Information Security HW3

1. Pseudo Primes in RSA

在 RSA 裡面

$pq = N$

加密

$c = n^e \mod N$，成立，因為 e 只要求與 $\phi(N)$ 互值，如果為費馬偽質數，
$\phi(N) = \phi(p) * \phi(q) = (p-1)(q-1)$ 仍然成立，因為p, q 的階仍然為 $p-1$ 跟 $q-1$，所以結果會一樣。

解密

$c^d = n^{ed} = n^{1+k\phi(N)} = n(\mod N)$仍然成立，先看 $ed = 1 + h\phi(N)$ 的部分，這是一開始找的定義，d 一定是 $\phi(N)$ 的模逆元，跟 N 無關，是找出來的，所以成立

再來是 $n^{\phi(N)} \mod (N) = 1$，仍然成立，作業二證明過 n,N 不須互質仍然成立。

因此得證 RSA 依舊成立

2. Key Switch

Let $h = PK_{global} = \Pi_{i=1}^{n} h_i = g^{\Sigma x_i}$

$c = Enc(PK_{global}, m) = (c1, c2) = (g^y, mh^y)$

$c_1^{x_i} = (g^y)^{x_i} = (g^{x_i})^y = h_i^y$

假設我們想要轉換 c 讓 user u 可以用

定義

$c_i = (c_{1,i}, c_{2,i}) = (c_{1,i-1} g^{y_i}, (c1^{x_i})^{-1} h_u^{y_i})$ where y_i 是每輪產生的隨機數

初始情況為 $c_0 = (1, c_2)$，c_i 是不同 user 產生的密文

轉換流程為，除了 user u 不會執行該流程，其他的 user 都要執行一次，總計 $n-1$ 次，順序沒差，執行完後會得到$c' = (g^{\Sigma y_i}, c2 * (\Pi_i c1^{x_i})^{-1} * h_u^{\Sigma y_i})$，user u 可以由原本的方式解密得到:

Dec(c') for user u

$s = (c_1')^{x_u} = g^{\Sigma x_u y_i} = h_u^{\Sigma y_i}$

$c_2'' = c_2' * s^{-1} = c2 * (\Pi_i h_i^y)^{-1} * h_u^{\Sigma y_i} * (s)^{-1} = m * (\Pi_i h_i^y) * h_u^y * (\Pi_i h_i^y)^{-1} * h_u^{\Sigma y_i} * (h_u^{\Sigma y_i})^{-1} = m * h_u^y$

再一次解密:

Dec((c_1, c_2''))

$s = c_1^{x_u} = h_u^y$

$c_2'' * s^{-1} = m * h_u^y * (h_u^y)^{-1} = m$

成功

3. Identity-Based Encryption: Security Proof

$\text{Enc}(PK, ID, m) \to K_{ID}$

$c = (g^r, m * e(g^{rs}, H(ID))) = (g^r, m * e(g^{rs}, g^k))$ for some k, $= (g^r, m * e(g, g)^{rsk})$

根據 Decisional Bilinear Diffie-Hellman Assumption，$e(g, g)^{rsk}$ 跟 $Z \in \mathbb{G}_T$ 無法分辨，因此無法分辯 $m_1 * e(g, g)^{r_1 sk}$ 跟 $m_2 * e(g, g)^{r_2 sk}$，就算 $m1 = m2$，因為有一個隨機的r_i，所以每次結果都會不一樣。

4. Fugisaki-Okamoto Commitment

1.

$c = g^m h^r = g^m g^{kr} = g^{m+kr}$，縱使我們有 c, g，基於離散對數難題，很難找到 $m + kr$，就算找到 $m + kr$，基於離散對數難題，由於很難找到 k，因此很難找到 m

2.

$c = g^m h^r = g^m g^{kr} = g^{m+kr}$，假設找到的參數是$(m', r')$

$h^{r'} = \frac{c}{g^{m'}}$，根據離散對數難題，我們很難找到 r'，因此很難構造 m'

5. Programming: One Way Hash Chain

套件(On Mac with Homebrew)

```
1  brew install openssl cmake
```

程式碼在 hasher 資料夾下，進去之後下以下指令來編譯

```
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release -DOPENSSL_ROOT_DIR=/opt/homebrew/opt/openssl -DOPENSSL_LIBRARIES=/opt/homebrew/opt/openssl/lib ..
make
```

使用方法

```
./hasher {videofile_path}
```

6. Lab: Hash Length Extension Attack

- Task 1: Send Request to List Files

利用題目給的 query format

```
myname=<name>&uid=<need-to-fill>&lstcmd=1&mac=<need-to-calculate>
```

