Dgoodrick3

CS6035 InfoSec

Project 3 writeup

Task 2

Is the hash of the salted password still vulnerable? Why or why not?
Response: Yes. Aside from the normal vulnerabilities, this case uses a common password for the salt. Hackers already have the hash for these common passwords so it isn't much more secure than using an unsalted common password.

What steps could be taken to enhance security in this situation?
Response: Don't use common passwords for salt. Use something unique, like a userID or a random key. Use HMAC-SHA1 to hash the password database since it uses a secret key in the hash process.
https://benlog.com/2008/06/19/dont-hash-secrets/

Task 3

What steps did you follow to get the private key?
Response: With p and q provided, the totient is easy to calculate – (p-1)*(q-1). With the totient, I just solved for d. I used $e*d \equiv 1 \bmod \varphi(N)$ to find a d that would agree with this congruency. Solving for d requires calculating the modular inverse of e such that $d \equiv e^{-1} \bmod \varphi(N)$. Inverting e required two helper functions – one to run the extended Euclidian algorithm on e and φ(N) and one to convert the result to a positive integer (and check that the solution is valid). I used the extended Euclidian algorithm on wikibooks.org as a guideline for both helper functions.

https://en.wikibooks.org/wiki/Algorithm_Implementation/Mathematics/Extended_Euclidean_algorithm#Python

Task 4

What makes the key generated in this situation vulnerable?
Response: The public key (N) is the product of two primes, p and q. In this case, the primes, p and q were reused for the creation of other public keys. This means any public keys that have a common factor can be broken down into the known factor (p in our case) and the unknown factor (q in our case). With p and q known, the totient of both keys can be calculated the attacker can solve for the private (unknown) using the steps outlined below. The project write-up claims that a faulty random number generator was used to produce the numbers, and it is true that a lack of entropy in key generation reduces the security of the encryption. However, that weakness was not exploited in my research so I can't speak to its role in the situation.

What steps did you follow to get the private key?
Response: First I searched for public keys that had a common factor (besides 1) with my public key. Then I took that common factor, p, and calculated q with integer division of my public key with p since p*q = N. With that I could calculate the totient (the number of relatively prime factors between two prime factors) by multiplying (p-1)*(q-1). This was really easy since p and q are prime numbers; all numbers less than p and q are relatively prime to p and q. With the totient and e, I was able to solve for d; $d \equiv e^{-1} \bmod \varphi(N)$.

Task 5

How does this attack work?
Response: When a single message M is encrypted with a number of public RSA keys, the result is N different ciphers that all represent the same underlying message. If an attacker can intercept enough of these ciphers, and if they somehow understand that they are all copies of M, their common congruency can be exploited with the Chinese Remainder Theorem to recover the original message M. The number of necessary ciphers is the value of e used in the RSA key. In our case, it was 3. The way to counteract this attack is to use large, sparse e values. The values of e should be sparse because of the Hamming distance so modular exponentiation is efficient (performance is not adversely affected). A common, safe value for e is: 0x10000001. Also, one should use randomized (not linear) padding so no ciphers actually represent the same message (M+padding). Note: the UTC paper referenced in this section (and other places) claim that k > e but in our case, we were successful with k=e. I am not sure where their claim came from or why it wasn't necessary in our case.

https://www.utc.edu/center-academic-excellence-cyber-defense/pdfs/course-paper-5600-rsa.pdf

https://crypto.stackexchange.com/questions/5572/in-rsa-encryption-does-the-value-of-e-need-to-be-random

What steps did you follow to recover the message?
Response: Given I instances of N, calculate the product of all N_i, called pN. Divide pN/N_i to get array n. Add the ciphers C_i to array C. Compute the inverse modulus, X_i, of each n_i to get the coefficients for the following congruency: $X_i = 1 \bmod (N_i)$. The Chinese remainder theorem says that $C^* = \text{sum}_{\text{over all } i}(C_i * X_i * n_i)$. $C^* = M^3 \bmod (pN)$ so M reproduced by modding C* by pN and taking the cube root of the remainder to produce M. However, in our Python implementation, it was necessary to find an alternate method of taking the cube root of M since RSA integers are too big to be converted to floats. I used binary search to find M by comparing mid^3 to C*/pN and returning the lower bound once the search zeroed in on a solution (lower>upper).