EITN50 Object security

Martin Gottlander, bte15mgo@student.lu.se
Elias Rudberg, el0032ru-s@student.lu.se
Viktor Joelsson, vi0842jo-s

September 2019

1 Introduction

Object security invokes interest as a new way of communicating securely, in the meanwhile IoT is on the rise. The purpose of this report is to investigate a possible solution by using object security as a way of secure communication between IoT devices. To test the subject in question a prototype will be developed. By discussing why and specify which security mechanism will be used the end goal is to answer the question if it can be implemented in a new series of IoT devices.

2 Architecture overview

The Elliptic Curve Diffie-Hellman Key Exchange algorithm (ECDH) was implemented mainly because the mathematical aspect that makes it possible for smaller keys in comparison with RSA. The generation of keys in ECDH also require significantly less power then the generation of keys in RSA. Both algorithm offer strong encryption, but since IoT devices (with limited process capability) is the main target for the security system smaller keys, hence, smaller packages is to be considered.

If the package sent is bigger then 64 bytes we fragment the data into smaller packets. Each packet sent has a 4 byte long header which contains

The key generation and exchange is carried out as follow:

Client side

- (1) Client generate a private key using elliptic curve
- (2) Client generate public key from the private key

Server side

- (1) Server generate a private key using elliptic curve

- (2) Server generate public key from the private key

Key exchange

- (1) Client send public key to the server
- (2) Server send public key to the client

Client/server side

Client/server use the opposite sides public key to calculate the shared key (The shared key is later used to encrypt/decrypt).

Encryption of the data is performed before the package is sent over the channel, hence, the principle of object security is fulfilled.

For encryption, the class Fernet from the recipe layer in the cryptography library has been used. Fernet use AES with CBC mode for encryption. The fernet key used for encryption is derived from the shared secret that was agreed upon during the handshake using the key derivation function HKDF. HKDF is a key derivation function which is based on HMAC using SHA256.

To ensure that no one is trying to modify messages to break the Diffie-Hellman key-agreement, a pre-shared secret was implemented. To be able to agree on a shared secret for forward secrecy the client first has to become authenticated. This is done by creating an HMAC on the first message sent from the client which can be verified with a pre-shared key. When the public key has been verified the server can continue the key agreement handshake.

We are securing ourselves against replay attacks by saving the timestamps that Fernet introduces to its encryption in a vector. We then check every packets timestamp versus that vector to see if that timestamp has been used once already. if i thas, we throw away the packet. After a set amount of time we will empty the vectors of timestamps, and then let Fernet.decrypt() take care of replay protection with it's ttl parameter. Fernet.decrypt() will check the timestamp and see if the packet is older than the ttl parameter, and if it is, throw an exception. And making the ttl parameter equal to the time interval at which we empty the vector of used timestamps we can make sure that they complement eachother, making sure that slow packets wont get denied by the ttl parameter and making sure that the vector of timestamps wont grow in size out of control.

The principle of perfect forward security can be achieved through ephemeral keys. By generating a new set of ECDH keys every session, without storing them and/or reusing them the principle is considered.

3 Evaluation of the implementation

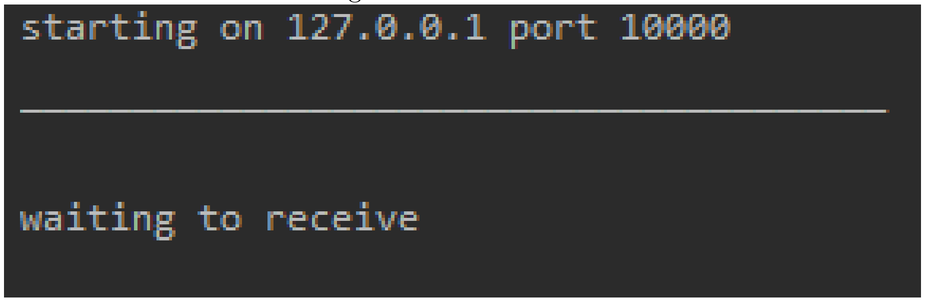
The prototype has some restrictions concerning testing different aspects. For example we have a solution for replay-attacks but since we actually have to perform an attack by sniffing packages this has not been tested. Different protection mechanism is embedded in Fernet. If Fernet suffer from vulnerabilities so will the prototype/implementation. Since the implementation concern IoT devices, the encryption SHA256 can be considered excessive. However since the lack of specification of the IoT devices it is more or less disregarded. The result of the implementation was that one package was split into several segments to adjust to the specification, max 64 bytes/package. We are missing the implementation of authentication which is supposed to be in the beginning of the handshake.

