Pledge: *I pledge my honor that I have abided by the Stevens Honor System.* - **Eric Altenburg**

Problem 1: Domain-Extension MAC implementation

The use of `https://en.wikipedia.org/wiki/CBC-MAC` was useful during the implementation of the `mac()` and `verify()` methods.

(1)

The file `Mac.java` that contains the implementation of the `mac()` and `verify()` methods can be found in this zip file.

(2)

They were proven correct through the use of this MAC tag after using the default key:
`29BA1525FA2E2E390574CEAB96BF3F3F`

(3)

The algorithm I implemented was the CBC-MAC one because of its use of block chaining that builds upon the previous blocks encryption. With the use of this, the attacker would still not be able to crack the algorithm and predict the plaintexts since they would need to know the key to the block cipher.

(4)

My implementation of CBC-MAC only securely handles messages of fixed lengths because the key used can is only used on messages of a fixed length. Because of this, the algorithm is susceptible to an attack in which the attacker can input messages of varying length. Say we pass a message m_1 though the algorithm and get a tag mac_1, then XOR that with into the first block with a variable length message m_2, then we can obtain the Mac on the modified version of m_2 which is essentially the Mac of the combined m_1 and m_2. However, this can be prevented by simply adding the message length into the first block for message m_1 before copying it to m_2.

Problem 2: Data outsourcing & public-key infrastructure

(1)

Eve can collaborate with Mallory to decrypt all subsequence messages sent to Bob by using a replay attack. While it is true that Eve was able to get access to Bob's secret key sk_B, it is now practically useless because Bob ended up changing his public-key pair. Since Mallory has access to all of the records that contain Bob's public keys along with proof from a CA, she can then send Bob's old public key when people, like Eve, request his new public key. This is significant because Eve still has the secret key sk_B which will correspond to his old public key. The old key she'll receive from Mallory will also be verified because it was previously verified from the CA. Therefore, with the help from Mallory, Eve can now successfully see all the incoming messages to Bob without his knowledge.

(2)

Timestamped signatures can be periodically used by the CA to provide a solution to the above replay attack by allowing the users to know whether or not they are being subject to an attack. Before, Bob was unaware that he was victim to a replay attack and that there was fraudulent behavior with this public keys. With these timestamped signatures, whenever Bob updates his public key, the CA can place a timestamp on this request; he will then know when Mallory updated his key. So with this in place now, if Mallory were to alter his public key using the Merkle-trees, he will know something is wrong based on the timestamp and can therefore take the necessary actions to remedy the situation.

Problem 3: Intrusion resilience

The use of the following paper `https://people.csail.mit.edu/rivest/pubs/JR13.pdf` was helpful in understanding the applications of honeywords.

(1)

The split architecture improves security because the authentication required depends on both of the two servers (the red and blue given in the example from the prompt). Each of the two servers will host different password verification methods, hence why access to only one provides little to no advantage for an attacker. With one server holding all of the passwords which contains one correct one while the rest are honeywords, the other stores the honeychecker which has the indices of the users' actual passwords. When a user inputs their password, the first server will end up verifying that the input is found within the user's set. If it is found, then it will return the password and the indices for the second server to then verify the password. If the password ends up not being valid and is instead a honeyword, then it will end up sending a notification letting the user or system admin know the situation is a sign of potential theft or other threat. For example, if the attacker has access to only the first server, then it can get a list of potential passwords to choose from but without the second honeychecker server, the attacker will not be able to properly authenticate the *password* they chose. On the other hand, if the attacker has access to the honeychecker, then they will not be able to supply a valid password to begin with and they will fail the first server's verification not even making it to the second server, thus leaving them no better than when they started.

(2)

Honeywords make password cracking detectable because say an attacker manages to get access to the first server mentioned above where it holds both the correct user password and all honeywords; which it should be noted that all the honeywords are going to be very similar to the actual password making the guess for the right one really difficult to get right with only one try. When attempting to login with what the attacker thinks is the password but is in reality a *honeyword*, the second server (honeychecker) will know that a honeyword was used and can then notify the user that an attacker had tried to gain access to their account.

