Assignment 2

Name: Hongzhi Fu Student ID: 1058170

1.

(a) To achieve higher performance of calculating exponential modulo operation, we need to create a function called `fast_expo_mod`, which takes arguments a, b, n and returns $a^b \bmod n$.

```
def fast_expo_mod(a, b, n):
    # fast calculation of a ^ b % n
    mod = 1  # initialization
    while b > 1:
        if b & 1 == 1:  # take the last digit of b in binary representation
            mod = mod * a % n
        a = a * a % n
        b = b >> 1  # shift 1 bit to the right
     return mod * a % n
```

Then, we can implement function `sign`.

```
def sign(m, d, n):
    return fast_expo_mod(m, d, n)
```

(b) Having implemented `fast_expo_mod`, we can easily write function `verify`.

```
def verify(s, e, n, m):
    return m == fast_expo_mod(s, e, n)
```

(c) `blind_sign` takes auguments m, e, d, n and x(a multiplicative argument) as input, compute m' as $x^e \cdot m$, and then obtain signature $s' = m'^d \bmod n = x \cdot m^d \bmod n$. The final step is derive forged signature: $s = s' \cdot x^{-1} \bmod n = x \cdot x^{-1} \cdot m^d \bmod n$.

```
def blind_sign(m, e, d, n, x):
    m_prime = x ** e * m  # compute m'
    s_prime = sign(m_prime, d, n)  # compute s'
    inverse_x = inverse_modulo(x, n)  # compute inverse modulo of x
     return inverse_x * s_prime % n
```

(d) After calling function `sign`, we have the signature:

842612069114304437277254620943803888193744323898497110421760207090491607854318308181376616122303678866250511591457563365047834904328420649048090111863476069393598680838267426915262169806349160418904394767148530074654724786749045 2

(e) original message signature:

463537989197757726080618016081027098741354387880292271435773264081119374791313030748577134416706490834108073135225971455228203713672286497977055891978399277918043028466260885152035443156020115664139443188024488006400963600865142780

blinded message signature:

842612069114304437277254620943803888193744323898497110421760207090491607854318308181376616122303678866250511591457563365047834904328420649048090111863476069393598680838267426915262169806349160418904394767148530074654724786749045 2

2.

(a) Since the plaintext space is small (26 characters), the opponent can apply chosen plaintext attack by encrypting each character and match the incoming ciphertext. For example, if Bob sends a message

m encrypted as m^e, the attacker can encrypt each 26 characters and match it with ciphertext m^e respectively. If it matches, the attacker can know what plaintext m is.

(b) One countermeasure is instead of encrypting each character, we encrypt a block of characters by concatenating each character. Thus, the plaintext space is exponentially proportional to block size, and it is not practical for attackers to recover the message by chosen plaintext attack.

3.

(a) Since e_1 and e_2 are relatively prime numbers, then we can find x and y such that:

$$e_1 x + e_2 y = 1$$

The above equation can be effectively calculated by Extended Euclidean Algorithm.

Having x and y, we can recover original message M by:

$$\begin{aligned} M &= C_1^x * C_2^y \bmod n \\ &= (M^{e_1})^x * (M^{e_2})^y \bmod n \\ &= M^{e_1 x + e_2 y} \bmod n \\ &= M \end{aligned}$$

(b) One simple strategy is Jiajia requests Jaiden's public key e_1 and since Jiajia knows n, she can easily derive $k \cdot \phi(n) = e_2 \cdot d_2 - 1$, where k is a coefficient that can be computed by the following formula:

$$k = (e_2 \cdot d_2 - 1) \,//\, n \; + 1$$

where // is floor division, e.g. 15 // 4 = 3.

Having the coefficient k, we can derive $\phi(n) = \frac{(e_2 \cdot d_2 - 1)}{k}$.

For example, if $p = 5,\ q = 11,\ e = 3,\ d = 27,\ n = 55$, we can derive k by $(e_2 \cdot d_2 - 1) \,//\, n + 1 = (3 \cdot 27 - 1) \,//\, 55 + 1 = 2$, and finally get $\phi(n) = \frac{(3 \cdot 27 - 1)}{2} = 40$, which is exactly equal to $(p-1) \cdot (q-1) = 4 \cdot 10 = 40$.

Finally, The private key of Jaiden can be calculated by $d_1 = e_1^{-1} \equiv 1 \pmod{\phi(n)}$.

4.

(a) Step 1: Initiator A asks responder B to establish connection between them by transmitting identity of A(ID_A) concatenated with a unique identifier, Nonce A, which encrypted by the master key of A.

Step 2: After receiving identity message of A, responder B also generates its unique number N_B, and concatenate it with B's identity ID_B encrypted by B's master key, then sends a request message which contains both identity information of both A and B to KDC for a session key between them.

Step 3: KDC responds with a message containing two parts: one is encrypted by B's master key, thus B can verify the authenticity of session key K_s and the previous information ID_A and N_B are not replayed by any other opponents. Similarly, another part is encrypted by A's master key which contains K_s, ID_B and N_A. Note that this message cannot be tampered by others because only A knows its master key.

Step 4: After verifying the authenticity of session key from KDC, B sends K_s, ID_B and N_A to A. Thus, A can verify the message is indeed from KDC by comparing received N_A with original one. Meanwhile, A also knows B's identity.

(b) A and B can compare received nonce number N_A and N_B with their corresponding original one respectively, and sometimes the nonce number is timestamp which generated at a particular time thus impossible to be replayed. If two number are identical, responders can convince themselves that the key is fresh and not replayed.

(c) Because only A and B knows its corresponding master key K_A and K_B, which is distributed by KDC

before session key request. If neither K_A nor K_B is compromised, A and B know that the session key distributed is not available to other opponents.

(d) No, A and B are not authenticated with each other. In step 1 and 4, two participants only send an identity to each other, which is not enough to ensure the message is indeed coming from someone. One such attack is by interception and masquerade. Suppose there is an intruder D, which intercepts the message $ID_A \,||\, E(K_a,\ N_a)$ in step 1, then generates their own nonce number N_D, requests KDC to obtain session key and sends $E(K_a,\ [K_s \,||\, ID_D \,||\, N_A])$ back to A, so A will think I have shared session key with B, but actually with D, which cannot ensure the authenticity of both A and B.

5.

(a) It satisfies on condition that unfilled block is padded with some special symbols. Since any message can be broken into several blocks, this hash function can take a variable-length message as input.

(b) It satisfies. Since the block size is fixed, and the result is constructed by XOR all encrypted blocks, the hash function always outputs a fixed-length message.

(c) Since the calculation involved only includes exponent modulus and XOR, the efficiency can be guaranteed.

(d) It satisfies, because if we want to recover the original message, then each block of hash value is constructed by RSA algorithm encrypted by public key, so it involves solving RSA problem, which is computationally infeasible to handle.

(e) It does not satisfy. Suppose the hash value of original message is $H(M)$, we can easily find an alternative one that all blocks of hash value is 0 except the one which is equal to $H(M)$, i.e. $H(M_1) = H(M_2) = \cdots = H(M_{n-1}),\ H(M_n) = H(M)$. Thus, $M_1 = M_2 = \cdots = M_{n-1} = 0$, and our problem is reduced to solve one block RSA problem. If the block size is small enough, it is not difficult to recover M_n.

(f) It does not satisfy. One simple example is that one message contains only one block, which we say M_1, and another message is three blocks of M_1. The reason for that is the XOR result of any two or even number of identical blocks are all 0 and any message XORed with all 0 message does not change. Thus, we can conclude that it is easy for us to find two different messages that share the same hash code.