Project: Object Security

Axel Holmqvist, tfr14ah2
Fredrik Magnusson, fr2231ma-s
Group 10

September 2020

1 Introduction

This purpose of this project was to get an understanding of object security and how you can make the communication between different *Internet of Things* (IoT) devices more secure. We got the task to implement the secure connection between two parties, in our case a client and a server. The connection had to provide confidentiality, integrity and replay protection. It was also important that it provided perfect forward secrecy. The implementation also had to utilize UDP as the transport protocol.

This short report describes how the task was implemented, what types of resources that were used and a final conclusion where the result is discussed.

2 Architectural overview

The communication channel was done using Python and some external libraries such as *cryptography* and *Crypto*, which took care of the encryption and key exchange algorithms. To be able to communicate the parties had to use UDP as the choice of transport protocol. Python comes with a *socket* module which helped establish a UDP connection.

In terms of security, Elliptic-curve Diffie-Hellman (ECDH) was used as the key agreement protocol. Diffie-Hellman, with ephemermal keys, provides perfect forward secrecy and the shared secret is later used to encrypt and decrypt the messages that are sent over the insecure line. The decision to utilize Elliptic curve (EC) instead of for example RSA was made since the key size is smaller in EC. A smaller key size means less transmitted data which is desirable when working with object security. Also, key generation is faster with EC than RSA, and since we made the shared key *ephemeral*, meaning that it is changed for each session, it is important with a fast and energy-efficient key exchange. Finally, the key is derived using HKDF which is based on HMAC using SHA-256.

Once the handshake is complete and the two parties have their own shared secret they can start sending messages to each other (after the client has authenticated itself). In our case the client is allowed to send one message to server after this process, then the session is shutdown. This can of course be expanded with a simple while loop for receiving more than one message, and threading for receiving from many clients at the same time.

The messages are encrypted using *Advanced Encryption Standard* (AES) with CFB mode which is considered to be a safe symmetric encryption standard.

2.1 Functions

In this section we will briefly address each function used in our implementation. The implemented functions includes:

`start_server() / start_session():`

- Waits for a 'Hello' to be received/sent from/to socket.
- Initializes handshake.
- (Authentication)
- Receives and decrypts data / Encrypts and sends data.

`ecdh_handshake():`

- Generates a private and a public key on both server and client side.
- Exchanges public keys between server and client.
- Generates shared keys on both sides (from private keys and the shared public keys)
- Derives the shared keys.

`encrypt():`

- Generates an initialization vector.
- Chooses encryption suite.
- Encrypts and encodes the message before it is sent to server.

`decrypt():`

- Chooses the decryption suite.
- Decrypts and decodes the message.

`send():`

- Utilizes encrypt() to generate IV and encrypt message.
- Sends IV.
- Sends encrypted message.

`receive():`

- Receives IV.
- Receives encrypted message.
- Utilizes decrypt() to decrypt message.

`(simple_authentication():`

- Utilizes receive() to receive and decrypt the username and password sent from the client.

- Checks if the information matches ”admin” and ”supersafepw”.
- Let’s client send one message if information is accurate.)

3 Evaluation

The finished implementation satisfies all the requirements for a secure communication. With the help of ECDH with ephemeral key that are in a compressed point format, the information sent is both safe and lightweight. The choice of utilizing elliptic curves instead of for example RSA is also more suitable for IoT devices, which is in a way the scope of this project. This is because the keys are shorter and generated faster, thus resulting in less transmitted data and a faster process, which are the principles of object security. No message in our solution is allowed to be greater than 64 bytes, and does not have to be due to what was just mentioned. Integrity and confidentiality was achieved using encryption, in our case AES with a 32-byte shared key. The implementation is also resistant to replay attacks since the keys are ephemeral, meaning that they are refreshed each time a message is sent, which results in perfect forward secrecy.

4 Conclusion

A secure, and small enough, communication has been established. This would (on a very basic level) work for a IoT device.

5 Appendix

5.1 Log print

```
(root) Axels-MacBook-Pro-3:ObjectSecurity axelholmqvist$ python server.py

Starting server...
Server is up and running!

[127.0.0.1:59440]: Hello

Initializing handshake...
Public key sent to client
Public key received from client
Calculating shared key...
Generating derived key...
Handshake is finished!

127.0.0.1:59440 successfully authenticated

[127.0.0.1:59440]: Hi IoT-device, do you want to hear a secret?
```

```
(root) Axels-MacBook-Pro-3:ObjectSecurity axelholmqvist$ python client.py

Starting client...
Client is up and running!

Sent to server: Hello

Handshake started...
Public key received from server
Public key sent to server
Calculating shared key...
Generating derived key...
Handshake is finished!

Username: admin
Password: supersafepw

[SERVER]: Successful authentication

Safe message to server: Hi IoT-device, do you want to hear a secret?
```

5.2 Run program

5.2.1 Libraries

$ pip install cryptography to install *cryptography*

$ pip install crypto to install *Crypto*

5.2.2 Commands

$ python server.py to start server.

