MÁSTER INTERUNIVERSITARIO EN CIBERSEGURIDAD

Information Security
Lab 1

Héctor Núñez Fernández

In order resolve this lab, I've created this code that can encrypt, send, receive, and decrypt arbitrary messages via nested encryption by using a MQTT message broker as intermediate forwarder between agents. The code is located in https://github.com/hectornunfer/SI-Munics/tree/main/Lab1. It's composed by 3 files, the first one is **tor.py.** In that file there are all the necessary encrypt/decrypt functions:

1- One to read my private key and another one to read my public key:

```
import cryptography
import os
import base64
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
# RSA-OAEP Encrypting and decrypting
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
import paho.mqtt.client as mqtt
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from base64 import b64decode,b64encode
pubkey_dictionary = {

}
# Open my private key
with open("id_rsa", "rb") as key_file:
    private_key = serialization.load_ssh_private_key(
        key_file.read(),
        password=None,
        backend=default_backend()
    )

# Open my public key
with open("id_rsa.pub", "rb") as key_file:
    public_key = serialization.load_ssh_public_key(
        key_file.read(),
        backend=default_backend()
    )
```

Listing 1: Read keys

2- Format the user-id to made him have 5 characters, it's used to format all the tags, incluiding the end one.

```
# Add padding to user_id to made him have 5 characters
def format_userid(user_id):
    return (b'\x00' * (5-len(user_id)) + user_id)
```

Listing 2: Format id's

3- This function is used to find the public key asigned to a user with a specific id, this iterates over the list of public keys called "pubkey-dictionary".

```
# Find public key by Id
def findPublicKeyById(id):
```

```
        pkey = pubkey_dictionary.get(id)
        if pkey is not None:
            return serialization.load_ssh_public_key(
                ('ssh-rsa ' + pkey).encode('ascii'),
                backend=default_backend()
            )
        else:
            return pkey
```

Listing 3: Find the public key of an user.

4- This pair of functions implements the AESGCM cipher, the first one encrypts a plaintext with a key and the second one decrypts a ciphertext with a key.

```
# Encrypt a plaintext with a key using AESGCM cipher
def encrypt_aesgcm(key, plaintext):
    aesgcm = AESGCM(key)
    nonce = key
    ciphertext = aesgcm.encrypt(nonce, plaintext, None)
    return ciphertext

# Decrypt a plaintext with a key using AESGCM cipher
def decrypt_aesgcm(key, ciphertext):
    aesgcm = AESGCM(key)
    nonce = key
    plaintext = aesgcm.decrypt(nonce, ciphertext, None)
    return plaintext
```

Listing 4: AESGCM cipher

5- This pair of functions implements the RSA cipher, the first one encrypts a plaintext with a given public key and the second one decrypts a ciphertext with a private key, in this case only my private key.

```
# Encrypt a plaintext with a key using RSA cipher
def encrypt_rsa(pub_key, plaintext):
    ciphertext = pub_key.encrypt(
        plaintext,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return ciphertext

# Decrypt a plaintext with a key using RSA cipher
def decrypt_rsa(ciphertext):
    plaintext = private_key.decrypt(
        ciphertext,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return plaintext
```

Listing 5: RSA cipher

6- Hybrid encryption and decryption.

```
# Encrypt a text using hybrid encryption, first AESGCM to cipher
# the text and then RSA to cipher the key used in AESGCM.
def encrypt_hybrid(pub_key, plaintext):
    key = AESGCM.generate_key(bit_length=128)
    c_aesgcm = encrypt_aesgcm(key, plaintext)
    c_rsa = encrypt_rsa(pub_key,key)
    return c_rsa + c_aesgcm
# Decrypt a ciphertext using hybrid encryption, first RSA to recover
# the AESGCM key and then AESGCM to recover the plaintext.
def decrypt_hybrid(ciphertext):
    key_length = private_key.key_size // 8
```

```
    cipher_k = ciphertext[:key_length]
    cipher_plaintext = ciphertext[key_length:]
    key = decrypt_rsa(cipher_k)
    plaintext = decrypt_aesgcm(key, cipher_plaintext)
    return plaintext
```

Listing 6: Hybrid encryption

7- It uses the hybrid cipher and nested encrypt over the relays of the path given.

```
# Encrypt nested throught the path using hybrid cipher
def encrypt_nested_hybrid(path, plaintext):
    # Revealed sender and end tags
    m_end = format_userid(b"end") + format_userid(b"hnf") + format_userid(plaintext)
    pkey_last = findPublicKeyById(path[-1])
    c = encrypt_hybrid(pkey_last,m_end)
    # Iterate through relays in reverse order and encrypt with their public keys
    for i in range(len(path) - 2, 0, -1):
        pkey_i = findPublicKeyById(path[i])
        c = path[i+1] + c
        c = encrypt_hybrid(pkey_i, c)
    return c
```

