Lab 7: RSA

Deadline: 3 August 2024

Objectives

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

Part I: Setup

Make sure you have pycryptodome installed.

- Install: `pip install pycryptodome --upgrade`
- Check that you can import it

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

The documentation for pycryptodome can be found here.

Part II: RSA Without Padding

Use your previous Python script implementation of square and multiply to do large integer exponentiation.

Encrypt `message.txt` using RSA, using the public key `mykey.pem.pub`. You can `Crypto.PublicKey.RSA.importKey()` function to import the public key as an RSA object in PyCrypto. You can use the n and e attributes to obtain the modulus and the exponent numbers of the public key:

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

```
print(rsakey.n)
print(rsakey.e)
# private key
print(rsakey.n)
print(rsakey.d)
```

Encryption

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

$m \equiv x^e \bmod n$

Note: Use square and multiply algorithm to do the exponentiation!

Decryption

Use the private key `mykey.pem.priv`, same way as how you imported and used the public key.
Define a function to decrypt a message using the following formula:

$m \equiv x^d \bmod n$

Note: Use square and multiply algorithm to do the exponentiation!

Task

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.
$s \equiv x^d \bmod n$

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

$x' \equiv s^e \bmod n$

Submit a script `ex2.py` that performs the above task, printing the check showing the final hash is the same.

Part III: 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:

1. Encrypt an integer (e.g. 100) using the public key from previous part, e.g. `y`
2. Choose a multiplier `s` equal to 2 and calculate: $y_s \equiv s^e \bmod n$
3. Multiply the two numbers: $m \equiv y \times y_s \bmod n$
4. Decrypt using the private key from the previous part.

Submit a script `ex3.py` that performs the above task. Your script should print something as follows at the end:

```
Part II-------------
Encrypting:  100
Result:
FRGXuy5uEmfkIgEy2y0e+7Age5/x2gDDD/m48XVxe9eEgGFU5Ru7669AGrR9rdxiIm2U
/miColuSFRX2z/moF6v/Lz+o/FhABDzWWe4R4Xqt9rDdVNDeaQq4qoQE7PmsW/PTep/s
im5lRt1TNWJO3jK6dpDIqiwtE0GDtpRGa8=
Modified to:
dpr87xC5fGMy86yvEb/8qyDxScczfhZwJ48hyuEQ1lnwQDhaJTR6lNVFSaQXQdJzs4R
xzPzWeZCf1CC0P8xa5yCl3Y+OiM1y6HMNic3/zSSY+ZJHKZvCw6tzWFhFffxInqCeh1Z
3ExTlvVJPRBdxafr+kjSx4nBJ0j19fx5sV9tfu4RwAqJmJ2vu11wcZFPUfbSXRUU/Fkf
A5uYMihLv5ezf93p7n+ArijN31DFaNAKoqTe+G5kelbwy+IO9B9iuy+AR7zByBm9C2Wq
twbTVvmxpIa39uUdSsI17JfgI774ITwbdmqlHCVfWK6fzxDTUXzfLCG/R14By0WENRg2
dQ==
Decrypted:  200
```

Part IV: RSA Digital Signature Protocol Attack

In this attack, on behalf of Alice, you send a message/signature pair `(x,s)` to Bob who will then use Alice's public key to verify the signature. The verification done by Bob is expected to be successful
Alice's part:

1. Take the public key from the previous section as Alice's public key.
2. Choose any 1024-bit integer s.
3. Compute a new message from s using the public key: $x \equiv s^e \bmod n$ (note: this `x` can look random)
4. On behalf of Alice, send the signature `s` and the message `x` to Bob.

Bob's part:

Bob verifies the signature using Alice's public key (by following normal RSA signature verification process):

1. Using the public key, Bob gets a new digest x': $x' \equiv s^e \bmod n$
2. Bob checks whether x' == x is true
3. If true, `s` is a valid signature for message `x` and Bob will accept the message/signature pair `(x,s)`

Task

Write a script `ex4.py` to demonstrate the above exchange.

Part V: Defending 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 functions:

1. `generate_RSA(bits=1024)` which generates 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. Use `RSA.generate()` to generate the keys.
2. `encrypt_RSA(public_key_file,message)` which encrypts a string message using the public key, stored in the file name `public_key_file`. The function returns the ciphertext in base64. Use PyCrypto class `Crypto.Cipher.PKCS1_OAEP` to encrypt the message. To do so:
    - 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.
3. `decrypt_RSA(private_key_file,cipher)` which decrypts cipher text in base64 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. This is similar to encrypt but use decrypt method.
4. `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 base64. Use PyCrypto class `Crypto.Signature.PKCS1_PSS`. Use SHA-256 to create a digest of the data before signing.
5. `verify_sign(public_key_file,sign,data)` which verifies 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.

Task

1. Encrypt `mydata.txt` using your public key
2. Decrypt `mydata.txt` using your private key.
3. Sign the text `mydata.txt` using your private key. Verify to check that it the signature is valid.

Write a script `ex5.py` to do the following:

1. Redo the protocol attack with the new RSA.
2. Report if the attack works

Answer in `ans.txt`: Explain one limitation of RSA protocol attacks.

Submission

eDimension Submission

Lab 7 submission:

Upload a zip file with the following: file name format: lab7_name_studentid

Upload a **zip file** with the following:

- `ex2.py`
- `ex3.py`
- `ex4.py`
- `ex5.py`
- `ans.txt`
- **`Note: Ensure all Dependencies are enclosed in the ZIP File`**

Deadline: 3 August 2024