RSA Public-Key Encryption Lab

0 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.

0.1 Learning Objectives

The learning objective of this lab are

1. Students to gain hands-on experiences on the RSA algorithm
2. Require students to go through every essential step of the RSA algorithm on actual numbers, applying the learned theories

NOTE: You will learn the theoretic part of the RSA algorithm in lecture. The lecture will discuss:

1. How to generate public/private keys
2. How to perform encryption/decryption
3. How to perform signature generation/verification

This lab covers the following security-related topics:

- Public-key cryptography
- The RSA algorithm and key generation
- Big number calculation
- Encryption and Decryption using RSA

0.2 Lab environment.

You will need to install some packages for this lab

1. SSH into your VM
2. Execute the following command `sudo apt update`
3. Execute the following command `sudo apt install libssl-dev -y`

Error Message If you receive an error message, at any point, similar to the following:

```
Waiting for cache lock: Could not get lock /var/lib/dpkg/lock-frontend.
It is held by process 2394 (apt-get)...
```

Execute the following commands

- Ctrl-C
- `sudo rm /var/lib/apt/lists/lock`
- `sudo rm /var/lib/dpkg/lock-frontend`
- `sudo rm /var/cache/apt/archives/lock`
- `sudo dpkg --configure -a`
- Reissue the command that gave you the error message

1 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 `bn1` and `bn2`, you need to use `bn1*bn2` in your program. However, if they are big numbers, you cannot. Instead, you need to use an algorithm to compute their products.

There are several libraries that can perform arithmetic operations on integers of arbitrary size. You will use the Big Number library provided by `openssl`. To use this library, you will define each big number as a `BIGNUM` type, and then use the APIs provided by the library for various operations, such as addition, multiplication, exponentiation, modular operations, etc.

1.1 BIGNUM APIs

All the big number APIs can be found from `https://linux.die.net/man/3/bn`. The following, APIs 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 hold BIGNUM temporary variables used by library functions. You 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 *bn1 = BN_new()
```

- There are a number of ways to assign a value to a BIGNUM variable.

```
// Assign a value from a decimal number string
BN_dec2bn(&bn1, "12345678901112231223");

// Assign a value from a hex number string
BN_hex2bn(&bn1, "2A3B4C55FF77889AED3F");

// Generate a random number of 128 bits
BN_rand(bn1, 128, 0, 0);

// Generate a random prime number of 128 bits
BN_generate_prime_ex(bn1, 128, 1, NULL, NULL, NULL);
```

- Print out a big number.

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

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

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

- Compute `res` = $bn1 - bn2$ and `res` = $bn1 + bn2$:

```
BN_sub(res, bn1, bn2);
BN_add(res, bn1, bn2);
```

- Compute `res` = $bn1 * bn2$. It should be noted that a `BN_CTX` structure is need in this API.

```
BN_mul(res, bn1, bn2, ctx)
```

- Compute `res` = $bn1 * bn2$ `mod bn3`:

```
BN_mod_mul(res, bn1, bn2, bn3, ctx)
```

- Compute `res` = $bn1^{bn2}$ `mod bn3`:

```
BN_mod_exp(res, bn1, bn2, bn3, ctx)
```

- Compute modular inverse, i.e., given `bn1`, find `bn2`, such that $bn1 * bn2$ `mod bn3 = 1`. The value `bn2` is called the inverse of `bn1`, with respect to modular `bn3`.

```
BN_mod_inverse(bn2, bn1, bn3, ctx);
```

1.2 A Complete Example

Below is a complete example. This sample.c is located in the Lab3 folder.

In this example, the code initializes three BIGNUM variables, `bn1`, `bn2`, and `bn3`. Next, $bn1 * bn2$ and `(`$bn1^{bn2}$ `mod bn3)` are computed.

```
/*sample.c provided in the Lab3 folder*/
#include <stdio.h>
#include <openssl/bn.h>

#define NBITS 256

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

int main ()
{
  BN_CTX *ctx = BN_CTX_new();

  BIGNUM *bn1 = BN_new();
  BIGNUM *bn2 = BN_new();
  BIGNUM *bn3 = BN_new();
  BIGNUM *res = BN_new();


  // Initialize bn1, bn2, bn3
  BN_generate_prime_ex(bn1, NBITS, 1, NULL, NULL, NULL);
  BN_dec2bn(&bn2, "273489463796838501848592769467194369268");
  BN_rand(bn3, NBITS, 0, 0);

  // res = bn1*bn2
  BN_mul(res, bn1, bn2, ctx);
  printBN("bn1 * bn2 = ", res);

  // res = bn1^bn2 mod bn3
  BN_mod_exp(res, bn1, bn2, bn3, ctx);
  printBN("bn1^bn2 mod bn3 = ", res);

  return 0;
}
```

Task 1.1

Compilation. Use the following commands to compile `sample.c` (**NOTE:** The character after sample.c is the - and the letter ℓ, not the number 1. This command tells the compiler to use the `crypto` library).

