Security 1: Mandatory hand in I

Author: Nedas Surkus

September 2022

Contents

Instructions to run the code

Ensure that Python 3 is installed

Run the script using any Python supporting IDE or by typing the following command to the terminal: " python3 main.py "

(This command might not work on Windows. If this is the case, run : " py main.py ")

Introduction

In this specific assignment, the students were tasked in completing three tasks.

1. Send “2000” from a person named Alice to person named Bob using ElGamal public key method.
2. Take Alice’s encrypted message and then use it to find Bobs private key and reconstruct Alice’s message.
3. Take Alice’s encrypted message and then modify the amount from 2000 to 6000.

In these tasks, the students were given a shared base called g, a shared prime p and Bob’s public key.

Additionally, it was assumed that in task 2 and 3, it is know how the message is encrypted, the shared base and shared prime and Bob’s public key. But, the contents of the message are unknown in task 2 as well it is unknown what is Bob’s private key.

Having this in mind, a simple python project was constructed that will

(a) Generate both Public and Private key for bob.

(b) Will encrypt the message Alice will send using Bobs public key.

(c) Will decrypt the message Alice sends using Bobs public and private keys.

(d) A method that will figure out what is Bob’s private key.

(e) A method that will change the value of Bobs message.

In addition, the project contains 2 Objects.

Public key which will store, g and p (which are the shared based and prime) and h which will contain the actual public key generated for Bob.

Private key which will store, g and p (which are the shared based and prime) and x which will be Bobs private key.

This is used to store and access values more easily.

Methodologies

A specific methodology will be used to find out Bobs private key and find out what value Alice picked for encoding cipher text.

Since we know that bobs private key must be an element such that GCD(p,x) = 1 or in other words the value x must be between 2<= x <=p-2. The public key provided is very small, thus we can assume that the x is not very large and is susceptible to a brute force attack.

In order terms, we will try to generate the exact public key that Bob uses by trying out every

possible x combination from 0 to p-2. Once we get an exact match, we will know exact private key that Bob used to generate his public key. Thus, we will have the ability to decrypt Alice's messages.

Same brute force algorithm will be used to discover what kind of secret key Alice used when encoding her message to Bob by brute forcing the first part of the cipher text.

In order terms, we will try out every possible combination of x from 0 until p-2 until we will have an exact match.

Part 1

Initial considerations in the first task

Before starting on the first task, we must consider if it is possible to accomplish with given data. We are provided with a shared P, shared G and Bobs public key for this assignment, but for us Bobs secret X is unknown until we are forced to find it out in task 2.

When encoding the data, Alice picks out a random value K that will be used to encode the message. We only need to know the shared g, p and Bobs public key to generate first and second part of the cipher text.

However, when decoding the message. We cannot do it without knowing Bobs secret key. Thus, we are unable to verify that Bob can decrypt the message and view Alice's message.

To establish correctness, in Task 1 Bob will generate a new public key by using a randomly selected private key. By using new public key to encrypt the message and new private key to decrypt the cipher text, we can verify that the code is correct and Bob can access the content of the message.

Once correctness is established, we will use the provided Bobs public key to encrypt messages, but we will not see Alice's message until we complete task 2.

Implementation

Firstly, we created a method called "Generate keys" which will output Bobs Private key object and Bobs Public key object.

```
1  def generate_keys(shared_p=6661, shared_g=666):
2      p = shared_p
3      g = shared_g
4
5      private_key = random.randint(1, (p - 1) // 2)
6      public_key = pow(g, private_key, p)
7
8      bob_public = PublicKey(p, g, public_key)
9      bob_private = PrivateKey(p, g, private_key)
10
11     return {'privateKey': bob_private, 'publicKey': bob_public}
```

Secondly, we created a method called "encrypt" in which we will pass Bobs public key and a message we want to encrypt. This method will output an array of 2 integers. First value will be used in decrypting the message and second value will contain the encrypted message.

In this specific algorithm, Alice will pick a random k value that is between 0 but less than shared p. This K value will be used to encrypt both parts of the cipher pairs.

