CSC321 Public Key Cryptography Report

Implementation notes

To better simulate real world message passing conditions, as well as more clearly delineate the portions controlled by each party, communication has been simulated using the `Channel` class, which is just a asynchronous `Queue` with additional logging capability for the demo.

Most shared functionality has been abstracted into various python modules as follows:

- `aes.py`: simple AES-CBC message encryption from strings
- `bad_rsa.py`: (awful) RSA impl for task 3
- `channel.py`: channel abstraction
- `dh.py`: Diffie-Hellman key exchange impl
- `modmath.py`: implementation of specialized modular math functions, such as modular inverse

Task 1: Diffie-Hellman Key Exchange

As can be seen in the file `task_1.py` D-H key exchange has been implemented according to spec.

This can be run using the following command, as can any other script mentioned:

```
./task_1.py # or python task_1.py
```

On any UNIX-like system.

Task 2: Implement MITM key fixing & negotiated groups

Part 1

Implementing the protocol described and setting A and B to p is implemented in `task_2_1.py`, using the same infrastructure as before.

Part 2

Implementing the tampering of the generator g is in `task_2_2_1.py` (for modification of g for both parties) and `task_2_2_2.py` (for modification of g only in transit).

It's based on the idea that by setting g to either 1, p, or p - 1, whatever the values of a and b will produce known values of the encrypted text A and B.

There are 3 cases:

$$g^p \bmod p = 1^x \bmod p = 1$$

$$g^p \bmod p = p^x \bmod p = 0$$

$$g^p \bmod p = (p-1)^x \bmod p = \begin{cases} 1 & x \text{ is even} \\ p-1 & x \text{ is odd} \end{cases}$$

Assuming Mallory can set g for both parties, this allows her to decrypt both parties using these known values of s, as shown in the demo. However, in the protocol specified, Mallory can only modify g when it is sent to Bob, so Bob would still correctly derive s, meaning that Mallory could only decrypt Alice's messages, and Alice couldn't decrypt Bob's messages either due to the tampering.

Task 3: Implement "textbook" RSA & MITM Key Fixing via Malleability

Part 1

RSA is implemented in `RSA.py` and usage example can be found in `task_3_1.py`.

Part 2

There are two ways to approach this task, both of questionable validity.

Approach 1 The simplest way to bypass this encryption scheme is to simply overwrite the value of c to be a multiple of n, including zero, similar to the MITM attack on D-H key exchange. This is implemented in `task_3_2.py`. This, however, doesn't use the fact that RSA is malleable whatsoever.

Approach 2 A more complex approach that actually does use malleability is a chosen ciphertext attack. However, this is only possible if Mallory has access to s after being decrypted by Alice, which makes no sense in this context.

It it were, though, Mallory could modify c by multiplying it with any arbitrary number x to using the same public key.

$$c' = F(c) = (c \cdot x^e \bmod n) \bmod n$$

Mallory can then extract the original s from the decrypted s' created by Alice like so:

$$s = (s' \cdot x^{-1} \bmod n) \bmod n$$

This works since Alice has effectively already decrypted the message, so Mallory need only reverse her transform on it.

This is implemented in `task_3_3.py`

Another Example The most obvious example of an issue with would be with regards to a proper example of the usage of a chosen ciphertext attack, where an attacker could violate confidentiality by getting the "Alice" to decrypt some seemly unimportant ciphertext, that can then be used to get the real message of interest.

Signature Verification The same malleability property poses an issue for signatures.

Given two signatures for messages m_1 and m_2 a new signature can be constructed from multiplying them, as demoed in `task_3_4.py` based on the fact that modulo multiplications produces another valid signatures:

$$\text{Sign}(m_1, d) = m_1^d \bmod n$$

$$\text{Sign}(m_2, d) = m_2^d \bmod n$$

$$m_3 = m_1 \cdot m_2 = (m_1^d \bmod n) \cdot (m_2^d \bmod n) = (m_1^d \cdot m_2^d) \bmod n = (m_1 \cdot m_2)^d \bmod n$$

Questions

1. Given the large prime numbers, it would be moderately difficult, but should be within the realm of possibility given modern computing power. Even a basic brute force attack where a fast computer guesses appropriate a and b values less than p until hitting one that works should work for smaller values of
   p.
2. With a brute force approach things get unfeasible for large values of p, like those used for the "real-life" example. As a result, other approaches would have to be explored. This is because a brute-force approach would have to guess any value less than p, where p is on the order of $10^3$08, which simply isn't possible in our lifetimes or that of the universe.
3. These attacks are possible because Mallory can modify data without detection. This could be solved with something like signatures, for example, to ensure no data has been tampered with in transit (ideally).
4. Since in RSA the modulus is computed from two prime numbers $p \cdot q$ then you know if n is shared then p and q must also be shared due to the properties of prime factorization. Therefore, assuming they share e, they can decrypt each other's messages, which is very bad for security.