Project 3: CS6035

Alexis Oswald
aoswald9@gatech.edu

1 TASK 2

In order to attempt to prevent the use of common passwords, safeguards can be put in place when passwords are created to avoid passwords being created without numbers and/or special characters. By requiring minimum password lengths, variation in capitalization, and requiring special characters/numbers to be included in the password, it will make the chosen password more difficult to guess. The more random the combination of letters, numbers, and special characters are, the more difficult it will be for a password cracking tool to guess a password (*"Help secure your accounts"*, 2020).

2 TASK 3

Another consensus mechanism that can be used as an alternative to proof-of-work is called proof-of-stake. Instead of mining for coins as is done in the proof-of-work mechanism, coin transactions are validated using 'forging' in proof-of-stake, which means that no actual mining occurs in the proof-of-stake mechanism (*"Proof of Work vs Proof of Stake Comparison"* 2020).

One of the most prominent benefits of the proof-of-stake mechanism is that due to the fact that it does not require as much technology (because it does not require any actual computational work) it is much more energy efficient. This also makes proof-of-stake more environmentally friendly and removes any gateway barriers for people to get involved because high tech computers with high computing power is not necessary (Greenfield, 2017).

However, a downfall to the proof-of-stake method is the concern that proof-of-stake is "helping the rich get richer" due to the fact that the winner of each block is randomly chosen based on the amount of money that they have staked. The same can technically be argued about the proof-of-work mechanism, though, because of the cost of the amount or the power of technology that is necessary to solve the complex sum in order to win the transaction fee (which is the overall reward) (M., 2020).

3 TASK 4

The first step that was taken in order to find the private key was to find the prime factors (p and q) of the given value of n. In order to do this, I implemented a prime factorization method that was called within the `get_factors` metho. This was an implementation of a prime factorization method called "Brent's Factorization Method". What this factorization method allows us to do is to quickly find the prime factors by identifying when there is a cyclic trend in the values that are being checked, in order to speed up the process. This is optimized in Brent's al-gorithm by checking if the current index is an integral power of 2, which allows us to bypass the unnecessary check of many values (Barnes, 2004).

Brent's method of factorization is based off of Pollard's Rho's method of factorization, which gained popularity due to the identification of periodicity that it included which helped optimize it for large numbers (https://en.wikipedia.org/wiki/Pollard's_rho_algorithm). Using a brute force method of checking for the two prime factors of a number would be impossible within a reasonable amount of time for numbers as large as the ones we are dealing with in RSA (Barnes, 2004).

Once we run the prime factorization method to obtain the values of p and q, we needed to use those values to find the totient using the given formula, $\varphi(N) = (p-1)*(q-1)$. Then, $\varphi(N)$ had to be used in another formula provided in the project instructions, $d = e^{-1} \bmod \varphi(N)$. However, in order to solve the equation for 'd', the private key, we had to find the modular inverse (due to e^{-1} in the private key equation).

In order to find the modular multiplicative inverse, I utilized an algorithm called "Extended Euclidean Algorithm". This is essentially Euclid's algorithm that is used to find the greatest common denominator, but in the reverse direction. Due to the nature of 'p' and 'q', and the fact that we found them through the prime factorization of a number, we know that 'p' and 'q' are coprime, and therefore the greatest common denominator of these two values is always 1 ("*Calculating RSA private exponent when given*", 2014).

Therefore, not only does the extended Euclid's algorithm compute the GCD of e and φ(N), but it also computes the value of x and y in the equation ax+by=gcd(a,b), the coefficients of Bézout's identity. It then uses those coefficients to solve the equation until the remainder is zero, signaling that factors were found ("Extended Euclidean algorithm", 2020). Once this algorithm has completed, it will return 'd', the private key that we were looking for.

4 TASK 5

The public key that is used in this task is vulnerable due to the random number generator that was used while generating the public keys. Because of the faulty random number generator not being truly random, we were able to factor the public keys until we found a pair that had a greatest common denominator. The reason that it is possible for key generation to have issues is the chance of entropy issues in a system; due to the ease for the greatest common denominator to be calculated, if there are two RSA moduli that share a factor, that is a major security issue because that allows for the private key to be calculated (Heninger, Durumeric, Wustrow, & Halderman, 2012).

In order to derive the private key, I first traversed the list of public keys provided as a parameter to task 5. For each public key, I computed the greatest common denominator of 'given_public_key_n' and the private key at the current index. If the GCD calculated was greater than 1, then that indicates that a value does exist that divides evenly into both 'given_public_key_n' and the current public key from the list. And if a factor does exist, then that means that the same random prime can be divided into both, and the other factor can also be found. The only way that this can occur is if there is a fault in the random number generator that was involved in the generation of the private keys, due to the generation not truly being random (Heninger, Durumeric, Wustrow, & Halder-man, 2012).

Once the two associated keys are found, I then calculated the GCD of 'given_public_key_n' and the current private key in the list and stored that value as 'p'. Then, I divided 'given_public_key_n' by 'p' using integer division (to avoid inadvertent conversion to float) and stored that value as 'q'. This gives the two factors of the n value of the public key.

