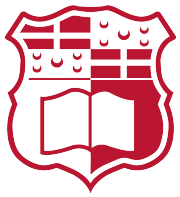


CPS3232: Applied Cryptography

Hands-on Crypto

MARC FERRIGGI

Supervised by Dr Mark Vella

Department of Computer Science

Faculty of ICT

University of Malta

January, 2019

A assignment submitted in partial fulfilment of the requirements for the degree of B.Sc. (hons.) Conputer Science AND Statistics and Operations Research.

Statement of Originality

I, the undersigned, declare that this is my own work unless where otherwise acknowledged and referenced.

Candidate Marc Ferriggi

Signed _____

Date January 16, 2019

Contents

1 Block Cipher Modes and Message Authenticity **1**

1.1 Introduction 1
1.2 ECB Mode Implementation 2
1.3 CBC Mode Implementation 3
1.4 Hardened Implementation 5
1.5 Demonstrating Failed Attacks 6

2 Transparent Access Control on the Blockchain **8**

2.1 Introduction 8
2.2 Implementing the Reference Monitor 8
2.3 Attack 1 10
2.4 Attack 2 11

3 Setting up CA trees and HTTPS servers. **13**

3.1 Introduction 13
3.2 Configuring the CA tree 13
3.3 Signing a domain using the CA 15
3.4 Traffic Analysis on HTTP and HTTPS 17

1

Block Cipher Modes and Message Authenticity

1.1 Introduction

In this task, a secure communication system will be modelled that makes use of an AES library for an e-Banking application for money transfers. The text-based protocol that's used by this e-Banking company has the following message format:

`Accnt:`*[10 bytes]*`Accnt:`*[10 bytes]*`Descr:`*[58 bytes]*`Amount:`*[9 bytes]*

The following libraries and functions were imported in order to run all the code presented in this chapter:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from Crypto.Util.Padding import unpad
from Crypto.Random import get_random_bytes
from base64 import b16encode
from base64 import b16decode
import os
import json
import hmac
import hashlib
```

The required data was then initialized as follows:

```
#Initialize Data
data = b"Accnt:0003336669Accnt:0123456789Descr:00000000000000
                000000000000000000000000000000000000000000A
```

```
                mount:023456789"
key = get_random_bytes(16)
print(b16encode(key).decode('utf-8'))
```

1.2 ECB Mode Implementation

The first task was to implement the e-Banking system using AES in ECB mode and demonstrate the weakness of using this approach through an attack which swaps the accounts thus reversing the direction of the transfer.

The data was encrypted in ECB mode using the following code:

```
ECB_cipher = AES.new(key, AES.MODE_ECB) #Create a new AES Cypher
 #encrypt the padded data
ciphertext_bytes = ECB_cipher.encrypt(pad(data, AES.block_size))
#Encode the cyphertext to 16 bit encoding to have a more clear output
#use the decode function to remove the b'...' in the output
ciphertext = b16encode(ciphertext_bytes).decode('utf-8')
print(ciphertext)
```

Output: 4CDEB5515885FEC7390A48D4F9747BD53A309C3CA17838A5A1FD18A9A433737E9
D35FE123067946375B9B59FAD318E1CE9C1DC732606DAA7644898E4D577B90DE9C1DC732606
DAA7644898E4D577B90DE9C1DC732606DAA7644898E4D577B90DBD18D9394E2B939E4EA7D9E
8C14C0B04FAF69C2C9877EE7A8805287F5B34C232

The ciphertext was then immediately decrypted to ensure it was encrypted successfully:

```
decoded_ciphertext = b16decode(ciphertext)
decrypted_text = unpad(ECB_cipher.decrypt(decoded_ciphertext),
                                    AES.block_size)
print(decrypted_text)
```

Output: b'Accnt:0003336669Accnt:0123456789Descr:00000000000000000000000000000000
00000000000000000000000000Amount:023456789'

The attack was then demonstrated by Oscar by simply swapping the desired bits of the ciphertext, and once Bob decodes the message he gets a valid transaction and is thus none the wiser that the accounts have been swapped.

