Project 3:
CS6035

Marcus Anderson
manderson332@gatech.edu

1 TASK 2

The purposes of salt values are to add random hashed data to an existing password, effectively making the password less vulnerable to attacks. This unique addition protects passwords against different attacks like hash table, which cracks passwords stored in a database using a rainbow hash table. The salt approach also helps slow down brute force attack methods like the dictionary offline attack. These two reasons are why this scheme would effectively protect a "weaker" password. This approach is not perfect and there are still ways of defeating it. Since hashed passwords are not unique, the same password would actually have the same hash value. Regardless of this, salt values still heavily mitigate the risks of common password attacks, which makes this approach so effective.

One way I would improve password security would be through two-factor authentication. This adds a second layer of security, in addition to a password, when a user logs into their account. Some of the most common examples is a confirmation text or email that gets sent to the registered user's account to confirm who they are after entering the correct password. This helps deter attackers who may know a victim's password, but cannot access the confirmation code.

2 TASK 3

The proof-of-work scheme is a common example of a consensus mechanism participant. Another common alternative is the proof-of-stake participant, which processes transactions and creates new blocks in a blockchain (Frankenfield 2021). This schema allows owners to validate block transactions based on the quantity of coins a validator stakes during a transaction. This means the owners offer up their coins as collateral damage for block validation. Blocks can only be validated by more than one validator, and once the predetermined number of validators verify the accuracy of a block, it's then closed.

Some strengths of the proof-of-stake mechanism is that it picks the validators randomly to validate a block, rather than proof-of-work, which is more competition-based. This "levels the playing field" a bit more for investors since they cannot gain any sort of advantages on the competition. This allows the proof-of-stake mechanism to resolve the scalability and environmental sustainability concerns that proof-of-work is plagued with.

On the other hand, the proof-of-stake incentivizes the gathering of coins to increase the opportunity of winning a block and reaping the benefits. This mechanism allows individual owners to create any number of validators, at a very low upfront cost. If an individual controls the majority of tokens, they could also control the majority of the work (Conway 2022).

3 TASK 4

Finding the initial signature was relatively straightforward since we were given both the private key (d) and modus value (n). Using the hashed transaction value as our cipher (c), we are able to use the decryption formula to find the signature (m). In the scenario, since the public key of the signer passed the signature value directly into the function, we're able to skip this step. However, even though the signature is given to us in this scenario, it's still important to note that the signature value is derived from the integer transaction value. This is important because we now know that in order to check the validity of a transaction value, we need to confirm that the signature value that is passed through is equal to the value m calculated from the decryption formula. In order to accomplish this, we just need to calculate c, using the encrypt formula, and confirm that this value is the same as the transaction value. For this we would use the values e and n that were given, and the public key signature provided as our m value. Once we get our "verify check" signature value, all we then need to do is compare this to the transaction value and return the resulted boolean outcome.

4 TASK 5

In order to find the private key, I needed to reverse engineer the public key, which is built from integer pairs e and N. In this task, I was already provided these values, so the first step towards finding the private key involved finding the values of p and q. The formula to find N is as follows: $N = p*q$, where p and q represent big prime numbers.

Knowing the formula, I knew that I only needed to find either p or q to find the other, since I can then use N / *(p or q)*. This called for a factorization formula that could efficiently find the prime factors of a large integer (N). There are several existing formulas out there that could be used to achieve this, such as: Trial Division, Fermat's Factorization and Miller-Rabin. During the beginning of my development phase for this step, I tried to implement Fermat's method, which is based off the formula: $N = a^2 - b^2$, and uses the various values of a until $a^2 - N = b^2$ results in a square. I would then use $a - sqrt(b^2)$ and $a + sqrt(b^2)$ to find p and q. I was able to get this method to work for the provided test case, but none of the other edge cases I found. This is because Fermat's method works well on smaller numbers. The bigger the number, the more inaccurate the result may be. Because of this, I pivoted over to a more brute force tactic, like trial division, where I loop through different values until I found one that satisfied: N % *value* == 0. Although it's more inefficient, I was able to consistently get correct results for any edge case.

After finding p and q, the next step was to find the value of φ(N), from the formula: $\varphi(N) = (p - 1) * (q - 1)$. This step was pretty straightforward, since I already found values p and q. Finally, in order to get the private key (d), I needed to find the modular inverse of e using the underlying mathematical principal: $e^{-1} \bmod \varphi(N)$ or its equivalent, $d*e = 1 \bmod \varphi(N)$. I first need to find the greatest common divisor (GCD) between e and $\varphi(N)$ using the Extended Euclid's GCD method. This is a recursive method used to find the only number that can satisfy the equation $(d * e) \bmod \varphi(N) = 1$, where d is the final private key.

Once the GCD value (private key), is returned from this method, I plugged the value into the equation stated earlier to confirm that the result is 1. Once confirmed, I check to make sure the private key is a positive integer, if not, I add $\varphi(N)$ to return the correct positive private key. If the private key is already positive, I just return that by default.

5 TASK 6

There are various vulnerabilities when it comes to RSA public keys, one of the most common ones displayed in this task are *repeated keys*. Even though this is a common vulnerability, this is sometimes an approach that's implemented on purpose. This is because many of the most commonly repeated keys appeared

in shared hosting situations (Heninger et al., 2012). These types of keys are known as default keys, that are preconfigured within the firmware of several devices that shares the same key pair. Through reverse engineering, we are able to find the private key. Task six showcases this vulnerability through the given list of one hundred public keys and one default public key. Using the underlying mathematical principle: greatest common divisor (GCD), the first step would be finding which key pair shares a common divisor. That would be the p value we could use to find q, or vice versa, in association with our default key. Once p and q are found, the last step to finding our private key would be to use the modular inverse formula, and calculate the private key. This is how someone can reverse engineer a private key from a list of repeated keys, and judging from the solution I programmed for this task, it only took less than a second to achieve.

