server responded and sent a download request as the request response.

b. Task 2 – Create Padding

We have developed a Python program in order to perform padding for messages.

Here is the algorithm performed for padding;

Paddings for SHA256 consist of;

- a. One byte of x80
- b. Number of 0's
- c. 64-bit (8-byte) length field (the length is the number of bits in the M)

The message that was sent is as follows;

123456:myname=MazenTala&uid=1001&lscmd=1.

The padding code that is written is shown in Figure 11 below.



Figure 12. Padding Result

The padding code we have written is appended in the Appendix section below.

c. Task 3 – The Length Extension Attack

The objective of this task is to generate a counterfeit request based on a known valid request R and its MAC, without having access to the MAC key. It is assumed that both the size of the MAC key and the MAC of R are known. The goal is to produce a fake request that can produce a valid MAC.

The MAC was generated using the following message:

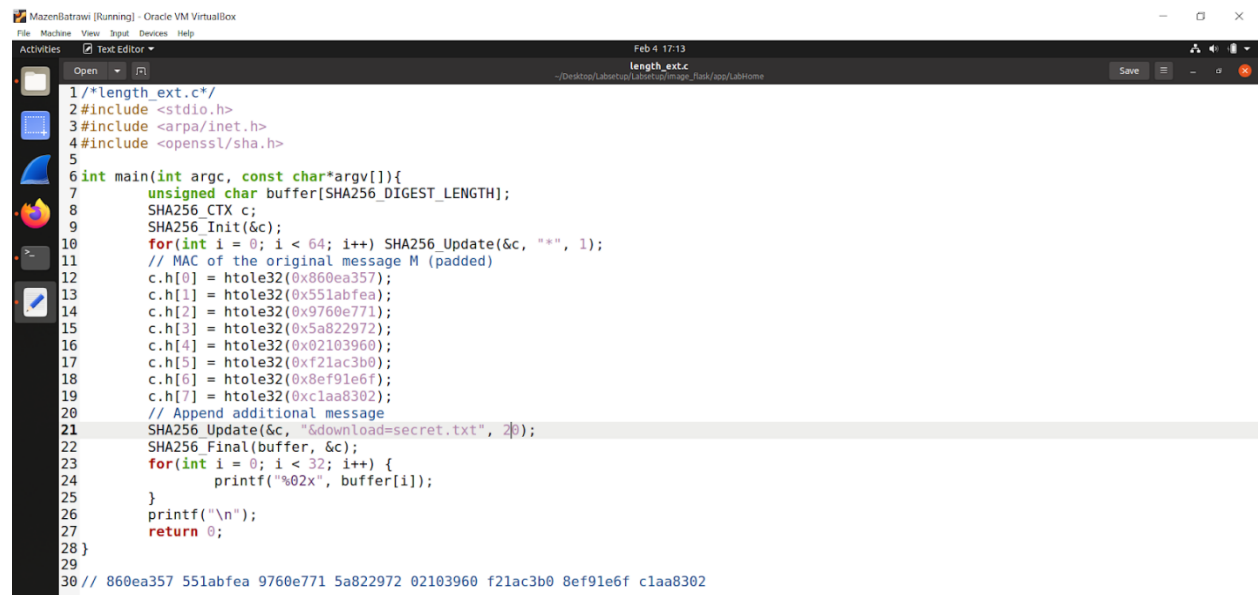
123456:myname=MazenTala&uid=1001&lscmd=1

The padding was generated as in task 2

%80%01%48

Then, the new MAC was generated using the original MAC as shown in the code in Figure 13. The original MAC was;

