**CYBER SECURITY & ETHICAL HACKING**

**Assessment - 3**

**NAME:** M N Surya Teja Reddy

**REG.NO:** 20BCN7021

**CAMPUS:** VIT AP University

**SYMMETRIC KEY – AES ALGORITHM**

* Block Division - First, the data is divided into blocks. Under this method of encryption, the first thing that happens is that your plaintext (which is the info that you want to be encrypted) is separated into blocks. The block size of AES is 128-bits, so it separates the data into a four-by-four column of sixteen bytes (there are eight bits in a byte and 16 x 8 = 128).
* Key Expansion - Key Expansion involves taking the initial key and using it to come up with a series of other keys for each round of the encryption process. These new 128-bit round keys are derived with Rijndael’s key schedule, which is essentially a simple and fast way to produce new key ciphers. Although they look like random characters each of these keys is derived from a structured process when AES encryption is actually applied. We’ll come back to what these round keys are used for later on.
* Add Round Key - In this step, because it is the first round, our initial key is added to the block of our message This is done with an XOR Cipher, which is an additive encryption algorithm. While it looks like you can’t actually add these things together, be aware that it is actually done in binary. The characters are just a stand-in to try and make things easier to understand.
* Substitute Bytes - In this step, each byte is substituted according to a predetermined table. This is kind of like the example from the start of the article, where the sentence was coded by changing each letter to the one that comes after it in the alphabet. This system is a little bit more complicated and doesn’t necessarily have any logic to it.
* Shift rows - Shift rows are a straightforward name, and this step is essentially what you would expect. The second row is moved one space to the left, the third row is moved two spaces to the left, and the fourth row is moved three spaces to the left.
* Mix Columns - This step is a little tricky to explain. To cut out most of the maths and simplify things, let’s just say that each column has a mathematical equation applied to it in order to further diffuse it.
* Add Round key again - Remember those round keys we made at the start, using our initial key and Rijndael’s key schedule? Well, this is where we start to use them. We take the result of our mixed columns and add the first round key that we derived
* Many more rounds - If you thought that was it, we’re not even close. After the last round key was added, it goes back to the byte substitution stage, where each value is changed according to a predetermined table. Once that’s done, it’s back to shift rows and moving each row to the left by one, two or three spaces. Then it goes through the mix columns equation again. After that, another round key is added.

It doesn’t stop there either. At the start, it was mentioned that **AES has key sizes of either 128, 192 or 256-bits.** When a 128-bit key is used, there are nine of these rounds. When a 192-bit key is used, there are 11. When a 256-bit key is used, there are 13. So the data goes through the byte substitution, shift rows, mix columns and round key steps up to thirteen times each, being altered at every stage.

After these nine, 11 or 13 rounds, there is one additional round in which the data is only processed by the byte substitution, shift rows and add round key steps, but *not* the mix columns step. The mix columns step is taken out because at this stage, it would just be eating up processing power without altering the data, which would make the encryption method less efficient.

**ASYMMETRIC KEY – RSA ALGORITHM**

Let’s say you want to tell your friend a secret. If you’re right next to them, you can just whisper it. If you are on opposite sides of the country, that obviously won’t work. You could write it down and mail it to them, or use the phone, but each of these communication channels is insecure and anyone with a strong enough motivation could easily intercept the message.

If the secret was important enough, you wouldn’t risk writing it down normally–spies or a rogue postal employee could be looking through your mail. Likewise, someone could be tapping your phone without your knowledge and logging every single call you make.

One solution to prevent eavesdroppers from accessing message contents is to encrypt it. This basically means to add a code to the message which changes it into a jumbled mess. If your code is sufficiently complex, then the only people who will be able to access the original message are those who have access to the code.

If you had a chance to share the code with your friend beforehand, then either of you can send an encrypted message at any time, knowing that you two are the only ones with the ability to read the message contents. But what if you didn’t have a chance to share the code beforehand?

This is one of the fundamental problems of cryptography, which has been addressed by public-key encryption schemes (also known as asymmetric encryption) like RSA.

Under RSA encryption, messages are encrypted with a code called a public key, which can be shared openly. Due to some distinct mathematical properties of the RSA algorithm, once a message has been encrypted with the public key, it can only be decrypted by another key, known as the private key. Each RSA user has a key pair consisting of their public and private keys. As the name suggests, the private key must be kept secret.

Public key encryption schemes differ from symmetric-key encryption, where both the encryption and decryption process use the same private key. These differences make public key encryption like RSA useful for communicating in situations where there has been no opportunity to safely distribute keys beforehand.

Symmetric-key algorithms have their own applications, such as encrypting data for personal use, or for when there are secure channels that the private keys can be shared over.