構造如下query

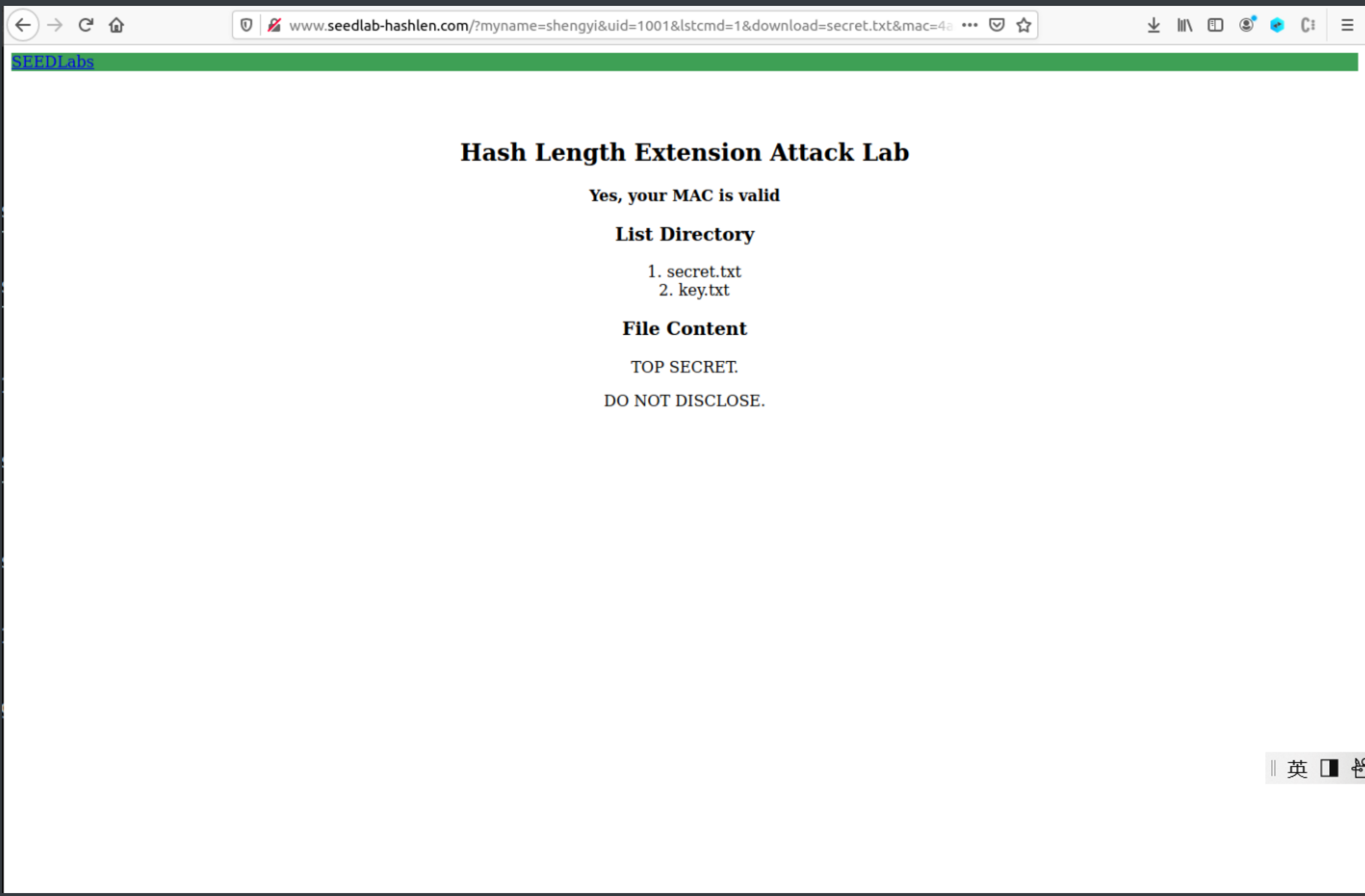
```
myname=shengyi&uid=1001&lstcmd=1&download=secret.txt
```

用 UID 1001 的 secret key:123456 (題目給的)，構造出如下的結果並丟到 SHA256 計算 MAC

```
[05/14/23]seed@VM:~/.../LabHome$ echo -n "123456:myname=shengyi&uid=1001&lstcmd=1&download=secret.txt" | sha256sum
4aa403e04346e490ebaecf58cbccc756a71c8c6ade544017dcf402c0d042f19b  -
```

最終得到如下URL

```
http://www.seedlab-hashlen.com/?myname=shengyi&uid=1001&lstcmd=1&download=secret.txt&mac=4aa403e04346e490ebaecf58cbccc756a71c8c6ade544017dcf402c0d042f19b
```



- Task 2: Create Padding

123456:myname=shengyi&uid=1001&lstcmd=1%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%01%38