860ea357551abfea9760e7715a82297202103960f21ac3b08ef91e6fc1aa8302



```
1 /*length_ext.c*/
2 #include <stdio.h>
3 #include <arpa/inet.h>
4 #include <openssl/sha.h>
5
6 int main(int argc, const char*argv[]){
7     unsigned char buffer[SHA256_DIGEST_LENGTH];
8     SHA256_CTX c;
9     SHA256_Init(&c);
10    for(int i = 0; i < 64; i++) SHA256_Update(&c, "", 1);
11    // MAC of the original message M (padded)
12    c.h[0] = htobe32(0x860ea357);
13    c.h[1] = htobe32(0x551abfea);
14    c.h[2] = htobe32(0x9760e771);
15    c.h[3] = htobe32(0x5a822972);
16    c.h[4] = htobe32(0x02103960);
17    c.h[5] = htobe32(0xf21ac3b0);
18    c.h[6] = htobe32(0x8ef91e6f);
19    c.h[7] = htobe32(0xc1aa8302);
20    // Append additional message
21    SHA256_Update(&c, "download=secret.txt", 16);
22    SHA256_Final(buffer, &c);
23    for(int i = 0; i < 32; i++) {
24        printf("%02x", buffer[i]);
25    }
26    printf("\n");
27    return 0;
28 }
29
30 // 860ea357 551abfea 9760e771 5a822972 02103960 f21ac3b0 8ef91e6f c1aa8302
```

Figure 13. Task 3 code

The new mac is: 528575c6660b7f068da0b981f9dae958263ceab830d6c1acc3bd971ae9853fcc, as shown in Figure 14.

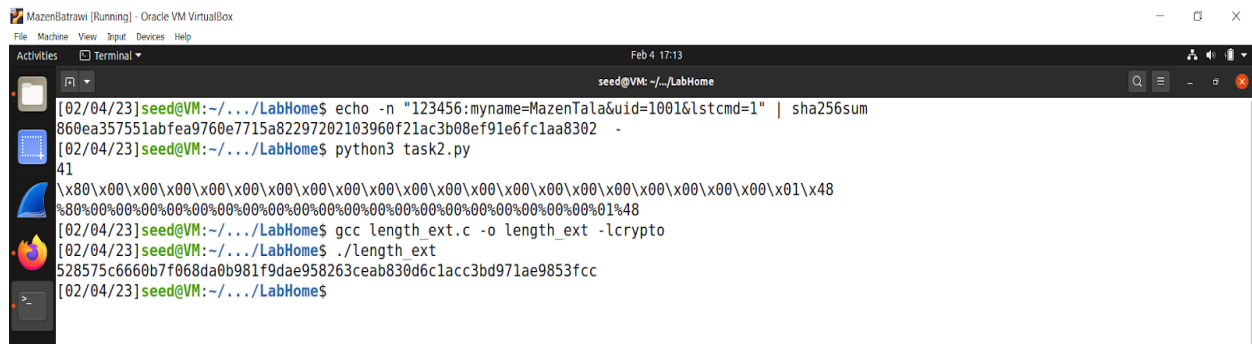


Figure 14. Task 3 original MAC + new MAC

Then, the URL was opened using the new MAC.

The URL is:

http://www.seedlab-hashlen.com/?myname=MazenTala&uid=1001&lscmd=1%80%01%48&download=secret.txt&mac=528575c6660b7f068da0b981f9dae958263ceab830d6c1acc3bd971ae9853fcc

The URL was opened using Firefox, as shown in Figure 15.

