RSA Public-Key Encryption and Signature Lab

In this lab, students will

- gain hands-on experiences on the RSA algorithm.
- Understand RSA by requiring them to go through every essential step of the RSA algorithm on actual numbers

This lab particularly aims at implementing the RSA algorithm using the C program language.

Table of Contents

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 C 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

Readings and videos. Detailed coverage of the public-key cryptography can be found in the following:

• Chapter 23 of the SEED Book, Computer & Internet Security: A Hands-on Approach, 2nd Edition, by Wenliang Du. See details at https://www.handsonsecurity.net.

Lab Tasks

Task 1: A Complete Example of BIGNUM

The program below shows a complete example of BIGNUM. This program uses three BIGNUM variables, a, b, and n; and then compute a ∗ b and (a b mod n).

```
#include <stdio.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 ()
{
        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, "273489463796838501848592769467194369268");
        BN_rand(n, NBITS, 0, 0);

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

        // res = aˆb mod n
        BN_mod_exp(res, a, b, n, ctx);
        printBN("aˆc mod n = ", res);
        return 0;
}
```

Commands

Execute the following command to compile the above program

```
$gcc -o task1 task1.c -lcrypto
$./task1
```

Give your observation with screen shot along with full plaintext.

Task 2: Deriving the private key

The objective of this task is to derive private key. Given are the hexadecimal values of p, q, e, and public key pair (e,n).

p = F7E75FDC469067FFDC4E847C51F452DF
q = E85CED54AF57E53E092113E62F436F4F
e = 0D88C3

Task2.c:

```
#include <stdio.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 ()
{
        BN_CTX *ctx = BN_CTX_new();
        BIGNUM *p= BN_new();
        BIGNUM *q= BN_new();
        BIGNUM *e= BN_new();
        BIGNUM *d = BN_new();
        BIGNUM *res1= BN_new();
        BIGNUM *res2= BN_new();
        BIGNUM *res3= BN_new();
        BIGNUM *one = BN_new();

        //initialize
        BN_hex2bn(&p,"F7E75FDC469067FFDC4E847C51F452DF");
        BN_hex2bn(&q,"E85CED54AF57E53E092113E62F436F4F");
        BN_hex2bn(&e,"0D88C3");
        BN_dec2bn(&one,"1");

        BN_sub(res1,p,one);
        BN_sub(res2,q,one);
        BN_mul(res3,res1,res2,ctx);
        BN_mod_inverse(d,e,res3,ctx);
```

```
                printBN("d = ",d);
                return 0;
        }
```

Commands:

```
$gcc -o task2 task2.c -lcrypto
$./task2
```

Give your observation with screen shot

Q1. Explain your understanding (in terms of mathematical statements) of what the above code does.

Task 3: Encrypting a message

The objective of this task is to encrypt a given message. Given are the hexadecimal values of n, e, M (you can use whatever message you want). The value of "d" is also given to verify the result.

n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
e = 010001 (This hex value equal to decimal 65537)
M = A top secret!
d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D

<u>**Step1:**</u> convert the ASCII string message to a hex string
Command:
$python -c 'print("A top secret!".encode("hex"))'

Give your observation with screen shot.

<u>**Step2:**</u> Execute the below program to encrypt the message M and verify it by decrypting it.

```
        #include <stdio.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()
{
        BN_CTX *ctx = BN_CTX_new();
        BIGNUM *m = BN_new();
        BIGNUM *e = BN_new();
        BIGNUM *n = BN_new();
        BIGNUM *d = BN_new();
        BIGNUM *enc = BN_new();
        BIGNUM *dec = BN_new();

        // Initialize p, q, e
        BN_hex2bn(&m,"/*Enter the hex encoded message from previous step
        here*/");
        BN_hex2bn(&e,"010001");
        BN_hex2bn(&n,
        "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242
        FB1A5");
        BN_hex2bn(&d,
        "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD
        7D30D");

        // Encryption : ree mod n
        BN_mod_exp(enc, m, e, n, ctx);
        printBN("Encrypted Message = ", enc);

        //Decryption: enc^d mod n
        BN_mod_exp(dec, enc, d, n, ctx);
        printBN("Decrypted Message=",dec);
        return 0;
}
```

Commands

```
$gcc -o task3 task3.c -lcrypto
$./task3
```

__**Step 3:**__ convert the hex value (output of previous step) to ASCII code

```
$python -c 'print("//previous step output hex value".decode("hex"))'
```

Give your observation with screen shot.

