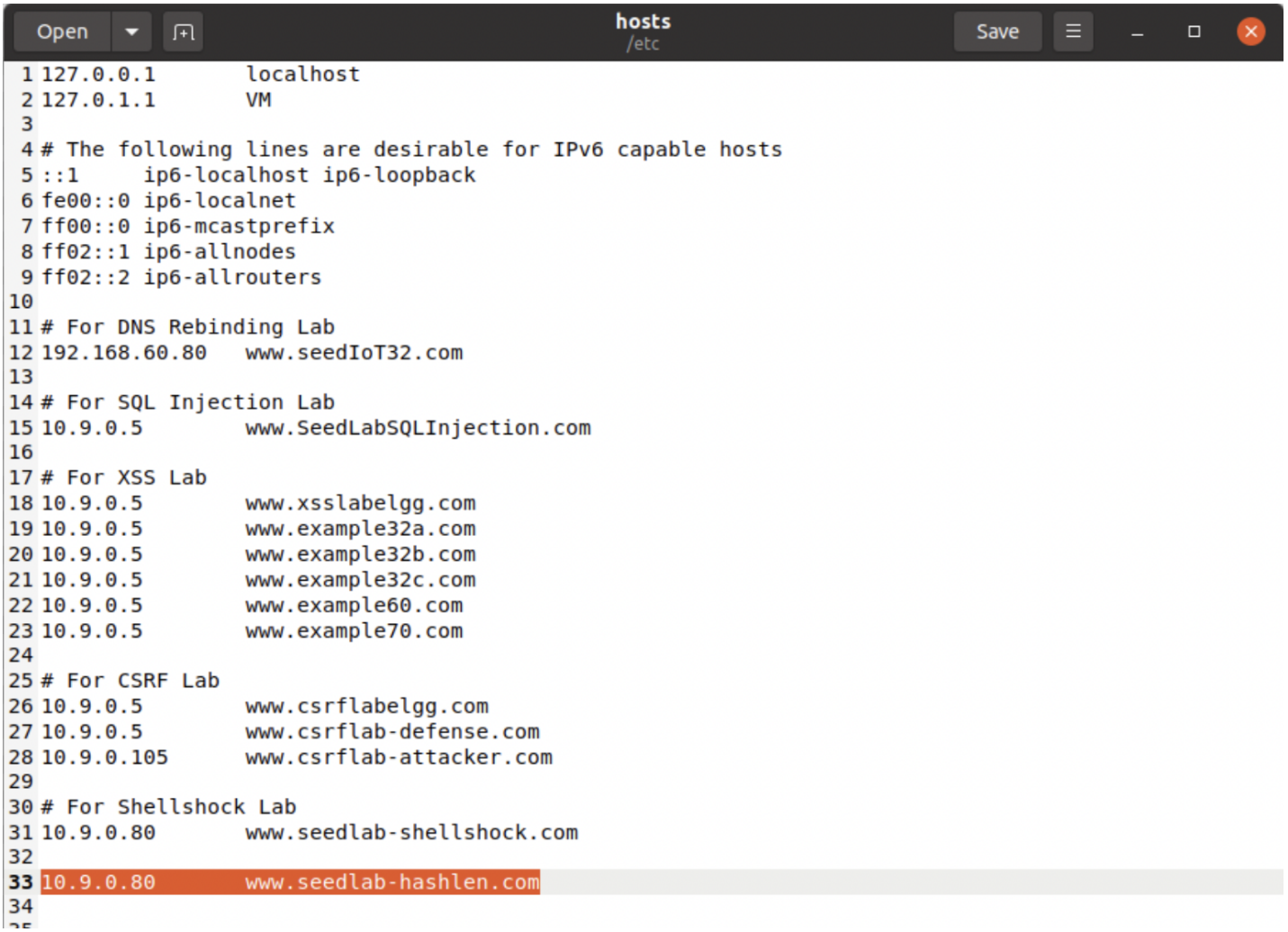
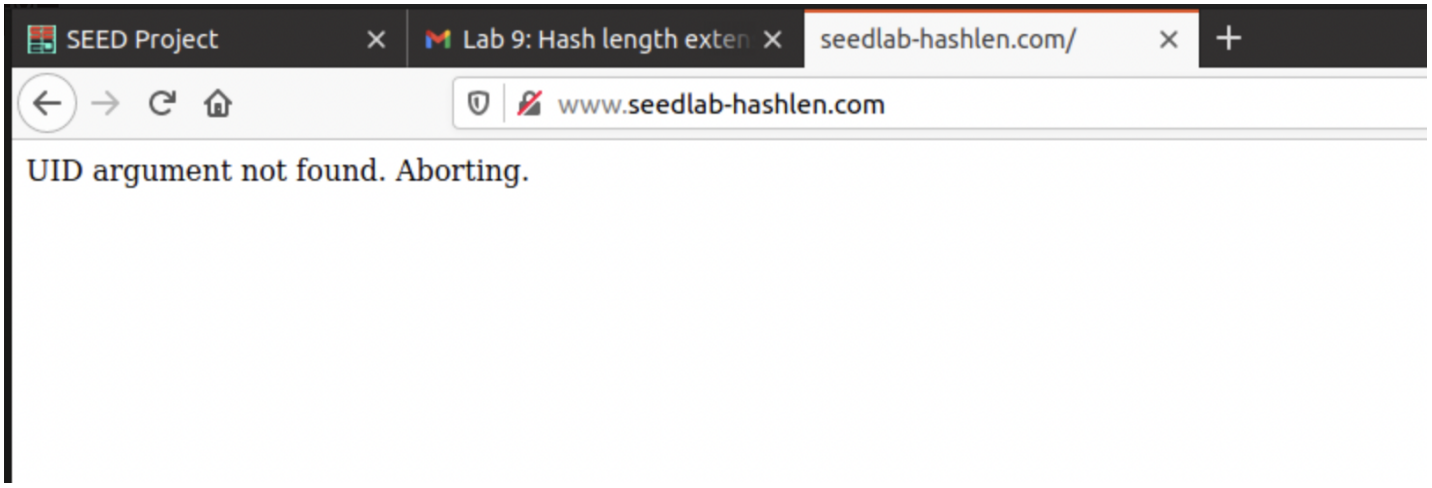
Applied Cryptography Lab-09 Manual