Listing 7: Encrypt nested

The second one is **mqtt-listener.py.** Here is the implementation of the listener, it keeps listening for a message, and when something is received, it decrypts and checks if I'm the final destination. If I'm not, I send the message to the next hop of the path. As you can see comented on the code, it creates a client and then keeps listening for any message directed to the client's suscribed topic.

```
import paho.mqtt.client as mqtt
import tor
import mqtt as mqtt_client

MQTT_SERVER = "18.101.47.122"
MQTT_USER = "sinf"
MQTT_PASSWORD = "HkxNtvLB3GC5GQRUWfsA"
MQTT_PORT = 1883
MQTT_KEEPALIVE = 60
MQTT_TOPIC = "hnf"

# It gets the first 5 bytes of a plaintext and ignores the 0 byte \x00
def split_received_message(plaintext):
    return plaintext[:5].strip(b'\x00').decode('ascii')

# When a message is received
def on_message(client, userdata, msg):
    # It decrypts using hybrid decryption
    plaintext = tor.decrypt_hybrid(msg.payload)
    # Check if the first 5 bytes are equal to "end"
    source = split_received_message(plaintext[:5])
    # The encrypted message to be sent if i'm not the destination
    message = plaintext[5:]
    if source == "END" or source == "end":
        # Same as above, if we are the destination, we get the source, that will be in
    the next 5 bytes
        source = split_received_message(plaintext[5:])
        # Getting the original message
        message = plaintext[10:].decode('ascii')
        print("From source: " + str(source))
        print("Message: " + message)
    else:
        # If I'm not the destination, i send it to the next hope
        print("Sending message to: " + str(source))
        client.publish(str(source), message)

# Configuration of the client
def mqtt_client(server, port, topic, user, password, keepalive):
    client = mqtt.Client()
    client.on_message = on_message
    client.username_pw_set(user, password)
    client.connect(server, port, keepalive)
```

```
    client.subscribe(topic)
    return client
# It keeps listening while it receives any key from terminal
def listen():
    while True:
        client.loop_start()
        leave = input("\nPresiona una tecla para salir: \n")
        if leave != "":
            client.loop_stop()
            break

# Create the MQTT client
client = mqtt_client(MQTT_SERVER, MQTT_PORT, MQTT_TOPIC, MQTT_USER, MQTT_PASSWORD ,
    MQTT_KEEPALIVE)

# Keeps listening
listen()
```

Listing 8: Python example

The third and the last one is **mqtt.py.** It's just to send messages, it creates a client and configurates the conection, after that, the message is sent to a path given.

```
import paho.mqtt.client as mqtt
import tor

MQTT_SERVER = "18.101.47.122"
MQTT_USER = "sinf"
MQTT_PASSWORD = "HkxNtvLB3GC5GQRUWfsA"
MQTT_PORT = 1883
MQTT_KEEPALIVE = 60
MQTT_TOPIC = "hnf"

# Configuration of the client
def mqtt_client(server, port, topic, user, password, keepalive):
    client = mqtt.Client()
    client.username_pw_set(user, password)
    client.connect(server, port, keepalive)
    client.subscribe(topic)
    return client

# Create the MQTT client
client = mqtt_client(MQTT_SERVER, MQTT_PORT, MQTT_TOPIC, MQTT_USER, MQTT_PASSWORD ,
    MQTT_KEEPALIVE)

cipher_message = b"Probando"
# The path to send the message, the first relay have to be me always, it can't not be
    ommited due the implementation
path = ["hnf", "hnf", "hnf"]
# Implements the nested encryption with a path given
encrypted_to_send = tor.encrypt_nested_hybrid(path,cipher_message)
# Publishing a message on the topic of the FIRST RELAY, not the last one.
result = client.publish("hnf", encrypted_to_send)
```

Listing 9: Python example

Tests.

Before all, you need to install the libraries from **requirements.txt**., to do that, just run pip install requirements.txt.

In order to demonstrate the correct functioning of the code, I'll execute some test. First one is with the message ("Probando"), that will be the same in all cases, and the path ["hnf", "hnf", "hnf"].
Second one is with the path ["hnf", "hnf"].
Third one, just to test a long path ["hnf", "hnf","hnf", "hnf","hnf", "hnf","hnf", "hnf","hnf", "hnf","hnf", "hnf","hnf", "hnf","hnf", "hnf","hnf", "hnf"].

