CS 4235 PROJECT 3 Reflection Report

Name: CHANG Yingshan
GT Account: ychang363
GT ID: 903457645

Task 2

1. Is the hash of the salted password still vulnerable? Why or why not?
Answer: still vulnerable!
It cannot be denied that adding salt of certain length and complexity to the original password before computing the hash is a good practice. The idea is that the salt itself is random enough to generate a hash which will not exist in a pre-computed look-up table. By eliminating the possibility of pre-computed look-up tables, an attacker is forced to go down the route of brute force attack. Even though it sounds extremely difficult, high end hardware with fast GPU can compute billions of hashes per minute. This means the hash of the salted password is still vulnerable.

2. What steps could be taken to enhance security in this situation?
Answer:
One idea to protect a password from being brute force attacked is to slow down the hash function, which is a specially crafted algorithm that is hardware intensive. This technique is called "key-stretching". Currently popular key-stretching algorithms are PBKDF2, bcrypt and scrypt.

Task 3

1. The steps that I followed to get the private key
Answer:
Get_factors: It is known that n = p*q, where p and q are prime numbers. In this function I assume p<=q. p = 2 is a special case, so I deal with it separately, by checking if n % 2 = 0. If p != 2, it is certain that p and q are odd numbers. Firstly, I found the square root of n and converted it to the nearest odd number and assigned this value to p. Starting from that, I iteratively decrease p by 2, checking if n % p = 0 at each iteration. Once I got p such that n % p = 0, n/p should be the correct answer for q.
Get_private_key_from_p_q_e: Given p, q, e, the remaining task is to find the private key d. following the steps of RSA, it is known that d*e = 1 (mod phi), where phi = (p-1)*(q-1). This means d*e = 1+k*phi, where k is a positive integer. Thus, I let a temporary variable to be 1 initially, and iteratively add phi to it, checking if the temporary variable is a multiple of e at each iteration. Once I got temp % e == 0, temp/e should be the correct answer for d.

Task 4

1. What makes the key generated in this situation vulnerable?
Answer:

In this case, the public keys n1 and n2 share a common factor other than 1. The common attack is that we can use the "greatest common factor" algorithm to find the common factor of n1 and n2, then, it would be quite easy to find the second prime factor of n1 and n2. Thus, it is easy to factor n1 and n2 into to prime numbers, and subsequent calculation of the private keys should be easy as well.

2. The steps I followed to get the private key

Answer:

I implemented is_Waldo function by applying the "greatest common factor" algorithm to n1 and n2, and checking if the greatest common factor of n1 and n2 is 1. If it is not 1, then Waldo is found. Then, in the next part, I firstly found p, the common factor of n1 and n2 by the gcd algorithm. Then n1/p should be the second prime factor of n2, which is also know as q. After that, I followed the same steps in task 3 to compute the private key d, given p, q and e.

Task 5

1. How does this attack work?

Answer:

If the same message was encrypted by three different public keys and with the same public exponent e=3, we get these three equations:

C1 = m^3 mod n1

C2 = m^3 mod n2

C3 = m^3 mod n3,

where C1, C2, C3 are the encrypted message. We can solve m^3 from the three equations. The problem is also known as Chinese Remainder Problem. After solving this problem, we can get the cubic root of m^3. This approach let us find the original message without knowing the private key.

2. The steps I followed to recover the message.

Answer:

The basic idea of the solution-forming process of Chinese Remainder Problem is described as follows.

Assume there exist alpha, beta and gamma such that:

alpha % n1 = 1, alpha % n2 = 0, alpha % n3 = 0

beta % n2 = 1, beta % n1 = 0, beta % n3 = 0

gamma % n3 = 1, gamma % n1 = 0, gamma % n2 = 0

Then, if we let m^3 = (C1*alpha + C2*beta + C3*gamma) % n1*n2*n3, this solution would fit into the three equations.

Then, we can construct alpha, beta and gamma with certain constraints in the following way:

Let n1_i = multiplicative inverse of n2*n3 mod n1,

n2_i = multiplicative inverse of n1*n3 mod n2,

n3_i = multiplicative inverse of n1*n2 mod n3.

alpha = n1_i*n2*n3, beta = n1*n2_i*n3, gamma = n1*n2*n3_i.

Then m^3 = (C1*alpha + C2*beta + C3*gamma) % n1*n2*n3.

After the value of m^3 is computed, we can find the cubic root of m^3 by binary search. This is implemented by the helper function "find_root". The basic idea is described as follows.

Firstly, I decided the searching domain by finding low, up such that up = low*2 and low <= m^3 < up. Secondly, I used binary search to find m in the range [low, up].

Reference

https://dusted.codes/sha-256-is-not-a-secure-password-hashing-algorithm
https://crypto.stanford.edu/pbc/notes/numbertheory/crt.html
https://github.com/sriram-anne2/IIS_CS6035