I gave a brief overview on how we could reverse engineer the repeated key scenario in the previous prompt. Now I'll provide a more granular explanation of each step involved in my solution. As explained before, I first needed to find which key in the list of one hundred public key list coupled with the given public key passed in the function has a GCD greater than 1. If there is no common divisor between the given key and the iterated key in the list, the GCD would result in 1, hence the greater than 1 check mentioned earlier. Out of the list of public keys, there was one key coupled with the given public key that resulted in a GCD value greater than 1. Once found, I set the GCD value to p and broke out of the loop. The next step was to find q, with the formula $q = n / p$, and plug p, q, and e into my modular inverse function. Now comes the final step of our reverse engineering process, using the modular inverse formula to find the private key using the underlying mathematical principal: $d*e = 1 \mod \varphi(N)$. Once I find $\varphi(N)$, the next step is to find the GCD of values e and $\varphi(N)$ using Extended Euclid's GCD method. Once this value is successfully extracted, I checked to make sure our private key value satisfied the equation above. The final check was to see if the private key was negative, and in this case, the calculated value actually was negative. After adding the value of $\varphi(N)$ to this, I was able to get a positive integer, and return the now finalized private key.

6 TASK 7

The RSA broadcast attack, commonly known as Hastad's broadcast attack, is an improvement on Coopersmith's theorem. Let's say one party wants to send an

encrypted message to a certain number of parties, each with their own RSA public key. This same party encrypts these public keys to their respective ciphertexts. The vulnerability comes in when the attacker can spy on this connection and collect the number of transmitted ciphertexts (lists of N and c). The attacker can apply the Chinese Remainder Theorem (CRT) on the amount of cipher texts that satisfies $C' = M^3 \ mod \ N_1 \ N_2 \ N_3$, if there were three messages sent for example. Once the attack finds C', all they would then need to do is find the real cube, or n^{th}, root of this value to get the message integer, which is then decoded into plain text.

As stated in the previous prompt, the main steps I implemented to find the plain text message from the list on public keys and cipher texts were CRT and an n^{th} root method. The first step was finding C' from the CRT using both public key and cipher text lists provided, three of each. During the CRT step, I first calculated the product of all three values in the public key list, and labelled this value m. Then, while looping through the list of public keys, I first got the value of n' from the equation $n' = m \ / \ n[i]$, i being the iterated list value. This is then applied to the equation: $x = c[i] * n' * modular \ inverse \ of \ (n', n[i])$, for each public key and added to variable x. Once the loop is complete, I return the result of $x \ mod \ m$.

After getting C', I then had to find the cubic root of this result to get the final m value. Since the size of the integer is so large, I was not able to calculate this using the built function $pow(crt, (1/3))$. So instead, I used a binary search-based method to find the root of C'. This is the most efficient and effective route I found to find the root of such a large integer. Once the binary search was complete, the value returned was my final message integer that was used in the conversion method provided to generate the plain text message.

7 REFERENCES

1. Arias, D. (2021). Adding Salt to Hashing: A Better Way to Store Passwords. Autho - Blog. Retrieved 2022, from https://autho.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/

2. Bauer, G. (2018). Programming Blockchains Step-by-Step from Scratch (Zero). Starting with Crypto Hashes... Programming blockchains step-by-step from scratch (zero). starting with crypto hashes... (Book edition).

Retrieved 2022, from https://yukimotopress.github.io/programming-blockchains-step-by-step

3. Chang, R. (2009). The Extended Euclid Algorithm. UMBC CMSC203, Discrete Structures, Spring 2009. Retrieved 2022, from https://www.csee.umbc.edu/%7Echang/cs203.s09/exteuclid.shtml#:%7E:text=Here's%20the%20pseudo%2Dcode%20for,%2Ct)%20%3B%20%7D%20Note%20that

4. Conway, L. (2022). Proof-of-Work vs. Proof-of-Stake: Which Is Better? Blockworks. Retrieved 2022, from https://blockworks.co/proof-of-work-vs-proof-of-stake-whats-the-difference/

5. Math 5410 Factoring. (n.d.). Math 5410 Factoring. Retrieved 2022, from http://math.ucdenver.edu/%7Ewcherowi/courses/m5410/ctcfactor.html

6. Pitts, J. (2005). Chinese Reminder Theorem. Chinese Reminder Theorem. Retrieved 2022, from https://www.math.tamu.edu/~jon.pitts/courses/2005c/470/supplements/chinese.pdf

7. Rasure, E. (2021). Proof-of-Stake (PoS). Investopedia. Retrieved 2022, from https://www.investopedia.com/terms/p/proof-stake-pos.asp

8. Tutorials Point. (2021). Data Structure and Algorithms Binary Search. Retrieved 2022, from https://www.tutorialspoint.com/data_structures_algorithms/binary_search_algorithm.htm

9. Wikipedia contributors. (2022). Modular multiplicative inverse. Wikipedia. Retrieved 2022, from https://en.wikipedia.org/wiki/Modular_multiplicative_inverse

10. YouTube. (2019). Chinese Remainder Theorem. Retrieved 2022, from https://www.youtube.com/watch?v=zIFehsBHB8o&ab_channel=MathswithJay