RSA Public-Key Encryption and Digital Signature Applications

Background Information

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, we cannot do that anymore; instead, we need to use an algorithm (i.e., a function) to compute their products.

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. To use this library, we 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.

Lab Environment

This lab has been adapted from SEED Project. https://seedsecuritylabs.org/labsetup.html

Install VirtualBox and Ubuntu 20.04 on your computer according to the instructions given the link above.

BIGNUM APIs

All the big number APIs can be found from https://linux.die.net/man/3/bn. In the following, 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 a number of 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);
```

Check whether a and b are relatively prime

```
BN_gcd(res, a, b, ctx);
if (!BN_is_one(res)) {
  printf("Not prime!");
  exit(0);
}
```

Hint: You will need (p-1)(q-1) (and more 😊) to find out private key. Note that you cannot subtract 1 directly from a BIGNUM. To do this, you can use "BN_value_one()" and subtraction command given above.

A Sample Code

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). You can change this code to perform the tasks given. Note that, this code only gives an idea of the BIGNUM operations. You need to find out the formulas to create necessary values with the given information in the tasks.

```
/* bn_sample.c */
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256
//Print out a big number.
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;
}
```

Compilation. We can use the following command to compile bn_sample.c (the character after - is the letter l, not the number l; it tells the compiler to use the crypto library).

$ gcc bn_sample.c -lcrypto

Task 1: Encrypting a Message

Let (e, n) be the public key. Encrypt the message "Acayip gizli bir mesaj!" (the quotations are not included). We need to convert this ASCII string to a hex string.

The following python commands can be used to convert a plain ASCII string to a hex string.

```
$ python3
>>> import binascii
>>> x=b' Acayip gizli bir mesaj!'
>>> x=binascii.hexlify(x)
>>> x
b'4163617969702067697a6c6920626972206d6573616a21'
```

Note that, the numbers within ‘’ above is the hex representation of the message. In our code, we must convert this hex string to a BIGNUM using the hex-to-bn API BN hex2bn().

The public keys are listed in the followings (hexadecimal). Private key d is provided to help you verify your encryption result.

n=BB300643E39AA365612115898C2737D969635148A40AAAD9F2A92E60A7BB1BB7DA9
A09F339FE02761FF451FF0FAFAFEA1C792D3C0114B2D4234FCFEABF1249C1

e = 0D88C3

M = Acayip gizli bir mesaj! (hex:4163617969702067697a6c6920626972206d6573616a21)

d=8D017DAF61EB9E6E08A74841F2F9B2F50D6913D605C98E416E06D8441DDBE94F5F05
8E2FF8B629B59C98D4A6B799909455018CDE39C9FC3A4A74A6E483E45C07

Encrypt the message above and provide ciphertext (hex) as the proof of your work.

Task 2: Encrypting a Message by Derived Key

Let p, q, and e be three prime numbers. Let n = p*q. We will use (e, n) as the public key. 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 256 bits).

p= C353136B52414B12B4149F7FA641AE97A07C98292D4358227DFE0EA3BC4DAD7F

q= F555DEEF7084C34D2FB95C3B942BB4CCF06A8FD18CE63A87D63275CE06FE28BF

e = 010001

M=Bu da ikinci gizli mesaj

Encrypt the message above and provide ciphertext (hex) as the proof of your work.

Task 3: Decrypting a Message

The public/private keys used in this task are the same as the ones used in previous task. Decrypt the following ciphertext C (so, you need to calculate private key d to achieve this) and convert it back to a plain ASCII string.

C=7AA0FF25F5D5C94FBEA7109F8AA34A43ADA883EF30CE12A4595BBD92D36D91FBE
43A841400345177D6572F6587882FAB78549D6155500F9D319F892F8E74F07F

The following python commands can be used to convert a plain ASCII string to a hex string.

```
$ python3
>>> import binascii
>>>
bytes.fromhex('4163617969702067697a6c6920626972206d6573616a21').decode('utf-8')
'Acayip gizli bir mesaj!'
```

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 = Sana 1 milyon lira borcum var.

Please make a slight change to the message M, such as changing 1 to 2, and sign the modified message. Compare both signatures and describe what you observe.

Task 5: Verifying a Signature

Bob receives a message M = "Launch a missile." from Alice. We know that Alice's public key is (e, n). For some reason, Bob got two signatures (S1 and S2) from Alice. The public key and signature (hexadecimal) are listed in the following:

M = Launch a missile.

S1=643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F

S2=4E96B0012354774DD6C90215F0A51D356D08D9D64064C8703962C414378CE7F3

e=010001

n=AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115

Find out which signature belongs to Alice. Show the results of your verification for both signatures.