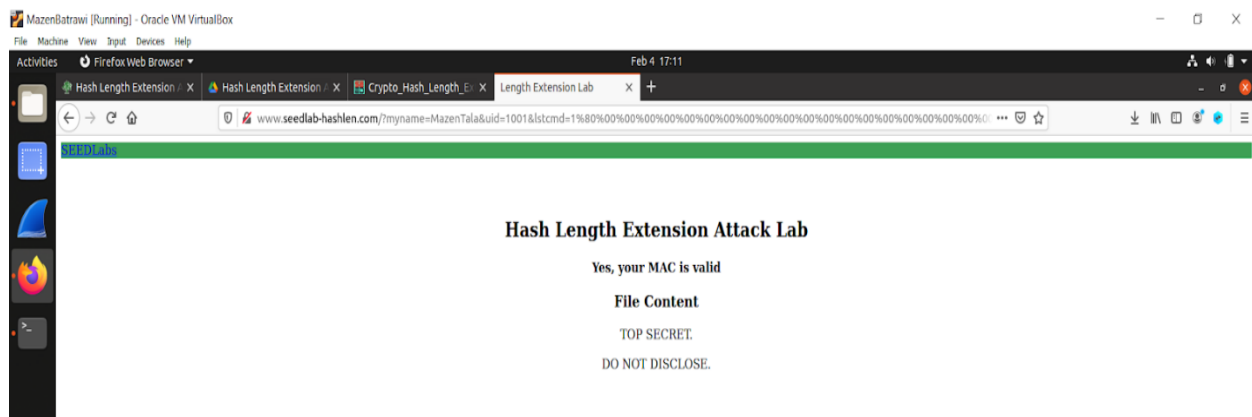


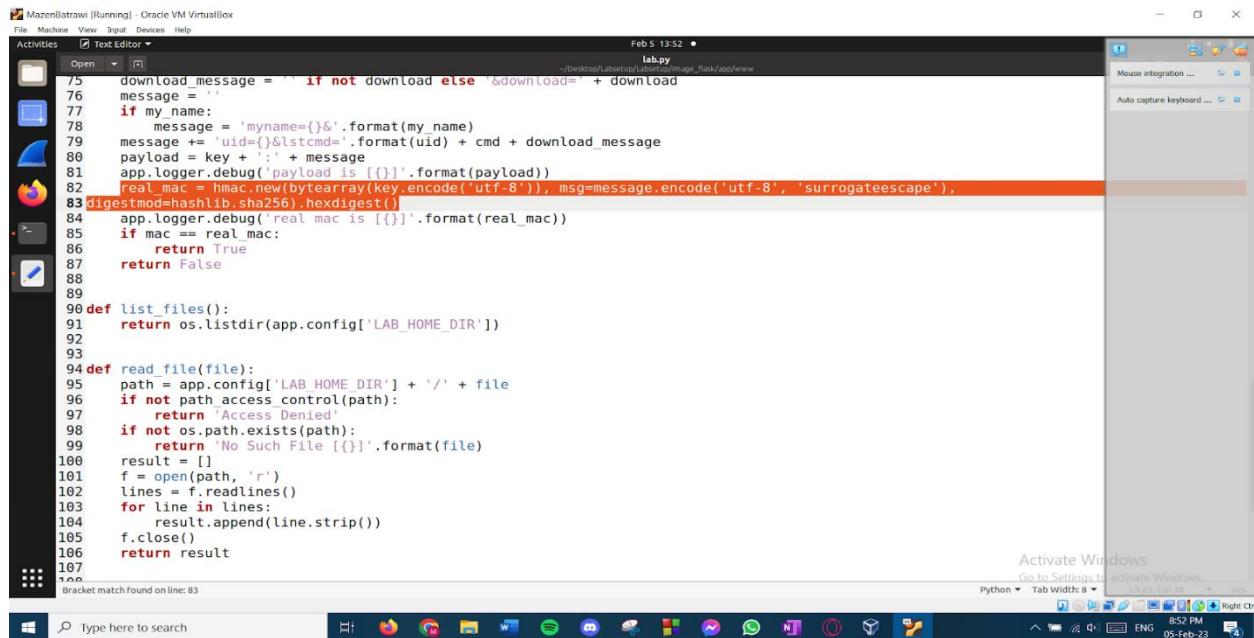
Figure 15. Task 3 result

A valid MAC was created for a URL in the absence of the MAC key. On the basis of R, a fake request was created that allowed for the calculation of a legitimate MAC. This was made achievable by knowing the size of the MAC key as well as the MAC of a valid request R.

This task's code will be appended in the Appendix section below.

d. Task 4 – Attack Mitigation using HMAC

So far, the dangerous consequences of a developer improperly computing a MAC by simply concatenating the key and message have been observed in previous jobs. This job aims to rectify this mistake. The standard formula used to compute MACs is HMAC. The `verify_mac()` method of the server program must be altered and the MAC calculation will be performed using the HMAC package in Python. The function can be found in the `lab.py` file. By providing a key and message of string type, the HMAC can be calculated using the following method as shown in Figure 16.