```
#Swap the account numbers to reverse the direction of the transfer
modified_ciphertext = ''.join([ciphertext[32:64],
```

```
        ciphertext[0:32], ciphertext[64:]])
#decrypt message:
modified_result = unpad(ECB_cipher.decrypt(b16decode
        (modified_ciphertext)), AES.block_size)
print(modified_result.decode('utf-8'))
```

Output: Accnt:0123456789Accnt:0003336669Descr:00000000000000000000000000000000000000000000000000000000000000000000Amount:023456789

1.3 CBC Mode Implementation

The encryption was then implemented using CBC mode and another attack was demonstrated by modifying the amount since the implementation does not verify for valid format field labels.

The data was encrypted in CBC mode using the following code:

```
iv = os.urandom(16) #initializing vector
#Create new AEC Cipher in CBC mode
CBC_cipher = AES.new(key, AES.MODE_CBC,iv)
#Encrypt padded data
ciphertext_bytes = CBC_cipher.encrypt(pad(data, AES.block_size))
#Encode the cyphertext to 16 bit encoding to have a more clear output
#use the decode function to remove the b'...' in the output
CBC_ciphertext = b16encode(ciphertext_bytes).decode('utf-8')
#Save as Json in order to simulate passing the whole thing over the network
json_data = json.dumps({'iv':b16encode(iv).decode('utf-8'),
         'ciphertext':CBC_ciphertext})
```

The ciphertext was again immediately decrypted to ensure it was encrypted successfully:

```
#Read the data from the json file
b16 = json.loads(json_data)
iv = b16decode(b16['iv'])
#Create a new Cipher for decryption using the same key and iv
CBC_cipher2 = AES.new(key, AES.MODE_CBC, iv)
#Decode, unpad and display the results
CBC_dec_ciphertext = b16decode(b16['ciphertext'])
CBC_decoded_plaintext = unpad(CBC_cipher2.decrypt(
        CBC_dec_ciphertext),AES.block_size)
print(CBC_decoded_plaintext)
```

Output: b'Accnt:0003336669Accnt:0123456789Descr:00000000000000000000000000000000000000000000000000000000000000000000Amount:023456789'

An XOR function was then defined to be used in the attack:

```
def xor(A, B):
        return hex(int(A, 16) ^ int(B, 16))[2:].upper()
```

The attack as explained previously was then demonstrated as follows:

```
#Attacker can also reda the data from the json file
Oscar_b16 = json.loads(json_data)
Oscar_ciphertext = Oscar_b16['ciphertext']
Oscar_iv = Oscar_b16['iv']
block_to_alter = Oscar_ciphertext[160:192] #block containing amount
byte_to_change = block_to_alter[14:16] #amount

#xor 1st bit of plaintext with the hex value you want in new plaintext
m = xor('0x30', '0x39') #0x30 is decimal 0, 0x39 is decimal 9

#perform bit flipping
new_byte = xor(hex(int(m, 16)), hex(int(byte_to_change, 16)))
while len(new_byte) < 2:
        new_byte = "".join(['0', new_byte])
attacked_ciphertext = ''.join([Oscar_ciphertext[0:174],
                new_byte, Oscar_ciphertext[176:]])

result = json.dumps({'iv':Oscar_iv, 'ciphertext':attacked_ciphertext})
```

Finally, the attack was decoded by Bob and as can be seen in the output, the amount was successfully altered.

```
#Create a new Cipher for decryption using the same key and iv
CBC_cipher_Oscar = AES.new(key, AES.MODE_CBC,b16decode(Oscar_iv))
#Decode, unpad and display the results
CBC_dec_ciphertext = b16decode(attacked_ciphertext)
CBC_decoded_plaintext = unpad(CBC_cipher_Oscar.decrypt(
                CBC_dec_ciphertext),AES.block_size)
print(CBC_decoded_plaintext)
```

*Output:*b'Accnt:0003336669Accnt:0123456789Descr:00000000000000000000000000000000000000000000\xbbs\x14\x81\x92xC\x975\xbfb\xdf\xaa\\\xc8DAmount:923456789'

1.4 Hardened Implementation

The system was then hardened in order to provide transaction integrity. Message authentication codes were used to protect from content tampering and a challenge-response was st up to protect from replay attacks.

Initially, a function was set up to compare the MACs:

```
def compare_mac(a, b):
        a = a[1:]
        b = b[1:]
        different = 0

        for x, y in zip(a, b):
                different |= x ^ y
        return different == 0
```

The data was then encrypted in CBC mode but with added MAC using HMAC-SHA256 standard.