$ python client.py to start client (which immediately sends a 'Hello' to the server and starts a session).

5.2.3 "Authentication"

Username: admin **Password:** supersafepw

5.3 Listing of the code

5.3.1 server.py

```
import socket
import time
import threading
import pickle
import random
import string
import base64
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.serialization import Encoding, PublicFormat
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import hashes
from Crypto.Cipher import AES

""" Server socket values """
UDP_IP = "127.0.0.1"
UDP_PORT = 5004
BUFFER_SIZE = 64

admin_user = ['admin', 'supersafepw']

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((UDP_IP, UDP_PORT))

def ecdh_handshake(client_ip, client_port):
    """
    Function that manages the handshake performed with the ECDH algorithm.
    Returns the derived key (shared secret)
    """
    print("\nInitializing handshake...")
    time.sleep(1)

    server_private_key = ec.generate_private_key(ec.SECP384R1, default_backend())
    server_public_key = server_private_key.public_key()
    encoded_server_public_key = server_public_key.public_bytes(Encoding.X962, PublicFormat.Compres

    print("Public key sent to client")
    sock.sendto(encoded_server_public_key, (client_ip, client_port))
```

```
    time.sleep(1)

    print("Public key received from client")
    client_public_key, _ = sock.recvfrom(BUFFER_SIZE)
    #print(client_public_key)

    print("Calculating shared key...")
    shared_key = server_private_key.exchange(ec.ECDH(), ec.EllipticCurvePublicKey.from_encoded_poi
    #print(shared_key)

    print("Generating derived key...")
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data',
    ).derive(shared_key)

    print("Handshake is finished!")
    #print(derived_key)

    return derived_key

def aes_encrypt(message, key):
    """
    Function responsible for encrypting the messages using AES.
    Returns the initialization vector and encrypted message.
    """
    iv = ''.join(random.choice(string.ascii_uppercase + string.ascii_lowercase + string.digits) fo
    encoded_iv = iv.encode('utf-8')

    encryption_suite = AES.new(key, AES.MODE_CFB, iv)
    encrypted_message = encryption_suite.encrypt(message)
    encoded_encrypted_message = base64.b64encode(encrypted_message)

    return encoded_iv, encoded_encrypted_message

def aes_decrypt(encrypted_message, key, iv):
    """
    Function that decrypts the messages received from the client.
    Returns the decrypted and decoded message.
    """
    decryption_suite = AES.new(key, AES.MODE_CFB, iv)
    decrypted_message = decryption_suite.decrypt(base64.b64decode(encrypted_message))
    return decrypted_message.decode()

def simple_authentication(derived_key, client_ip, client_port):
    username = receive(derived_key)
```

```
    password = receive(derived_key)

    if username == admin_user[0] and password == admin_user[1]:
        send("Successful authentication", derived_key, client_ip, client_port)
        print(f"\n{client_ip}:{client_port} successfully authenticated")
        return True
    else:
        send("Unsuccessful authentication", derived_key, client_ip, client_port)
        print(f"\n{client_ip}:{client_port} unsuccessfully authenticated")
        return False

def send(message, derived_key, client_ip, client_port):
    """
    Function for encrypting and sending a message (+ the iv).
    """
    iv, encrypted_message = aes_encrypt(message, derived_key)
    sock.sendto(iv, (client_ip, client_port))
    sock.sendto(encrypted_message, (client_ip, client_port))

def receive(derived_key):
    """
    Function for receiving and decrypting an encrypted message (+ the iv).
    Returns the decrypted message.
    """
    iv, _ = sock.recvfrom(BUFFER_SIZE)
    encrypted_message, _ = sock.recvfrom(BUFFER_SIZE)
    decrypted_message = aes_decrypt(encrypted_message, derived_key, iv)
    return decrypted_message

def start_server():
    """
    Starts the server and the session. The server listens for a 'Hello' from the client.
    The session consist of the handshake and receiving a message (if authenticated).
    """
    print("\nStarting server...")
    time.sleep(1)
    print("Server is up and running!")

    hello_message, address = sock.recvfrom(BUFFER_SIZE)
    client_ip = address[0]
    client_port = address[1]
    print(f"\n[{client_ip}:{client_port}]: {hello_message.decode()}")
    derived_key = ecdh_handshake(client_ip, client_port)

    if simple_authentication(derived_key, client_ip, client_port):
        iv, _ = sock.recvfrom(BUFFER_SIZE)
        encrypted_message, _ = sock.recvfrom(BUFFER_SIZE)
        decrypted_message = aes_decrypt(encrypted_message, derived_key, iv)
```

```
        print(f"\n[{client_ip}:{client_port}]: {decrypted_message}")
        print("\n")
    else:
        pass

start_server()
```