The screenshot shows a code editor window titled 'lab.py' with the following Python code:

```
75 download_message = 'if not download else &download=' + download
76 message =
77 if my_name:
78     message = 'myname={}&'.format(my_name)
79 message += 'uid={}&lscmd='.format(uid) + cmd + download_message
80 payload = key + ':' + message
81 app.logger.debug('payload is {}'.format(payload))
82 real_mac = hmac.new(bytearray(key.encode('utf-8')), msg=message.encode('utf-8', 'surrogateescape'),
83 digestmod=hashlib.sha256).hexdigest()
84 app.logger.debug('real mac is {}'.format(real_mac))
85 if mac == real_mac:
86     return True
87 return False
88
89
90 def list_files():
91     return os.listdir(app.config['LAB_HOME_DIR'])
92
93
94 def read_file(file):
95     path = app.config['LAB_HOME_DIR'] + '/' + file
96     if not path.access_control(path):
97         return 'Access Denied'
98     if not os.path.exists(path):
99         return 'No Such File {}'.format(file)
100 result = []
101 f = open(path, 'r')
102 lines = f.readlines()
103 for line in lines:
104     result.append(line.strip())
105 f.close()
106 return result
107
108
```

Figure 16. Editing the lab.py code

After these changes, the docker was shut down and then built and brought up as in section 1 in the procedure.

A MAC was generated using the following message using the code in Figure 17:

myname=DweikatBatrawi&uid=1001&lscmd=1

A screenshot of a text editor window titled 'task4.py' within an Oracle VM VirtualBox environment. The code is a Python script that uses the hmac and hashlib libraries to generate a MAC. It defines a hashlibkey, a message, and calculates a mac using hmac.new and hashlib.sha256. The final output is printed.

```
1#!/bin/env python3
2import hmac
3import hashlib
4
5hashlibkey='123456'
6message='myname=DweikatBatrawi&uid=1001&lscmd=1'
7mac = hmac.new(bytearray(hashlibkey.encode('utf-8')),
8msg=message.encode('utf-8', 'surrogateescape'),
9digestmod=hashlib.sha256).hexdigest()
10print(mac)
```

Figure 17. Task 4 first message code

As shown in Figure 18;

the generated MAC was:

ad23f30516aa63d28d55391887e732642786033c0163520dd12d178b1ef921bf

A screenshot of a terminal window showing the execution of the task4.py script. The user runs 'gedit task4.py' and then 'python3 task4.py', which outputs the generated MAC.

```
[02/04/23]seed@VM:~/.../www$ gedit task4.py
[02/04/23]seed@VM:~/.../www$ python3 task4.py
ad23f30516aa63d28d55391887e732642786033c0163520dd12d178b1ef921bf
[02/04/23]seed@VM:~/.../www$
```

Figure 18. Task 4 first message MAC

The URL is:

<http://www.seedlab-hashlen.com/?myname=DweikatBatrawi&uid=1001&lscmd=1&mac=ad23f30516aa63d28d55391887e732642786033c0163520dd12d178b1ef921bf>

The URL was opened as in Figure 19.