- Task 3: The Length Extension Attack

1. 計算 checksum for 123456:myname=shengyi&uid=1001&lstcmd=1

```
[05/14/23]seed@VM:~$ echo -n "123456:myname=shengyi&uid=1001&lstcmd=1"
 | sha256sum
db4343cbbe48862b295029507e2d19c69ff90ee1fcacd3d5baa5f16a9bea6d1c  -
```

2. 計算 original + padding + "&download=secret.key" checksum

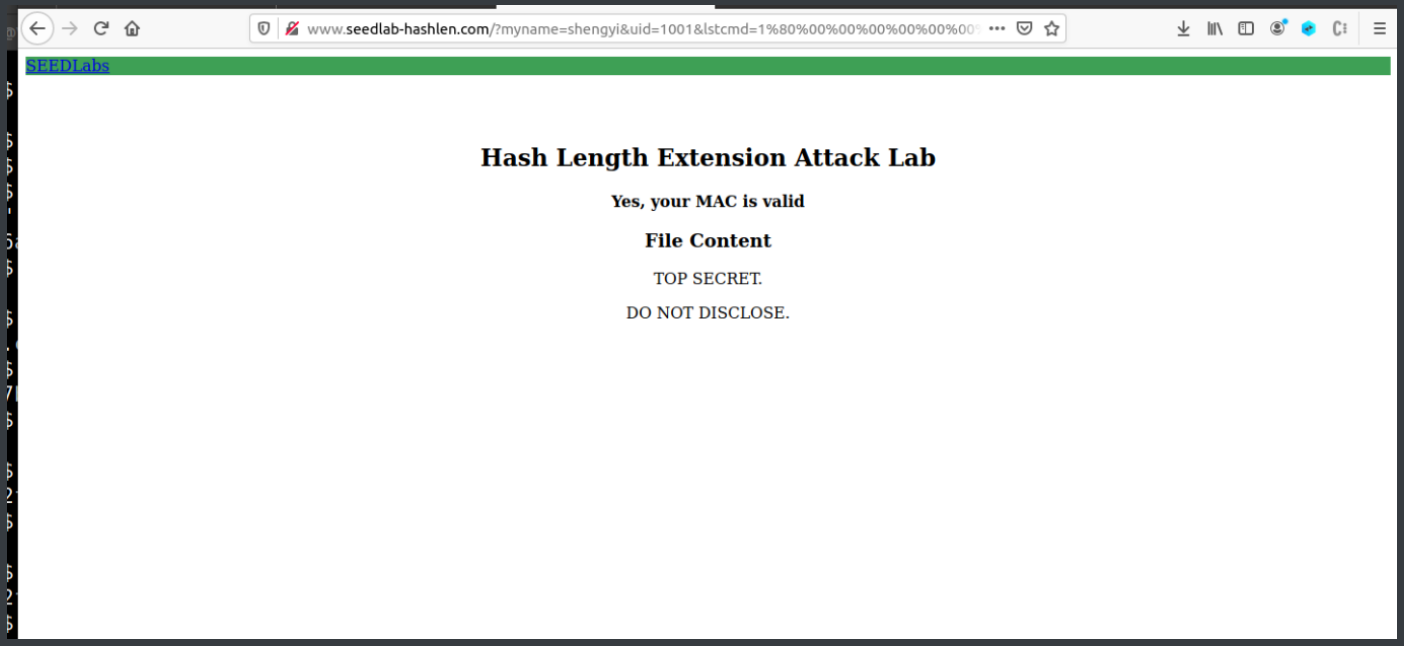
```c
/* length_ext.c */
#include <stdio.h>
#include <arpa/inet.h>
#include <openssl/sha.h>
int main(int argc, const char *argv[])
{
  int i;
  unsigned char buffer[SHA256_DIGEST_LENGTH];
  SHA256_CTX c;
  SHA256_Init(&c);
```

```c
  for(i=0; i<64; i++)
    SHA256_Update(&c, "*", 1);
  // MAC of the original message M (padded)
  c.h[0] = htole32(0xdb4343cb);
  c.h[1] = htole32(0xbe48862b);
  c.h[2] = htole32(0x29502950);
  c.h[3] = htole32(0x7e2d19c6);
  c.h[4] = htole32(0x9ff90ee1);
  c.h[5] = htole32(0xfcacd3d5);
  c.h[6] = htole32(0xbaa5f16a);
  c.h[7] = htole32(0x9bea6d1c);
  // 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;
}
```

```
[05/14/23]seed@VM:~/.../LabHome$ ./length_ext
9b5dcbf75fcd42b1d076eb3a0f1c9bd853a741180d333c0f31b8bffce4391a06
```

3. query original + padding + "&download=secret.key" + Mac =

```
http://www.seedlab-hashlen.com/?myname=shengyi&uid=1001&lstcmd=1%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%01%38&download=secret.txt&mac=9b5dcbf75fcd42b1d076eb3a0f1c9bd853a741180d333c0f31b8bffce4391a06
```



