Exercise 2

Cryptography

Log into your VM (`user` / `1234`), open a terminal and type in `infosec pull 2`.

- When prompted, enter your username and password
- Once the command completes, your exercise should be ready at `/home/user/2/`

When you finish solving the assignment, submit your exercise with `infosec push 2`.

Question 1 (70 pt)

In this exercise you will focus on a simple stream cipher called **Repeated Key cipher**. A repeated key cipher works by XOR-ing the bytes of the plaintext with the bytes of the key similarly to a one-time pad:

- The first byte of the plaintext is XOR-ed with the first byte of the key
- The second byte of the plaintext is XOR-ed with the second byte of the key
- ...
- **If the plaintext is longer than the key, start using the key again from the beginning** (first it's first byte, then it's second byte, etc.)

Part A (10 pt)
Inside `q1.py`, implement the `encrypt` method in the `RepeatedKeyCipher` class.
Tip: To get the Unicode/ASCII code of a char, use ord. To convert a code back to a char, use chr.

Part B (5 pt)
Inside `q1.py`, implement the `decrypt` method in the `RepeatedKeyCipher` class. **Add the smallest amount of code possible** on what you already implemented in the previous part.

Part C (20 pt)
Inside `q1.py`, implement the `plaintext_score` method in the `BreakerAssistant` class. The method should take a string (of any length) and return a numeric score such that a string containing a plausible text in English will receive (with a high probability) a higher score than a random string of the same length.

- For example, it should give this string `"I am a sentence!"` a higher score than `"\xc6\xc9u\xd0v\x14\xe2\xcf\xc5\xf5\x1eZ\x10Yd\xd3"`[1].
- As you will be using this method in the following parts, **we recommend implementing a simple method now and improving it if necessary as you proceed in the next parts**.

Document your solution inside `q1c.txt`.

Part D (10 pt)

Inside `q1.py`, implement the `brute_force` method in the `BreakerAssistant` class. The function receives a key length, attempts all possible keys of the given length, evaluates the plaintext possibilities using the `plaintext_score` function you have written, and **returns the "correct"[2] plaintext**.

In `q1d.cipher` we attached a sample encrypted text you can try breaking (it has a short key - 2 bytes long). You can run your code on it with `python decrypt.py 1d`. **Your solution must decrypt the attached ciphertext to the correct plaintext**.

Document your solution inside `q1d.txt`.

Part E (25 pt)

The problem with the brute force method is, well, that it's brute force. This means we try many attempts without much thought. Trying to a break a key longer than a few bytes is going to be impractical (as we need $2^{8\cdot(\text{number of bytes})}$ attempts).

In this part we urge you to try and break they key in a different way - find a way which is much faster, and for example, can break a key of length 10 or higher. For this part, **you may assume the ciphertext is long**.

Inside `q1.py`, implement the `smarter_break` method in the `BreakerAssistant` class using the technique you devised. As before, the method should receive the key length and return the "correct" plaintext.

In `q1e.cipher` we attached a sample encrypted text you can try breaking (it has a loooong key - 16 bytes long). You can run your code on it with `python decrypt.py 1e`. **Your solution must work on this text**.

Document your solution inside `q1e.txt`.

---

[1] The notation `\x<2 hex digits>` is used in Python to denote a char by it's hexadecimal code. We mainly use this where writing the char directly would result in binary gibberish or chars we can't read.

[2] While you can't know what is the correct plaintext, we hope that the plaintext with the highest score is indeed the correct one.

Question 2 (45 pt)

In this question you will break several simplistic schemes that are based on RSA. While solving this question, keep in mind the following two points:

1. This exercise assumes you understand the basic **mechanics and math used by RSA**. This is not a cryptography course, so we don't assume deep understanding, but **you will need to understand RSA basics.**

2. The documentation of the Python Crypto library (the library we use for the encryption) is available at http://pythonhosted.org/pycrypto.
    - The links specific to RSA are here and here.
    - You don't need to read this in advance. This is here for reference if something is unclear in our code.

Overview

Inside `q2_atm.py`, you can find the code for a (hypothetical) ATM. After inserting the credit card, the machine sends both the credit card and the PIN code the user entered, to a remote server for verification. Once the server responds, the ATM verifies the response and it's valid, it gives the user his money.

Part A (15 pt)

When a user enters a 4-digit PIN code, the machine encrypts it with a 2048 bit RSA public key before sending it to the server.

Inside `q2.py`, implement the `extract_PIN` method in the `RSABreaker` class. The method receives an encrypted PIN and returns the original PIN.

In `q2a-pin.txt` we attached a sample encrypted PIN you can try breaking. You can run your code on it with `python decrypt.py 2a`. **Your solution must work on this PIN**.

Document your solution in `q2a.txt`.

Part B (15 pt)

The same ATM machine also sends the encrypted credit card number, a 9 digit number[3], to the server. Since the developers of the ATM were paranoid, they encrypted the credit card number with a different 2048 bit RSA key.

[3] Israeli credit cards are 8-9 digits (unlike the international ones that have at least 11 digits)

Inside `q2.py`, implement the `extract_card` method in the `RSABreaker` class. The method receives an encrypted credit card and returns the original card number.

In `q2b-card.txt` we attached a sample encrypted card you can try breaking. You can run your code on it with `python decrypt.py 2b`. **Your solution must work on this credit card**.

Document your solution in `q2b.txt`.

Part C (15 pt)
When the central server sends a response to the ATM, it sends a status code (these are detailed in `q2_atm.py`) and a signature. The ATM uses the signature to verify the server response (see `verify_server_approval` in the `ATM` class).

Inside `q2.py`, implement the `forge_signature` method in the `RSABreaker` class. The method should return a `ServerResponse` object that passes the verification of the `ATM` class.

Document your solution in `q2c.txt`.

Final notes:

- The sum of all points in this exercise is higher than 100.
  - Yes, we know - it's intentional.
- **All answers should run in at most 60 seconds per question**. If anything is running significantly longer than that, your solution is not efficient enough and is probably wrong.
- **Document your solutions.**
- Don't use any additional third party libraries that aren't already installed on your machine (i.e. don't install anything).
- If your answer takes an entire page, you probably misunderstood the question.