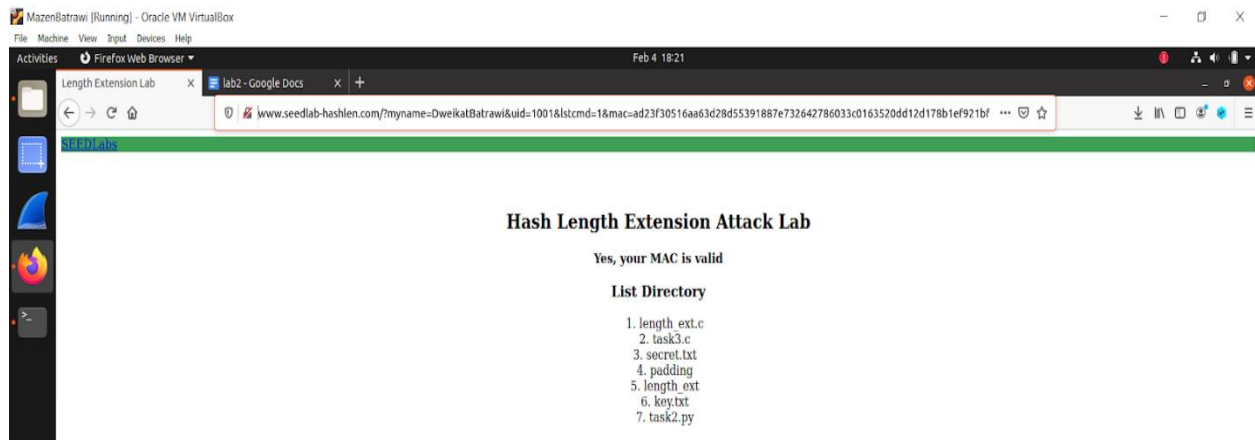


Figure 19. Task 4 first message result

Then, a MAC for download was generated after editing the message in the code, as shown in Figure 20.

The new message is as follows;

myname=DweikatBatrawi&uid=1001&lscmd=1&download=secret.txt



Figure 20. Task 4 second message code

As shown in Figure 21;

the generated MAC was:

9651b270a209b458fa4df7901a0ba3032e2d54d9ad88bcd77d858f4b7df5415



Figure 21. Task 4 second message MAC

The URL is:

<http://www.seedlab-hashlen.com/?myname=DweikatBatrawi&uid=1001&lscmd=1&download=secret.txt&mac=9651b270a209b458fa4df7901a0ba3032e2d54d9ad88bcd77d858f4b7df5415>

The URL was opened as in Figure 22.

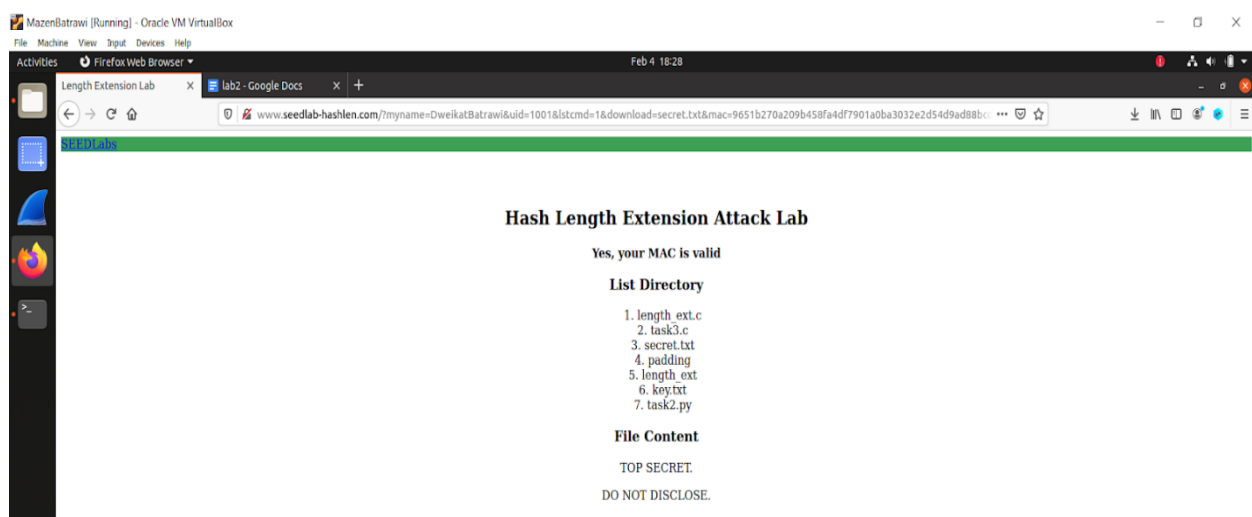


Figure 22. Task 4 second message result

It should be noted that the directory has the following files:

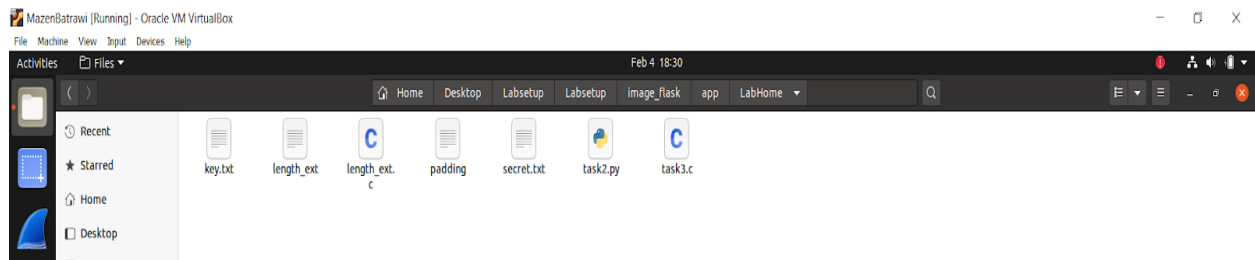


Figure 23. The directory files

This task's code will be appended in the Appendix section below.

Now a new request using SHA256 will be sent.

The MAC will be generated as shown in figure 24.

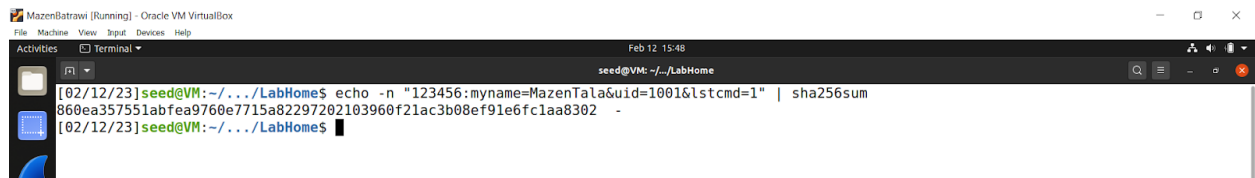


Figure 24. Generating SHA256 MAC

The URL is: <http://www.seedlab-hashlen.com/?myname=MazenTala&uid=1001&lscmd=1&mac=860ea357551abfea9760e7715a82297202103960f21ac3b08ef91e6fc1aa8302>

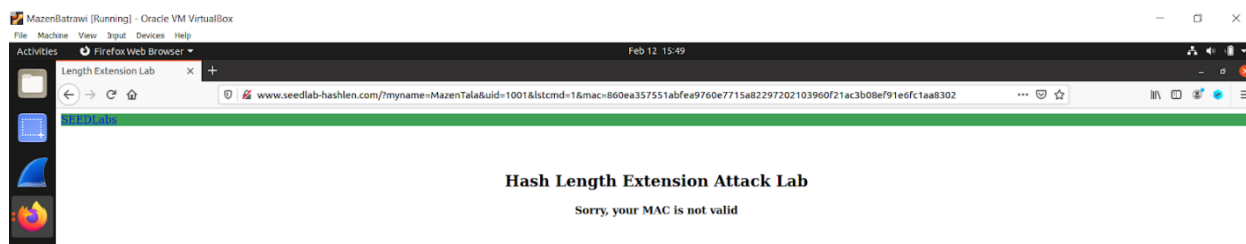


Figure 25. SHA256 Response Due to HMAC

The request was sent using Firefox and got the result shown in figure 25. In Figure 25, the server response indicates that the MAC is invalid due to the usage of HMAC. To evaluate the impact of a Hash Length Extension attack, the original MAC and padding were obtained, as demonstrated in Figure 26 below.



Figure 26. Getting the padding and the MAC

[illegible]

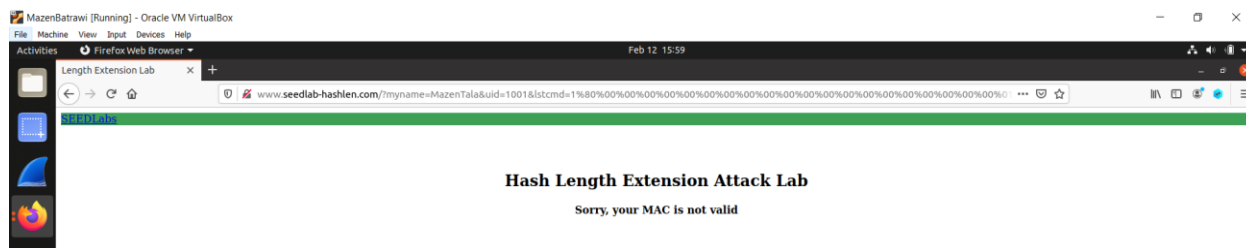


Figure 27. Hash Length Extension Attack Failure

As depicted in Figure 27, the attempt to carry out a Hash Length Extension attack was unsuccessful due to the use of HMAC, which provides security against such attacks and verifies the integrity of the data.