Submitted by: Mahika Gupta
SRN: PES1UG20CS243

Date: 26/11/2022

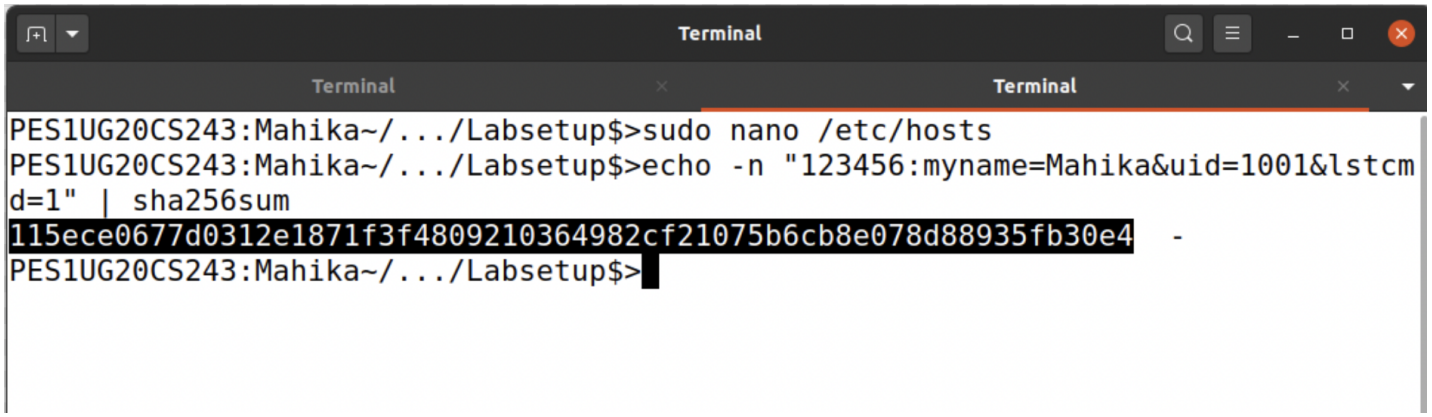
Step 1: Unzip Labsetup.zip file to your working directory.
open a terminal in the folder Labsetup and run the following commands:
$ docker-compose build
$ docker-compose up
Step 2: Open /etc/hosts as root in a text editor of your choice and append the following line:
10.9.0.80 www.seedlab-hashlen.com



