Hash Length Extension Attack Lab

The objective of this lab is to understand how the length extension attack works. MAC is computed using a key and message by concatenating them and hashing. While seemingly secure, this method is vulnerable to a length extension attack, enabling attackers to modify messages and generate valid MACs without knowing the key.

Task 1: Send Request to List Files

In the first task, we will start by sending a benign request to the server so we can see how it works and how the server responds.

The format of the request is the following:

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

We will use the following data:

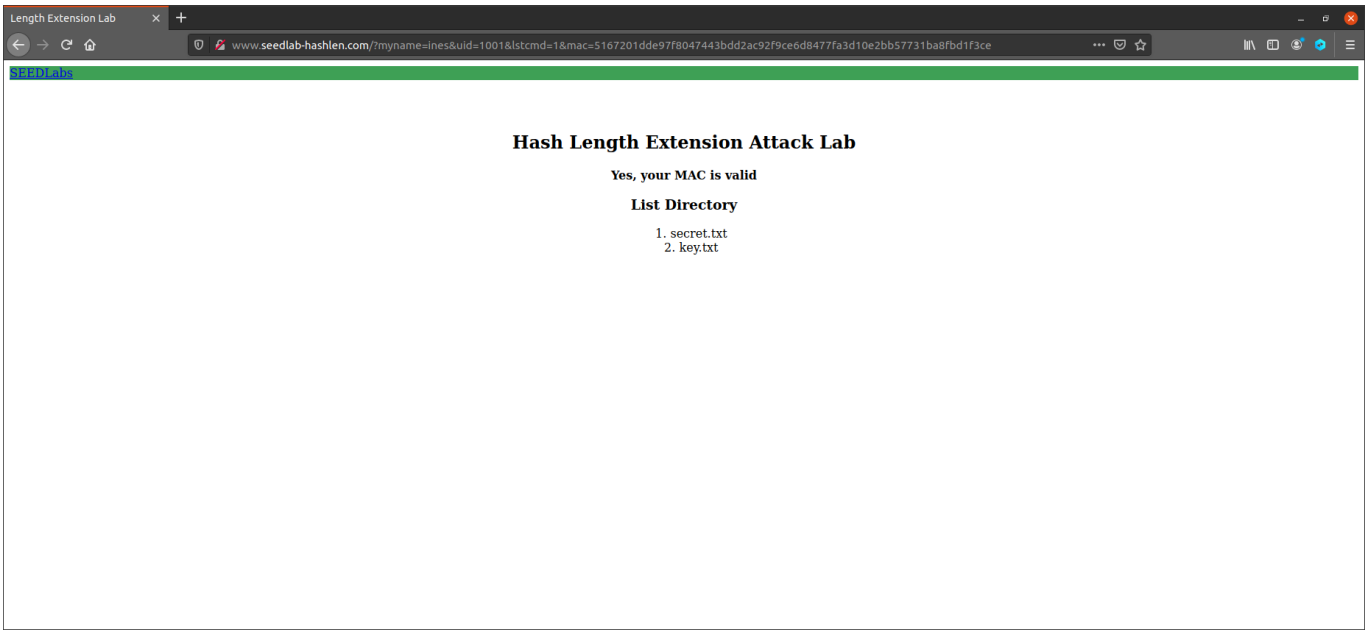
- Name: ines
- UID: 1001
- Key: 123456

The combination of UID and key was obtained from the key.txt file, as they suggested just for these exploratory tasks.

Now, we only need to calculate the mac with the following command:

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

> 5167201dde97f8047443bdd2ac92f9ce6d8477fa3d10e2bb57731ba8fbd1f3ce
```

Visiting the crafted request we get the following page: (http://www.seedlab-hashlen.com/?myname=ines&uid=1001&lstcmd=1&mac=5167201dde97f8047443bdd2ac92f9ce6d8477fa3d10e2bb57731ba8fbd1f3ce)



Task 2: Create Padding

The second task requires us to understand how padding works in SHA-256, and it asks us to create the padding for the message we sent before.

The block size is 64, and the message has size 36.

So, the padding will have 64 - 36 = 28 bytes.

The first byte of the padding is fixed with the value of 0x80, and the last bytes correspond to the hex number of the number of bits of the message.

In another words, the message has 36 * 8 = 288 bits which corresponds to the hex 0x120, so the last two bytes will be 0x01 and 0x20.

The rest of the bytes will be filled with 0x00.

Here the result:

```
\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\x01\x20
```

Task 3: The Length Extension Attack

In this task, our objective is to generate a valid MAC for a forged request without possessing the private key. To achieve this, we will leverage the MAC obtained from the previous request and recalculate it to execute an additional action.

First, we need to clearly understand how we can extend the original message and regenerate its MAC. In short words, we will pad the original message to fill a block size of the SHA-256 so the crafted message is a new block. Subsequently, the new MAC can be calculated performing a new iteration of the algorithm.

For our attack, we start by generating a valid MAC for our original message (used in Task 1).

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

> 5167201dde97f8047443bdd2ac92f9ce6d8477fa3d10e2bb57731ba8fbd1f3ce
```

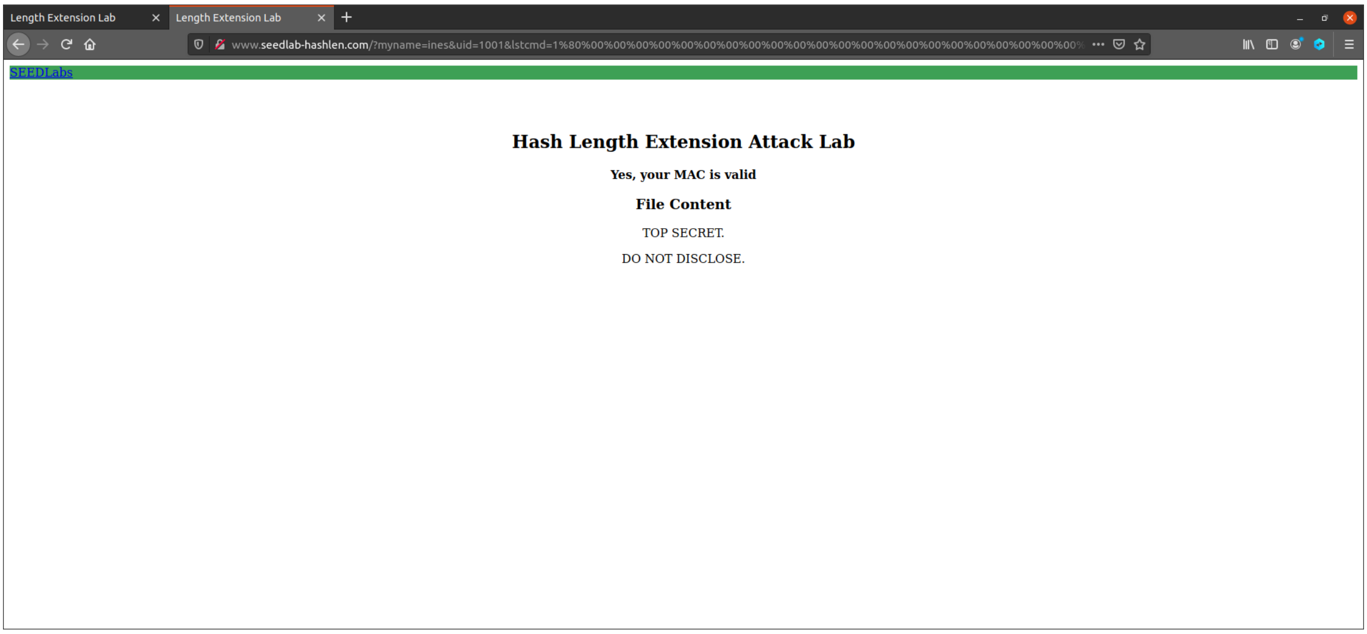
Now, based on this MAC and this message, we're going to forge a new request that includes the `download` command.

To do this, we wrote the following script:

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

int main(int argc, const char *argv[])
{
    int i;
    unsigned char buffer[SHA256_DIGEST_LENGTH];
    SHA256_CTX c;

    SHA256_Init(&c);
    for(i=0; i<64; i++)
        SHA256_Update(&c, "*", 1);

    //MAC
    c.h[0] = htole32(0x5167201d);
    c.h[1] = htole32(0xde97f804);
    c.h[2] = htole32(0x7443bdd2);
    c.h[3] = htole32(0xac92f9ce);
    c.h[4] = htole32(0x6d8477fa);
    c.h[5] = htole32(0x3d10e2bb);
    c.h[6] = htole32(0x57731ba8);
    c.h[7] = htole32(0xfbd1f3ce);

    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;
}
```

The resulting MAC is `f8b76bcd90331283275ff245ba12650805fb03eda80197d86bcb98622b88c186` and with the following URL we confirmed the attack worked.