As you can see in the image 1, the 3 test have been executed succesfully. Message sending is represented in the upper terminal and message receiving in the lower one.

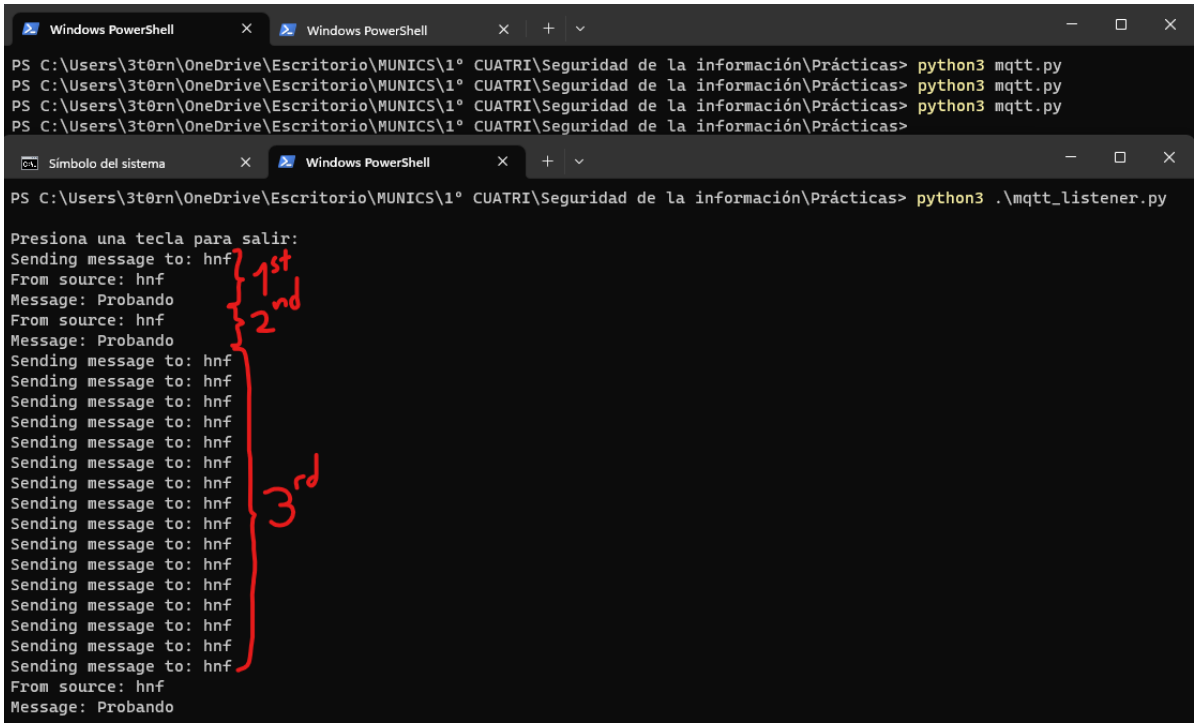

Figure 1: Tests local.

Now, I'll test my code with my classmate Daniel. Firstable, I will use him as a relay of my path 2:

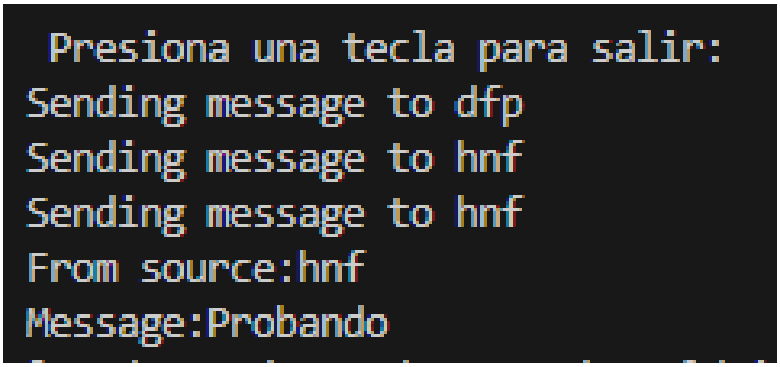

Figure 2: Classmate as a relay.

Then, I ask him to send me a message, it works good again 3:

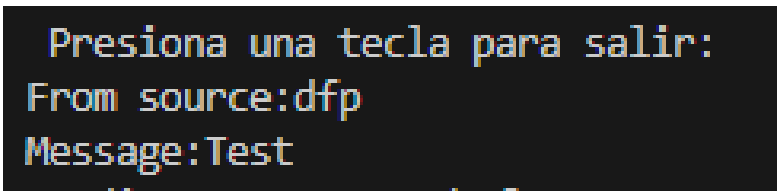

Figure 3: Receiving classmate's message.