(3)

Real Password: `pa$$word5`

Possible Honeywords:

1. `pa$$word1`

2. `pa$$word6`
3. `pa$$word9`
4. `pa$$w0rd5`
5. `pAs$word5`
6. `PA$$WORD5`
7. `Pa$$word3`
8. `Pa$$W0rd5`
9. `pa$$words`
10. `pas$words`

There are plenty other different honeywords that can be made out of the real password, but the above ten were just a few.

(4)

I would choose `Blink-182` from the honeywords list as the actual password. The reason for this is because `Blink-123` seems to be an attempt at an easy variation of `Blink-182`. `itWb!%s453gMoI00286!*mooewTi409##21jUi` is simply far too complex for any human to realistically create and remember; there would be too much room for error in inputting characters. Finally, the reason why I think `Blink-182` is not a honeyword and is the password is because it is also a band, and humans tend to want their passwords to be more significant this way they are easy to remember. Therefore, it could be reasonable to assume that Blink-182 might be a registrar employee's favorite band.

Problem 4: On the RSA cryptosystem

The use of `https://en.wikipedia.org/wiki/RSA_(cryptosystem)` was particularly useful in helping me understand the workings of key generation. Other useful sources were these two links `https://crypto.stackexchange.com/questions/2575/chinese-remainder-theorem-and-rsa` and `https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture12.pdf` which showed me the way in which the Chinese Remainder Theorem can apply to the RSA decryption.

(1)

Modular exponentiation can be written as $b^e \% m$. Found in `rsa_skeleton.py`, it is implemented in a more efficient way by constantly checking to see whether e is odd, and if it is, the result gets updated and then e is bitshifted to the right one time ($e = \frac{e}{2}$). This is done until e is no longer greater than 0. In a naive implementation, the calculation of b^e can be extremely slow as it will grow to extremely large numbers causing the mathematical computation to take longer as well.

The way in which RSA uses this fast implementation of module exponentiation allows it to encrypt and decrypt messages. Given a message m with a public key(e, n), RSA will then iterate through the plaintext p character-by-character using modular exponentiation to help generate the respective ciphertext characters. For decryption, the process is similar. RSA will iterate through

the ciphertext using modular exponentiation helping it convert back to the plaintext given that a correct privte key is used.

Going back to the equation form of modular exponentiation, for encrypting, b would be the character coming from the plaintext; e and m come from the public key tuple. In terms of decrypting, b is now coming from the ciphertext and e and m would be from the private key tuple.

The RSA decryption and signing can be further accelerated through the use of the Chinese Remainder Theorem. When decrypting, we can write the equation as such: $C^d\%n$ where C is the ciphertext value and d and n are the old e and m respectively as stated above. However, raising C^d can be extremely demanding and take a lot of time as d and large as is n. But since we know the the prime factorization of the modulus n being p and q it allows us to use the CRT.

This can give us:

$$V_p = C^d\%p$$
$$V_q = C^d\%q$$

But before we continue, some extra values must be computed:

$$X_p = q * (q^{-1}\%p)$$
$$X_q = p * (p^{-1}\%q)$$

Then applying the CRT we can get:

$$C^d\%n = (V_pX_p + V_qX_q)\%n$$

We can also use Fermat's Little Theorem to speed up the calculations of V_p and V_q, V_p requires $C^d\%p$. Since p is prime, then C and p are coprimes, and thus we can write:

$$V_p = C^d\%p = C^{u*x(p-1)+v}\%p = C^v\%p$$

For some u and v. Since $v < d$, it'll be faster to compute $C^v\%p$ than $C^d\%p$. This combination of theorems allows for the calculation of $C^d\%n$ to be done in nearly a quarter of the time it takes normally.

However, there are some risks associated with this as it is prone to Side Channel Attacks like Fault Injection Attacks since the processor can now show the values of the prime factors p and q which would allow an attacker to miscalculate the values V_p or V_q.

(2)

The python file `rsa_skeleton.py` containing the implementation of the RSA key generation algorithm along with the required lab deliverables can be found in this zip file.