4 Conclusion

When developing security you have to some degree trust already evaluated techniques, by implementing these techniques it helps to understand why and where they are important. The implementation was a success and could be used for future IoT devices. Which encryption to use should be considered. Since the package size is important considering the limited process power. It might be considered harmless if an unwanted third party successfully decrypt a package several weeks later from when it was sent.

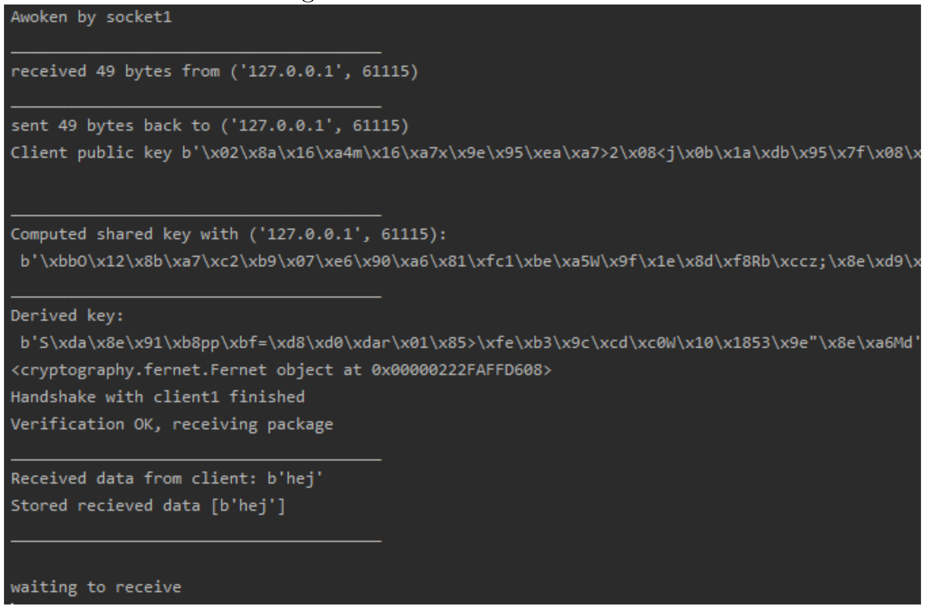
5 Appendix

Figure 1: Cache initiate



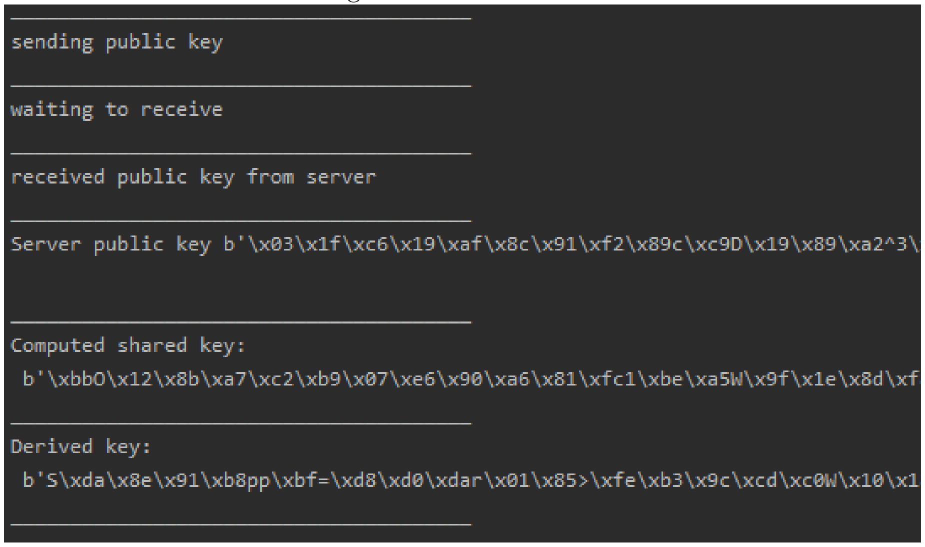
```
starting on 127.0.0.1 port 10000

________________________________________________


waiting to receive
```

