Textbook RSA

Mark Nakamae

Cal Poly SLO

August 2020

Description

This project looks closely at [RSA protocols](https://en.wikipedia.org/wiki/RSA_(cryptosystem)) and the relation between both public and private key pairs. The most notable equations for this portion of the assignment are the encryption [E(m) = me mod n] and decryption [m(c) = cd mod n] equations given public (e, n) and private (d, n) key pairs. We also explored malleability attacks in which attackers are allowed to create their own ciphertexts due to the mathematical properties of the encryption/decryption algorithms. As a result, it is very important to also use padding when using any RSA schemes. Both the public and private keys are derived from randomly generated prime numbers, p and q.

“Textbook” RSA and MITM Key Fixing via Malleability

*While it’s very common for many people to share an e (common values are 3,7, 216+1), it is very bad if two people share an RSA modulus* n*. Briefly describe why this is, and what the ramifications are.*

Since the d value is directly calculated from the value of the RSA modulus n, any users with this n value could also calculate the private keys of one another, making the keys insecure.

“Textbook” RSA Implementation

RSA begins by defining two prime numbers p and q. From these values, it then determines the value of the modulus n = p \* q and λ(n) = (p - 1) \* (q - 1). The value of the exponent e is defined differently per implementation, but for this implementation, the value 65,537 was used. Lastly the value d is calculated using the extended euclidean algorithm, d = e-1 mod λ(n). These values make up what are known as the public key and private key pairs used to encrypt and decrypt messages, respectively. From this step, RSA encryption and decryption are a series of straightforward computations.

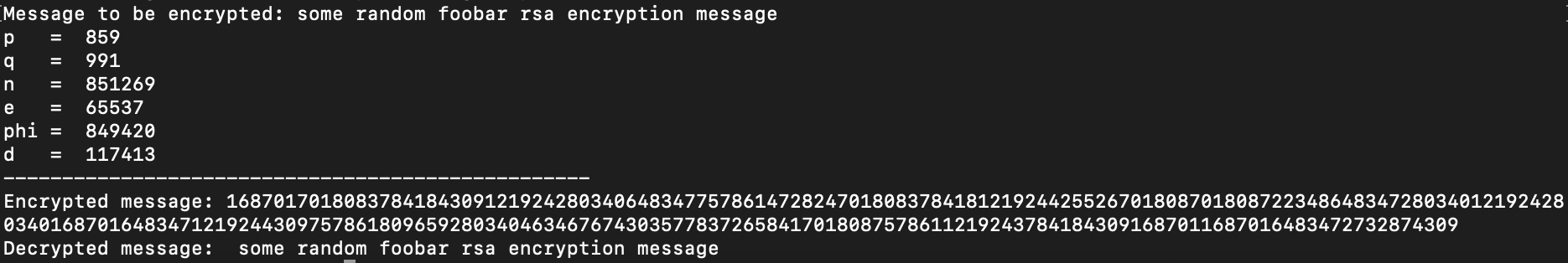
To encrypt a plaintext message m, the public key (e, n) and the following equation are used, where E(m) is the resulting ciphertext:

E(m) = me mod n

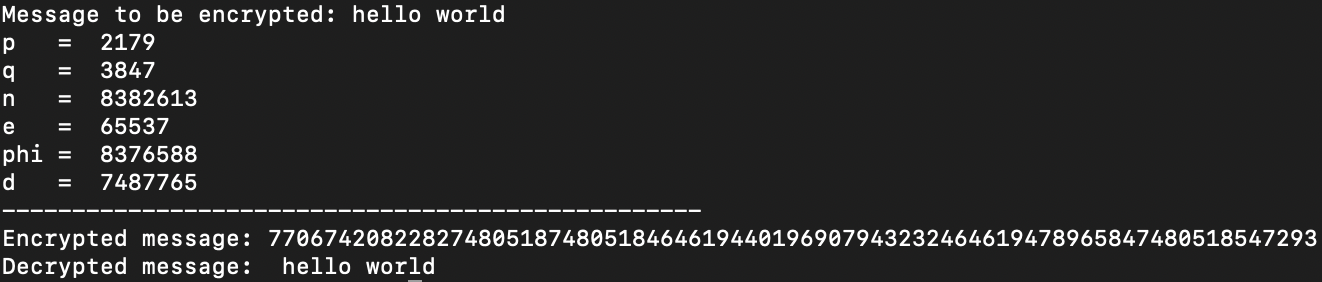
To decrypt a ciphertext message c, the private key (d, n) and the following equation are used where m(c) is the resulting plaintext:

m(c) = cd mod n

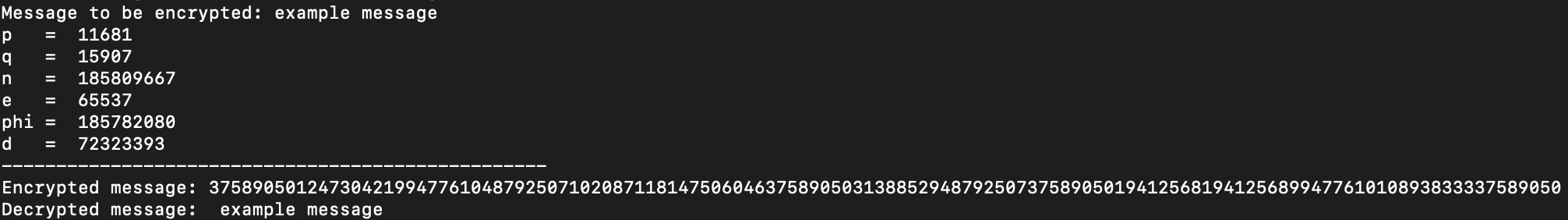
Example 1:



Example 2:



Example 3:

****

It’s also worth noting that larger bit values for p and q cause longer runtimes, more complicated computations, and longer encrypted messages.

PART B

*Find the operation F(∙) that Mallory needs to apply to the ciphertext c that will allow her to decrypt the ciphertext c0. HINT: Mallory knows Alice’s public key (n,e) and can encrypt her own messages to Alice.*

The generalizable equation to decrypt a ciphertext back to the plain text m(c), given private key pair (d, n) is the following:

m(c) = cd mod n

Since modulus n is equal to p \* q, the factors of n are 1, p, q, and n since both p and q are prime numbers. After solving for p and q by finding the greatest common denominators of n, one could calculate λ(n) = (p - 1) \* (q - 1). From this value, since we know e from the public key pair, we can then calculate the value of d from the following equation:

d \* e = 1 mod λ(n) or d = e-1 mod λ(n)

Now that we know both d and n, we know the private key pair (d, n) and can therefore decrypt any message sent, including ciphertext c0.

*Give another example of how RSA’s malleability could be used to exploit a system (e.g. to cause confusion, disruption, or violate integrity).*

Just from looking at the equations used to encrypt with RSA, given a plaintext message m one does the following to get the resulting ciphertext E(m):

E(m) = me mod n

This is given the public key values (e, n). Similarly, due to the malleability of RSA, an attacker is able to create their own encryption for a message mt for any given t where:

E(m) \* te mod n = (mt)e mod n = E(mt)

This clearly violates any original integrity in the system, and allows attackers to create messages, which is why padding is typically used in addition to any RSA encryption.

Code

**from** Crypto.Util **import** number

# wrapper function used to encrypt plaintext

**def** rsa\_encrypt(plain, e, n):

cipher\_tmp = []

**for** char **in** plain:

cipher\_tmp.append(pow(ord(char), e, n))

**return** cipher\_tmp

# wrapper function used to decrypt ciphertext

**def** rsa\_decrypt(cipher, d, n):

plain\_tmp = []

**for** char **in** cipher:

plain\_tmp.append(chr(pow(char, d, n)))

**return** "".join(plain\_tmp)

# print encrypted cipher text

**def** print\_cipher(cipher):

**print**("Encrypted message: ", end = "")

**for** char **in** cipher:

**print**(str(char), end = "")

**print**()

# extended euclidean algorithm used to find d

**def** multiplicative\_inv(e, phi):

tmp = e % phi;

**for** i **in** range(1, phi):

**if**((tmp \* i) % phi == 1):

**return** i

**return** 1

**def** main():

msg = input("Message to be encrypted: ")

# Get key pair values

p = number.getPrime(10) # prime number

q = number.getPrime(10) # prime number

n = p \* q # value used for key pairs

e = 65537 # exponent

phi = (p - 1) \* (q - 1) # totient of phi(n)

d = multiplicative\_inv(e, phi) # private value used for key pairs

# print values used to calculate keys

**print**("p = ", p)

**print**("q = ", q)

**print**("n = ", n)

**print**("e = ", e)

**print**("phi = ", phi)

**print**("d = ", d)

**print**("--------------------------------------------------")

#encrypt and decrypt input message

encrypted = rsa\_encrypt(msg, e, n)

print\_cipher(encrypted)

**print**("Decrypted message: ", rsa\_decrypt(encrypted, d, n))

**if** \_\_name\_\_ == '\_\_main\_\_':

main()