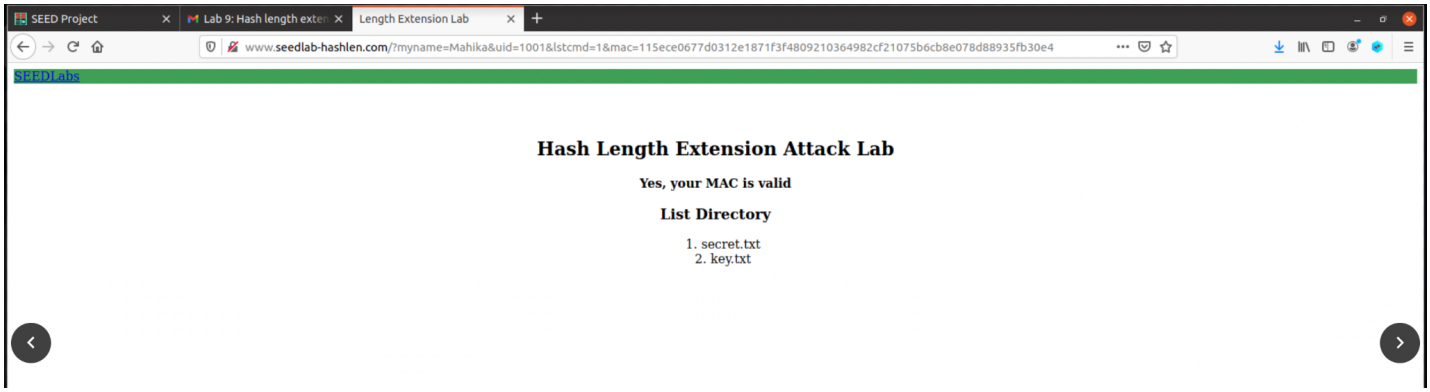


Task 1: Send Request to List Files

In this task, we send a benign request to the server so we can see how the server responds to the request. The request we want to send is as follows:
http://www.seedlab-hashlen.com/?myname=<name_or_srn>&uid=<uid>&lstcmd=1
&mac=<need-to-calculate>
Step 1: Finding the uid
Go to Labsetup/image_flask/app/LabHome and open key.txt
This file contains multiple uid:key pairs. Choose one and use that uid in the request.
Step 2: Calculating mac
Command:
$ echo -n "<key>:myname=<name_or_srn>&uid=<uid>&lstcmd=1" | sha256sum