To generate the first pair of the cipher text, we use the following formula:

$$g^k \ mod \ p$$

To generate the second part of the cipher text we utilize the following formula:

$$msg * (bobs \ public \ key)^k \ mod \ p$$

This is reflected slightly differently in Python code, however the execution is identical to the formula provided.

```
def encrypt(key, msg):
    # pick random K
    k = random.randint(0, key.p)

    y1 = pow(key.g, k, key.p)
    y2 = (msg * pow(key.h, k, key.p)) % key.p

    cipher_pairs = [y1, y2]

    return cipher_pairs
```

Lastly, we created a decrypt method in which we pass Bobs private key and cipher text. This method will output the message in plain text.

```
def decrypt(key, cipher_array):
    y1 = cipher_array[0]
    y2 = cipher_array[1]

    normal_message = (int(y2) * (pow(int(y1), key.p - 1 - key.x, key.p))) % key.p

    return normal_message
```

We can decrypt the message using the following formula:

$$y2 * y1^{(p-1-(bobs \ private \ key))} \ mod \ p$$

Thus, using a randomly generated private and public key, we get the following output :

1. `Alice's unencrypted message:  2000`
2. `Alice's encrypted message:  [4137, 2416]`
3. `Bob decrypt the following message:  2000`

This proves that Bob will receive Alice's message and will be able to decrypt it.

With this in mind, we can replace the randomly generated public key with Bobs public key and assume that he will be able to decrypt it.

However, to ensure correctness we will need to find his true private key. This will be done in the following section.

Part 2

Finding bobs secret key

As mentioned previously, we are unaware what is Bobs private key. This prevents us from decrypting Alice's message and checking for correctness.

Before attempting to find out Bobs private key, we must analyze the data provided. We know that to generate Bobs public key we use the formula of: $g^x \, mod \, p$

Thus, if we can find out what is the X (Bobs private key) we could use it to generate an identical public key to the one provided for the assignment.

Additionally, after analysing Bobs public key, we can see that is short (2227) which indicates that x is not large and thus is susceptible to a Brute force attack by guessing every combination of $g^x \, mod \, p$. until we find the exact x that will generate Bobs public key.

Lastly, we know that x must be a value between 2<= x <=p-2. Thus, we can predict that we will not need to check more than 6661 values which is a manageable amount for modern day computers.

Thus the following code is created:

```
1  def brute_force(base, power, answer):
2      result = -1
3      expect_public_key = answer
4      for i in range(power):
5          public_key = pow(base, i, power)
6          if public_key == expect_public_key:
7              result = i
8              break
9
10     return result
```

This code uses the shared base and power and an answer we know (in this case, Bobs public key) to try multiple combinations of x until we finally find the x that generates the public key.

Lastly to check if it is correct, we can insert Bobs private key into generate keys method and check if it is generating the same public key that we are provided with.

Using the newly found private key we can decrypt Alice's message to reveal that Alice wrote "2000"

```
1
2      private_key = 66
3      public_key = pow(g, private_key, p) # Output is 2227
```

1. `Alice's unencrypted message:  2000`
2. `Alice's encrypted message:  [3334, 1744]`
3. `Bob decrypt the following message:  2000`

Part 3

Changing Alice's message from 2000 to 6000

Last part of the assignment states that we have intercepted Alice's message, and we have to change the value of 2000 to 6000.

To do so, we must attempt to break Y2 encryption. However, a significant problem is that we use modulus when generating the message and it's impossible to reverse the modulus to get the original message and change it.

Thus there is a couple of ways we can approach this problem.

Method 1

We can notice the same vulnerability that Bobs secret key has. Secret key K , that is picked by Alice, is used to generate the first cipher text pair and the first cipher text pair does not use any other values that would be unknown to us (Example: Alice's message).

Additionally, K is a value between 0 and shared P, and thus we would only need to check a low number of combinations to discovered what kind of K produces an identical first cipher text pair. Thus, we can utilize the same brute force method to find out K value.

Once we have K, we can use it to generate the second pair of the cipher text and replace the message with our own message.

Once we have the new second pair, we can simply replace the existing second pair in the cipher text array and send it to Bob.

```
1  def change_msg_without_knowing_message(cipher_array, key, new_msg):
2      # Get the first cypher key
3      y1 = cipher_array[0]
4
5      # Find out K used to generate y1
6      k = brute_force(key.g, key.p, y1)
7
8      # Encrypt a new message using the known public key and discovered k
9      new_y2 = (new_msg * pow(key.h, k, key.p)) % key.p
10
11     # Replace existing second part of the cipher
12     cipher_array[1] = new_y2
13
14     return cipher_array
```

Since we did not modify the first cipher pair or the way the second cipher pair is generated, Bob can decrypt the message and it will output "6000".