```
hmac_key = get_random_bytes(16)
plaintext = pad(data, AES.block_size)
iv_bytes = get_random_bytes(AES.block_size)
HMAC_cipher = AES.new(key, AES.MODE_CBC, iv_bytes)
encrypted_data = HMAC_cipher.encrypt(plaintext)
iv = iv_bytes + encrypted_data #secret prefix mac
signature = hmac.new(hmac_key, iv, hashlib.sha256).digest()
```

Finally, the signature was verified and the data decrypted if it wasn't tampered with.

```
new_hmac = hmac.new(hmac_key, iv, hashlib.sha256).digest()
if not compare_mac(new_hmac, signature):
print("Incorrect decryption")
cipher = AES.new(key, AES.MODE_CBC, iv_bytes)
dec_plaintext = cipher.decrypt(encrypted_data)
print(unpad(dec_plaintext, AES.block_size).decode('utf-8'))
```

Output: Accnt:0003336669Accnt:0123456789Descr:00000000000000000000000000000000000000000000000000000000000000000000Amount:023456789

1.5 Demonstrating Failed Attacks

Two failed attacks were then demonstrated to show the new system's security. The first attack attempts to tamper the data, while the second failed attack attempts to perform a replay attack.

The attack function attempts the data tampering similar to the first attack shown when the data was encrypted in CBC mode.

```
#Like Task B's attack
def attack(data):
        b16 = json.loads(data)
        ciphertext = b16['ct']
        iv = b16['iv']
        block_to_alter = ciphertext[160:192]
        byte_to_change = block_to_alter[14:16]

        m = xor('0x30', '0x39')

        #perform bit flipping
        new_byte = xor(hex(int(m, 16)), hex(int(byte_to_change, 16)))
        while len(new_byte) < 2:
                new_byte = "".join(['0', new_byte])
        new_ciphertext = ''.join([ciphertext[0:174],
                        new_byte, ciphertext[176:]])

        result = json.dumps({'iv':iv,
                        'ct':new_ciphertext})
        return result
```

The *decrypt_message* function performs the required checks before decrypting the data in order to ensure the data hasn't been tampered with, similar to the decryption step shown in the previous section.

```
#Like Task C's Decryption
def decrypt_message(encrypted_data, iv_bytes, signature, shared_key,
                        hmac_key):
        iv = iv_bytes + encrypted_data
        new_hmac = hmac.new(hmac_key, iv, hashlib.sha256).digest()
        if not compare_mac(new_hmac, signature):
                print("Attack!")
```

```
        cypher = AES.new(shared_key, AES.MODE_CBC, iv_bytes)
        plaintext = cypher.decrypt(encrypted_data)
        return unpad(plaintext, AES.block_size)
```

The tamper attack was done as follows:

```
iv = b16encode(iv_bytes).decode('utf-8')
ciphertext = b16encode(encrypted_data).decode('utf-8')
ct_json = json.dumps({'iv': iv, 'ct': ciphertext})
altered_ciphertext = attack(ct_json)
altered_ciphertext = b16decode(json.loads(altered_ciphertext)['ct'])
decrypt_message(altered_ciphertext, iv_bytes, signature, key, hmac_key)
```

Output: Attack!

Finally, the replay attack was also demonstrated:

```
altered_ciphertext = b"".join([encrypted_data[32:64],
        encrypted_data[0:32], encrypted_data[64:]])
decrypt_message(altered_ciphertext, iv_bytes, signature,
        key, hmac_key)
```

Output: Attack!

2

Transparent Access Control on the Blockchain

2.1 Introduction

In this task, a reference monitor was implemented where access request decisions took the form of Blockchain transactions. In particular, n parties were provided access to a controlled resource through a shared secret s. Fiat-Shamir's Zero Knowledge Proof was used in order not to disclose the shared secret s.

2.2 Implementing the Reference Monitor

The reference monitor was implemented in Jupyter Notebook. The RSA modulus was generated using PyCryptodome's *RSA.generate* function, where the number of bits was set to 2048 as this was deemed a sufficient length for 2017 and the public exponent e was set to be 65537 as this is the FIPS standard. The generated key (which contains the RSA modulus $n = p.q$) was then saved to a file protected by a password in order to simulate that only Rene the reference monitor has access to it.

