Project #3 All Things Cryptography

CS 6035: Introduction to Information Security

Fall 2019

Prashanth Subrahmanyam
prashsub@gatech.edu

Contents

Task 2 – Can I Have Some Salt With My Password?

Why is this hashing scheme insecure?

This hashing scheme is insecure, because at the fundamental level, the password itself is too weak, as it's from the 1000 most commonly used passwords on the internet. More importantly, even the salt was from the same list of 1000. This allows a hacker to generate the hash values of each possible password and salt combination that could be possible – i.e. a mammoth hash value tables or the rainbow table. This attack is based on trading off time for space, by precomputing and storing hash values.

What steps could be taken to enhance security in this situation?

This hashing scheme can be secured by the use of a larger salt value as well as making sure that the salt values are not being taken from the commonly used password list. A larger salt value makes it difficult for the attacker to store entire data in the memory, as the rainbow table will become too large. Similarly, selection of a unique salt from a randomized set rather than commonly used list of passwords would make it difficult for the attacker to create a good rainbow table.

Additionally, of course, if possible, the password also should be set to something unique and not from a list of 1000, making it even harder for the attacker to guess the password and sat.

Task 3 – Attack A Small Key Space

What steps did you follow to get the private key?

Since, the key size was small in size (64 bit), I first observed that this is a factoring vulnerability, as it didn't satisfy the requirement for public-key encryption, that it

should be infeasible to determine d, given e and n. In this case, it was feasible to determine d, since n wasn't a sufficiently large number, and could be factored easily.

First, I found the factors of the given public key, "n". I did this using the brute force method, although to keep the process fast, I limited the search space to square root of "n".

Once I knew the factors for the public key "n", I could calculate the Euler totient function for n, which essentially is a count of the integers that are smaller than n and are relatively prime to "n".

In summary, by finding the factors "p" and "q" of public key "n", we were able to calculate the Euler totient function of "n". Using this function, in conjunction with "e", which is also part of the public key, we can determine "d" which is part of the private key. This can be calculated using the following equation.

$$d \equiv e^{-1} (mod\ \emptyset(n))$$

To solve this, we use the Extended Euclidean Algorithm which allows us to compute integers x and y, given "a" and "b", such that:

$$ax + by = \gcd(a, b)$$

In this algorithm, we supply the value of e and the Euler Totient function as "a and "b" and calculate the resulting value of "d".

Task 4 – Where's Waldo

What makes the key generated in this situation vulnerable?

The key generated in this situation is vulnerable because the Random Number Generator (RNG) is vulnerable due to inadequate/insufficient randomness. This inadequate randomness may be since sometimes, the RNG may have limited entropy sources. Henninger, Durumeric, Wustrow and Halderman (2012) found that one of the reasons could be "Linux's RNG can exhibit a boot-time entropy hole that can cause

"urandom" to produce deterministic output, under conditions likely to occur in headless and embedded devices".

What steps did you follow to get the private key?

The Task 4 provided the information that all the public keys for both me and all the classmates were generated on a vulnerable RNG on the same system.

Henninger (2012) described that if two different moduli shared only 1 factor which was prime, instead of both unique prime factors, this could result in a vulnerability. The vulnerability is that, although the public keys may appear distinct, we could easily calculate the common prime factor, by calculating the Greatest Common Divisor (GCD). Calculating the GCD is a relatively easy task, compared to factoring a number, and can be done, even on a secure 1024-bit key.

Once we calculate the common prime factor using GCD, we can easily calculate the other prime factor as well, and thus, we would effectively have "factored" the public key "n" to have calculated the factors "p" and "q".

Once we have the factors, it becomes a simple factorization vulnerability and the private key can be calculated using the following equation.

$$d \equiv e^{-1} (mod\ \emptyset(n))$$

We can solve this equation using the Extended Euclidean Algorithm, as described in above in the reflection for Task 3.

Task 5 – Broadcast RSA Attack

How does this attack work?