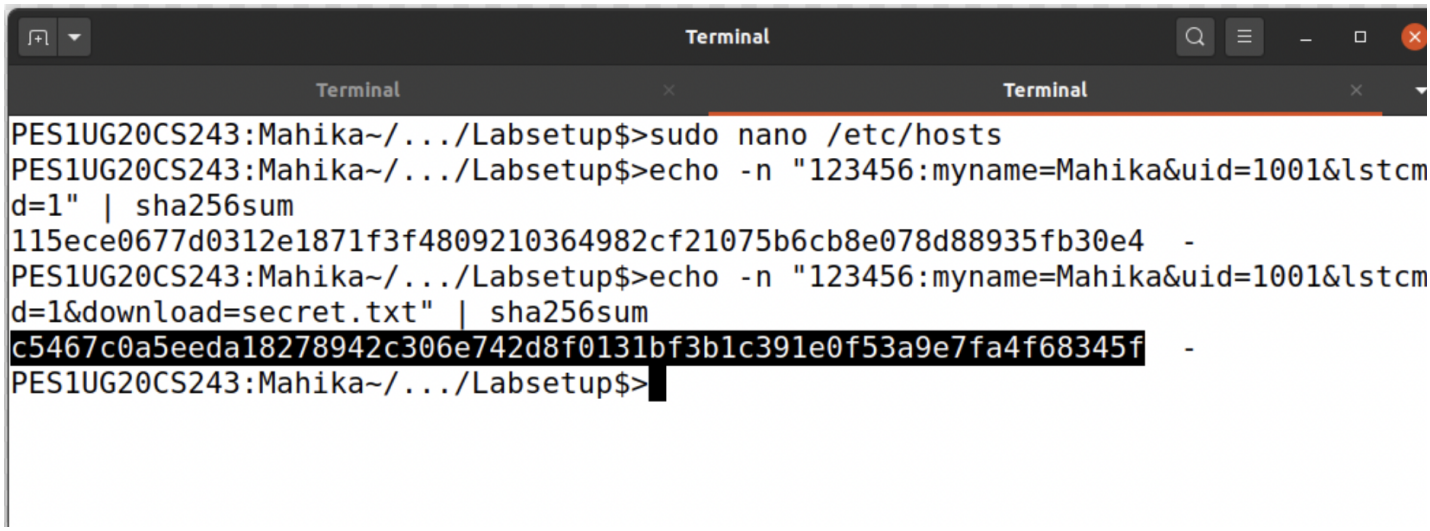
Step 3: Sending the request
Fill in the uid and mac in the request and paste it in your browser. Note that cURL or wget will not work.

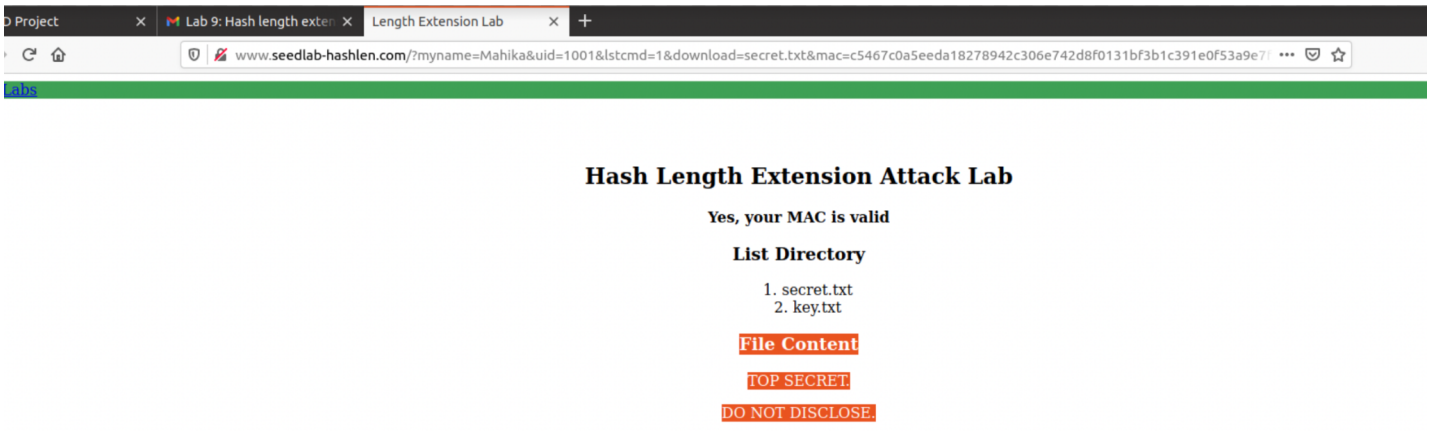


After entering the valid mac along with the uid the contents are visible.