```
%reset
from Crypto.PublicKey import RSA
import random

#Setting up FS
#Generate n:
key = RSA.generate(2048,e=65537)
#nbits = 2048 (Sufficient length for 2017), e -> FIPS Standard
#save the secure key to a file to only be accessed by Rene
passphrase = 'Str0ngPassw0rd!%'
```

```
f = open('mykey.pem','wb')
#Use a passphrase since
f.write(key.exportKey('PEM',passphrase=passphrase))
f.close()
n = key.n #assume n is public knowledge

#Rene will then generate s, which lies in integer ring (Z_n)
s = random.randint(1,key.n+1)
#Assume this is sent to Alice and Bob over a secure channel

#Delete Secrets:
del key
del passphrase
#Authorised parties will then be given mykey.pem and
#with knowledge of the passphrase will have access
#to the secure key
```

Once the RSA key was set up, Rene then set v by using the equation $v = s^2 \bmod n$.

```
#Rene:
#get the key
f = open('mykey.pem','r')
Rene_key = RSA.importKey(f.read(),passphrase='Str0ngPassw0rd!%')
f.close()
#set v
v = (s**2)%Rene_key.n
```

The interactive protocol was then implemented setting $t = 100$, thus having a successful attack probability of 2^{-100}.

```
t = 100
for i in range(t):
        #Interactive Protocol
        #Alice
        #pick random r
        r = random.randint(1,n+1)
        x = (r**2)%n #compute x
        #x is now sent to Rene

        #Rene
        e = random.randint(0,1)
```

```
        #e is sent to Alice

        #Alice
        y = (r*(s**e)) % n
        #y is sent to Rene

        #Rene:
        if (y**2 % n != (x*(v**e))%n):
                print('Attack!')
                break
```

2.3 Attack 1

The implementation was then weakened in a way so that Alice always commits to the same $r \in Z_n$. An attack that discloses s which hinges on the fact that whenever $e = 0$ Alice sends $y = r \ mod \ n$ as a response was then shown. Given the fact that r wasn't changing, Oscar could simply extract r by intercepting y when $e = 0$. Once Oscar has r then, when $e = 1$, Oscar intercepts y and can extract s by using the equation $s = y.r^{-1} \ mod \ n$. The modified code to the previous implementation can be seen below.

```
from sympy import mod_inverse

gotR = 0

t = 100
#Alice will pick a random r and keep it fixed:
r = random.randint(1,n+1)
for i in range(t):
        #Interactive Protocol
        #Alice:
        x = (r**2)%n #compute x
        #x is now sent to Rene

        #Rene
        e = random.randint(0,1)
        #e is sent to Alice on open channel,
        #Oscar can intercept this

        #Alice
        y = (r*(s**e)) % n
        #y is sent to Rene on an open channel,
```

```
        #Oscar can intercept this

        #Oscar's Attack:
        if e==0 and gotR == 0:
                oscar_r = y
                gotR = 1
        if e==1 and gotR == 1:
                oscar_s=(y*mod_inverse(r,n)) % n
                print(s==oscar_s)
                break

        #Rene:
        if (y**2 % n != (x*(v**e))%n):
                print('Attack!')
                break
```

Output: True

2.4 Attack 2

The original implementation was then weakened in a way that Rene would always challenge with $e = 1$. In this case, its possible to spoof Alice without knowing s. For this attack it also assumed that Oscar has access to r, or has somehow guessed it. Oscar then uses a man in the middle attack to intercept x and send over $x/v \ mod \ n$ (which is equivalent to $r^2v^{-1} \ mod \ n$) to Rene. Oscar will continue the attack and intercept y and simply send over r instead of y to Rene. When Rene checks to ensure $y^2 \equiv x * v^e \ mod \ n$, this would return true, thus granting access to Oscar. The modified code for the attack can be seen below.

```
t = 100
for i in range(t):
        #Interactive Protocol
        #Alice
        #pick random r
        r = random.randint(1,n+1)
        x = (r**2)%n #compute x
        #x is now sent to Rene on an open channel

        ####################################
        #Oscar intercepts x with a man in the
        #middle attack, changes it and sends
```

```
#his own x to Rene
Oscar_x = (x*gmpy2.invert(v,n))%n
#####################################


#Rene receive's Oscar's x and sends 1
e = 1
#e is sent to Alice, Oscar can intercept

#Alice receives e so she's clueless to the attack
y = (r*(s**e)) % n
#y is sent to Rene on the open channel

#####################################
#Oscar intercepts y
Oscar_y = r % n
#####################################

#Rene will check with Oscar's values and thus
#let him in:
if (Oscar_y**2 % n != (Oscar_x*(v**e))%n):
        print('Attack!')
        break
```

3

Setting up CA trees and HTTPS servers.

3.1 Introduction

In this task, the `openssl toolkit` was used to set up a Certificate Authority and HTTPS servers in order to provide client/server authentication along with message confidentiality and integrity by passing the HTTP traffic over TLS. Note that although the CA has been set up in the `/root/ca` directory, the file structure has been copied over to the `tasks/Task 3` folder in the provided `.zip` file.

3.2 Configuring the CA tree

Initially, the terminal was set to run as root by running the command `sudo -i` and entering the password when prompted. The following commands were then run to create the directory structure as well as set restricted access to the directory.