Finding the two factors 'p' and 'q' provides us with enough information to find the private key by using the same function that was used in task 4: computing φ(N), and then using that value in the Extended Euclidean algorithm in order to compute the modular inverse. After 'get_private_key_from_p_q_e' is called, the private key that we were looking for is returned ("*Calculating RSA private exponent when given*", 2014).

5 TASK 6

The broadcast RSA attack works by using the three public keys, 'n_1', 'n_2', and 'n_3', and the public exponent, 'e', to decrypt and find the original message from the cipher texts 'c_1', 'c_2', and 'c_3'. We are able to do this using the application of Coopersmith's Theorem called Hastad's Broadcast Attack, due to the same message being encrypted and sent at least three times. This attack works when one unique message, 'm', is encrypted using 3 different public keys and results in 3 different encrypted messages, 'c_1', 'c_2', and 'c_3'. Meanwhile, each ciphertext follows the equation $C_i = M^3 \bmod N_i$, where 'i' is the index of the encrypted message (Alrasheed, "*RSA Attacks*").

If there was a pair of n values that were given that were not coprime, then we could factor them as we did in task 4 and work from there to decrypt the message. Instead, we have to use the Chinese Remainder Theorem in order to get a value of 'c' that satisfies the following: $0 \leq c < N_1 * N_2 * N_3$. Also, by the Chinese Remainder Theorem, it is true that $c = m^3 \bmod N_1 * N_2 * N_3$, and therefore $c = m^3$ by the remainder theorem (Trenwith, 2018).

To calculate the value of 'c' using the Chinese Remainder Theorem, I developed a function called 'find_crt' which used all of the values of 'n' and 'c' that were provided in the task as parameters and added them into two separate arrays- 'c_vals' and 'n_vals'. This allowed for iteration through the arrays as the calculations were occurring. First the product of all N's was calculated, then that product was divided by each N value, and the resulting quotient was used with its corresponding n value in the Extended Euclidean algorithm in order to return the bezout value. That bezout value is then used in the calculation of 'ans' which is the original value of 'c' multiplied by the bezout value and the quotient. The sum of all 3 derived 'ans' values were summed, then the function returned the value of 'ans' mod 'product' to give us our value of 'c' ("*Chinese re-mainder theorem*", 2020).

From there, the only thing left to do was to take the cubed root of our 'c' value, which gave us the integer value of the message, 'm', which was done by a custom cubed root function, 'find_root'. The built-in root function in Python cannot handle numbers as large as we have to deal with in RSA, which is the reason for a custom function. I implemented a function that used binary sort in order to find the correct value of the cubed root ("Find cubic root of a number", 2018). I did not implement any kind of error handling due to the nature of the project (all input is expected to be valid and functional). Once we have the cubed root, we have the integer value of 'm' that is then converted to get the actual message text.

6 REFERENCES

1. Alrasheed, A. (n.d.). RSA Attacks. Retrieved November 1, 2020, from https://www.utc.edu/center-academic-excellence-cyber-defense/pdfs/course-paper-5600-rsa.pdf
2. Barnes, C. (2004, December 07). Integer Factorization Algorithms. Retrieved November 1, 2020, from http://connellybarnes.com/documents/factoring.pdf
3. Calculating RSA private exponent when given public exponent and the modulus factors using extended euclid. (2014). Retrieved November 01, 2020, from https://crypto.stackexchange.com/questions/5889/calculating-rsa-private-exponent-when-given-public-exponent-and-the-modulus-fact
4. Chinese remainder theorem. (2020, October 13). Retrieved November 02, 2020, from https://en.wikipedia.org/wiki/Chinese_remainder_theorem
5. Extended Euclidean algorithm. (2020, October 09). Retrieved November 01, 2020, from https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm
6. Find cubic root of a number. (2018, May 29). Retrieved November 02, 2020, from https://www.geeksforgeeks.org/find-cubic-root-of-a-number
7. Greenfield, R., IV. (2017, August 24). Vulnerability: Proof of Work vs. Proof of Stake. Retrieved November 01, 2020, from https://medium.com/@robertgreenfieldiv/vulnerability-proof-of-work-vs-proof-of-stake-f0c44807d18c
8. Help secure your accounts with these strong password tips. (2020, January 07). Retrieved October 31, 2020, from https://us.norton.com/internetsecurity-how-to-how-to-secure-your-passwords.html
9. Heninger, N., Durumeric, Z., Wustrow, E., & Halderman, J. (2012, July 11). Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network

Devices. Retrieved November 1, 2020, from https://factorable.net/weakkeys12.extended.pdf

10. M., L. (2020, September 11). Proof of Work vs Proof of Stake: What's The Difference? Retrieved November 01, 2020, from https://www.bitdegree.org/crypto/tutorials/proof-of-work-vs-proof-of-stake
11. Proof of Work vs Proof of Stake Comparison. (2020, September 26). Retrieved November 01, 2020, from https://www.devteam.space/blog/proof-of-work-vs-proof-of-stake-comparison/
12. Trenwith, B. (Director). (2018, July 13). *Discrete Maths for Computer Science - 4.4.2.5 - Hastad's Broadcast Attack* [Video file]. Retrieved November 1, 2020, from https://youtu.be/DKWnvyCsh9A