Repeat steps 1, 2 and 3 for the request http://www.seedlab-hashlen.com/?

myname=<name_or_srn>&uid=<uid>&lstcmd=1&&download=secret.txt&mac=<need-to-calculate>





The contents of the file secret.txt are visible after using the correct mac.

Task 2: Create Padding

To conduct the hash length extension attack, we need to understand how padding is calculated for a one-way hash. 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 many 0's, followed by a 64-bit (8 bytes) length field (the length is the number of bits in 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 terms of bits is 22*8 = 176 = 0×B0.

We generate padding using the following script:

```
payload = bytearray("<key>:myname=<name>&uid=<uid>&lstcmd=1",
"utf8") length_field = (len(payload) * 8).to_bytes(8, "big")
padding = b"\x80" + b"\x00" * (64 - len(payload) - 1 - 8) +
length_field print("".join("\\x{:02x}".format(x) for x in padding))
# for url-encoding
print("".join("%{:02x}".format(x) for x in padding))
```

Note down the paddings generated
Give screenshots of your output with observation

```
PES1UG20CS243:Mahika~/.../Labsetup$>python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> payload = bytearray("123456:myname=Mahika&uid=1001&lstcmd=1","utf-8")
>>> length_field=(len(payload)*8).to_bytes(8,'big')
>>> padding = b"\x80" + b"\x00" * (64 - len(payload) - 1 - 8) + length_field
>>> print("".join("\\x{:02x}".format(x) for x in padding))
\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x01\x30
>>> print("".join("%{:02x}".format(x) for x in padding))
%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%01%30
>>>
```

Here we can see the hex and url encoded padding generated.

Task 3: The Length Extension Attack

In this task, we will generate a valid MAC for a URL without knowing the MAC key. Assume that we know the MAC of a valid request R, and we also know the size of the MAC key. Our job is to forge a new request based on R, while still being able to compute the valid MAC.

Step 1: Substitute in the required values, then compile and run the following code

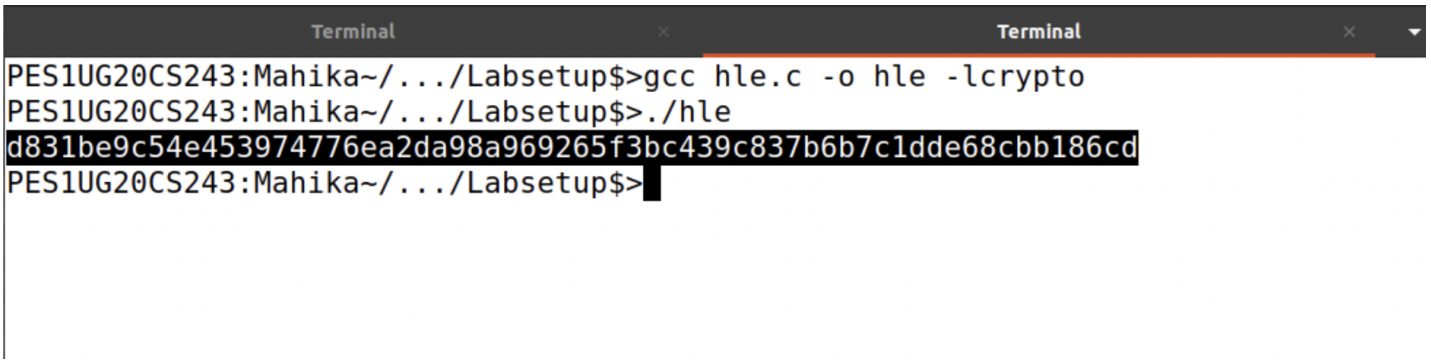
```c
#include <stdio.h>
#include <openssl/sha.h>
int main(int argc, const char *argv[])
{
    SHA256_CTX c;
    unsigned char buffer[SHA256_DIGEST_LENGTH];
    int i;
    SHA256_Init(&c);
    SHA256_Update(&c,
                  "123456:myname=<name>&uid=<uid>&lstcmd=1<padding_generated_in_task
_ 2>"
                  "&download=secret.txt",
```



```c
                  "&download=secret.txt",
                  64 + 20);
    SHA256_Final(buffer, &c);
    for (i = 0; i < 32; i++)
    {
        printf("%02x", buffer[i]);
    }
    printf("\n");
    return 0;
}
```

