RSA Public-Key Encryption and Signature Lab

(adapted for Python from SEED Labs – M.EEC/SSR@FEUP 2022/23)

1 Overview

RSA (Rivest–Shamir–Adleman) is one of the first public-key cryptosystems and is widely used for secure communication. The RSA algorithm first generates two large random prime numbers, and then use them to generate public and private key pairs, which can be used to do encryption, decryption, digital signature generation, and digital signature verification. The RSA algorithm is built upon number theories, and it can be quite easily implemented with the support of libraries.

The learning objective of this lab is for students to gain hands-on experiences on the RSA algorithm. From lectures, students should have learned the theoretic part of the RSA algorithm, so they know mathematically how to generate public/private keys and how to perform encryption/decryption and signature generation/verification. This lab enhances student's understanding of RSA by requiring them to go through every essential step of the RSA algorithm on actual numbers, so they can apply the theories learned from the class. Essentially, students will be implementing the RSA algorithm using the Python program language. The lab covers the following security-related topics:

- Public-key cryptography
- The RSA algorithm and key generation
- Big number calculation
- Encryption and Decryption using RSA
- Digital signature
- X.509 certificate

2 Background

The RSA algorithm involves computations on large numbers, typically more than 512 bits long. Not all programming languages directly support arithmetic operations on integers longer than 32 or 64-bits. The original RSA Encryption and Signature Lab demonstrates how to use the Big Number library provided by `openssl` to perform arithmetic operations on integers of arbitrary size using the C programming language: `https://seedsecuritylabs.org/Labs_20.04/Crypto/Crypto_RSA/`

The Python programming language natively supports computations on large numbers. To assign a number to a variable and perform addition, subtraction or multiplication is straightforward:

```
a = 0x10AB12EF
b = 0xAC120816
res = a+b
res = a-b
res = a*b
```

The modulo operator is '`%`'. To compute `res = a * b mod n`, you simply type:

```
res = (a*b)%n
```

To exponentiate, you may use the native function pow. Computing `res = a`^{c}` mod n` is:

```
res = pow(a, c, n)
```

To compute the modular inverse, i.e., given `a`, find `b`, such that `a * b mod n = 1` (the value `b` is called the inverse of `a`, with respect to modular `n`) in Python 3.8+:

```
b = pow(a, -1, n);
```

As usual, you can either type all operations in a Python terminal:

```
seed@VM:~$ python3
>>> a = 0xF7E75FDC469067FFDC4E847C51F452DF
>>> b = 0xE85CED54AF57E53E092113E62F436F4F
>>> a * b
```

Or you can write your code to a file:

```
#!/usr/bin/env python3
a = 0xF7E75FDC469067FFDC4E847C51F452DF
b = 0xE85CED54AF57E53E092113E62F436F4F
print(a*b)
```

And then run it:

```
// Make it executable
seed@VM:~$ chmod u+x code.py
// Run it
seed@VM:~$ ./code.py
```

3 Lab tasks

To avoid mistakes, please avoid manually typing the numbers used in the lab tasks. Instead, copy and paste them from this PDF file.

3.1 Task 1: Deriving the Private Key

Let `p`, `q`, and `e` be three prime numbers. Let `n = p*q`. We will use (`e`, `n`) as the public key. Please calculate the private key `d`. The hexadecimal values of `p`, `q`, and `e` are listed in the following. It should be noted that although `p` and `q` used in this task are quite large numbers, they are not large enough to be secure. We intentionally make them small for the sake of simplicity. In practice, these numbers should be at least 512 bits long (the one used here are only 128 bits).

```
p = 0xF7E75FDC469067FFDC4E847C51F452DF
q = 0xE85CED54AF57E53E092113E62F436F4F
e = 0x0D88C3
```

3.2 Task 2: Encrypting a Message

Let (`e`, `n`) be the public key. Please encrypt the message "`A top secret!`" (quotations not included). We need to convert this ASCII string to a hex string, and then convert the hex string to an integer. The following python code can be used to convert plain ASCII strings to hex strings:

```
>> import codecs; print(codecs.encode("A top secret!".encode(), "hex"))
b'4120746f702073656372657421'
```