Conclusion

Finally, this task gave us practical experience with length extension attacks and how they may be used to tamper with message authentication codes (MACs) produced by one-way hashes. We now understand the many phases of the attack, from making requests to list files to using the length extension attack to take advantage of the vulnerability. Additionally, we discovered how to use HMAC to lessen such attacks. SEED Ubuntu was utilized to complete this assignment. It is intended to make it easier to comprehend the practical effects of security flaws in cryptographic systems.

References

- [1] https://en.wikipedia.org/wiki/Length_extension_attack
- [2] <https://deeprnd.medium.com/length-extension-attack-bff5b1ad2f70>
- [3] <https://www.techtarget.com/searchsecurity/definition/message-authentication-codeMAC>
- [4] <https://www.okta.com/identity-101/hmac/>
- [5] <https://www.jscape.com/blog/what-is-hmac-and-how-does-it-secure-file-transfers>
- [6] <https://cryptography.io/en/latest/hazmat/primitives/padding/>
- [7]: <https://en.wikipedia.org/wiki/HMAC>

Appendix

- **Task 2 Code:**

```
content = bytearray("123456:myname=MazenTala&uid=1001&lstcmd=1", 'utf8')
print(len(content))
lenf = (len(content) * 8).to_bytes(8, 'big')
padding = b'\x80' + b'\x00' * (64 - len(content) - 1 - 8) + lenf
print(''.join('\x{:02x}'.format(x) for x in padding))
print(''.join('%{:02x}'.format(x) for x in padding))
```

- **Task 3 Code:**

```
/*length_ext.c*/
#include <stdio.h>
#include <arpa/inet.h>
#include <openssl/sha.h>

int main(int argc, const char*argv[]){
    unsigned char buffer[SHA256_DIGEST_LENGTH];
    SHA256_CTX c;
    SHA256_Init(&c);
    for(int i = 0; i < 64; i++) SHA256_Update(&c, "*", 1);
    // MAC of the original message M (padded)
    c.h[0] = htobe32(0x860ea357);
    c.h[1] = htobe32(0x551abfea);
    c.h[2] = htobe32(0x9760e771);
    c.h[3] = htobe32(0x5a822972);
    c.h[4] = htobe32(0x02103960);
    c.h[5] = htobe32(0xf21ac3b0);
    c.h[6] = htobe32(0x8ef91e6f);
    c.h[7] = htobe32(0xc1aa8302);
    // Append additional message
    SHA256_Update(&c, "&download=secret.txt", 20);
    SHA256_Final(buffer, &c);
    for(int i = 0; i < 32; i++) {
        printf("%02x", buffer[i]);
    }
    printf("\n");
    return 0;
}
```

// 860ea357 551abfea 9760e771 5a822972 02103960 f21ac3b0 8ef91e6f c1aa8302

- **Task 4 – First Message Code:**

```
#!/bin/env python3
import hmac
import hashlib

hashlibkey='123456'
message='myname=DweikatBatrawi&uid=1001&lstcmd=1'
mac = hmac.new(bytearray(hashlibkey.encode('utf-8')),
msg=message.encode('utf-8', 'surrogateescape'),
digestmod=hashlib.sha256).hexdigest()
print(mac)
```

- **Task 4 – Second Message Code:**

```
#!/bin/env python3
import hmac
import hashlib

hashlibkey='123456'
message='myname=DweikatBatrawi&uid=1001&lstcmd=1&download=secret.txt'
mac = hmac.new(bytearray(hashlibkey.encode('utf-8')),
msg=message.encode('utf-8', 'surrogateescape'),
digestmod=hashlib.sha256).hexdigest()
print(mac)
```