MÁSTER INTERUNIVERSITARIO EN CIBERSEGURIDAD

Information Security
Lab assignment I

1 Problem Description

We ask you in this lab assignment for a direct inmplementation of **nested encryption and onion routing**. You have to write the code to encrypt, send, receive, and decrypt arbitrary messages via nested encryption by using a MQTT message broker as intermediate forwarder between agents. More specifically,

1. Generate a pair of public-private keys for your program with the command `ssh-keygen`
2. Choose a short user-id as your send/recive identity (three or four letters), and inform the instructor of that user-id and of your public key created at step 1.
3. Write the code for sending and receiving messages with nested encryption and onion routing. To that end:
   - For sending a message to a recipient r, choose first a route $(s, h_1, h_2, \ldots, h_k, r)$ where s is your user-id, h_i is the user-id of the i-th intermediate hop, and r is the recipient's user-id. This route can be arbitrarily long, it can contain loops, and can be random if you wish.
   - Use nested encryption as seen in the lecture to compose the encrypted message. Let m be the plaintext you wish to send. Then implement the following simple algorithm:

Algorithm 1 Nested hybrid encryption.

Input: a path $p = (s, h_1, h_2, \ldots, h_n)$ from source s to recipient $r = h_n$; the set $\mathcal{K} = \{pk_1, pk_2, \ldots, pk_n\}$ of public keys for the relays h_i; plaintext m
Revealed sender: $m \leftarrow (s, m)$
Or anonymous sender: $m \leftarrow (\mathsf{none}, m)$
End-of-path marker: $m' \leftarrow (\mathsf{end}, m)$
$c \leftarrow \textsc{Encrypt}(pk_n, m')$
for $i = n - 1$ to 1 **do**
    $c \leftarrow \textsc{Encrypt}\big(pk_i, (h_{i+1}, c)\big)$
end for
procedure $\textsc{Encrypt}(pk, \mathsf{bytes})$
    $k \leftarrow \mathcal{R}$ ▷ Generate a random key k (128 bits)
    $c \leftarrow (E_{\text{pub}}(pk, k), E_{\mathsf{AESGCM}}(k, \mathsf{bytes}, iv = k)$ ▷ Encrypt the opaque bytes with the random key ▷ The initialization vector iv is equal to the key k
end procedure

   - Send the ciphertext c through the MQTT server to the first relay h_1
   - Receive messages directed to you by reading messages from the MQTT server in the channel tagged with your user-id. For each message, apply the relay or decode algorithm (Algorithm 2). Note that the recipient gets a message formatted as (end, s, m) where s may be equal to `none`. Also, note the following properties of the encoding, relaying, and decoding algorithms: (i) the length of the path is not revealed to any intermediate node, only the destination is able to recognize that forwarding ceases at that point; (ii) the length of the

Algorithm 2 Decode & relay messages at node h.

Input: a message $c_h = (c_{1h}, c_{2h})$; private key sk_h
Get the symmetric key: $k \leftarrow D_{\text{priv}}(sk_h, c_{1h})$
Decrypt message: $(\text{next-hop}, c_{h+1}) \leftarrow D_{\mathsf{AESGCM}}(k, c_{2h}, iv = k)$
if next-hop $\neq$ `end` **then**
    Forward c_{h+1} to the `next-hop` through MQTT broker
else
    Return $c_{h+1} = (s, m)$
end if

plaintext is not revealed to the intermediate nodes. As an exercise, think of this question: is this consistent with the general definition of semantic security?

4. Test the code with your classmates and play fun.

2 Key generation, encryption and decryption

You do not need to understand the inner working of key generation, encryption and decryption. We will cover in full detail all the background material in the following lectures during the course. The snippet below shows a possible implementation in Python of these operations just for purposes of illustration. Install first the Python library `cryptography`

```
# Generating a key
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
private_key = rsa.generate_private_key(
       public_exponent=65537,
       key_size=2048,
       backend=default_backend()
   )
public_key = private_key.public_key()

# Storing the keys
from cryptography.hazmat.primitives import serialization
pem = private_key.private_bytes(
       encoding=serialization.Encoding.PEM,
       format=serialization.PrivateFormat.PKCS8,
       encryption_algorithm=serialization.NoEncryption()
   )
with open('private_key.pem', 'wb') as f:
   f.write(pem)

pem = public_key.public_bytes(
       encoding=serialization.Encoding.PEM,
       format=serialization.PublicFormat.SubjectPublicKeyInfo
   )
with open('public_key.pem', 'wb') as f:
   f.write(pem)

# Reading the keys back in (for demonstration purposes)
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
with open("private_key.pem", "rb") as key_file:
       private_key = serialization.load_pem_private_key(
           key_file.read(),
           password=None,
           backend=default_backend()
       )
with open("public_key.pem", "rb") as key_file:
```

```
        public_key = serialization.load_pem_public_key(
            key_file.read(),
            backend=default_backend()
        )

# RSA-OAEP Encrypting and decrypting
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding

message = b'encrypt me!'
encrypted = public_key.encrypt(
        message,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
original_message = private_key.decrypt(
        encrypted,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

# AES-GCM encryption and decryption
import os
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
data = b"a secret message"
key = AESGCM.generate_key(bit_length=128)
aesgcm = AESGCM(key)
nonce = key
ciphertext = aesgcm.encrypt(nonce, data, None)
aesgcm.decrypt(nonce, ciphertext, None)
```

3 Explanations

- The MQTT server IP address is `18.101.47.122`. Messages can be read/written with the user `sinf` (the password will be posted in the course website). The Python library paho-mqtt can be used to write easily the code for publishing and subscribing to messages. The directory with user-ids and public keys will be available once you start the implementation. Check Moovi.

- user-ids should not exceed 5 bytes; the format (h, c) must use the first 5 bytes for holding the user-id h.

- The special user-id `none` can be used as sender address when you want to send messages anonymously.

- The suggested code snippet uses a padding scheme called OAEP (Optimal Asymmetric Encoding Padding) that is the recommended way for encryption. However, with OAEP, the length in bytes of the plaintext is given by

$$m = k - 2 \cdot h - 2$$

  where k is the number of bytes of the key and h is the number of bytes of the hash tags. Since $h = 32$ in our case (`SHA256`), we get $m = 190$ bytes. To overcome this restriction, we use *hybrid encryption*: public key encryption is used to set a single-use random symmetric key of 128 bits between the sender and each intermediate relay, and next the message content is encrypted with that symmetric key using the AES algorithm GCM. Thus, the messages transmitted to relay h_i

have the form $c_i = (ek, s_i)$ where ek is the symmetric key k encrypted with `RSA-OAEP`, and s_i is the symmetric encryption of (h_{i+1}, c_{i+1}) with `AES-GCM` and key k. The latter can have an arbitrary length. Incidentally, this is exactly the approach used in TLS: the client and server first exchange and agree securely on a secret key using a protocol based on public key cryptography, and next the session data is encrypted with a symmetric block cipher. TLS does not use nested encryption, however.