ECE404 Introduction to Computer Security: Homework 06

Spring 2024
Due Date: 5:59pm, February 27, 2024

1 Introduction

The goal of this homework is to give you a deeper understanding of RSA encryption and decryption, and its underlying principles. Before starting this assignment, make sure that you understand the relationship between the modulus and the block size for the RSA cipher and how RSA is made feasible by the fact that the modular exponentiation possesses a fast implemenation.

As always, please read the homework document in its entirety before coming to office hours with your questions. The teaching staff have spent a long time writing the assignment to cover many common questions you might have.

2 Problem 1: RSA Encryption and Decryption

Write a Python object oriented program to implement a 256-bit RSA algorithm for encryption and decryption. The plaintext message has been provided in the zip file and is called message.txt. Your data block from the text will be 128-bits. For the reasons explained in Lecture 12.4, prepend your 128-bit data block with 128 zeros on the left to make it a 256-bit block. If the overall length of the plaintext is not an integral multiple of 128-bits, make sure to pad the appropriate number of zeros from the right before prepending the 128 zeros mentioned in the previous step. This method of creating the blocks can be a little tricky so make sure to understand it, or you will face issues when trying to get the correct encryption results.

2.1 Problem 1 Program Requirements

Your solution should accept the following command-line syntax:

```
python3 rsa.py -g p.txt q.txt
python3 rsa.py -e message.txt p.txt q.txt encrypted.txt
python3 rsa.py -d encrypted.txt p.txt q.txt decrypted.txt
```

An explanation of the command-line syntax is as follows:

- **For Key Generation** (indicated by '-g' in line 1)
  - The generated values of p and q will be written to `p.txt` and `q.txt` respectively.
  - The `.txt` files should contain the number as an integer represented in ASCII.
  - For example if $p = 7$, the corresponding text file will display 7 when opened in a text editor.
- **For Encryption** (indicated by '-e' in line 2)
  - Given the p and q values found in `p.txt` and `q.txt` respectively, encrypt the plaintext message in `message.txt` using the RSA algorithm, and write the output to `encrypted.txt`
  - **The key generation step mentioned in the previous bullet is there to simply make you aware of the necessity in real world applications. Make sure to use the p and q value we provided to check the fidelity of your implementation.**
- **For Decryption** (indicated by '-d' in line 3)
  - Given the p and q values found in `p.txt` and `q.txt` respectively, decrypt the ciphertext in `encrypted.txt` using the RSA algorithm, and write the output to `decrypted.txt`

2.2 Important Implementation Details

Regarding key generation, keep the following points in mind while writing your solution:

- The priority in RSA is to select a particular value of e and then choose p and q accordingly. For this assignment use $e = 65537$.
- Using the logic in `PrimeGenerator.py` (found in Lecture 12), to generate values of p and q. Both p and q must satisfy the following conditions:
  1. The two leftmost bits of p and q must be set
  2. p and q should not be equal
  3. $(p - 1)$ and $(q - 1)$ should both be co-prime to e
  4. If any of the above conditions are not satisfied, repeat the process until they are.

Regarding decryption, keep the following points in mind

- RSA specifies that the recovered plaintext can be computed as $C^d \bmod n$.
- However, d is roughly the same size as the modulus n meaning the above modular exponentiation is an expensive process.
- To circumvent this obstacle, use the Chinese Remainder Theorem (CRT) explained in Lecture 12.5 to compute the modular exponentiation for decryption.

2.3 RSA Class Skeleton File

Below is a skeleton file to get you started on this assignment.

```python
class RSA():
    def __init__(self, e) -> None:
        self.e = e
        self.n = None
        self.d = None
        self.p = None
        self.q = None

    # You are free to have other RSA class methods you deem necessary for your solution

    def encrypt(self, plaintext:str, ciphertext:str) -> None:
        # your implemenation goes here
    def decrypt(self, ciphertext:str, recovered_plaintext:str) -> None:
        # your implemenation goes here

if __name__ == "__main__":
    cipher = RSA(e=65537)
    if sys.argv[1] == "-e":
        cipher.encrypt(plaintext=sys.argv[2], ciphertext=sys.argv[5])
    elif sys.argv[1] == "-d":
        cipher.decrypt(ciphertext=sys.argv[2], recovered_plaintext=sys.argv[5])
```

3 Problem 2: Breaking RSA for small values of e

Lecture 12.3.2 describes a method for breaking RSA encryption for small values of e. In this scenario, a sender, say Party A, sends the same message

M to 3 different receivers using their respective public keys. All of the public keys have the same value of e, but different values of n. An attack can intercept all three ciphertexts and use the Chinese Remainder Theorem to calculate the value of $M^3 \bmod N$, where N is the product of the three ns. Then, he or she can simply solve the cube-root to recover the plaintext message M. Your task is to replicate this scenario with **a Python script** that does the following:

1. Generates three sets of public and private keys with $e = 3$
2. Encrypts the given plaintext with each of the three public keys. (You should have three ciphertexts after this step)
3. Take the three ciphertexts generated in step 2 and use the CRT to recover the original plaintext.

3.1 Problem 2 Program Requirements

Your solution should accept the following command-line syntax:

```
python3 breakRSA.py -e message.txt enc1.txt enc2.txt enc3.txt
    n_1_2_3.txt
python3 breakRSA.py -c enc1.txt enc2.txt enc3.txt n_1_2_3.txt
    cracked.txt
```

An explanation of the command-line syntax is as follows:

- **For Encryption** (indicated with the '-e' argument)
  - Encrypt the plaintext in `message.txt` with three different self generated public keys and write each ciphertext to `enc1.txt`, `enc2.txt`, `enc3.txt` respectively.
  - Also write the moduli used (i.e. n_1, n_2, n_3) to `n_1_2_3.txt`
- **For Cracking the Encryption** (indicated with the '-c' argument)
  - Given the three ciphertexts and the respective public keys used to compute them, recover the original plaintext and write it to a file called `cracked.txt`

3.2 Important Implementation Details for Problem 2

This section details some implemenation details that are important to consider

- Problem 1 and 2 share a lot of overlap in terms of operations that need to be performed. Thus, we highly recommend doing a good job of defining your RSA class in problem 1 so that it can be directly used in problem 2.
- Because python's pow() method does not provide enough precision to solve the cube-root, we have provided a method called `solve_pRoot()` that provides the necessary precision.

4 Submission Instructions

- Make sure to follow program requirements specified above. **Failure to follow these instructions may result in loss of points!**.
- For this homework you will be submitting a zip file titled `HW06_<last_name>_<first_name>.zip` to Brightspace containing:
  - The file `rsa.py` containing your code for Part 1.
  - The file `breakRSA.py` containing your code for Part 2.
  - You can import `PrimeGenerator.py` and `solve_pRoot.py` into your .py files with the assumption that it will be in the same directory as your files when being graded.
  - a pdf titled `HW06_<last_name>_<first_name>.pdf` containing a detailed explanation of how you implemented RSA in part 1 and the CRT to break RSA in part 2.