This attack is also known as the Coppersmith's attack, and it is a specific case from a class of attacks based on the Coppersmith method. In this case, this is specifically also known as Håstad's broadcast attack.

The premise of this attack is as follows is based on the same encrypted message being sent to multiple recipients, with each message being encrypted using a small public exponent (in the given case, e =3). In such cases, the message is no longer encrypted securely, as soon as it sent to 3 or more recipients.

This is because now there is one of the two following vulnerability:

1. The three messages have different moduli (N_i, e) such that their GCD is 1, i.e. they do not share a prime factor
2. The three messages have a GCD >1

In the first case, we could solve the equation for C using the Chinese Remainder Theorem such that:

$$C_i \equiv C \times mod\ N_i$$

From this, once C is calculated, we can further calculate the plain text message using $C = M^i$ and thus break the encryption.

In the second case, if the GCD>1, we can easily compute the shared prime factor, and thus reduce this to a factorization attack.

What steps did you follow to recover the message?

I first observed that the task had mentioned that this was an RSA Broadcast Attack, and that the public exponent was provided as e=3.

Additionally, the problem statement had provided us with the 3 encrypted messages, C1, C2, and C3 as well as the 3 public keys N1, N2 and N3.

Thus, we could write the 3 encrypted messages such that:

$$C_1 \equiv M^3 \times mod\ N_1$$
$$C_2 \equiv M^3 \times mod\ N_2$$
$$C_3 \equiv M^3 \times mod\ N_3$$

Re-arranging the equations, we could solve the following set of equations using the Chinese Remainder Theorem:

$$M^3 \equiv C_1 \times mod\ N_1$$
$$M^3 \equiv C_2 \times mod\ N_2$$
$$M^3 \equiv C_3 \times mod\ N_3$$

Steps to solve using Chinese Remainder Theorem

First, I defined N as a product of all given public keys N1, N2, N3.

Secondly, for each given public key, we can further define another variable, $b_i = N // N_i$ where in b_i and N_i are relatively prime to each other. This is a floor division, and hence only integer value of the division will be retained and not a float number.

Third, for each given public key, we can compute $z_i \equiv b_i^{-1} * \text{mod } N_i$ using Extended Euclidean Algorithm.

Fourth, we multiply each of these values as below to get the following set of equations:

$$x_1 \equiv C_1 \times b_1 \times z_1$$
$$x_2 \equiv C_2 \times b_2 \times z_2$$
$$x_3 \equiv C_3 \times b_3 \times z_3$$

Finally, we add all these values as below:

$$x \equiv (x_1 + x_2 + x_3) \times mod\ (N_1 N_2 N_3)$$

Now that we know x as well as value of e=3, we can calculate the plaintext message M as an integer with a cube root of x as below:

$$M = \sqrt[3]{x}$$

Bibliography

Chinese Remainder Theorem. Brilliant.org. Retrieved 19:15, November 3, 2019, from https://brilliant.org/wiki/chinese-remainder-theorem/

Extended Euclidean Algorithm. Brilliant.org. Retrieved 11:12, November 3, 2019, from https://brilliant.org/wiki/extended-euclidean-algorithm/

Heninger, N., Durumeric, Z., Wustrow, E., & Halderman, J. A. (2012). *Mining your Ps and Qs: Detection of widespread weak keys in network devices*. In Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12) (pp. 205-220).

Stallings, William, and Lawrie Brown. *Computer Security: Principles and Practice*. 4th ed., Pearson Education, Inc., 2018.

FGrieu. "Deciphering the RSA encrypted message from three different public keys" *Stack Exchange*, 24 Oct. 2017, https://crypto.stackexchange.com/questions/52504/deciphering-the-rsa-encrypted-message-from-three-different-public-keys

Wikipedia contributors. (2019, July 1). Coppersmith's attack. In Wikipedia, The Free Encyclopedia. Retrieved 01:41, November 4, 2019, from https://en.wikipedia.org/w/index.php?title=Coppersmith%27s_attack&oldid=904277988