

# Lab 9: RSA

50.020 Security

Hand-out: November 24  
Hand-in: December 1, 11am

## 1 Objective

By the end of this lab, you should be able to:

- generate keys, encrypt, decrypt, sign and verify using RSA
- explain the importance of padding in RSA digital signature

## 2 Part 0: Preparation

- Check whether you have PyCrypto installed or not.

```
>>> from Crypto.PublicKey import RSA
```

- If you do not have PyCrypto, install it using:

```
$ pip install pycrypto
```

- API for PyCrypto can be found at:

– <https://www.dlitz.net/software/pycrypto/api/current/>

## 3 Part I: RSA Without Padding

- Use your previous Python script implementation of square and multiply to do large integer exponentiation.
- Encrypt `message.txt` using RSA:
  - Download the public key `mykey.pem.pub` from eDimension.
  - Use `Crypto.PublicKey.RSA.importKey()` function to import the public key as an RSA object in PyCrypto.

```
>>> key=open('mykey.pem.pub','r')
>>> rsakey=RSA.importKey(key)
```

You can use the `n` and `e` attribute to obtain the modulus and the exponent numbers of the public key.

```
>>> print rsakey.n
>>> print rsakey.e
```

- Define a function to encrypt a message using the following formula:

$$m \equiv x^e \bmod n \quad (1)$$

Use square and multiply algorithm to do the exponentiation.

- Decrypt the encrypted message:

- Download the private key `mykey.pem.priv` from eDimension.
- Use `Crypto.PublicKey.RSA.importKey()` function to import the private key as an RSA object in PyCrypto. You can use the `n` and `d` attribute to obtain the modulus and the exponent numbers of the private key.

```
>>> print rsakey.n
>>> print rsakey.d
```

- Define a function to decrypt the encrypted message using the following formula:

$$m \equiv x^d \bmod n \quad (2)$$

Use square and multiply algorithm to do the exponentiation.

- Create a signature the plaintext `message.txt` using the **private key**. Rather than exponentiation of the actual message, signature using RSA is usually applied to the hash of the message. First, hash the plaintext using SHA-256 (use `Crypto.Hash.SHA256`), then exponentiate the digest using the following equation.

$$s \equiv x^d \bmod n \quad (3)$$

- Verify the signature using the **public key**. The resulting exponentiation must be the same as the hash value of the plaintext.

$$x' \equiv s^e \bmod n \quad (4)$$

## 4 Part II: Protocol Attack

RSA has an undesirable property, namely that it is malleable. An attacker can transform the ciphertext into another ciphertext which leads to a transformation of the plaintext.

- RSA Encryption Protocol Attack:

- Encrypt an integer (e.g. 100) using the public key from previous part, e.g. `y`.
- Choose a multiplier `s` equal to 2 and calculate

$$y_s \equiv s^e \bmod n \quad (5)$$

- Multiply the two numbers:

$$m \equiv y \times y_s \bmod n \quad (6)$$

- Decrypt using the private key from the previous part. You should display something as follows.

Part II-----

Encrypting: 100

Result:

FRGXuy5uEmfkIgEy2y0e+7Age5/x2gDDD/m48XVxe9eEgGFU5Ru7669AGrR9rdxiIm2U  
/miColuSFRX2z/moF6v/Lz+o/FhABDzWWe4R4Xqt9rDdVNDeaQq4qoQE7PmsW/PTep/s  
im5lRt1TNWJ03jK6dpDIqiwtE0GDtpRGa8=

Modified to:

dpr87xC5fGMy86yvEb/8qyDxSccczfhZwJ48hyuEQ1lnwQDhaJTR6lNVFSaQXQdJzs4R  
xzPzWeZCf1CCOP8xa5yCl3Y+0iM1y6HMNic3/zSSY+ZJHKZvCw6tzWFhFffxInqCeh1Z  
3ExTlvVJPRBdxaf+r+kjSx4nBJ0j19fx5sV9tfu4RwAqJmJ2vu11wcZFPufbSXRUU/Fkf  
A5uYMiHlV5ezf93p7n+ArijN31DFaNAKQqTe+G5kelbwY+I09B9iuy+AR7zByBm9C2Wq  
twbTVvmxpIa39uUdSsI17JfgI774ITwbdmqlHCVfWK6fzxDTUXzfLCG/R14ByOWENRg2  
dQ==

Decrypted: 200

- RSA Digital Signature Protocol Attack:

- Obtain the public key from the previous section.
- Generate a random signature  $s$ .
- Compute an new digest from the signature using the **public key**.

$$x \equiv s^e \bmod n \quad (7)$$

- Send the signature  $s$  and the digest  $x$ .
- Verify the signature  $s$  for the digest  $x$  using the **public key**.

$$x' \equiv s^e \bmod n \quad (8)$$

- Since  $x' = x$ , it is a valid signature!

## 5 Checkoff 1

- Demo encryption and decryption of RSA.
- Demo signing a message and verifying it.
- Demo protocol attack to both encryption and digital signature.
- Explain the limitation of protocol attack.

## 6 Part III: Implementing RSA with Padding

The way to make RSA more secure is to use padding. In this implementation, we will use *Optimal Asymmetric Encryption Padding* (OAEP) for RSA encryption and *Probabilistic Signature Standard* (PSS) for RSA digital signature.

- Create an implementation of RSA with the following basic building blocks:
  - `generate_RSA(bits=1024)`, which generate the private key and public key in PEM format. The input parameter is the number of bits in the key which has default value of 1024 bits. *Hint: use `RSA.generate()` to generate the keys.*
  - `encrypt_RSA(public_key_file,message)`, which encrypt a string message using the public key stored in the file name `public_key_file`. The function returns the ciphertext in base 64. Use PyCrypto class `Crypto.Cipher.PKCS1_OAEP` to encrypt the message. *Hint:*
    - \* Read the public key from a file.
    - \* Use `RSA.importKey()` to import the key.
    - \* Create a new `PKCS1_OAEP` object and use its `encrypt` method rather than using your own square and multiply. This will encrypt RSA with some padding following OAEP.
  - `decrypt_RSA(private_key_file,cipher)`, which decrypt cipher text in base 64 using the private key stored in the file name `private_key_file`. The function returns the plaintext. Use PyCrypto class `Crypto.Cipher.PKCS1_OAEP` to decrypt the message. *Hint: similar to encrypt but use `=decrypt=` method.*
  - `sign_data(private_key_file,data)`, which signs the data string using a private key stored in the file name `private_key_file`. The function returns a signature string in base 64. Use PyCrypto class `Crypto.Signature.PKCS1_PSS`. Use SHA-256 to create a digest of the data before signing.
  - `verify_sign(public_key_file,sign,data)`, which verify the signature sign of a given data. The function returns either `True` or `False`. Use PyCrypto class `Crypto.Signature.PKCS1_PSS` instead of using the square and multiply routine you created. Use SHA-256 to create a digest of the data before verifying.
- Publish your public key on the internet. You can put it on your website, or via a social website.
- Ask your friend to encrypt a message using your public key. Decrypt the message using your private key.
- Sign the text `mydata.txt` using your private key. Send your data and its digital signature to a friend. Ask your friend to verify it.
- Similarly, get some data and its digital signature from a friend, and verify it.
- Redo the protocol attack with the new RSA.

## 7 Checkoff 2

- Demo decryption of a message from a friend using RSA.

- Explain the purpose of Optimal Asymmetric Encryption Padding (OAEP) to encrypt and decrypt using RSA. Explain how it works.
- Explain the purpose of Probabilistic Signature Scheme (PSS) to sign and verify using RSA. Explain how it works.