1. `Alice's unencrypted message:  2000`
2. `Alice's encrypted message:  [1271, 4649]`
3. `Bob decrypt the following message:  2000`
4. `But Eve intercepted the message and brute forced bobs private key.`
5. `Bobs private key :  66`
6. `Also Mallory intercepted the message and changed the message.`
7. `Mallory new cypher:  [1271, 625]`

8. `Bob decrypt the following message:  6000`

While this method is ideal and works perfectly, in the task it is stated that Mallory has a constrained device. A constrained device means that it might not have enough processing power to brute force Alice's secret key.

Thus we can use an alternative method to change the message Alice sent.

Method 2

As previously discussed, we use $msg * (bobs\ public\ key)^k\ mod\ p$ to encrypt the second part of the cipher text.

Modulus gives back the result of the division, thus we can't reverse the modulus to get the initial value.

However, if we know the original message we are able to modify the result. By multiplying the second cipher pair by 3. We can increase the result to 6000. Since 2000 * 3 = 6000.

Afterwards we simply replace the second pair inside the ciper text and send it to Bob.

```
1  def change_msg_knowing_message(cipher_array):
2      # Get the second cypher key
3      y2 = cipher_array[1]
4
5      # Since we know that result of y2 = 2000. By multiplying with 3, we get 6000
6      y2 = y2 * 3
7
8      # Replace existing second part of the cipher
9      cipher_array[1] = y2
10
11     return cipher_array
```

We can note that this produces a completely different second pair of the cipher text.

The newly produced second pair is notably longer and it could tip off Bob that Alice's message is modified. However, this cipher will still produce 6000. Thus, we succeeded in our attempt to modify the message.

This a less CPU intensive method and due to this, it fits our requirements of Mallory having a constrained device.

1. `Alice's unencrypted message:  2000`
2. `Alice's encrypted message:  [1449, 4817]`
3. `Bob decrypt the following message:  2000`
4. `But Eve intercepted the message and brute forced bobs private key.`
5. `Bobs private key :  66`
6. `Also Mallory intercepted the message and changed the message.`
7. `Mallory new cypher:  [1449, 1129]`
8. `Bob decrypt the following message:  6000`
9. `Alternatively if we know the content of the message.  We can change the cipher as well.`
10. `Mallory new cypher:  [1449, 14451]`
11. `Bob decrypt the following message:  6000`

Conclusion

In this specific assignment managed to send Bob an encrypted message and Bob can decrypt it and see its contents.

We also managed to find out Bobs private key by utilising a simple brute-force attack.

Lastly we managed to alter the message sent in two different ways. In the first method, we utilised another brute-force attack to figure out what secret key Alice used to encrypt the message and we encrypted a new message using Alice's secret key. In the second method we managed to manipulate the message by multiplying the result by 3. Thus manipulating the result of the decryption.

As a result we can make the following conclusions:

ElGamal encryption method is a valid way to encrypt messages which does not require the person sending the message to know every detail about the receiver, this allows the message to have confidentiality as only the receiver is able to decypther message. Additionally, the sender must select a random secret key, which is used to prevent messages from being tampered with and provides the message with integrity.

However, this encryption method only provides such benefits if there are good and long values selected for generating keys and encrypting. If good values are not selected, the ElGamal can become vulnerable to brute-force attacks and message-tampering. Weak secret key combined with small prime and g can generate a weak public key that can be reversed engineered by a simple brute-force attack and the receivers secret key can be discovered. This compromises confidentiality and allows the attacker to eavesdrop on the conversation and the messages being sent to the receiver.

When encrypting the message a strong secret key must be selected by the sender or the same kind of brute-force attack can be utilised to find the secret key and allow the attacker to masquerade as the sender. Additionally, it is crucial that messages should not only be simple numbers as it is vulnerable to message tampering. By knowing the message being sent, the attacker can use simple arithmetic's to manipulate the value being sent and compromising the integrity of the message.

To prevent the encryption from failing, the receiver must select big numbers for p,g and secret

key to ensure that he is not vulnerable to brute force attacks. Bigger numbers will require more combinations to be checked and would require significantly more time to reverse engineer (if a sufficient large number is selected. Then the secret key might take millions or even more years to crack even if we utilize the most modern computers available).

When encrypting messages, it is advisable to encrypt messages by selecting a large secret key and it would be advisable that the message would be a string as each character of a string needs to be encrypted individually making it less vulnerable to message tampering as using simple arithmetic's will change the message to unreadable gibberish.