Figure 2: Cache communication



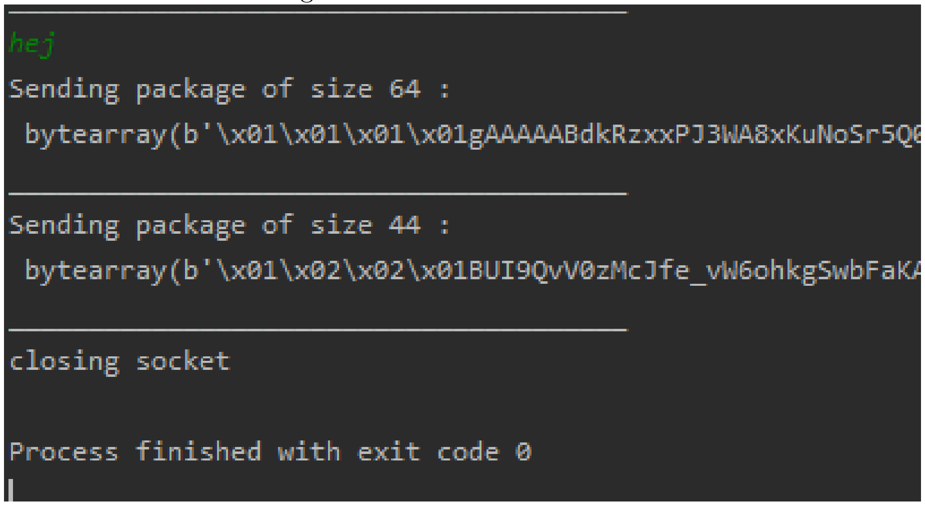
```
Awoken by socket1
_______________________________________
received 49 bytes from ('127.0.0.1', 61115)
_______________________________________
sent 49 bytes back to ('127.0.0.1', 61115)
Client public key b'\x02\x8a\x16\xa4m\x16\xa7x\x9e\x95\xea\xa7>2\x08<j\x0b\x1a\xdb\x95\x7f\x08\x


_______________________________________
Computed shared key with ('127.0.0.1', 61115):
 b'\xbbO\x12\x8b\xa7\xc2\xb9\x07\xe6\x90\xa6\x81\xfc1\xbe\xa5W\x9f\x1e\x8d\xf8Rb\xccz;\x8e\xd9\x
_______________________________________
Derived key:
 b'S\xda\x8e\x91\xb8pp\xbf=\xd8\xd0\xdar\x01\x85>\xfe\xb3\x9c\xcd\xc0W\x10\x1853\x9e"\x8e\xa6Md'
<cryptography.fernet.Fernet object at 0x00000222FAFFD608>
Handshake with client1 finished
Verification OK, receiving package
_______________________________________
Received data from client: b'hej'
Stored recieved data [b'hej']
_______________________________________

waiting to receive
```

