Introduction

This tutorial is meant to act as a help for possible inclusion in the 1st assignment. It is just an indicative approach on how to create a simple DH key agreement scheme using python. If you are considering to use such a protocol in the design and implementation of on securing the modbus transactions then the code might be of use. Also, it is a good additional exercise/tutorial for those of you that worked on python on the first four laboratories.

Initial Notes

To better understand which part was already done (in python docs and HowTos) and which part was written as an example for you, in this document the code is placed with comments that shows the approach that was followed. Note, that it is not mandatory to use every part of the code if it does not fit your proposed solution.

The Server

The imports

At the beginning we only need socketserver and we want some sort of exit (although not very elegant) from the server that listens for client requests. So for the exit we need sys. So for the server the imports are

```
import socketserver
import sys
```

main() function

As usual I used #!/usr/bin/env python3 to tell bash to call the env executable which will find python3 interpreter to read and execute the rest of the file. The file can be made executable by issuing chmod +x dh_server.py and then it can be run by simply issuing ./dh_server.py. Then, typically, we can use def main() to define a main function and the condition if __name__ == '__main__': to call main if the code is run from command line. So the code so far

```
#!/usr/bin/env python3
import socketserver
import sys
```

```
def main():
    pass
if __name__ == '__main__':
    main()
```

Python’s socketserver

Now is time to write the server code. We will use python’s socketserver. To do so we need to tell on what hostname or IP address and what port to listen to. We also need to decide whether we want to use TCP or UDP. I chose TCP as I want a reliable and in order delivery of messages. We need to then write our own request handler by overriding methods of a class defined in python’s socketserver for this purpose. Let’s call this request handler for our DH server Dh_Handler. This will do all the required work to respond to client’s requests. We do not need to worry about TCP sides of things as all of that work is done by python’s socketserver.

So the code so far

```
#!/usr/bin/env python3
import socketserver
import sys

def main():
    # choosing to listen on any address and port 7777
    host, port = '', 7777
    # create an instance of python's tcp server class, we specify which ip address or hostname
    # and what request handler to use which in this case is the one we defined as DH_Handler
    dh_server = socketserver.TCPServer((host, port), Dh_Handler)
    # we don't bother with threading and forking but we want to stop the server and shutdown the
socket
    # so we capture the KeyboardInterrupt exception
    try:
        # this will start to listen in an infinite loop
        dh_server.serve_forever()
    except KeyboardInterrupt:
        # we need to be able to stop the service, for now we don't care if our implementation is
ugly
        # we just want it to work
        dh_server.shutdown()
        sys.exit(0)

if __name__ == '__main__':
    main()
```

Now the try: and except KeyboardInterrupt is an ugly way of exiting the code (hitting Ctrl+C) which sometimes even does not properly shutdown the service but the purpose was to give a quick example so for now we put up with this ugliness.

The Request Handler

This will be a new class derived from socketserver.BaseRequestHandler. So all we have to do is to override some of the methods and leave the rest for socketserver. We need to do this as no one else knows what our server will do. So which methods should we override? Well we may need some initialisation, for instance in our case we need to read the DH MODP parameters from a file. For this we override __init__() function by defining it for our class. We can now read the file and store it in a variable for the class. We do this by using self.VARIABLE_NAME so the variable is unique for each instance. We can create a state variable to keep track of the state (although we are not running this as a multi-threaded or forking process so this is not really a concern). Then we just need to pass the variables we receive from socketserver to the socketserver.BaseRequestHandler init function. Next we will override/implement the handle method which does nothing in socketserver.BaseRequestHandler.

Where I find more information?

If you wish to gain a better understanding of the python's socketserver module check the documentations at https://docs.python.org/3.4/library/socketserver.html.