Commands

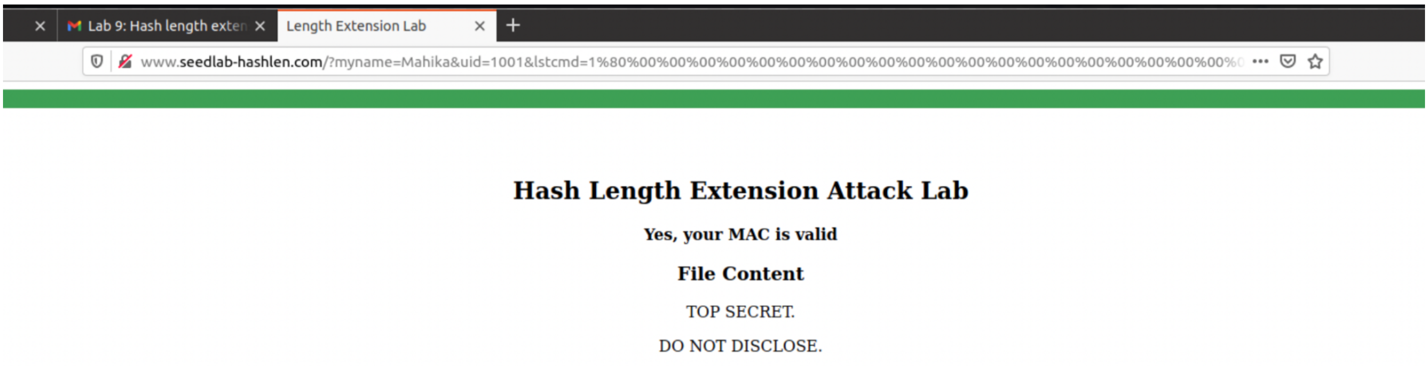
```
gcc calculate_mac.c -o calculate_mac -lcrypto
./calculate_mac
```



Note down the hash generated

Step 2: Visit the url

Format the attack URL as follows:
http://www.seedlab-hashlen.com/?myname=<name>&uid=<uid>&lstcmd=<url_encoded_padding>&download=secret.txt&



After using the right mac, we can see that it is valid and contents of the file are displayed.

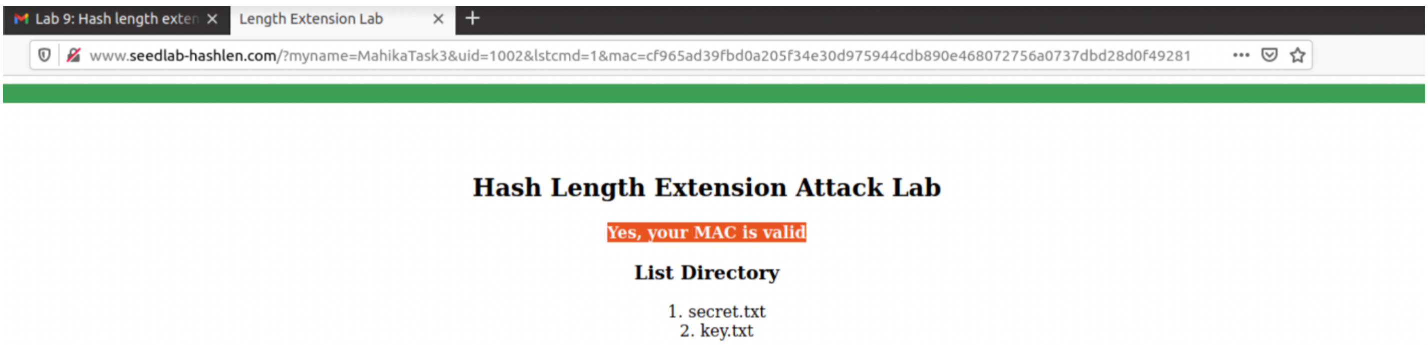
Step 3: Perform Hash Length Extension Attack without the knowledge of the key