Task 4: Decrypting a message

The objective of this task is to decrypt a given ciphertext. given are the hexadecimal values of n, e, d from the above task

C= 8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F

```
#include <stdio.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()
{
        BN_CTX *ctx=BN_CTX_new();
        BIGNUM *m = BN_new();
        BIGNUM *e = BN_new();
        BIGNUM *n =BN_new();
        BIGNUM *d = BN_new();
        BIGNUM *enc =BN_new();
        BIGNUM *dec=BN_new();

        // Initialize p, q, e
        BN_hex2bn(&n,
        "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5"
        );

        BN_hex2bn(&d,
        "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D
        ");
        BN_hex2bn(&enc,
        "8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F"
        );
        //Decryption: enc^d mod n
        BN_mod_exp(dec, enc, d, n, ctx);
        printBN("Decrypted Message = ", dec);
        return 0;
}
```

Commands
$gcc -o task4 task4.c -lcrypto
$./task4

Decode the output using the command from the previous task.

Give your observation with screen shot.

Task 5: Signing a Message

The objective of this task is to generate a signature for the following message. Use the public/private key set from task3
M= I owe you $2000

step1: generate hex for M
Commands
$python -c 'print("I owe you $2000.".encode("hex")'

step2: Execute the following program to generate signature of the given message. Using the signing algorithm M^d mod n

```
#include <stdio.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()
```

```
{
 BN_CTX *ctx = BN_CTX_new();
 BIGNUM *m = BN_new();
 BIGNUM *n = BN_new();
 BIGNUM *d = BN_new();
 BIGNUM *sign = BN_new();

// Initialize p, q, e
 BN_hex2bn(&m,"/* hex encoded message here */");
 BN_hex2bn(&n,
"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
 BN_hex2bn(&d,
"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

// signing : m^e mod n
BN_mod_exp(sign, m, d, n, ctx);
printBN("encrypted Message = ", sign);
return 0;
}
```

Commands:
$gcc -o task5 task5.c -lcrypto
$./task5

<u>**Step3:**</u> execute the step 1 and 2 for message “I owe $3000”

Give your observation with screen shot.

Task 6: Verifying a Signature

The objective of this task is to verify if the signature received by Bob is Allice’s or not. Given are the Message M, signature S, Allice public key e and n.

```
#include <stdio.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()
{
                BN_CTX *ctx = BN_CTX_new();
                BIGNUM *s = BN_new();
                BIGNUM *n = BN_new();
                BIGNUM *e = BN_new();
                BIGNUM *message = BN_new();

               // Initialize p, q, e
               BN_hex2bn(&s,"643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CA
               B026C0542CBDB6802F");
                BN_hex2bn(&n,
               "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18
               116115");
                BN_hex2bn(&e, "010001");

               // signing : m^e mod n
                BN_mod_exp(message, s, e, n, ctx);
                printBN("encrypted Message = ", message);
               return 0;
}
```

Commands

```
$gcc -o task6 task6.c -lcrypto
$./task6
$python -c ‘print(“//output of task6”.decode(“hex”))’
```

Give your observation with screen shot.

Task 7: Manually verifying an X.509 Certificate

The objective of this task is to verify the signature of a public key certificate from a server and show that the signature matches.

To verify that a certificate was signed by a specific certificate authority we need the following details

1. public key of the certificate authority (issuer).
2. signature and algorithm used to generate signature from the server's certificate.

Step 1: download the certificate from any website (each student use a different website)
Command:
$ openssl s_client -connect www.example.org:443 -showcerts

Copy server certificate to c0.pem file and root certificate of the issuer to c1.pem

Give your observation with screen shot.

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.

Commands
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 |grep "Exponent"

Give your observation with screen shot.

Step 3: Extract the signature from the server's certificate. There is no specific opensslcommand 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, find another certificate).
Commands
$openssl x509 -in c0.pem -text -noout
//extract only the signature part and paste it in signature file
$ cat signature | tr -d '[:space:]:'

Give your observation with screen shot.

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, which can be used to parse a X.509 certificate.

Commands:

```
$ openssl asn1parse -i -in c0.pem -strparse 4 -out c0_body.bin -noout
$ sha256sum c0_body.bin
```

Give your observation with screen shot.

<u>**step 5:**</u> 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.

```
#include <stdio.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()
{
         BN_CTX *ctx = BN_CTX_new();
         BIGNUM *s = BN_new();
         BIGNUM *n = BN_new();
         BIGNUM *e = BN_new();
         BIGNUM *message = BN_new();

        // Initialize p, q, e
        /* Insert the values of n and e from step 2 */
        /* Insert the value of s from step 3 */
         BN_hex2bn(&s,"");
         BN_hex2bn(&n, "");
         BN_hex2bn(&e, "");
```

```
                // signing : m^e mod n
                 BN_mod_exp(message, s, e, n, ctx);
                 printBN("encrypted Message = ", message);
                return 0;
        }
```

$gcc task7.c -lcrypto
$ ./a.out

Give your observation with screen shot.

Submission

You need to submit a detailed lab report to describe what you have done and what you have observed, including screenshots and code snippets. You also need to provide explanation to the observations that are interesting or surprising. You are encouraged to pursue further investigation, beyond what is required by the lab description. Please submit in word or PDF format only.