Public Key Encryption and Signature Lab rev3

1. Overview

The RSA (Rivest–Shamir–Adleman) algorithm 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 and Public Key Encryption and Signatures, in general. From lectures, students should have learned the theory of the Public Key Cryptosystems and of the RSA algorithm specifically, so they know what is required mathematically to generate public/private keys and how to perform encryption/decryption and signature generation/verification. This lab enhances student's understanding of the RSA algorithms by requiring them to go through every essential step of the RSA algorithm on actual practical examples, so they can apply the theories learned from the class. Essentially, students will be implementing the RSA algorithm using the C and Python programming languages.

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 signatures
- X.509 certificates

Lab environment: This lab has been tested on the SEED[1] Ubuntu 20.04 VM.

The seed Account: In this lab, we need to telnet from one container to another.
We have already created an account called seed inside all the containers.
Lab_public.zip is also provided on Canvas with necessary files for this lab.

The username is seed and password is dees for all VMs and docker containers. Docker Manual for the Lab [here][2]

This lab is part of your assignment so please keep screenshots of your steps while

[1] This Lab exercise has been adopted from SEED Labs by Dr. Wenliang Du (https://seedsecuritylabs.org)
[2] https://github.com/seed-labs/seed-labs/blob/master/manuals/docker/SEEDManual-Container.md

you perform the lab to show your work.

2. Lab Environment

In this lab, you can use the SEED virtual machine from your previous lab.

3. Background

The RSA algorithm involves computations on large numbers. These computations cannot be directly conducted using simple arithmetic operators in programs, because those operators can only operate on primitive data types, such as 32-bit integer and 64-bit long integer types. The numbers involved in the RSA algorithms are typically more than 512 bits long. For example, to multiple two 32-bit integer numbers **a** and **b**, we just need to use **a*b** in our program. However, if they are big numbers that cannot be stored in build-in variables, we cannot perform the operations without relying on external libraries that can perform operations on large numbers. These libraries provide algorithms (i.e., functions) to compute their product of large numbers and other large-number operations.

There are several libraries that can perform arithmetic operations on integers of arbitrary size. In this lab, we will use the Big Number library provided by Openssl for C and numpy for python. To use this library, we will define each big number as **BIGNUM** type, and then use the APIs provided by the library for various operations, such as addition, multiplication, exponentiation, and modular operations among others.

3.1. C/C++ BIGNUM APIs

All the big number APIs can be found from:

https://linux.die.net/man/3/bn

https://www.openssl.org/docs/man1.1.0/man3/BN_sub.html

In the following section, we describe some of the APIs that are needed for this lab.

- Some of the library functions requires temporary variables. Since dynamic memory allocation to create BIGNUMs is quite expensive when used in conjunction with repeated subroutine calls, a BN CTX structure is created to holds BIGNUM temporary variables used by library functions. We need to create such a structure and pass it to the functions that requires it.

  ```
  BN_CTX *ctx = BN_CTX_new()
  ```

- Initialize a BIGNUM variable.

  ```
  BIGNUM *a =  BN_new()
  ```

- There are several ways to assign a value to a BIGNUM variable:

  ```
          // Assign a value from a decimal number string
  ```

```
BN_dec2bn(&a, "12345678901112231223");

// Assign a value from a hex number string
BN_hex2bn(&a, "2A3B4C55FF77889AED3F");
// Generate a random number of 128 bits
BN_rand(a, 128, 0, 0);

// Generate a random prime number of 128 bits
BN_generate_prime_ex(a, 128, 1, NULL, NULL, NULL);
```

- Print out a big number.

```
void printBN(char *msg, BIGNUM * a)
{
// Convert the BIGNUM to number string
    char * number_str = BN_bn2dec(a);

// Print out the number string
    printf("%s %s\n", msg, number_str);

// Free the dynamically allocated memory
    OPENSSL_free(number_str);
}
```

- Compute res = a − b and res = a + b:

```
BN_sub(res, a,b);
BN_add(res, a, b);
```

