# Lab 5

In this lab, you will put all you have learned from cryptography lectures particularly Lab 3 and Diffie-Hellman code into action. You should now understand the concepts you need to secure communications between two parties (Bob and Alice) with an intruder Darth listening in and trying to listen in to the messages. The homework will focus on building an application that can send messages encrypted using the Custom Encryption technique we developed in Lab 3. As you may recall from Lab 3, the Custom encryption technique requires a numeric "N" which is used for encryption. In this homework, we will use the Diffie-Hellman key exchange to determine the value of the key which will be a shared secret between Bob and Alice and, use this key for the value "N". You will implement these techniques between two applications over an insecure UDP connection, which will help you think through how to securely send messages and ensure that they have not been eavesdropped on by intruders.
Learning Objectives:

- Construct an application that communicates over UDP
- Use the Diffie Hellman code that you will develop as part of this lab
- Use the Custom Encryption technique that you developed as part of Lab3

Task1: Diffie-Hellman algorithm

The Diffie-Hellman algorithm was developed to create secure communications over a public network. For this lab, we will consider four variables that include P, G, A, B:

- **P:** One prime number; publicly available.
- **G:** A generator of P.
- **A:** A user (Alice) picks private values for A and B and use them to generate a key to exchange publicly with a second user (Bob).
- **B:** The second user (Bob), receives the key from Alice and uses it to generate a secret key; this gives both users the same secret key to encrypt.

To get a better understanding, review the following five steps:

1. Alice and Bob get public numbers P = 23, G = 9.
2. Each user selects a private key:
    - Alice selected a private key a = 4
    - Bob selected a private key b = 3
3. Each user computes public values:
    - Alice: x = (9^4 mod 23) = (6561 mod 23) = 6
    - Bob: y = (9^3 mod 23) = (729 mod 23) = 16
4. Alice and Bob exchange public numbers:
    - Alice receives public key y = 16
    - Bob receives public key x = 6
5. Alice and Bob compute symmetric keys:
    - Alice: ka = y^a mod p = 65536 mod 23 = 9
    - Bob: kb = x^b mod p = 216 mod 23 = 9

The completed process generates 9, which is the shared secret. Notice this value was never shared between the two parties.
Write a Python program that implements Diffie-Hellman method.
Hint: Python pow() Function The pow() function returns the value of x to the power of y. If a third parameter is present, it returns x to the power of y, modulus z.

**Your dh program should have the three functions.**

```
#Write Function 1 power(a,b,p) that returns a^b mod p using Python's
pow function. Handle special case if b == 1
def power(a, b, p):
    <Enter your code here>

#Write Function 2 that generates and returns a public key using P,G,
and a privateKey chosen by the sender
def dh_generatePublicKey(P,G,privateKey):
    <Enter your code here>

#Write Function 3 that generates and returns a private key using the
publicKey, privateKey and P chosen by the sender

def dh_generateSecretKey(publicKey, privateKey, P):
    <Enter your code here>
```