The Server

All-right we start with the server, the assumption is that DH parameters are generated and stored in a file in local directory. You need to code that part yourself or use command line to generate the parameters. The code has plenty of comments explaining the process. It is not well designed or considers much of security for handling potential errors. The goal is just to provide a very simple example of writing a network protocol to play with the DH key exchange primitives.

```
#!/usr/bin/env python3
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.kdf.hkdf import HKDF

from cryptography.hazmat.primitives.serialization import *
# instead of importing the following methods and attributes
```

```
# individually we import them all by specifying a *
#from cryptography.hazmat.primitives.serialization import load_pem_parameters
#from cryptography.hazmat.primitives.serialization import load_pem_public_key
#from cryptography.hazmat.primitives.serialization import ParameterFormat
#from cryptography.hazmat.primitives.serialization import PublicFormat
#from cryptography.hazmat.primitives.serialization import Encoding
import binascii as ba
import socketserver
import sys


def load_dh_params():
    '''
    Load DH parameters from a file which is hard coded here for simplicity
    generating DH parameters is a time consuming operation so we rather use
    generated values in practice several defined primes and generators
    are hard-coded into programs
    '''
    with open('./dh_2048_params.bin', 'rb') as f:
        # the load_pem_parameters is part of serialization which reads binary
        # input and converts it to proper objects in this case it is
        # DH parameters
        params = load_pem_parameters(f.read(), default_backend())
    print('Parameters have been read from file, Server is ready for requests ...')
    return params

def generate_dh_prvkey(params):
    '''
    Generate a random private key (and a public key) from DH parameters
    '''
    return params.generate_private_key()

def check_client_pubkey(pubkey):
    '''
    Check whether the client public key is a valid instance of DH
    shouldn't we check whether the key is valid under the parameters
    sent by the server?
    '''
    if isinstance(pubkey, dh.DHPublicKey):
        return True
    else:
        return False
```

```python
class Dh_Handler(socketserver.BaseRequestHandler):
    '''
    The request handler class for DH server

    It is instantiated once per connection to the server.
    '''

    def __init__(self, request, client_address, server):
        ''' here we do our service specific initialisation
         in this case we want to load the DH parameters
         that we have generated in advance
        '''

        # the params variable of the class will store the DH parameters
        self.params = load_dh_params()
        # current state, received a request but not handled yet
        # the state variable is made up, it helps us keep track of what is happening
        self.state = 0
        # we just pass the variables we receive to the BaseRequestHandler to do
        # whatever tcp needs to do
        socketserver.BaseRequestHandler.__init__(self, request, client_address, server)

    def handle(self):
        '''
        This function handles the requests and sends proper responses
        '''

        # we read the first message sent by the client up to 3072 bytes
        # what if the message is longer?
        # should we check the size of the message?
        self.data = self.request.recv(3072).strip()

        # here we are inventing our own protocol so say the first message sent by the client
        # must be the text Hello, this message will only be valid if we are in state 0, ready
        # and just received the first request
        if self.state == 0 and self.data == b'Hello':
            # we have received proper request and the state changes to initiated
            self.state = 1
            # we print the received data and state on the server so we could follow how things
            # work
            print(self.data, self.state)
            # now let's say the proper response in our protocol to the client's Hello message
            # is the text message Hey There!
```

```
            response = b'Hey there!'
            # here we send this out to the client
            self.request.sendall(response)
        else:
            # we have received an invalid message since we can only expect a text Hello
            # in state 0, anything else is invalid and we end the communication and return
            response = b'I do not understand you, hanging up'
            self.request.sendall(response)
            return

        # so far so good, if we get here it means we have received a proper Hello
        # and have sent a proper Hey There!
        # now is time to read the next client request
        self.data = self.request.recv(3072).strip()
        # we define the request be the text Params? and if we are in initiated state
        if self.state == 1 and self.data == b'Params?':
            # change the state to parameters requested
            self.state = 2
            print(self.data, self.state)
            dh_params = self.params
            # here we convert the parameter object to binary so we could send it over the network
            response = dh_params.parameter_bytes(Encoding.PEM, ParameterFormat.PKCS3)
            self.request.sendall(response)
        else:
            # if we get here then something was not right so we end the communication and return
            response = b'I do not understand you, hanging up'
            self.request.sendall(response)
            return

        # Ok we have come a long way, time to read the next client message
        self.data = self.request.recv(3072).strip()
        # now in our protocol we define that when the client wants to send the public key
        # it would start the message with the text "Client public key:", we check if the message
        # starts with that. We convert the received binary data to bytearray and take the first
        # 18-byte slice of it which must be our expected text. Of-course we must be in state 2
        # (although we will not get here otherwise or would we?)
        if self.state == 2 and bytearray(self.data)[0:18] == b'Client public key:':
            # now we convert the binary message to bytearray so we can choose the public key
            # part of it and use key serialization method to turn it into an object
            client_pubkey = load_pem_public_key(bytes(bytearray(self.data)[18:]),
default_backend())
            # now if the public key is loaded (we might not get to this point otherwise,
            # something for you to check!)
```

```
            if client_pubkey:
                # client key is valid so we generate our own from the parameters
                server_keypair = generate_dh_prvkey(self.params)

                # we will send the public key to the client and we need to convert it to
                # binary to send over the network
                response = b'Server public key:' + server_keypair.public_key().public_bytes(
                    Encoding.PEM, PublicFormat.SubjectPublicKeyInfo)

                # then we will calculate the shared secret
                shared_secret = server_keypair.exchange(client_pubkey)

                # and we are done back to waiting
                self.state = 0
                print(self.data, self.state)
                self.request.sendall(response)
                # we print the shared secret on the server and return
                print('Shared Secret:\n{}'.format(ba.hexlify(shared_secret)))
                return
            else:
                # if we get here the client key is not right
                response = b'Invalid client public key, hanging up'
                self.request.sendall(response)
                return

def main():
    # choosing to listen on any address and port 7777
    host, port = '', 7777
    # create an instance of python's tcp server class, we specify which ip address or
    # hostname and what request handler
    # the request handler is the one we defined as DH_Handler
    dh_server = socketserver.TCPServer((host, port), Dh_Handler)
    # we don't bother with threading and forking but we want to stop the server and shutdown the
socket
    # so we capture the KeyboardInterrupt exception
    try:
        # this will start to listen in an infinite loop
        dh_server.serve_forever()
    except KeyboardInterrupt:
        # we need to be able to stop the service, for now we don't care if our implementation is
ugly
        # we just want it to work
        dh_server.shutdown()
```

```
        sys.exit(0)

if __name__ == '__main__':
    main()
```