- Compute res = a ∗ b. It should be noted that a BN CTX structure is need in this API.

```
BN_mul(res, a, b, ctx)
```

- Compute res = a ∗ b mod n:

```
BN_mod_mul(res, a, b, n, ctx)
```

- Compute res = a^c mod n:

```
BN_mod_exp(res, a, c, n, ctx)
```

- Compute 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.

```
BN_mod_inverse(b, a, n, ctx);
```

3.2. Putting it all together, a complete example

We show a complete example in the following. In this example, we initialize three BIGNUM variables, a, b, and n; we then compute a ∗ b and (a^b mod n).

```
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 512

void printBN(char *msg, BIGNUM * a)
{
/* Use BN_bn2hex(a) for hex string
   Use BN_bn2dec(a) for decimal string */
     char * number_str = BN_bn2hex(a);
     printf("%s %s\n", msg, number_str);
     OPENSSL_free(number_str);
}

int main ()
{

     BN_CTX *ctx = BN_CTX_new();
     BIGNUM *a = BN_new();
     BIGNUM *b = BN_new();
     BIGNUM *n = BN_new();
     BIGNUM *res = BN_new();

     // Initialize a, b, n
     BN_generate_prime_ex(a, NBITS, 1, NULL, NULL, NULL);
     BN_dec2bn(&b, "7966919325049983427477025466251372129123");
     BN_rand(n, NBITS, 0, 0);

     // res = a*b
     BN_mul(res, a, b, ctx);
     printBN("a * b = ", res);

     // res = (a^b) mod n
     BN_mod_exp(res, a, b, n, ctx); printBN("(a^c) mod n = ", res);

     return 0;
     }
```

Compilation. We can use the following command to compile Big_number_template.c (the character after "-" is the letter *L* not the number 1; it tells the compiler to use the crypto library).

```
$ gcc Big_number_template.c -lcrypto -o Big_number_template
```

For Mac OSX only, you need to add the explicit paths to OpenSSL library:

$gcc -I/usr/local/opt/openssl/include -L/usr/local/opt/openssl/lib -lcrypto Big_number_template.c -o Big_number_template

In Python there is build-in support for large integers, and we can use the Crypto.Util and random libraries to generate large primes and random numbers. Also, python can handle all large integer operations without any external libraries as you can see below:

```
from Crypto.Util import number
from random import getrandbits
nbits = 512

primeNum1 = number.getPrime(nbits)
primeNum2 = 79669193250499834274770254662513721 29123
moduloN = getrandbits(512)

multipl = primeNum1*primeNum2

print ("a*b=", hex(multipl))

modExp = (primeNum1 * primeNum2) % moduloN
print ("(a*c) mod n =", hex(modExp))
```

4. 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. Each task is 12 points except the extra credit which is 10 points.

4.1. 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 **decimal** values of N, e, and d 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 1024 bits long. You need p and q to be able to calculate the rest of the values. Here, we selected, p, q to be primes and e to be 65537 so you can use these values to verify the rest of your calculations.

```
p:
8771202078281035880601236696053048036367629088057503902559294535
8193408249897

q:
1028354713512644517084005764843012743470851886292219969511523140
10256656047547

e: 65537

N:
9019907000372206840004996398463812455275090884514014708960288439
4206645274616999106751706668715909735563734539583895781710596679
6387615449077754648985265 9
```

```
d:
2777113439026967813870345260359839597823226328146295814538363064
0802930990908698281933459321392252138493930716820396289607716754
74665521453093256445593377
```

Q1. Develop and execute your code to calculate the private key d (shown above for verification)

Q2. Calculate the time that it takes to generate private keys of the following sizes: 256, 512, 1024, 2048, 4096, and 8192. What do you observe?

4.2. Task 2: Encrypting a Message

Let (e, N) be the public key. Please encrypt the message ***"Hello, this is my first RSA message!"*** (the quotations are not included).

If you are planning to use C/C++ to implement your solution, you need to convert this ASCII string to a hex string, and then convert the hex string to a BIGNUM using the hex-to-bn API BN hex2bn(). Sample code snippet below:

```
#include <stdio.h>
#include <string.h>
#include <openssl/bn.h>
#define NBITS 256

void printBN(char *msg, BIGNUM * a)
{
/* Use BN_bn2hex(a) for hex string
   Use BN_bn2dec(a) for decimal string */
     char * number_str = BN_bn2hex(a);
     printf("%s %s\n", msg, number_str);
     OPENSSL_free(number_str);
}

int main ()
{
     char msg[] = "Hello, this is my first RSA message!";
     int len = strlen(msg);

     char msginhex[NBITS];

     // Convert text to hex and store it in hex[].
     for (int i = 0, j = 0; i < len; ++i, j += 2)
       sprintf(msginhex + j, "%02x", msg[i] & 0xff);

     BIGNUM *msgBN = BN_new();
     BN_hex2bn(&msgBN, msginhex);
     printf("Original message = %s\n",msg);
     printBN("Hex converted Message = ",msgBN);
}
```

Q1. Encrypt the ***"Hello, this is my first RSA message!"*** using the following set of keys. We provide all the keys and **hexadecimal values** for both the message and the ciphertext to help you verify your results.

N =
```
EF38064573FC9B1DF7BD8415B6BFB64402E5DB284FE8CAD9A85F0785BC3E3D07A3CFCFCEE6C8B
64C37966982472C36604EF8B5A4AA5178CD2758D0E443126C19
```

e = `010001`

d =
```
DB94C484EA239C4B14C5FC41663F71D1DA0B1D715270700AFCF745D3676885E0FFEB7067C95BD
ADE54A62BE6066801093A5E2D3C2B98A95B9D763AF437B09795
```

msg =
`48656C6C6F2C20746869732069732061206D792066697273742052534120657373616765`

Ciphertext =
`722732586FFB8B6639AED189A822D92D16630FAC0D54C9F7A7DF9810E2F2200476FAD29DB5DE2`
`AAA2C82F1F706CBD674E09CBAF5E7215CB179933B359FCA8997`

Decrypted =
`48656C6C6F2C20746869732069732061206D792066697273742052534120657373616765`

Q2. What will happen if the message is changed to ***" This is a much longer second RSA message. I am having so much fun!"*** using the same set of keys? If any errors arise, explain how you could address them.

4.3. Task 3: Decrypting a Message

Let (e, n) be the public key and d be the private key.

Q1. Decrypt the Ciphertext using the following set of keys. We provide all the keys and hex value for both the message and the ciphertext to help you verify your results.

N =
`BDDD9F7CF8B69B24810B0A0F02CE69549F5E94BAD865100F60698C13A5E190F24D8900`
`B8E9126461110D51FA7D5C7B1E0F2DA28568D36D96BE65D9062DD2EE89`

e = `010001`

d =
`6D7690B4E44FA332709384C112C51E45037CEC12AD1FD71A866353B72033E3F44FE76B`
`CC343CB4319CCD5049AE3B52CB65102249BAF44AB834311CC908E17461`

Ciphertex =
`35B8BC929DD26C75A17CDA4772FB9E6A0682ED019EE806D1507AFC064D4955BE031EAC`
`E40DD3B9F9421511EC0AF6600510E93E0C3D6F2270FF9A879C132476C`

You can use the following python command to convert a hex string back to a plain ASCII string.

```
$ python  -c 'print("4120746f702073656372657421".decode("hex"))'
A top  secret!
```

Q2. Generate your own set of keys and message and show that you can encrypt and decrypt it successfully.

4.4. Task 4: Signing a Message

The public/private keys used in this task are the same as the ones used in Task 3.

Q1. Using the keys from task 3, please generate a signature for the following message: ***"This is a contact for $20,000"*** (no quotes). Please directly sign this message, instead of signing its hash value.