```
mkdir /root/ca
cd /root/ca
mkdir certs crl newcerts private
chmod 700 private
touch index.txt
echo 1000 > serial
```

A configuration file was then set up, copying the contents from the *Appendix* of the provided tutorial. this was saved in the file `/root/ca/openssl.cnf`. This file is important as it tells `openssl` what options to use when setting up the CA as well as the policies

that must be followed by the CA intermediate signatures in order to get signed by the root CA. The root key was then created using `openssl` and this was encrypted using AES in order to be kept absolutely secure. The following code was run to do this, and a password was created when prompted.

```
openssl genrsa -aes256 -out private/ca.key.pem 4096
chmod 400 private/ca.key.pem
```

A root certificate was then set up using the newly created root key. The certificate was given an arbitrarily long expiry date of 7300 days for the sake of this assignment. After running the `openssl` command, details of the root CA were entered when asked to in order to be able to distinguish this CA from other CA's.

```
openssl req -config openssl.cnf \
    -key private/ca.key.pem \
    -new -x509 -days 7300 -sha256 -extensions v3\_ca \
    -out certs/ca.cert.pem
chmod 444 certs/ca.cert.pem
```

The intermediate CA was then set up. In real world scenarios, the intermediate CA will be used to sign certificates on behalf of the root CA, so that the root CA can be kept offline to improve the security of the structure. Thus, if the private key of the intermediate CA is compromised, the root CA will simply revoke the intermediate one. The following commands were run similar to those of creating the root CA in order to begin the setup of the intermediate CA. The last line in the following code adds a file to keep track of certificate revocation list, although this will not be used for this task, it was set up anyway to have a complete structure.

```
mkdir /root/ca/intermediate
cd /root/ca/intermediate
mkdir certs crl csr newcerts private
chmod 700 private
touch index.txt
echo 1000 > serial
exho 1000 > /root/ca/intermediate/crlnumber
```

The intermediate CA configuration file was then set up similar to the root's configuration file, with a few commands changed. this was saved to `/root/ca/intermediate/`

`openssl.conf`. Once again, a key was then set up and the intermediate certificate was created and encrypted using the set up key. Note that the intermediate certificate was set to be valid for 3650 days, which is a shorter period than the root certificate.

```
cd /root/ca
openssl genrsa -aes256 \
    -out intermediate/private/intermediate.key.pem 4096

chmod 400 intermediate/private/intermediate.key.pem
openssl req -config intermediate/openssl.cnf -new -sha256 \
    -key intermediate/private/intermediate.key.pem \
    -out intermediate/csr/intermediate.csr.pem

openssl ca -config openssl.cnf -extensions v3_intermediate_ca \
    -days 3650 -notext -md sha256 \
    -in intermediate/csr/intermediate.csr.pem \
    -out intermediate/certs/intermediate.cert.pem
chmod 444 intermediate/certs/intermediate.cert.pem
```

The `index.txt` file of the root CA was then checked to ensure that it contains an entry to the intermediate CA. After this, the certificate chain file was then created, this file tells the web browser how to complete the chain of trust when verifying a certificate in order to reach the root CA.

```
cat intermediate/certs/intermediate.cert.pem \
    certs/ca.cert.pem > intermediate/certs/ca-chain.cert.pem
chmod 444 intermediate/certs/ca-chain.cert.pem
```

3.3 Signing a domain using the CA

In this task, the newly set up intermediate CA was used to sign a certificate for the domain `www.example.com`, thus enabling HTTPS access to the website. Firstly, a key was created for the client domain. Since this was set to expire after only 1 year, 2048 bits was used for the key instead of 4096 bits as this would be sufficiently secure and would increase the performance of TLS handshakes.