```
cd Documents/Labs/Lab3/
gcc sample.c -lcrypto -o rsaSample
```

To Complete Task 1.1

1. Compile the `sample.c` into the execute about rsaSample
2. Execute rsaSample and capture the output

Task 1.2

In the `Initialize bn1, bn2, bn3` section change the values of `bn1`, `bn2`, and `bn3`. Run the program at least three more times where each time you change the values of `bn1`, `bn2`, and `bn3`. Change the name of the executable each time to `rsaSample#` where # is the number of execution.

To Complete Task 1.2

Capture the output of each run and clearly label them.

NOTE: I am trying to get you to look up and understand the three functions. I don't want you to just change the value of `bn2` everytime, but to understand the function calls and change the parameters being passed to the functions.

For Example: What do the three `NULL` values mean in the `BN_generate_prime_ex` and what happens if you change one or all of them?

2 Lab Tasks

Up to this point, your tasks hopefully helped you understand the basic C program for generating large values. The following tasks will allow you to apply the RSA algorithm to different scenarios.

NOTE: To avoid mistakes copy and paste the numbers from the PDF.

2.1 Task 2.1: Deriving the Private Key

Let `p`, `q`, and `e` be three prime numbers. Let `n = p*q`. You will use `(e, n)` as the public key.

To Complete Task 2.1

Calculate the private key `d`. The **hexadecimal** values of `p`, `q`, and `e` are listed in the following. You cna convert a hex string to a BIGNUM using the hex-to-bn API function `BN_hex2bn()`.

Note: Although `p` and `q` used in this task are quite large numbers (128 bits), they are not large enough to be secure. In practice, these numbers should be at least 512 bits long.

```
p = F7E75FDC469067FFDC4E847C51F452DF
q = E85CED54AF57E53E092113E62F436F4F
e = 0D88C3
```

From Euler theorem you can derive $e.d$ mod (p-1)(q-1) = 1

1. Copy sample.c to task2.1.c
2. Modify/declare the variables you will need
3. Initialize your variables
4. Calculate $p - 1$
5. Calculate $q - 1$
6. Calculate the results of step 4 times step 5
7. Calculate $e.dmod$ result of step 6
8. Information on modular multiplicative inverse can be found at https://tinyurl.com/modmultinv
9. Call the display function
10. Compile your C program producing the executable *rsaTask21*
11. Capture your C code, and, the compiling, execution and output of your program

2.2 Task 2.2: Encrypting a Message

Let `(e, n)` be the public key. Your task is to encrypt the message `A top secret!`. You need to convert this ASCII string to a hex string, and then convert the hex string to a BIGNUM. The following `python` command can be used to convert a plain ASCII string to a hex string.

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

The public keys are listed in hexadecimal below.

```
n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
e = 010001 (this hex value equals to decimal 65537)
M = A top secret!
```

To Complete Task 2.2

Encrypt the message `M` of `A top secret!` into `y` using the formula $y = m^e$ mod n.

1. Copy sample.c to task2.2.c
2. Modify/declare the variables you will need
3. Initialize your variables
4. Calculate y
5. Call the display function
6. Compile your C program producing the executable *rsaTask22*
7. Capture your C code, and, the compiling, execution and output of your program

2.3 Task 2.3: Decrypting a Message

The objective of this task is to decrypt a given ciphertext, given the hexadecimal values of `n`, `e`, and `d`

```
n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
e = 010001 (this hex value equals to decimal 65537)
d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D
```

To Complete Task 2.3

Decrypt the following ciphertext `C`, and convert it back to a plain ASCII string using the formula $M = encoded^d$ mod n.

```
C = 8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F
```

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

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

1. Copy sample.c to task2.3.c
2. Modify/declare the variables you will need
3. Initialize your variables
4. Determine M as a hexadecimal value
5. Call the display function
6. Compile your C program producing the executable *rsaTask23*
7. Display the converted M as a hexadecimal value as the original string.
8. Capture your C code, and, the compiling, execution and output of your program and the display of the original string.

3 Submission

Submit a single PDF containing:

- All screen captures
- All required code as a capture

Your captures should be labeled under the appropriate section. You will have a section title of the section and then the captures under that section.

Name your PDF your last name first letter of your first name Lab3.pdf. (Example: steinersLab3.pdf)