# EITN50 Project 2, Object Security

Group 16, Marcus De Lacerda and Martin H Tomicic
dat11mla@student.lu.se, dat15mto@student.lu.se

September 27, 2018

## Introduction

In this project we are tasked with exploring and educating ourselves about object security, specifically by implementing a secure connection that can be used by two parties. This report explains how we went about with our implementation of the connection as well as the inherent strengths and weaknesses of our design choices.

## 1 Design Choices

In the project description a few recommendations and instructions are stated on the implementation of the connection. The specifications are followed and implemented throughout the project. The programming language used in this project is Python.

### 1.1 Classes

The design of the project is largely simple using only three classes, `Peer`, `DHPeer`, and `Protocol`. The `Peer` class is responsible for the low level connection using UDP and sockets, as well as logging certain information. The `DHPeer` extends the `Peer` class and implements both the Diffie-Hellman handshake and the object encryption. Lastly, the `Protocol` class is simply a class for enumeration of protocol headers.

The base class `Peer` works by setting up two UDP sockets, one where the socket is used as server and one where it is used as a client. A thread is used to listen for incoming messages using the server socket while the client socket is used to send messages.

`DHPeer` (Diffie-Hellman Peer) is the main part of the project. This class is where all the security is implemented. It is responsible for storing data regarding EDH as well as performing handshakes. It is also what encrypts and decrypts objects being sent. The class has one more responsibility which is packaging data. This includes splitting data being sent so it fits into 64 byte packets as well as creating the headers.

## 1.2 Security

For the purpose of providing integrity, confidentiality and replay protection, as well as working with forward security we chose to implement an ephemeral Diffie-Hellman handshake using the python library `cryptography`. This is, as previously mentioned, implemented in the class `DHPeer`. The handshake works by first setting up common parameters meaning a common prime number and one of its primitive roots. After common parameters are established, both parties generate private and public keys. The public keys are shared and from these keys the common secret is computed which is then used to encrypt the data to be sent. For every data-packet being sent the handshake is repeated.

For the object security a Python library was used, called `python-jose` which is an implementation of *JSON Object Encryption and Signing*. The object to be sent, which has to be a `dict` in python, is encrypted and decrypted using the shared secret from EDH-handshake.

## 1.3 Protocol

The protocol is quite simple. It uses a 4-byte header where the first byte signifies what type of packet is being sent. The second byte contains information about how many packets the data being sent is split up into and the ordering of said packets. The two last bytes are simply used as a session ID indicating which packets are linked to which. We will now describe the function of the first two bytes in a bit more detail. In the first byte, each bit functions as a flag, currently we only use the first 3 bits and the 8th bit as is described in the table below.

| Name | Code | Meaning |
|---|---|---|
| `BASE` | `10000000` | Setup common base parameters |
| `SECRET` | `01000000` | Setup common secret |
| `SEND` | `00100000` | Send encrypted data |
| `ACK` | `00000001` | Acknowledge |

The reason why the acknowledge is the 8th bit is that it allows the construction of e.g. `SECRET_ACK` as `SECRET` + `ACK`. Note that only the acknowledge flag is not a valid header. Also, one might add that it seems wasteful to have 4 unused bits and the reason for that is simply that we wanted to deal with whole bytes and we thought that there might be a need for additional header-types.

Now to the second byte. The reason we need this byte is that we are using UDP, we have to create all acknowledgements ourself and we have no guarantee that all our packets get transmitted. We wanted a way to check if all packets corresponding to some data had been received. So, say you have some data which gets split up into 4 packets. The first one would have its 2nd byte set to 3 (4 - 1), indicating how many packets there are in total. The following packets would count up to 2 (4 - 2), starting at 0. This gives us the swell property that the value 0 in this field means that there is only packet in this session.

## 1.4 Example Log

What follows are the printouts from an example where common parameters are shared followed by a handshake and sending an encrypted object. The first log is from the peer initiating the communication, we can call that peer **A**. The second log is from the other peer which we will call **B**. **A** starts by generating its parameters followed by sending them to **B**. After that **A** generates a private key and public key. The public key is sent to **B** which generates its own private and public key as well computes the shared secret using **A**'s public key. **B** then sends its public key back to **A** which then also computes their shared secret. The shared secret is then used to encrypt an object which **A** sends and **B** receives and decrypts. We added the printout of the objects for clarity, something that should not be include otherwise.

Listing 1: Log of peer A

```
Entry 0:        Generating parameters.
Entry 1:        Sending parameters.
Entry 2:        Generating private key.
Entry 3:        Generating public key.
Entry 4:        Sending public key.
Entry 5:        Computing shared key.
Entry 6:        Object to send: {'en': 'Hello', 'swe': 'Hej', 'pt': 'Ola'}
Entry 7:        Encrypting object.
Entry 8:        Sending object.
```

Listing 2: Log of peer B

```
Entry 0:        Received parameters.
Entry 1:        Received peer public key.
Entry 2:        Generating private key.
Entry 3:        Generating public key.
Entry 4:        Sending public key.
Entry 5:        Computing shared key.
Entry 6:        Object received.
Entry 7:        Decrypting object.
Entry 8:        Object: {u'en': u'Hello', u'swe': u'Hej', u'pt': u'Ola'}
```

## 2 Evaluation

The implementation seems to work when running locally, meaning when both server and client have the same IP. However, there is at this point little robustness, no retransmitting or such. Also, our implementation of object security makes the objects as ephemeral as the Diffie-Hellman handshake, meaning it might be ill suited for caching. But, one could easily change this behavior by e.g sending the object key separately using EDH encryption instead of using the common secret as the object key. Furthermore, the proposed protocol is both flexible (in that we can add at least 4 more header-types) and allows for checking if all packets have arrived as well as sorting them. One thing to note however is that elliptic curve DH might have been a better choice for encryption, due to its smaller key size. At this point, the keys are split up into two packets.

## 3 Conclusion

We implemented a lightweight security protocol for IoT-devices over UDP. It has room for improvements in terms of robustness and, the object security uses the EDH shared secret for encryption which might not be optimal. However the groundwork has been done and this implementation should at the very least serve as a proof of concept.