- Task 4: Attack Mitigation using HMAC

1. 修改

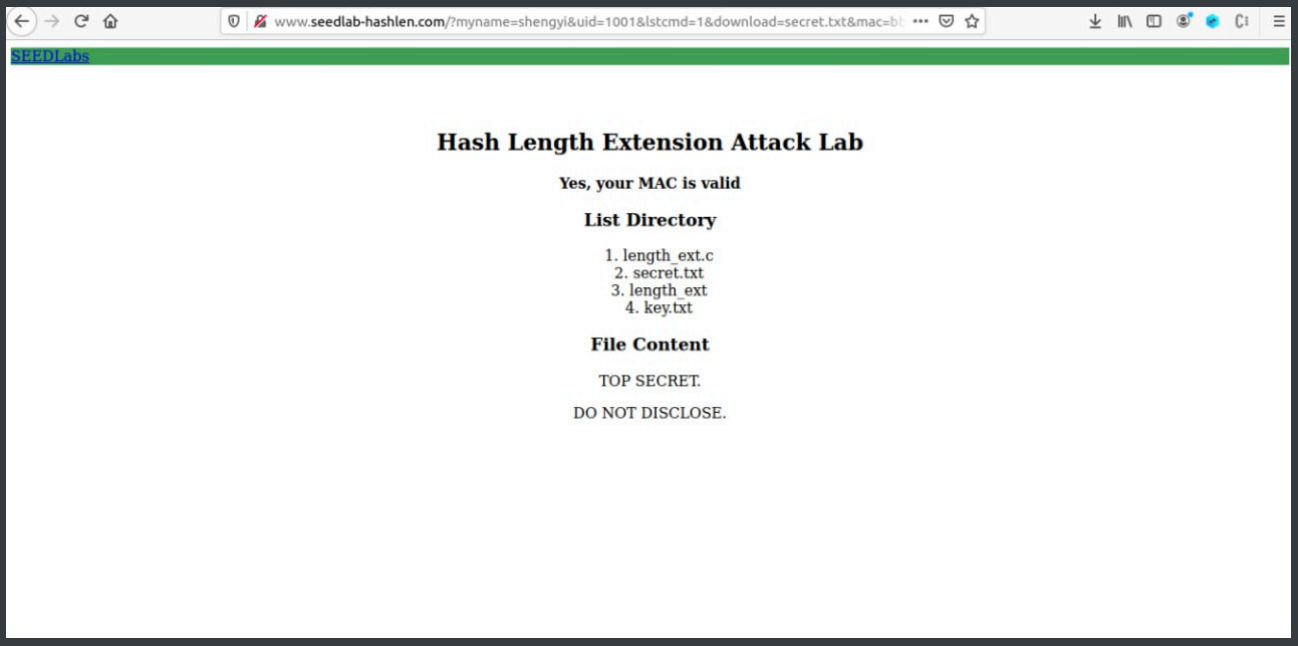
```
def verify_mac(key, my_name, uid, cmd, download, mac):
    download_message = '' if not download else '&download=' + download
    message = ''
    if my_name:
        message = 'myname={}&'.format(my_name)
    message += 'uid={}&lstcmd='.format(uid) + cmd + download_message
    payload = key + ':' + message
    app.logger.debug('payload is [{}]'.format(payload))
    real_mac = hmac.new(bytearray(key.encode('utf-8')),msg=message.enc
ode('utf-8', 'surrogateescape'),digestmod=hashlib.sha256).hexdigest()
    app.logger.debug('real mac is [{}]'.format(real_mac))
    if mac == real_mac:
        return True
    return False
```

2. Code:

```
#!/bin/env python3
import hmac
import hashlib
key='123456'
message='myname=shengyi&uid=1001&lstcmd=1&download=secret.txt'
mac = hmac.new(bytearray(key.encode('utf-8')),
msg=message.encode('utf-8', 'surrogateescape'),
digestmod=hashlib.sha256).hexdigest()
print(mac)
```

```
[05/14/23]seed@VM:~$ python3 calc.py
bbe0f1b9682d7a90d7b27a5f794869865821050e5cf9223fc36e90b20c10ef2a
```

3. query: http://www.seedlab-hashlen.com/?myname=shengyi&uid=1001&lstcmd=1&download=secret.txt&mac=bbe0f1b9682d7a90d7b27a5f794869865821050e5cf9223fc36e90b20c10ef2a



因為 hmac 做了兩次 hash, 縱使我們能延長外部的 hash_value，但是內部的 hash_value 是不可知的，因此這個攻擊不會成功

觀察：

可以發現，這個攻擊手法很優雅地避開原本的明文，用擴充的方式就可以修改明文，有點像是 SQL Injection 的感覺，非常有趣