The Client

A quick how to for python's client sockets can be found here https://docs.python.org/3/howto/sockets.html.

```
#!/usr/bin/env python3
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.serialization import *
import binascii as ba
import socketserver
import socket

def main():
    # we specify the server's address or hostname and port
    host, port = 'localhost', 7777
    # create a tcp socket for IPv4
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    # connect to the tcp socket
    sock.connect((host, port))
    # set the first request according to our protocol
    request = b'Hello'
    # send the request
    sock.sendall(request)
    # read the server's response
    received = sock.recv(3072).strip()
    # print what we have received from the server
    print('Received:\n{}'.format(received))
    # check if the response is valid acording to our protocol
    if received == b'Hey there!':
        # set the next request accordingly
        request = b'Params?'
        sock.sendall(request)
    else:
        # if we get here something is not right
```

```
        print('Bad response')
        # close the connection and return
        sock.close()
        return

    # this means we are still in the game and the next server response must be the DH parameters
    received = sock.recv(3072).strip()
    print('Received:\n{}'.format(received))
    dh_params = load_pem_parameters(received, default_backend())

    # check if the params are valid DH params (do we get here if the response was not valid or
    # do we get an error before getting here?)
    if isinstance(dh_params, dh.DHParameters):
        # based on received parameters we generate a key pair
        client_keypair = dh_params.generate_private_key()
        # create the next message according to the protocol, get the binary of the public key
        # to send to the server
        request = b'Client public key:' + client_keypair.public_key().public_bytes(
            Encoding.PEM, PublicFormat.SubjectPublicKeyInfo)
        sock.sendall(request)
    else:
        print('Bad response')
        sock.close()
        return

    # this means we are still in the game
    received = sock.recv(3072).strip()
    print('Received:\n{}'.format(received))
    # check the format of the message (or rather the beginning)
    if bytearray(received)[0:18] == b'Server public key:':
        # get the server's public key from the binary and its proper index to the end
        server_pubkey = load_pem_public_key(bytes(bytearray(received)[18:]), default_backend())
        if isinstance(server_pubkey, dh.DHPublicKey):
            # calculate the shared secret
            shared_secret = client_keypair.exchange(server_pubkey)
            # print the shared secret
            print('Shared Secret\n{}'.format(ba.hexlify(shared_secret)))
            # close the connection
            sock.close()
            return

    # if we get here it means something went wrong
```

```
    print('Failed')
    sock.close()
    return

if __name__ == '__main__':
    main()
```