* Generating primes

The trap door functions mentioned above form the basis for how public and private-key encryption schemes work. Their properties allow public keys to be shared without endangering the message or revealing the private key. They also allow data to be encrypted with one key in a way that can only be decrypted by the other key from the pair.

The first step of encrypting a message with RSA is to generate the keys. To do this, we need two prime numbers (*p* and *q*) which are selected with a primality test. A primality test is an algorithm that efficiently finds prime numbers, such as the Rabin Miller Primality test

The prime numbers in RSA need to be very large, and also relatively far apart. Numbers that are small or closer together are much easier to crack. Despite this, our example will use smaller numbers to make things easier to follow and compute.

Let’s say that the primality test gives us the prime numbers that we used above, 907 and 773. The next step is to discover the modulus (*n*), using the following formula:

n = p x q

Where p = 907 and q = 773

Therefore:

n = 907 x 773  
n = 701,111

* Carmicheal’s Totient function

Once we have *n*, we use Carmichael’s totient function:

*λ*(*n*) = *lcm* (*p* − 1, *q* − 1)

If it’s been a while since you’ve hit the math textbooks, the above might look a bit terrifying. You can skip over this part and just trust that the math works, otherwise stick with us for a few more calculations. Everything will be explained in as much detail as possible to help you get your head around the basics.

For those who aren’t aware, *λ(n)* represents Carmichael’s totient for *n*, while *lcm* means the lowest common multiple, which is the lowest number that both *p* and *q* can divide into. There are a few different ways to figure this out, but the easiest is to trust an [online calculator](https://www.calculatorsoup.com/calculators/math/lcm.php) to do the equation for you. So let’s put our numbers into the equation:

*λ*(*701,111*) = *lcm* (*907* − 1, *773* − 1)  
*λ*(*701,111*) = *lcm* (*906,* *772*)

* Generating the Public Key

Now that we have Carmichael’s totient of our prime numbers, it’s time to figure out our public key. Under RSA, public keys are made up of a prime number *e*, as well as modulus *n*(we will explain what modulus means in a few paragraphs). The number *e*can be anything between 1 and the value for *λ*(*n*), which in our example is 349,716.

Because the public key is shared openly, it’s not so important for *e* to be a random number. In practice, *e*is generally set at 65,537, because when much larger numbers are chosen randomly, it makes encryption much less efficient. For today’s example, we will keep the numbers small to make calculations efficient. Let’s say:

*e* = 11

Our final encrypted data is called the ciphertext (*c*). We derive it from our plaintext message (*m*), by applying the public key with the following formula:

*c* = *me mod n*

As we mentioned, *e mod n* is the public key. We have already devised *e* and we know *n*as well. The only thing we need to explain is *mod*. It’s a little bit out of the depth of this article, but it refers to a modulo operation, which essentially means the remainder left over when you divide one side by the other. For example:

10 *mod* 3 = 1

This is because 3 goes into 10 three times, with a remainder of 1.

Back to our equation. To keep things simple, let’s say that the message (*m*) that we want to encrypt and keep secret is just a single number, *4*. Let’s plug everything in:

*c* = *me mod* *n*

*c* = 4*11* *mod* *701,111*

*c* = 4,194,304 *mod* 701,111

c**= 688,749**

Therefore when we use RSA to encrypt our message, 4, with our public key, it gives us the ciphertext of 688,749. The previous steps may have seemed a little too math-heavy, but it’s important to reiterate what has actually happened.

We had a message of 4, which we wanted to keep secret. We applied a public key to it, which gave us the encrypted result of 688,749. Now that it is encrypted, we can securely send the number 688,749 to the owner of the key pair. They are the only person who will be able to decrypt it with their private key. When they decrypt it, they will see the message that we were really sending, 4.

* Generating the Private Key

In RSA encryption, once data or a message has been turned into ciphertext with a public key, it can only be decrypted by the private key from the same key pair. Private keys are comprised of *d* and *n*. We already know *n,* and the following equation is used to find *d*:

*d* =1/*e mod λ*(*n*)

In the Generating the public key section above, we already decided that in our example, *e* would equal 11. Similarly, we know that *λ*(*n*) equals 349,716 from our earlier work under Carmichael’s totient function. Things get a little more complicated when we come across this section of the formula:

1/*e mod*

This equation may look like it is asking you to divide 1 by 11, but that’s not the case. Instead, this just symbolizes that we need to calculate the modular inverse of *e* (which in this case is 11) and *λ*(*n*) (which in this case is 349,716).

