Nick Guo

I pledge my honor that I have abided by the Stevens Honor System.

Hw2 CS306

Problem 1.

The MAC tag (in hexidecimal) of the specifieddefault message using the specified default key:

8B572F8502BF835A25AB932B289CA95FB40FDC1A94F38EDCB2527121396BCFA13509E00C689C434090
A3E7A400D6573B828F3D05592A0118C44EE201A3D7512E29BA1525FA2E2E390574CEAB96BF3F3F

And when I verify it, I got:

Key

CS-306_Test_Key

Message

This is a test message. There are many like it but this one is mine.

Tag

8B572F8502BF835A25AB932B289CA95FB40FDC1A94F38EDCB2527121396BCFA13509E00C689C434090
A3E7A400D6573B828F3D05592A0118C44EE201A3D7512E29BA1525FA2E2E390574CEAB96BF3F3F

Success, tag was verified.

```java
public byte[] mac(byte[] message, Key key) {
    int bsize = getBlockSize();
    byte[] pads = pad(message, bsize);
    byte[] tag = new byte[pads.length];
    int numBlock = pads.length / bsize;
    byte[] temp = new byte[bsize];
    for (int i = 0; i < numBlock; i++){
        byte[] block = new byte[bsize];
        System.arraycopy(pads, i*bsize, block, 0, bsize);
        if (i == 0){
            try{
            temp = encryptBlock(block, key);
            System.arraycopy(temp, 0, tag, i*bsize, bsize);
            } catch (Exception e){
                e.printStackTrace();
            }
        }
        else{
            try{
                byte[] comb = xor(temp,block);
                temp = encryptBlock(comb, key);
                System.arraycopy(temp, 0, tag, i*bsize, bsize);
            } catch(Exception e){
                e.printStackTrace();
            }
        }
    }

    return tag;
}

public boolean verify(byte[] message, byte[] tag, Key key) {
    byte[] tag_1 = mac(message, key);
    return Arrays.equals(tag, tag_1);
}
```

My algorithm is that:

For mac(), since it's using CBC-MAC, and we know the block size, after we pad the plaintext, we know how many blocks there are for this plaintext. In other words, we can handle the message with any fixed size. Starting from the first plaintext block, I encrypt it with the key to get the ciphertext of the first block, and then with the ciphertext I just got, I xor it with the following block and encrypt the result with the key. And for the following blocks, I just did the same thing by using the ciphertext of the previous block, xor it, and use the result to encrypt with the key. If all blocks are done, the final result will be my tag for this plaintext.

For verify(), I first mac the message and the key to get the tag t', and check it with the passing tag t. If t = t', then we accept, else reject.

My algorithm has some features of the domain-extension because CBC-MAC is similar to it. Both of them pad the message m and view it as d blocks m1, m2 ,…, md with same size. And besides the blocks message mi, they add more things and xor them with the block message to generate the new tag/ciphertext.

Problem 2

1. Eve can collaborate with Mallory to get Bob's old public key from where Eve can access it. When someone requests Bob's public key from Mallory, Mallory can just send Bob's old public key instead of the new one. Eve then can still use the old secret key that is corresponding to this old public key so that Eve can decrypt the message and knows the plaintext and encrypt the message with the new public key of Bob from Mallory, within the new Public-key directory. Eve now can send a message to Bob and finish the man-in-the-middle attack.

2. Due to timestamped signatures, the CA can just authorize the new directory by tagging a new timestamp on the Merkle-tree hash digest on the new directory. The pointer of the key should move to the new timestamp on the hash when a new key is generated and when a user request Bob's public key. Therefore, when people get a Bob's public key, by checking the timestamp, they will know that this key is an old public key if the timestamp is out of date, so that they know Mallory is acting like an attacker(Maliciously).

Problem 3

1. The split architecture can be regarded as a double protection to improve the security, as both servers need to be compromised to make the attack successful. If the attacker wants to get the password, they need to find location and the password at the same time. Since the red server will hold the possible passwords and the blue server will hold the index of the correct password, each server will keep its own information and keep it safe so that it can make sure the password is safe. If the red server is compromised, the attacker will get a bunch of possible passwords(1 to k) and without knowing the index of the password, he/she can't get the password. When the blue server is compromised, the attacker only knows the index of the password, but he/she doesn't know what the password will look like. In this case, it's like a double safeguard so that it's hard for attackers to get the correct password.
2. Since Honeyword will create k possible passwords that are pretty similar to the correct the password, and the other server will store the correct index of the correct password, when attacker is trying to compromise a server, it needs to deal with k possible passwords which makes them more task to do in order to get the correct password. However, when a decoy password is detected, and when the attacker enter the decoy password to the server, the server will just alert to the attacker, and the attacker will know that this decoy password looks pretty close to the correct password. By doing this way, since the system only alert, the attacker can try to find the correct password by trying different passwords to get the correct password so there is a possible compromise underway.
3. Some possible honeyword passwords for pa$$word5 can be like: passwords, po$sWord4, p@ssw0rds, pa$$w0rd5, Pa$$word3, paS$word5, password7, pa$$w0rd58, pa$$w0rd!.
4. I would choose Blink-182, since it seems to be the base standard and an actual famous band compare to other possible passwords. I choose this because due to Blink-182 being an actual band that other passwords might be the honeyword or the similar passwords from this. Since people might choose their passwords according to their personal habits or hobbies or personal information. Blink-182 may be this person's favorite band and it is reasonable for him/her to choose this as his/her password. And especially the last password, it's too long and complicated so that I doubt the person will actually remember this password if it is the correct password.

Problem 4

1. RSA encryption use large prime numbers to protect the message. RSA choose two big prime numbers and multiply them to get number n = pq, where p, q is big prime number. And then choose a number e that is coprime to ϕ(n), and find e's multiplicative inverse d of e. Then the public key will be (n,e) and private key will be d. For encryption, $C = M^e \ mod \ n$, and decryption $M = C^d \ mod \ n$. It's because that $M^{ed} \equiv M \ mod \ n$.(in number theory, $ed \equiv 1 \ mod \ \phi(n)$) Since prime number is big, we have so many pairs of keys can choose and the decryption is going to be hard for attackers. By using Euclid's extended algorithm, RSA decryption can be accelerated since we can use $\mathrm{xa} + \mathrm{yb} = \mathrm{gcd}(\mathrm{a}, \mathrm{b})$ to get the multiplicative of a, which is x. Further methods to accelerate the RSA decryption/signing can be using Chinese Remainder Theorem to minimize the arithmetic operations.
2. Check the python program.