Typed/Copied it, what now?

If the code is typed/copied correctly and the DH parameters are generated, then you can open two terminals on the VM and run the server in one and the client in the other. Every time you run the client you should see a new shared secret being generated at the end of the communication. You can also open wireshark and listen to the loopback interface and you would be able to see the actual network communication. If you want to take this a step further, although it does not necessarily teaches you any thing new, you can create or use the provided container in order to mount two containers and copy the server to one and the client to the other. You can connect the two containers to the provided CORE network emulator configuration file “factory.imn” and gain communication as well as internet access (when you start the emulator, of course) Generally speaking, You need to install the python cryptography module for this to work following the procedure that was used in the laboratory notes. You can add command line options using argparse module of python to for instance tell where the DH parameters file is or for client to tell what is the server’s address.

Using this code or any other security code in the ModBus library

The above code can be merged on the a modbus client and server code that is provided with the uModbus library and is slightly costomized in the container file given to you in Moodle. If you decide to use DH key agreement then you must first agree on the session key and then use the agreed value to secure the Modbus protocol. The uModbus library can be slightly modified in order to make the securing process easier for you. The modbus server code can be written clearer thatn the original code by storing the whole TCP PDU message and reply package in a variable and only then process/parse it based on the Modbus protocol structure. Currently, the original library parses the TCP PDU on the server side in a strange manner. The Modbus Server handler can be changed as follows bollow in order to be made simpler. The changes are made in the __init__.py that resides inside the folder server of the library. Note, that if the uModbus library is installed using pip or pip3 (following the instructions on the library’s website) then it is installed in /usr/local/lib/python3.5/dist-packages/umodbus/ That means that whatever changes you may want to make in the library you should access the library files that are stored in this folder. The suggested changes on the server should be made in the /usr/local/lib/python3.5/dist-packages/umodbus/server/__init__.py and should be on the AbstractRequestHandler class as follows:

```
class AbstractRequestHandler(BaseRequestHandler):
    """ A subclass of :class:`socketserver.BaseRequestHandler` dispatching
    incoming Modbus requests using the server's :attr:`route_map`.

    """
    def handle(self):
        try:
            while True:
                try:
                      message=self.request.recv(1024)
                      if not message: break
                      #print(message)
                      #print(binascii.b2a_hex(message))
                      #print(len(message))

                      #At this Point the message from the client
                      #has been received and is ready to be processed
                      mbap_header = message[0:7]
                      remaining = self.get_meta_data(mbap_header)['length'] - 1
                      request_pdu = message[7:8+remaining]

                except ValueError:
                    print("issue")
                    return

                response_adu = self.process(mbap_header + request_pdu)
                #At this point the responce (response_adu) to the message
                #has been structured and is ready to be send to the client
                self.respond(response_adu)
        except:
            import traceback
            log.exception('Error while handling request: {0}.'
                          .format(traceback.print_exc()))
        raise
```

The above code does have a small bug which may generate exceptions but currenlty fully performs the Modbus server functionality.

The proposed changes have already been made in the provided container which is to be used as a Modbus client or as a Modbus server.