Security 1 - Mandatory excercise set 1

ElGamal Encryption

The ElGamal public key encryption scheme has been chosen to encrypt messages between two parties. The following tasks demonstrates how different parts of ElGamal works in practice.

The keys used for messaging are based on the following values:

$p = 666, g = 6661$, Bobs Private key - $BobPk = 2227$

Task 1 - Message Encryption

Given generator g, a prime p, and the receivers public key *recieverPk*, a ciphertext can be generated using the following formula:

$(g^r \ mod \ p, m * B^r \ mod \ p)$

Where B is the receivers public key, *receiverPk*. We also pick a random value between $0 ... p-2, r$ used to encrypt the message.

```
(int, int) Encrypt(int g, int p, int recieverPk, int m) {

    int r = Random.Shared.Next(1, (p - 1));

    int R = (int)BigInteger.ModPow(g, r, p);
    int S = (int)(m * BigInteger.Pow(recieverPk, r) % p);

    return (R, S);
}

(int, int) encryptedMessage = Encrypt(generator, prime, bobPk, message);
```

Task 2 - Bruteforce Bobs private key

To obtain Bobs private key, we can use the fact that we already know g, p and *bobPk*, and thus that the formula for bobs public key, $B = g^b \mod p = 2227$ only contains one unknown - Bobs private key, b. We also know that Bobs private key is in the range $0 \le b \le p-2$, or $0 \le b \le 666$, making it very realistic to brute force it. We simply have to find a b, for which $B = 6661^b$ mod $666 = 2227$. We will do this using a loop that tests all numbers from 0 to $p-2$, and returns b once the equation is true.

The code used to brute force the key is as follows:

```
int BruteForceKey(int pk) {

    for(int i = 0; i < prime; i++) {
            int computedKey = (int)BigInteger.ModPow(generator, i, prime);
            if (computedKey == pk) {
                return i;
            }
        }

    return 0;
}
```

Bobs private key can now be used to decypt Allice's message using the formula:

$R^{-b} * S \ mod \ p$

Where R in the code is called *sharedSecret*, and R^{-b} is called *inverseSharedSecret*

```
int DecryptMessage(int p, int recieverKey, (int, int) cipherText) {

    int R = cipherText.Item1;
    int S = cipherText.Item2;

        int sharedSecret = (int)BigInteger.ModPow(R, recieverKey, p);
        int inverseSharedSecret = (int)BigInteger.ModPow(sharedSecret, p - 2, p);
        int decryptedMessage = S * inverseSharedSecret % p;

        return decryptedMessage;
    }

    int bobKey = BruteForceKey(bobPk);

    Console.WriteLine(DecryptMessage( prime, bobKey, encryptedMessage));
}
```

Task 3 - Changing the message content

ElGamal is multiplicatively homomorphic, which means that multiplications done to the message of a ciphertext S, directly translates to corrosponding operations on the decrypted message. Thus we can leverage the fact that:

$(m * B^r \ mod \ p) * 3 = (R^{-b} * S \ mod \ p) * 3$

Thus multiplying the message, *2000*, even when encrypted, by 3, will yield 6000 once decrypted. The code for this is as follows:

```
(int, int) InterceptAndChangeMessage((int, int) cipherText) {
    return (cipherText.Item1, cipherText.Item2 * 3);
}

(int, int) ChangedEncryptedMessage = InterceptAndChangeMessage(encryptedMessage);
```

```
int ChangedDecryptedMessage = DecryptMessage(prime, bobKey, ChangedEncryptedMessage);
```