I will try to send a message to him now, and it also works 4:

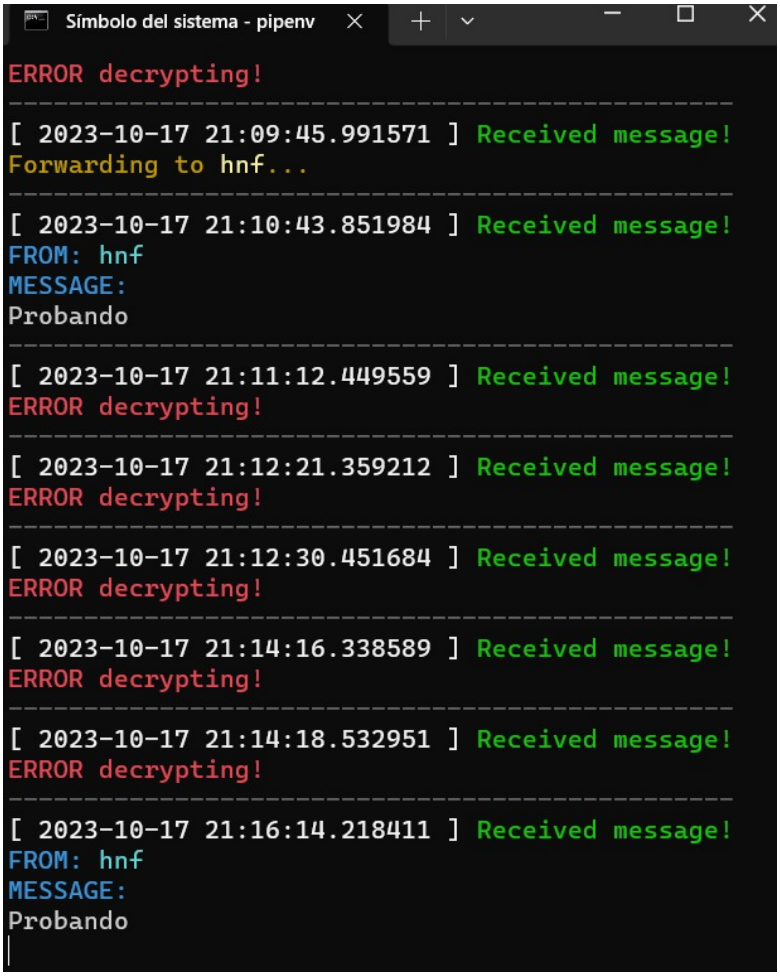

Figure 4: Sending message to classmate.

Note: the failed decryptions were because I was sending the message to the final relay's topic. So, the first succesfully decrypt was because the path was just me and Daniel. Then, when I was trying to send a message to this path = ["hnf","hnf","dfp"], the first publish was sended to the last relay's topic, not to the next one in the path. I fixed that and now it works properly.

Finally, to test everything, i will test an anonym message, and it works too 5::

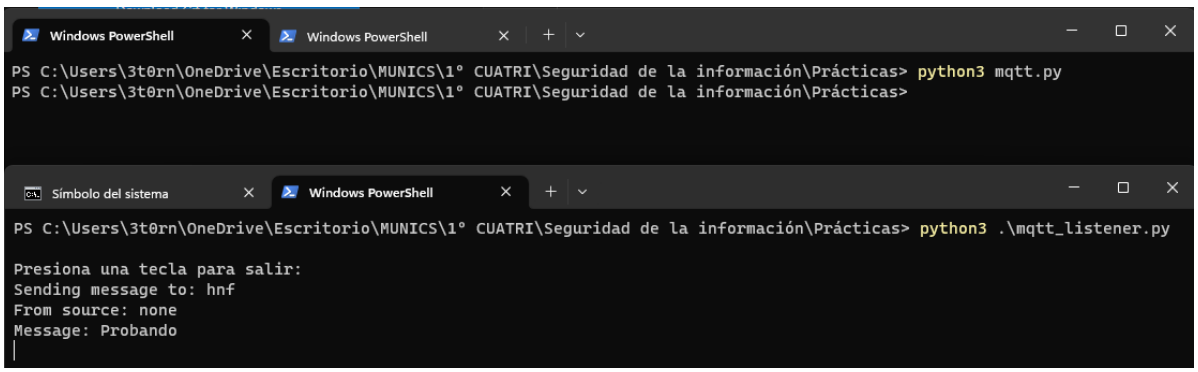

Figure 5: Anonym sender.