1. Choose an alternate uid:key pair (not the one you've been using so far)

```
PES1UG20CS243:Mahika~/.../Labsetup$>echo -n "983abe:myname=MahikaTask3&uid=1002&lstcmd=1" | sha256sum
cf965ad39fbd0a205f34e30d975944cdb890e468072756a0737dbd28d0f49281  -
PES1UG20CS243:Mahika~/.../Labsetup$>
```

We use a different uid key pair : 1002:9831be

2. Generate a legitimate request to list files using this uid:key pair (task 1). Note the URL down.



We can see that the mac is valid

We now know the uid, command and the hash. We do not know the key, which is required to generate new mac in case of a new command. The attack involves attempting to run a different command (viewing the contents of secret.txt) without knowing the key

Step 1: Run the following code after changing the parameters marked in <>

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <openssl/sha.h>
#include <string.h>
int main(int argc, const char *argv[])
{
    int i;
    unsigned char buffer[SHA256_DIGEST_LENGTH];
    SHA256_CTX c;
    char hex[] = "< mac_in_new_request >";
    char subbuffer[9];
    SHA256_Init(&c);
    for (i = 0; i < 64; i++)
        SHA256_Update(&c, "*", 1);
    // MAC of the original message M (padded)
    for (i = 0; i < 8; i++)
    {
        strncpy(subbuffer, hex + i * 8, 8);
        subbuffer[8] = '\0';
        c.h[i] = htole32(strtol(subbuffer, NULL, 16));
```



```
        c.h[i] = htole32(strtol(subbuffer, NULL, 16));
    }
    // Append additional message
    SHA256_Update(&c, "&download=secret.txt", 20);
    SHA256_Final(buffer, &c);
    for (i = 0; i < 32; i++)
    {
        printf("%02x", buffer[i]);
    }
    printf("\n");
    return 0;
}
```

Note down the new mac.

*Untitled Document 1 | key.txt | hle.c | len_ext.c

```
3 #include <arpa/inet.h>
4 #include <openssl/sha.h>
5 #include <string.h>
6 int main(int argc, const char *argv[])
7 {
8         int i;
9         unsigned char buffer[SHA256_DIGEST_LENGTH];
10        SHA256_CTX c;
11        char hex[] = "cf965ad39fbd0a205f34e30d975944cdb890e468072756a0737dbd28d0f49281";
12        char subbuffer[9];
13        SHA256_Init(&c);
14        for (i = 0; i < 64; i++)
15        SHA256_Update(&c, "*", 1);
16        // MAC of the original message M (padded)
17        for (i = 0; i < 8; i++)
18        {
19        strncpy(subbuffer, hex + i * 8, 8);
20        subbuffer[8] = '\0';
21        c.h[i] = htole32(strtol(subbuffer, NULL, 16));
22        }
23        // Append additional message
24        SHA256_Update(&c, "&download=secret.txt", 20);
25        SHA256_Final(buffer, &c);
26        for (i = 0; i < 32; i++)
27        {
28        printf("%02x", buffer[i]);
29        }
30        printf("\n");
31        return 0;
32 }
```

```
PES1UG20CS243:Mahika~/.../Labsetup$>gcc len_ext.c -o len_ext -lc
PES1UG20CS243:Mahika~/.../Labsetup$>./len_ext
fca5bbe6a9cba5745fbf2b5d0e1ca0cd65037748a2a1946e539f8c568dbcf77e
PES1UG20CS243:Mahika~/.../Labsetup$>
```

Step 2: Use the following script to generate a new padding

