RSA Public-Key Encryption and Signature Lab

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 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 environment. This lab has been tested on the SEED Ubuntu 20.04 VM. You can download a pre-built image from the SEED website, and run the SEED VM on your own computer. However, most of the SEED labs can be conducted on the cloud, and you can follow our instruction to create a SEED VM on the cloud.

Acknowledgment This lab was developed with the help of Shatadiya Saha, a graduate student in the Department of Electrical Engineering and Computer Science at Syracuse University.

2 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, we cannot do that any more; 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.

2.1 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);
```

2.2 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)`.

```
/* bn_sample.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 *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 = 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 `bn_sample.c` (the character after - is the letter ℓ, not the number 1; it tells the compiler to use the `crypto` library).

```
$ gcc bn_sample.c -lcrypto
```

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

3.2 Task 2: Encrypting a Message

Let `(e, n)` be the public key. Please encrypt the message `"A top secret!"` (the quotations are not included). We 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()`. 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 the followings (hexadecimal). We also provide the private key `d` to help you verify your encryption result.

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

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 = 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!
```

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 = 643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F
e = 010001 (this hex value equals to decimal 65537)
n = AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115
```

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 Task 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 Corporation for Assigned
     Names and Numbers/OU=Technology/CN=www.example.org
   i:/C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert SHA2 High Assurance
     Server CA
   -----BEGIN CERTIFICATE-----
   MIIF8jCCBNqgAwIBAgIQDmTF+8I2reFLFyrrQceMsDANBgkqhkiG9w0BAQsFADBw
   MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
   ......
   wDSiIIWIWJiJGbEeIO0TIFwEVWTOnbNl/faPXpk5IRXicapqiII=
   -----END CERTIFICATE-----
 1 s:/C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert SHA2 High
     Assurance Server CA
   i:/C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert High Assurance
     EV Root CA
   -----BEGIN CERTIFICATE-----
   MIIEsTCCA5mgAwIBAgIQBOHnpNxc8vNtwCtCuF0VnzANBgkqhkiG9w0BAQsFADBs
   MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
   ......
   cPUeybQ=
   -----END CERTIFICATE-----
```

The result of the command contains two certificates. 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, the second certificate belongs to an intermediate CA. In this task, we will use CA's certificate to verify a server certificate.

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 certificate (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 `c0.pem` and the second one `c1.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 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
...
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. 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.

```
$ cat signature | tr -d '[:space:]:'
84a89a11a7d8bd0b267e52247bb2559dea30895108876fa9ed10ea5b3e0bc7
......
5c045564ce9db365fdf68f5e99392115e271aa6a8882
```

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 is able to parse our X.509 certificate.

```
$ openssl asn1parse -i -in c0.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
   :0E64C5FBC236ADE14B172AEB41C78CB0
   ... ...
 1236:d=4  hl=2 l=  12 cons:     SEQUENCE
 1238:d=5  hl=2 l=   3 prim:      OBJECT            :X509v3 Basic Constraints
 1243:d=5  hl=2 l=   1 prim:      BOOLEAN           :255
 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 c0.pem -strparse 4 -out c0_body.bin -noout
```

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

```
$ sha256sum c0_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 students are required to use their own programs to do so, otherwise, they get zero credit for this task.

4 Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. 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.