Figure 3: Server initiate



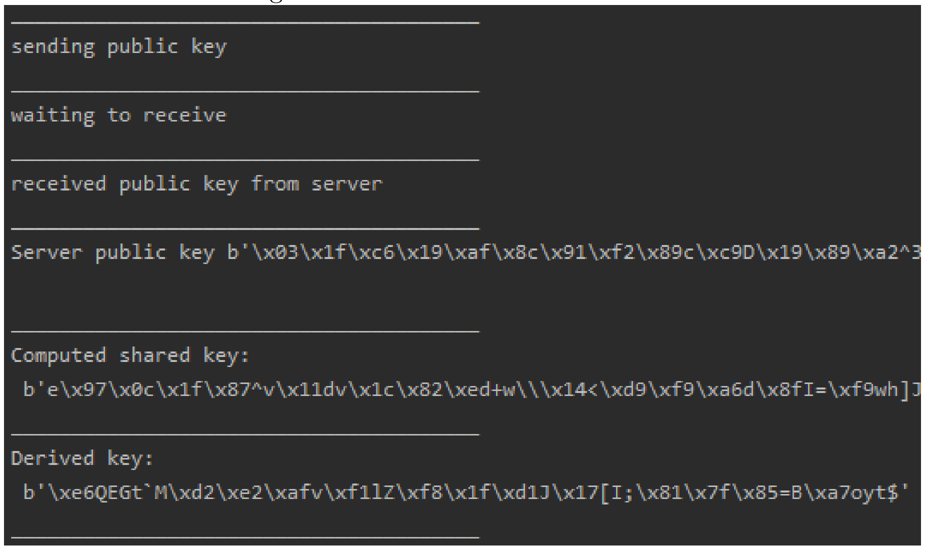
```
________________________________________
sending public key

________________________________________
waiting to receive

________________________________________
received public key from server

________________________________________
Server public key b'\x03\x1f\xc6\x19\xaf\x8c\x91\xf2\x89c\xc9D\x19\x89\xa2^3\



________________________________________
Computed shared key:
 b'\xbbO\x12\x8b\xa7\xc2\xb9\x07\xe6\x90\xa6\x81\xfc1\xbe\xa5W\x9f\x1e\x8d\xf

________________________________________
Derived key:
 b'S\xda\x8e\x91\xb8pp\xbf=\xd8\xd0\xdar\x01\x85>\xfe\xb3\x9c\xcd\xc0W\x10\x1

________________________________________
```

Figure 4: Server communication



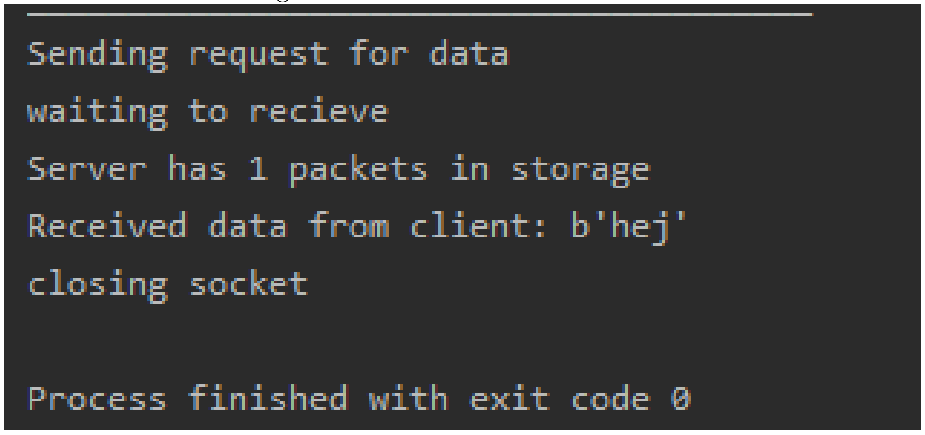
```
________________________________________
hej
Sending package of size 64 :
 bytearray(b'\x01\x01\x01\x01gAAAAABdkRzxxPJ3WA8xKuNoSr5Q

________________________________________
Sending package of size 44 :
 bytearray(b'\x01\x02\x02\x01BUI9QvV0zMcJfe_vW6ohkgSwbFaK

________________________________________
closing socket

Process finished with exit code 0
```