This essentially means that instead of performing a standard modulo operation, we will be using the inverse instead. This is normally found with the Extended Euclidean Algorithm. Now that we have the value for *d*, we can decrypt messages that were encrypted with our public key using the following formula:

*m* = *cd mod n*

We can now go back to the ciphertext that we encrypted under the Generating the private key section. When we encrypted the message with the public key, it gave us a value for *c*

**HASH FUNCTION – SHA 256**

Cryptographic hash functions are special types of hash functions that have a range of strange properties. Not only do they change data of any length into fixed-length values, but they are also deterministic, designed so that slight changes alter the output, fast to calculate, one-way function, Collision resistant.

SHA-256 results in a 256-bit hash and has a 512-bit block size. The message input is processed in 32-bit words, while the initialization variables and constants are also 32 bits in length. SHA-256 also involves 64 rounds.

In cryptography, **one-way compression functions take two fixed-length inputs and produce an output that is also a fixed length**. The process makes it difficult to figure out what the two inputs were if you only have access to the output. The inputs do not have to be the same length, as we will see in SHA-2.

These one-way compression functions should not be confused with the compression algorithms we use to make audio, video and other files smaller.

For **each block of data** being processed by SHA-256, the one-way compression function has inputs of:

* **512-bits of message data** — SHA-2 processes one block of data at a time. This input acts in a similar way that a key would in a normal block cipher (see the **The SHA-2 block cipher** section below). By this, we mean that it plays a controlling role in what the output will be. If there is more than one block of data that needs to be processed, these subsequent blocks become the inputs once the first block of data has gone through the SHA-2 algorithm. The final block must always be padded.
* **One set of 256-bit initialization variables** — These are broken up into eight parts for the initial processing of the first block. If there is more than one block, the 256-bit **intermediate hash** fills the role of the initialization variables in each of these rounds. The intermediate hash is essentially just the output from processing the prior block of data. These initialization variables and intermediate hashes take the place of the plaintext that would be encrypted in a normal block cipher (see the **SHA-2 block cipher** section below).

It produces an output of:

* **One 256-bit hash** — For a single block of data, the output of the compression function is the 256-bit hash. When processing multiple blocks of data, each block produces a 256-bit **intermediate hash**. As we mentioned above, these act as inputs for the next block. When all of the blocks have been processed, the output of the final block is the 256-bit hash.

Note that the compression functions in SHA-384, SHA-512, SHA-512/224 and SHA-512/256 each have inputs of:

* 1,024 bits of message data.
* One set of 512-bit initialization variables or a 512-bit intermediate hash.

They output:

* Hashes of either 384, 512,224 or 256 bits, depending on the algorithm. In the case of multiple blocks being processed, the intermediate hashes are 512 bits.

**IMPLEMENTATION**

I choose to implement RSA Algorithm in Python to encrypt and decrypt a file. The scenario I aim to solve is securely storing and transmitting sensitive files by encrypting them with RSA.

The RSA algorithm is a widely used public-key encryption algorithm named after its inventors Ron Rivest, Adi Shamir, and Leonard Adleman. It is based on the mathematical concepts of prime factorization and modular arithmetic.

The algorithm for RSA is as follows:

Select 2 prime numbers, preferably large, p and q.

Calculate n = p\*q.

Calculate phi(n) = (p-1)\*(q-1)

Choose a value of e such that 1<e<phi(n) and gcd(phi(n), e) = 1.

Calculate d such that d = (e^-1) mod phi(n).

Here the public key is {e, n} and private key is {d, n}. If M is the plain text then the cipher text C = (M^e) mod n. This is how data is encrypted in RSA algorithm. Similarly, for decryption, the plain text M = (C^d) mod n.

Suppose our message is M=31. You can encrypt and decrypt it using the RSA algorithm as follows:

Encryption: C = (M^e) mod n = 31^7 mod 33 = 4

Decryption: M = (C^d) mod n = 4^3 mod 33 = 31

Since we got the original message that is plain text back after decryption, we can say that the algorithm worked correctly.

Below is the Python code for the implementation of the RSA Algorithm:

CODE:

import math

*# step 1*

p = 3

q = 7

*# step 2*

n = p\*q

print("n =", n)

*# step 3*

phi = (p-1)\*(q-1)

*# step 4*

e = 2

while(e<phi):

    if (math.gcd(e, phi) == 1):

        break

    else:

        e += 1

print("e =", e)

*# step 5*

k = 2

d = ((k\*phi)+1)/e

print("d =", d)

print(f'Public key: {e, n}')

print(f'Private key: {d, n}')

*# plain text*

msg = 11

print(f'Original message:{msg}')

*# encryption*

C = pow(msg, e)

C = math.fmod(C, n)

print(f'Encrypted message: {C}')

*# decryption*

M = pow(C, d)

M = math.fmod(M, n)

print(f'Decrypted message: {M}')

OUTPUT:

