Assignment 2 - Public Key Distribution Authority

Aditya Kumar 2018MCS2013
Pushkar Singh 2018MCS2016

2019/02/28

1 Introduction

In this Assignment, we have simulated a Public Key Distribution Authority (PKDA) in python, using socket programming. The entire simulation consists of two clients, which rely on the PKDA to get each other's public key for communication that is encrypted using RSA.

In this document, a breakdown of how the code written by us which implements this is explained in some detail.

2 Modules

There are 2 main modules involved in the simulation as described above. These are

- Client
- PKDA

2.1 Client

Both communicating clients are represented by the same python file `client.py`. Clients can run in listen mode, where they wait for connections before starting communication, or connect mode, where they actively connect to the other client, which is listening.

In order to ensure that there are is no blockage on the main thread, both `sendMessage` and `sendMessage` functions are dispatched to separate threads, as shown:

```
t1 = threading.Thread(target=receiveMessage, args=[peer])
t1.start()
t2 = threading.Thread(target=sendMessage, args=[peer])
t2.start()
```

`sendMessage` and `receiveMessage` both simply implement the `send` and `recv` methods provided by the `socket` module.

2.2 PKDA

The Public Key Distribution Authority is represented by another python file called `pkda.py`. This code is set to always listen to connections, and when a client does connect, it checks the data, and depending on who's key is asked, returns that client's private key. This message is encrypted by the server's personal private key.

The encryption algorithm used is RSA, which is used from the `Crypto` module available for python.

```python
while True:
    client,address = s.accept()
    requestforKey = client.recv(2048)
    request = requestforKey
    clientId = int(request[0])
    if clientId == 1:
        Key = str(publicKeyA) + ";" + str(request)
    else:
        Key = str(publicKeyB) + ";" + str(request)
    Key = encrypt(privateKeyKdc,Key.encode('utf-8'))
    client.send(Key)
```

3 Command Flow

The flow of commands when using a Public Key Distribution Authority (PKDA) is fairly simple, if a bit verbose. The flow begins with one of the clients wanting to communicate securely with one of its peers. Since the client (say Alice) does not have the public key of its peer (say Bob), and for this Alice queries the PKDA for Bob's public key. The PKDA returns Bob's public key to Alice, and this response is encrypted by its own private key, to ensure that a spoofing attack isn't possible. It is assumed that both Alice and Bob have PKDAs public key accessible to them.

Alice now sends Bob a message with her own ID (in our case, the port number) and a nonce, encrypted by Bob's public key. Bob decrypts this and sees that he does not have Alice's public key. Just like earlier, Bob queries the PKDA for Alice's public key, and the PKDA returns this, encrypted by its own private key. Bob now replies to Alice with a function (usually +1) applied on the nonce, encrypted by Alice's public key.

At this point, both Alice and Bob have each other's public key, which means that the PKDA's role in the communication is complete, and Alice and Bob can start conversing in an encrypted manner.

4 Conclusion

We have, in python, implemented a PKDA and two clients which successfully perform the actions expected of them. The simulation is created in one machine, using socket programming, but supports the same simulation across a network too, without any changes.

There are two main assumptions taken in this program. The first is that Bob, Alice and PKDA know of each other's addresses. This is mostly a nominal assumption to make, since for most networks, communication is between machines where the address is essentially the identity of a machine. The second assumption is a little more substantial, which is the assumption that both Alice and Bob are aware of the PKDA's public key. This assumption can be reduced to that of only Alice knowing PKDA's public key, but the only reason that spoofing attacks aren't possible is because of this constraint. This means that this is a valid concern which needs addressing in order for this flow to be completely secure.