```
cd /root/ca
openssl genrsa -aes256 \
```

```
    -out intermediate/private/www.example.com.key.pem 2048
chmod 400 intermediate/private/www.example.com.key.pem
```

Similar to the previous tasks, the private key was then used to create the certificate which will be signed by the intermediate CA using the following code, where the details of the domain were entered when prompted.

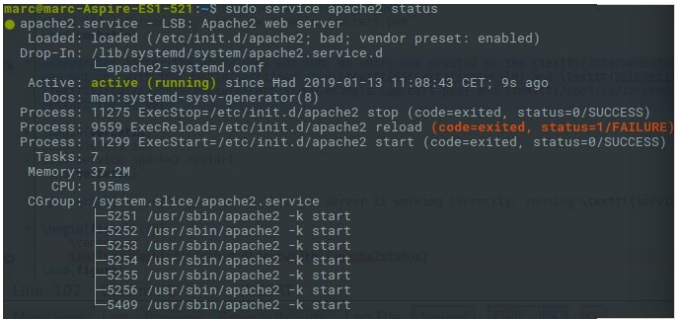
```
openssl req -config intermediate/openssl.cnf \
    -key intermediate/private/www.example.com.key.pem \
    -new -sha256 -out intermediate/csr/www.example.com.csr.pem

openssl ca -config intermediate/openssl.cnf \
    -extensions server_cert -days 375 -notext -md sha256 \
    -in interediate/csr/www.example.com.csr.pem \
    -out intermediate/certs/www.example.com.cert.pem
chmod 444 intermediate/certs/www.example.com.cert.pem
```

After the above code was run, an entry was created in the `intermediate/index.txt` which referred to the new certificate of `www.example.com`. Once the CA tree was set up, with the domain certified, the `SSLCertificateFile` and `SSLCertificateKeyFile` entries in the Apache 2 settings file were changed to point to `/root/ca/intermediate/certs/www.example.com.cert.pem` and `/root/ca/intermediate/private/www.example.com.key.pem` respectively. Finally, the following code was run to restart the server:

```
sudo a2enmod ssl
sudo service apache2 restart
```

In order to verify that the server is working correctly, running `service apache2 status` gave the following output:



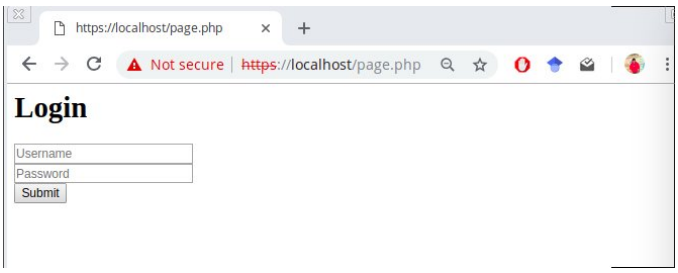
Furthermore, running `localhost` in a web browser or `https://localhost` both gave a running web page. Finally, the server's IP address was hard-coded in a DNS response cache file in order to carry out domain name resolution.

3.4 Traffic Analysis on HTTP and HTTPS

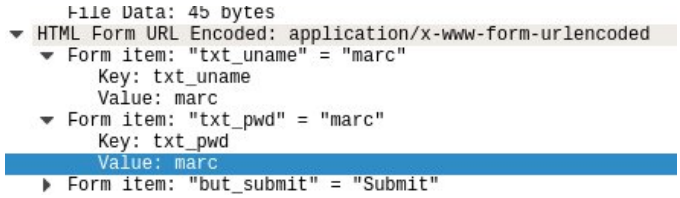
In order to be able to analyse the traffic over the Apache server, a login page was created using PHP and mySQL. Initially, mySQL was installed on the machine and run using the command `mysql -u root -p`, a schema called `crypto` was created and a 'users' table was created in this schema using the following code:

```
CREATE TABLE ‘users‘ (
    ‘id‘ int(11) NOT NULL PRIMARY KEY AUTO_INCREMENT,
    ‘username‘ varchar(80) NOT NULL,
    ‘name‘ varchar(80) NOT NULL,
    ‘password‘ varchar(80) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

The table was then populated with an entry and a login page was then created using PHP and saved in `/var/www/html` in order to be run by the Apache server. These files can be found saved in `/tasks/Task 3/html` in the provided `.zip` folder. The resultant login page running on `localhost` can be seen below:



After all of this was set up, `wireshark` was installed to perform traffic analysis. This was first done when running `localhost` on HTTP and the following image shows the exposed user name and password that was sent over the network with no encryption.



The next image shows the traffic analysis done when running `localhost` in HTTPS. It can be noted that no data was exposed since its now encrypted.