5.3.2 client.py

```
import socket
import time
import threading
import pickle
import random
import string
import base64
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.serialization import Encoding, PublicFormat
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import hashes
from Crypto.Cipher import AES

""" Client socket values """
UDP_IP = "127.0.0.1"
UDP_PORT = 5004
BUFFER_SIZE = 64
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

def ecdh_handshake():
    """
    Function that manages the handshake performed with the ECDH algorithm.
    Returns the derived key (shared secret)
    """
    print("\nHandshake started...")

    client_private_key = ec.generate_private_key(ec.SECP384R1, default_backend())
    client_public_key = client_private_key.public_key()
    encoded_client_public_key = client_public_key.public_bytes(Encoding.X962, PublicFormat.Compres

    print("Public key received from server")
    server_public_key, _ = sock.recvfrom(BUFFER_SIZE)
    #print(server_public_key)

    print("Public key sent to server")
    sock.sendto(encoded_client_public_key, (UDP_IP, UDP_PORT))
```

```
    time.sleep(1)

    print("Calculating shared key...")
    shared_key = client_private_key.exchange(ec.ECDH(), ec.EllipticCurvePublicKey.from_encoded_poi
    #print(shared_key)

    print("Generating derived key...")
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data',
    ).derive(shared_key)

    print("Handshake is finished!")
    #print(derived_key)

    return derived_key

def aes_encrypt(message, key):
    """
    Function responsible for encrypting the messages using AES.
    Returns the initialization vector and encrypted message.
    """
    iv = ''.join(random.choice(string.ascii_uppercase + string.ascii_lowercase + string.digits) fo
    encoded_iv = iv.encode('utf-8')

    encryption_suite = AES.new(key, AES.MODE_CFB, iv)
    encrypted_message = encryption_suite.encrypt(message)
    encoded_encrypted_message = base64.b64encode(encrypted_message)

    return encoded_iv, encoded_encrypted_message

def aes_decrypt(encrypted_message, key, iv):
    """
    Function that decrypts the messages received.
    Returns the decrypted and decoded message.
    """
    decryption_suite = AES.new(key, AES.MODE_CFB, iv)
    decrypted_message = decryption_suite.decrypt(base64.b64decode(encrypted_message))
    return decrypted_message.decode()

def send(message, derived_key, client_ip, client_port):
    """
    Function for encrypting and sending a message (+ the iv).
    """
    iv, encrypted_message = aes_encrypt(message, derived_key)
    sock.sendto(iv, (client_ip, client_port))
```

```
    sock.sendto(encrypted_message, (client_ip, client_port))

def receive(derived_key):
    """
    Function for receiving and decrypting an encrypted message (+ the iv).
    Returns the decrypted message.
    """
    iv, _ = sock.recvfrom(BUFFER_SIZE)
    encrypted_message, _ = sock.recvfrom(BUFFER_SIZE)
    decrypted_message = aes_decrypt(encrypted_message, derived_key, iv)
    return decrypted_message

def start_session():
    """
    Starts a new session with sending a 'Hello' to the server.
    Each session starts with a handshake. After the handshake is complete,
    encrypted messages are sent to the server.
    """
    print("\nStarting client...")
    time.sleep(1)
    print("Client is up and running!")
    hello_message = b"Hello"

    sock.sendto(hello_message, (UDP_IP, UDP_PORT))
    print(f"\nSent to server: {hello_message.decode()}")
    derived_key = ecdh_handshake()

    send(input("\nUsername: "), derived_key, UDP_IP, UDP_PORT)
    send(input("Password: "), derived_key, UDP_IP, UDP_PORT)
    authentication_status = receive(derived_key)

    print(f"\n[SERVER]: {authentication_status}")
    if authentication_status == 'Successful authentication':
        send(input("\nSafe message to server: "), derived_key, UDP_IP, UDP_PORT)
        print("\n")

start_session()
```