Figure 5: Client start communication



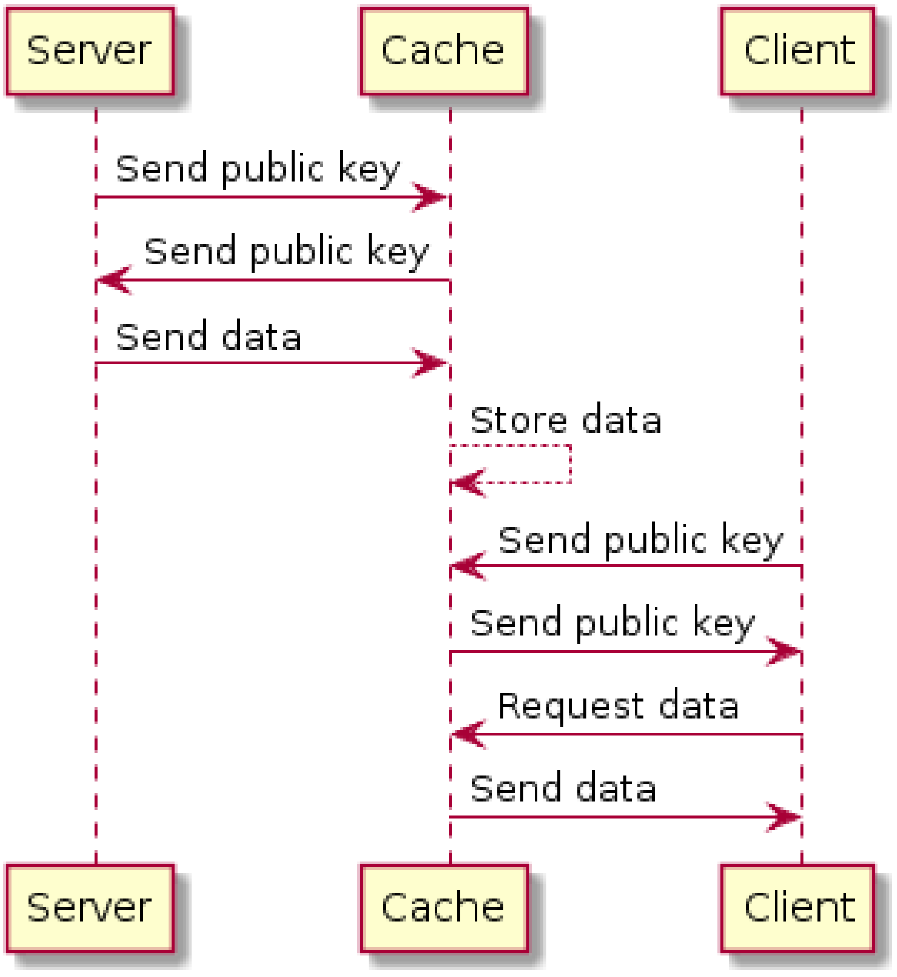
```
________________________________________
sending public key

________________________________________
waiting to receive

________________________________________
received public key from server

________________________________________
Server public key b'\x03\x1f\xc6\x19\xaf\x8c\x91\xf2\x89c\xc9D\x19\x89\xa2^3


________________________________________
Computed shared key:
 b'e\x97\x0c\x1f\x87^v\x11dv\x1c\x82\xed+w\\\x14<\xd9\xf9\xa6d\x8fI=\xf9wh]J

________________________________________
Derived key:
 b'\xe6QEGt`M\xd2\xe2\xafv\xf1lZ\xf8\x1f\xd1J\x17[I;\x81\x7f\x85=B\xa7oyt$'

________________________________________
```

Figure 6: Client end communication



```
________________________________________
Sending request for data
waiting to recieve
Server has 1 packets in storage
Received data from client: b'hej'
closing socket

Process finished with exit code 0
```

Figure 7: Sequence diagram


Server
Cache
Client
Send public key
Send public key
Send data
Store data
Send public key
Send public key
Request data
Send data
Server
Cache
Client