```
http://www.seedlab-hashlen.com/?
myname=ines&uid=1001&lstcmd=1%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%0
0%00%00%00%00%00%00%00%00%01%20&download=secret.txt&mac=f8b76bcd90331283275ff245ba
12650805fb03eda80197d86bcb98622b88c186
```



Task 4: Attack Mitigation using HMAC

Since it is possible to perform the attack with a simple MAC generated this way, the standard method currently used involves HMAC. First, we modify the hash method present in `lab.py` to generate MACs using the secure HMAC method:

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

After stopping and rebuilding the containers used in the lab, we recreate the hash for Task 1, but this time using HMAC as well:

```
import hmac
import hashlib

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

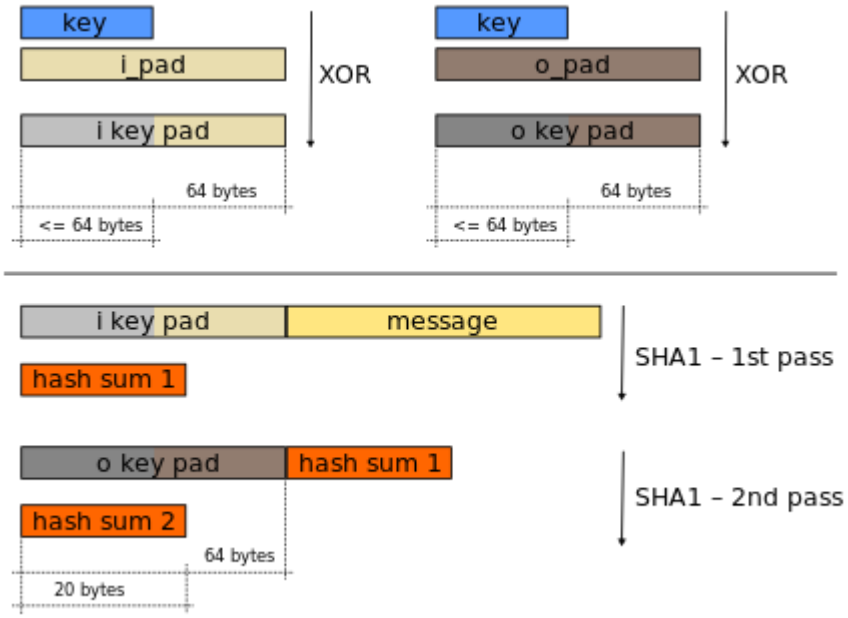
```
print(mac)
```

```
> fb25849259eec65bbf083f6b56e14ecc050a2ed3a8d395a82723b1893587b0c5
```

Attempting to execute the same attack now becomes impossible since HMAC is not prone to length extension attack.

```
http://www.seedlab-hashlen.com/?
myname=ines&uid=1001&lstcmd=1%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%0
0%00%00%00%00%00%00%00%01%20&download=secret.txt&mac=fb25849259eec65bbf083f6b56
e14ecc050a2ed3a8d395a82723b1893587b0c5
```

Why the Hash Length Extension Attack won't work anymore?



In HMAC, the inner hash alone is susceptible to a length-extension attack. This means an attacker could potentially compute a valid inner hash digest without needing access to the secret key. However, the outer hash, which incorporates both a secret key and the inner hash, is immune to such attacks. The attacker only has control over the variable-length input to the inner hash, not to the outer hash, which remains protected by the fixed-length key.

We found an interesting comment on the web:

```
The role of the inner function is to provide collision-resistance (to compress a
long message down to a short fingerprint, in a way so that someone who does not
know the key cannot find a pair of messages with the same fingerprint), and the
role of the outer function is to act as a message authentication code on this
fingerprint.
```

Author

G1:

- Alexandre Nunes (up202005358)
- Fábio Sá (up202007658)
- Inês Gaspar (up202007210)