**Your dh program should have the following main function**

```
def main():
    P = 0; G = 0; x = 0; a = x;
    y = 0; b = 0;
    ka = 0; kb = 0;

    # Both the users will be agreed upon the public keys G and P
    P = 23; # A prime number P is taken
    print("The value of P:", P);

    G = 9; # A primitive root for P, G is taken
    print("The value of G:", G);

    # Alice will choose the private key a
    a = 4; # a is the chosen private key
    print("The private key a for Alice:", a);
    <Enter code here which calls the appropriate function from above
to generate public key for Alice>

    # Bob will choose the private key b
    b = 3; # b is the chosen private key
    print("The private key b for Bob:", b);
    <Enter code here which calls the appropriate function from above
to generate public key for Bob>

    # Generating the secret key after the exchange of keys
    <Enter code here which calls the appropriate function from above
to generate secret keys for both Alice and Bob. Test that these keys
match>
    print("Secret key for the Alice is:", ka);
    print("Secret Key for the Bob is:", kb);

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

Note, while the above main program has specific values of P,G and private keys for Bob and Alice, your program should work for any user-specified values.

**Task 2: Constructing a Plaintext Communications Application**

The architecture for the application includes a server file (alice.py), a client file (bob.py), an intruder (darth.py) and a helper file for shared components used by all the files share many of the same components. Alice first shares the shared public key with Bob. This application assumes that Bob has already shared his public key with Alice. We are assuming this to keep the application simple for the homework. In the real world, Bob will share his key with Alice over the channel. Alice will then compute the shared Secret using Bob's public key and encrypt the message before sending it over the channel. Bob will also compute the shared Secret using Alice's key and decrypt the message. To iterate, to keep this example simple, we will only do one-way communication. The same techniques used in this chapter can be used between any applications that transmit data over UDP.

**Creating Code for Alice.py**

Our first step is to create an application (alice.py) that will use UDP sockets to send messages to Bob. She is not expecting to send messages to Darth, but Darth is eavesdropping on the messages. A skeleton code for alice.py is provided to you with instructions on where to introduce your code.

**Creating Code for Bob.py and Darth.py**

Bob.py and Darth.py will largely stay the same with the exceptions of the private key for each of them. A skeleton code for bob.py and darth.py is provided to you with instructions on where to introduce your code.

**Creating the Helper File**

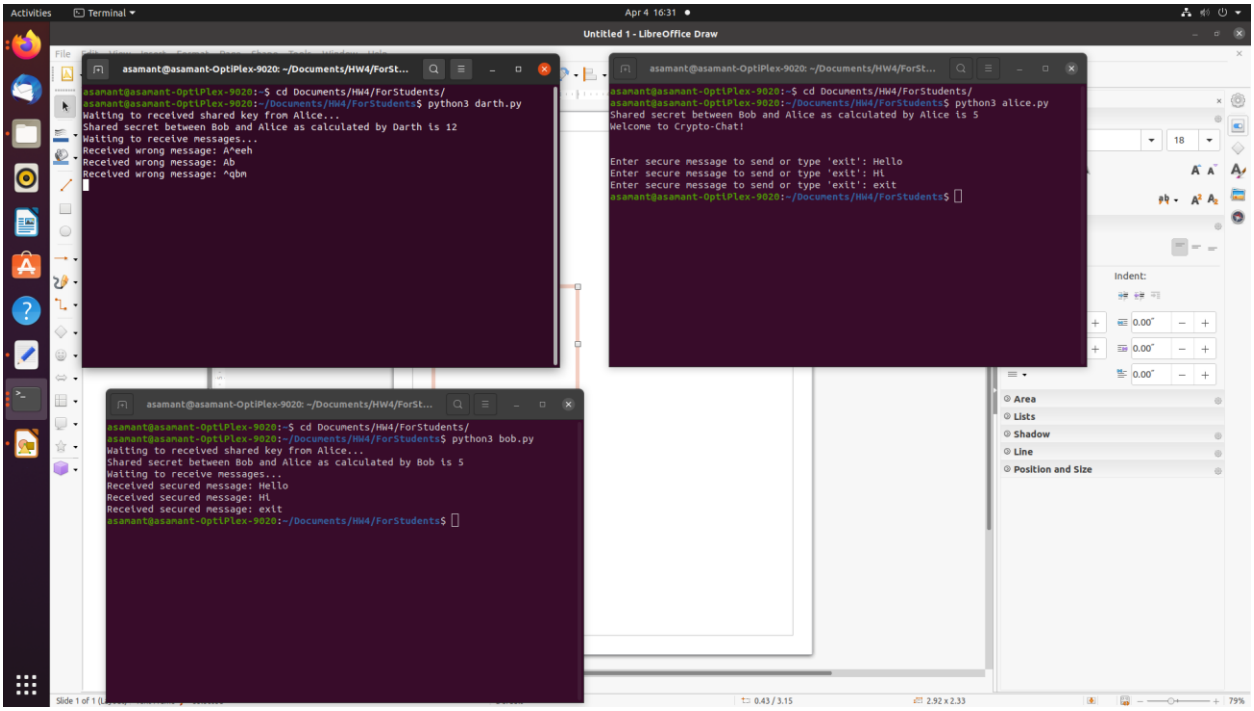
To keep the code as consistent as possible between each step in our application, we will create some system components that are used in alice.py, bob.py and darth.py. These components are saved in the file lab6_shared.py. The below figure shows a screenshot of the three Terminal Windows (Command Windows) running the three programs bob.py, alice.py, and darth.py. Please note that the listeners bob.py and darth.py should be run first before running alice.py. Also, while alice.py and bob.py programs will properly exit when they receive the text "exit", darth.py will not exit (as per current logic). You will have to manually terminate the window.

Note: Students running the code on Linux machines will be able to run all three programs together. For Windows OS, the following command

```
UDPSock.setsockopt(SOL_SOCKET, SO_REUSEPORT, 1)
```

does not work on Win7 and higher. Students can either find a different way for both bob.py and darth.py to listen to the same port or they can first run alice.py and bob.py together and then run alice.py and darth.py together. A quick demo is shown here:

https://www.youtube.com/watch?v=j-ggSi0Wj7E




What to submit, a zip file containing below files:

1. Updated version of alice.py
2. Updated version of bob.py
3. Updated version of darth.py
4. Updated version of lab6_shared.py
5. dh.py
6. screenshot of command windows showing the communication between alice, bob and darth

Assumptions:

1. The input text (message being sent) does not include any spaces between the words.

Rubric

| Submission Artifact | Points |
|---|---|
| Updated version of alice.py | 20 |
| Updated version of bob.py | 20 |
| Updated version of darth.py | 20 |
| Updated version of lab6_support.py | 20 |
| Screenshot of command windows | 20 |