```
payload = bytearray("******:myname=<name>&uid=<new_uid>&lstcmd=1",'utf8')
length_field = (len(payload)*8).to_bytes(8,'big')
padding = b'\x80' + b'\x00'*(64-len(payload)-1-8) +
length_field print(''.join('%{:02x}'.format(x) for x in
padding))
```

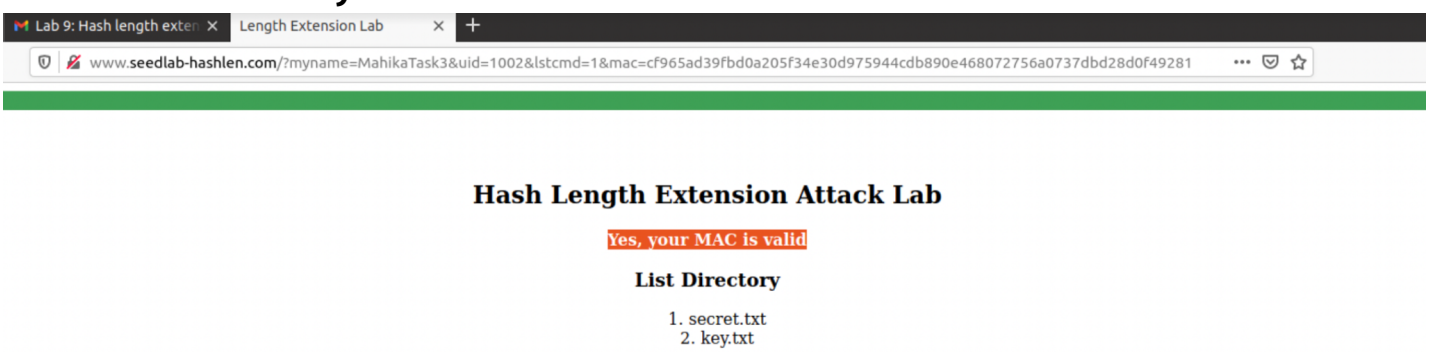
```
PES1UG20CS243:Mahika~/.../Labsetup$>echo -n "983abe:myname=MahikaTask3&uid=1002&
lstcmd=1" | sha256sum
cf965ad39fbd0a205f34e30d975944cdb890e468072756a0737dbd28d0f49281  -
PES1UG20CS243:Mahika~/.../Labsetup$>python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> payload = bytearray("******:myname=MahikaTask3&uid=1002&lstcmd=1","utf-8")
>>> length_field=(len(payload)*8).to_bytes(8,'big')
>>> padding = b"\x80" + b"\x00" * (64 - len(payload) - 1 - 8) + length_field
>>> print("".join("%{:02x}".format(x) for x in padding))
%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%01%58
>>>
```

The padding is generated when the key is unknown.

Step 3: Create a new request using the padding and hash generated in the previous steps

http://www.seedlab-hashlen.com/?myname=<name>&uid=<new_uid>

Step 4: Visit the above generated URL and provide a screenshot of your observations



The contents are successfully displayed even after using the mac with unknown key value, after using the program to generate the mac, using the old mac.

Task 4: Mitigation Using HMAC

Run the following script and describe why a malicious request using length extension and extra commands will fail MAC verification when the client and server use HMAC.

Script

```
import hmac
import hashlib
key="123456"
message="lstcmd=1"
mac = hmac.new(bytearray(key.encode("utf-8")),
msg=message.encode("utf-8", "surrogateescape"),
digestmod=hashlib.sha256).hexdigest()
print(mac)
```

```
PES1UG20CS243:Mahika~/.../Labsetup$>python3 hmac_mitigation.py
e374b19c9bb95fd3f29007cdf1c8e2edd3a16e769801a1c4417608c47c350d66
PES1UG20CS243:Mahika~/.../Labsetup$>echo -n "lstcmd=1" | openssl dgst -sha256 -hmac "123456"
(stdin)= e374b19c9bb95fd3f29007cdf1c8e2edd3a16e769801a1c4417608c47c350d66
PES1UG20CS243:Mahika~/.../Labsetup$>
```

We can see that the same hash value is generated in both.