And the following function converts ASCII strings directly to integers

```
import codecs
def string_to_int(string_message):
    strmsg = codecs.encode(string_message.encode(), "hex")
    return int(strmsg.decode(), 16)

>>> print(string_to_int("A top secret!"))
5159874845387593541659235021857
```

The public keys are listed in the followings (hexadecimal). We also provide the private key d to help you verify your encryption result.

```
n = 0xDCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
e = 0x010001 # this hex value equals to decimal 65537
M = "A top secret!"
d = 0x74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D
```

3.3 Task 3: Decrypting a Message

The public/private keys used in this task are the same as the ones used in Task 2. Please decrypt the following ciphertext C, and convert it back to a plain ASCII string.

```
C=0x8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F
```

You can use the following code to convert integers back to ASCII strings:

```
import codecs
def int_to_bytes(int_message):
    return codecs.decode(hex(int_message)[2:], "hex")

>>> print(int_to_bytes(5159874845387593541659235021857))
b'A top secret!'
```

If you already have the hex (binary) string, you can skip the hex conversion:

```
>>> hexstr = b'4120746f702073656372657421'
>>> codecs.decode(hexstr, "hex")
b'A top secret!'
```

3.4 Task 4: Signing a Message

The public/private keys used in this task are the same as the ones used in Task 2. Please generate a signature for the following message (please directly sign this message, instead of signing its hash value):

```
M = "I owe you $2000."
```

Please make a slight change to the message M, such as changing `$2000` to `$3000`, and sign the modified message. Compare both signatures and describe what you observe.

3.5 Task 5: Verifying a Signature

Bob receives a message M = "`Launch a missile.`" from Alice, with her signature S. We know that Alice's public key is (e, n). Please verify whether the signature is indeed Alice's or not. The public key and signature (hexadecimal) are listed in the following:

```
M = "Launch a missile."
S = 0x643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F
e = 0x010001 # this hex value equals to decimal 65537
n = 0xAE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115
```

Suppose that the signature above is corrupted, such that the last byte of the signature changes from `2F` to `3F`, i.e, there is only one bit of change. Please repeat this task and describe what will happen to the verification process.

3.6 Manually Verifying an X.509 Certificate

In this task, we will manually verify an X.509 certificate using our program. An X.509 contains data about a public key and an issuer's signature on the data. We will download a real X.509 certificate from a web server, get its issuer's public key, and then use this public key to verify the signature on the certificate.

Step 1: Download a certificate from a real web server. We use the www.example.org server in this document. Students should choose a different web server that has a different certificate than the one used in this document (it should be noted that `www.example.com` share the same certificate with `www.example.org`). We can download certificates using browsers or use the following command:

```
$ openssl s_client -connect www.example.org:443 -showcerts

Certificate chain
 0 s: C=US, ST=California, L=Los Angeles, O=Internet\C2\A0Corporation ...
......
```

The result of the command contains two certificates. The subject field (the entry starting with `s:`) of the certificate contains `CN = www.example.org`, i.e., this is `www.example.org`'s certificate. The issuer field (the entry starting with `i:`) provides the issuer's information. The subject field of the second certificate is the same as the issuer field of the first certificate. Basically, the second certificate belongs to an intermediate CA. In this task, we will use CA's certificate to verify a server certificate. Copy and paste each of the certificates (the text between the line containing `"-----BEGIN CERTIFICATE-----"` and the line containing `"-----END CERTIFICATE-----"`, including these two lines) to a file. Let us call the first one `server.pem` and the second one `issuer.pem`.

Step 2: Extract the public key (e, n) from the issuer's certificate. Openssl provides commands to extract certain attributes from the x509 certificates. We can extract the value of `n` using `-modulus`. There is no specific command to extract e, but we can print out all the fields and can easily find the value of `e`.

```
For modulus (n):
$ openssl x509 -in issuer.pem -noout -modulus

Print out all the fields, find the exponent (e):
$ openssl x509 -in issuer.pem -text -noout
```