The value of the signed message in hex should be:

```
4560BE3C1E1D9A42B6784C7F06CBB6908D2A56017BB02794B06F322E7D339395
2C03B3FB310084C59C30CD33350188662E8090364AD5E57A8149E2D795393A30
```

Q2. What happens if you modify the message slightly by changing the contract value or another character? Produce an example.

4.5. Task 5: Verifying a Message

Q1. Using your code from Task 4, sign and verify your own message. Please make sure that you include both your code and code output showing the signature and verification steps clearly.

Q2. Increase the size of the message you attempt to sign testing files of varying size from 1KB, 100KB, 1MB, to 10MB. You can generate files of fixed size and measure the time it takes to sign them. What do you observe? Does the key size have an impact in the signature process?

4.6. [Extra Credit 10pts] Using our code to Verify an Web (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.

More on the structure of X.509 digital certificates:

https://docs.microsoft.com/en-us/windows/win32/seccertenroll/about-x-509-public-key-certificates

https://dev.to/wayofthepie/structure-of-an-ssl-x-509-certificate-16b

Step 1: Download a certificate from a real web server.

We will use the **https://**www.example.org server as an example here. ***Students should choose a different web server that has a different certificate (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
```

The above command will print to the screen the following output (notice the command might take few seconds to fully return to the command line so be patient).

```
Certificate chain
 0 s:/C=US/ST=California/L=Los Angeles/O=Internet Corporation for Assigned
Names and Numbers/CN=www.example.org
   i:/C=US/O=DigiCert Inc/CN=DigiCert TLS RSA SHA256 2020 CA1
-----BEGIN CERTIFICATE-----
MIIG1TCCBb2gAwIBAgIQD74IsIVNBXOKsMzhya/uyTANBgkqhkiG9w0BAQsFADBP
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMSkwJwYDVQQDEyBE
[…]
/sgUKGiQxrjIlH/hD4n6p9YJN6FitwAntb7xsV5FKAazVBXmw8isggHOhuIr4Xrk
vUzLnF7QYsJhvYtaYrZ2MLxGD+NFI8BkXw==
-----END CERTIFICATE-----
 1 s:/C=US/O=DigiCert Inc/CN=DigiCert TLS RSA SHA256 2020 CA1
   i:/C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert Global Root CA
-----BEGIN CERTIFICATE-----
MIIE6jCCA9KgAwIBAgIQCjUI1VwpKwF9+K1lwA/35DANBgkqhkiG9w0BAQsFADBh
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
[…]
as6xuwAwapu3r9rxxZf+ingkquqTgLozZXq8oXfpf2kUCwA/d5KxTVtzhwoT0JzI
8ks5T1KESaZMkE4f97Q=
-----END CERTIFICATE-----
 2 s:/C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert Global Root CA
   i:/C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert Global Root CA
-----BEGIN CERTIFICATE-----
MIIDrzCCApegAwIBAgIQCDvgVpBCRrGhdWrJWZHHSjANBgkqhkiG9w0BAQUFADBh
[…]
YSEY1QSteDwsOoBrp+uvFRTp2InBuThs4pFsiv9kuXclVzDAGySj4dzp30d8tbQk
CAUw7C29C79Fv1C5qfPrmAESrciIxpg0X40KPMbp1ZWVbd4=
-----END CERTIFICATE----
```

The result of the command contains **three** certificates (can you point them?). The subject field (the entry starting with **s:**) of the certificate is 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, in some cases, the second certificate is the TLS certificate of an intermediate CA or the of the Root CA. In this task, we will use CA's TLS certificate to verify the server certificate. We could also verify the TLS certificate from the Root Certificate, but we will not perform that step. The third certificate is the Root certificate of DigiCert Inc (DigiCert Global Root CA). If you only get one certificate back using the above command, that means the certificate you get is signed by a root CA. Root CAs' certificates can be obtained from the Firefox browser installed in our pre-built VM.

Go to the Edit ⍰ Preferences ⍰ Privacy and then Security ⍰ View Certificates. Search for the name of the issuer and download its 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 separate files. Let us call the first one c0.pem, the second one c1.pem, and the third one c2.pem.

To store the output to a file to help with that last command, you can use the following command to redirect the output to the openssl_out.txt file:

```
$openssl s_client -connect www.example.org:443 -showcerts > openssl_out.txt &
```

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 the flag ***"-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 c1.pem -noout -modulus

Print out all the fields, find the exponent (e):
$ openssl x509 -in c1.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 c0.pem -text -noout
```

[Look at the bottom of the file]

```
Signature Algorithm: sha256WithRSAEncryption
         a7:2a:10:30:5c:b8:6b:7a:1b:f8:66:38:f6:e9:a0:0a:d5:13:
         82:82:f8:65:89:57:a5:b8:eb:13:29:1d:84:6c:ec:fb:e3:05:
[…]
9c:5e:d0:62:c2:61:bd:8b:5a:62:b6:76:30:bc:46:0f:e3:45:
         23:c0:64:5f
```

We need to remove the spaces and colons from the data, so we can get a hex formatted string that we can feed into our program. The following command commands can achieve this goal. The **tr command** is a Linux utility tool for string operations. In this case, the -d option is used to delete ":" and "space" from the data:

After you save the signature to a separate file called signature, you perform the following command:

```
$cat signature | tr -d '[:space:]:'

SignatureAlgorithmsha256WithRSAEncryptiona72a10305cb86b7a1bf86638f6e9a00ad513
8282f8658957a5b8eb13291d846cecfbe30511d71e315e0ee2c000e56d0648be3d556fbab7113
5b6eac4cf84f1304cbb339e11172bc9d2194b2cd0ad5f172384e1df17a23ba87f69297c48a661
5f263f75e23b5ba336b31ccde30457301ffcc9fa4b8e488058279ca2c7c326dc1702fae66cea8
1015c928fd3180817707ac2a34b6c3afae3cff6fe7ec956e5a54e1b144fa9989d79b11ec3abb1
```

```
0d1585a946b6e5c258e85afec814286890c6b8c8947fe10f89faa7d60937a162b70027b5bef1b
15e452806b35415e6c3c8ac8201ce86e22be17ae4bd4ccb9c5ed062c261bd8b5a62b67630bc46
0fe34523c0645f
```

Remove the bolded text (either after or before running tr).

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 and can parse our X.509 certificate.

```
$openssl asn1parse -i -in c0.pem

0:d=0  hl=4 l=1749 cons: SEQUENCE
4:d=1  hl=4 l=1469 cons:  SEQUENCE
8:d=2  hl=2 l=   3 cons:   cont [ 0 ]
10:d=3  hl=2 l=   1 prim:    INTEGER           :02
[…]
1477:d=1  hl=2 l=  13 cons:  SEQUENCE
1479:d=2  hl=2 l=   9 prim:   OBJECT            :sha256WithRSAEncryption
1490:d=2  hl=2 l=   0 prim:   NULL
1492:d=1  hl=4 l= 257 prim:  BIT STRING
```

The certificates have fields and offsets. Offsets are the numbers at the beginning of the lines. The field starting from offset **4** is the body of the certificate that is used to generate the hash; the field starting from **1477.** In our case, the certificate body is from offset 4 to 1476 (it finishes when the signature block begins), while the signature block is from 1477 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 c0.pem -strparse 4 -out c0_body.bin  -noout
```

This will generate the file "c0_body.bin".
Once we get the body of the certificate, we can calculate its hash using the following command:

```
$sha256sum c0_body.bin

373e90af567edd6dab8b435927ff7cc2db6e8a17337c6eb9aabea0e4907b0eff  c0_body.bin
```

Be careful to remove the bold part of the output and keep only the hash.

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 students are required to use their own programs to perform this task.

5. Lab Report as part of Assignment

As part of your assignment that will be posted on Canvas, you need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed as screenshots and description of steps as you see fit. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.