Step 3: Extract the signature from the server's certificate. There is no specific `openssl` command to extract the signature field. However, we can print out all the fields and then copy and paste the signature block into a file (note: if the signature algorithm used in the certificate is not based on RSA, you can find another certificate).

```
$ openssl x509 -in server.pem -text -noout
...
Signature Algorithm: sha256WithRSAEncryption
```

```
    84:a8:9a:11:a7:d8:bd:0b:26:7e:52:24:7b:b2:55:9d:ea:30:
    89:51:08:87:6f:a9:ed:10:ea:5b:3e:0b:c7:2d:47:04:4e:dd:
    ......
    5c:04:55:64:ce:9d:b3:65:fd:f6:8f:5e:99:39:21:15:e2:71:
    aa:6a:88:82
```

We need to remove the spaces and colons from the data, so we can get a hex-string that we can feed into our program. In Python, you can create a multi-line string using `"""string"""` as shown next. Then, you can use the `str.replace(old, new)` function to remove spaces, newlines and colons, so that you obtain a hex string that is easily converted to an integer:

```
>>> sign = """84:a8:9a:11:a7:d8:bd:0b:26:7e:52:24:7b:b2:55:9d:ea:30:
...     89:51:08:87:6f:a9:ed:10:ea:5b:3e:0b:c7:2d:47:04:4e:dd:
...     ......
...     5c:04:55:64:ce:9d:b3:65:fd:f6:8f:5e:99:39:21:15:e2:71:
...     aa:6a:88:82"""
>>> sign = sign.replace(':', '').replace(' ', '').replace('\n','')
>>> print(sign)
84a89a11a7d8bd0b267e52247b ... aa6a8882
>>> signature_as_int = int(sign, 16)
```

Step 4: Extract the body of the server's certificate. A Certificate Authority (CA) generates the signature for a server certificate by first computing the hash of the certificate, and then sign the hash. To verify the signature, we also need to generate the hash from a certificate. Since the hash is generated before the signature is computed, we need to exclude the signature block of a certificate when computing the hash. Finding out what part of the certificate is used to generate the hash is quite challenging without a good understanding of the format of the certificate.

X.509 certificates are encoded using the ASN.1 (Abstract Syntax Notation.One) standard, so if we can parse the ASN.1 structure, we can easily extract any field from a certificate. Openssl has a command called `asn1parse` used to extract data from ASN.1 formatted data:

```
$ openssl asn1parse -i -in server.pem
    0:d=0 hl=4 l=1522 cons: SEQUENCE
    4:d=1 hl=4 l=1242 cons:  SEQUENCE ❶
    8:d=2 hl=2 l=   3 cons:   cont [ 0 ]
   10:d=3 hl=2 l=   1 prim:    INTEGER     :02
   13:d=2 hl=2 l=  16 prim:   INTEGER     :0E64C5FBC236ADE14B172AEB41C78CB
   ......
 1246:d=5 hl=2 l=   2 prim:     OCTET STRING  [HEX DUMP]:3000
 1250:d=1 hl=2 l=  13 cons:  SEQUENCE ❷
 1252:d=2 hl=2 l=   9 prim:   OBJECT        :sha256WithRSAEncryption
 1263:d=2 hl=2 l=   0 prim:   NULL
 1265:d=1 hl=4 l= 257 prim:  BIT STRING
```

The field starting from ❶ is the body of the certificate that is used to generate the hash; the field starting from ❷ is the signature block. Their offsets are the numbers at the beginning of the lines. In our case, the certificate body is from offset 4 to 1249, while the signature block is from 1250 to the end of the file. For X.509 certificates, the starting offset is always the same (i.e., 4), but the end depends on the content length of a certificate. We can use the `-strparse` option to get the field from the offset 4, which will give us the body of the certificate, excluding the signature block.

```
$ openssl asn1parse -i -in server.pem -strparse 4 -out cer_body.bin -noout
```

Once we get the body of the certificate, we can calculate its hash using the following command:

```
$ sha256sum cer_body.bin
```

Step 5: Verify the signature. Now we have all the information, including the CA's public key, the CA's signature, and the body of the server's certificate. We can run our own program to verify whether the signature is valid or not. Openssl does provide a command to verify the certificate for us, but the goal of this task is to use our own program to do so.

The signature is computed based on a specific message, that not only includes the hash, but also includes padding, and information about the hash algorithm. Remember to convert the signature back to a hex string using the `hex(...)` function when performing your comparison.