RSA and Digital Signatures

Import

The cryptographic algorithm**RSA**is a cryptosystemwith a public key. It is an asymmetric cryptographic algorithm where each user has two keys, one private and one public. It uses elements from number theory in combination with particulars***oversized keys*** achieves encryption in remainder arithmetic that makes decryption by factorization impossible. It is still considered a secure cryptographic algorithm.

General Framework

The RSA algorithm involves calculations on large numbers. These calculations cannot be performed directly on the machine using simple arithmetic operators in programs because these operators can only operate on classical data types such as 32-bit and 64-bit integers. The numbers involved in RSA algorithms are typically larger than 512 bits.

For example, to multiply two 32-bit integers**a**and**b**, we just have to use**a*b**in the programus. However, if the numbers are large, this cannot be done. Instead, we must use an algorithm (ie a function) to calculate their products, thus allowing us to overcome the limits of the machine and specifically of its arithmetic and logic unit circuits.

There are various libraries that can perform arithmetic operations on integers of arbitrary size. In this lab, we will use the large numbers library**bn**(multiprecision integer arithmetics) provided by the**openssl**[1]. This library provides the formula **BIGNUM**[2], which can represent any large number. It also provides an API that implements various operations such as addition, multiplication, exponentiation, remainder arithmetic, etc.
In this exercise, all the material of the theoretical part of the RSA course uploaded to eclass will be used.

[1] https://www.openssl.org
[2] https://www.openssl.org/docs/man1.0.2/man3/bn.html

BIGNUM APIs

All APIs for large numbers provided by the librarybn exist in https://www.openssl.org/docs/man1.0.2/man3/bn.html . In the continuation of the exercise, some of the functions needed for this laboratory exercise are described.

• Some library functions require**temporary**variables. Given that the dynamic memory allocation for creating the**BIGNUMs**is quite expensive, when used in conjunction with repeated subroutine calls, a structure is created**BNX_CTX**to hold temporary variablesBIGNUM that used by library functions. We need to create such a structure and pass it (pass as an argument) to the functions that require it.

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

- **Initialization**of a variableBIGNUM:

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

- **Assignment**value to a variableBIGNUM: There are many ways to assign one price to oneBIGNUM 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);
```

- **Printing**a large 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);


}
```

• Calculation**removal**and**addition**:

```
//Calculate res = a - b
BN_sub(res, a, b);
//Calculate res = a + b
BN_add(res, a, b);
```

• Calculation**multiplication**(requireda structureBN_CTX in thisAPI):

```
//Calculate res = a * b
BN_mul(res, a, b, ctx)
```

• Calculation**res = a ∗ b mod n**(requireda structureBN_CTX in thisAPI):

```
//Calculate res = a ∗ b mod n
BN_mod_mul(res, a, b, n, ctx)
```

• Calculation**res = $a_c$mod n**(requireda structureBN_CTX in thisAPI):

```
//Calculate res = a^c mod n
BN_mod_exp(res, a, c, n, ctx)
```

• Calculation**reverse to remainder arithmetic**, that is, for one**a**to be found**b**, such that:

a * b mod n = 1,that is**(a * b ≡ 1 mod n)**.

The price**b**is called its inverse**a**, in balance arithmetic**n**and there isonly if **gcd(a,b)=1**
.

```
//Calculate mod inverse
BN_mod_inverse(b, a, n, ctx);
```

Example

A complete example of using the library is presented below**bn**. In it, we initialize three BIGNUM variables, a, b, and n, and then calculate and we print it**a*b**and the**(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);



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

    // calculate and print res = aˆb mod n
    BN_mod_exp(res, a, b, n, ctx);
    printBN("aˆc mod n = ", res); return 0;

}
```

Dubbing: To compile the program**bn_sample.c**use it below command.

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

Laboratory Activities

To perform the following activities you will need to use the virtual machine**SEED Ubuntu 16.04**. In it you will find all the necessary applications (*browser, terminal*) as well as tools such as*openssl*,*python*.

Activity 1: Generate a private key

Let it be**p**and**q**two prime numbers and**e**a number. Let it be**n = p * q**. We will use it**(e, n)** as the public key.**Calculate**the private key**d**. The hexadecimal values ofp, q, and e are listed below.

```
p  =  953AAB9B3F23ED593FBDC690CA10E703
q  =  C34EFC7C4C2369164E953553CDF94945
e  =  0D88C3
```

Note: It should be noted that although the**p**and**q**used in this activity are large numbers, not large enough to be safe. For the purposes of the exercise and for the sake of simplicity, we use 128-bit numbers. In practice, these numbers should be at least 512 bits long.

Activity 2: Encrypt a message

Consider the message**M**consisting of theyour name (in Latin characters), in the following form:

```
Onoma Eponymo
```

You should write the code to encrypt this message. The public key **(e,n)**which you will use will be the same as the one you used in Activity 1. Next, with the private key**d**that you found in the Activity1, you should verify the encryption result.

Note: To proceed with encryption you must first convert it your message from string ASCII to hexadecimal (hex) string and to then convert the hexadecimal string to BIGNUM using the functionBN_hex2bn().

The following commandpython can be used to convert a simple string**ASCII**in one **hexadecimal**string.

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

Accordingly, to convert one**hexadecimal**string to one**ASCII**string (readable format) you can use below commandpython.

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

Activity 3: Decipher a message

The public and private keys used in this activity are as follows:

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

Decrypt the following C cipher, and convert it back to oneASCII string in readable format.

```
C= B3AF0A70793BB53492B5311AED5EA843D94661924C97A446E9DD75846DF860DF
```

Activity 4: Signing a message

The public and private keys used in this activity are the same as in activity 3. Consider a short message**M**<u>of your choice</u> (in Latin characters). Create one**digital signature**of your message. Then make a small change to the message**M,**such as changing a letter or number, and re-sign the modified message. Finally, compare the two signatures and describe what you notice.

Note:for the purposes of the exercise, sign**the message directly**this instead of sign the hash value:

Activity 5: Signature Verification

Perform signature verification in the following two cases:

Case A

Alice receives a message M = “Launch a missile.” from Bob, along with his signature S. We know that Bob's public key is**(e, n)**. Confirm the signature is Bob's or not. The public key and signature (in hexadecimal) are given below:

```
M = Launch a missile.
S = 643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F e =
010001 (this hex value equals to decimal 65537)
n = AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115
```

The signature is then considered to be corrupted (destroyed) so that the last byte of the signature changes from 2F to 3F, i.e., there is only one bit that has changed. Repeat this activity and describe what will happen during the signature verification process.

Case B

Bob receives the message M=“Please transfer me $2000.Alice.” by Alice, along with her signature S. We know that Alice's public key is**(e, n)**. Confirm the signature is Alice's or not. The public key and signature (in hexadecimal) are given below:

```
M = Please transfer me $2000.Alice.
S = DB3F7CDB93483FC1E70E4EACA650E3C6505A3E5F49EA6EDF3E95E9A7C6C7A320 e =
010001 (this hex value equals to decimal 65537)
n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
```

(Hint: you can also use the function if you want**BN_cmp**for the comparing two numbers – see also the link: https://www.openssl.org/docs/man1.0.2/man3/ BN_cmp.html)

Activity 6: Manually verify an X.509 certificate

In this activity, you will need to manually check a certificateX.509 using the program you developed in a previous activity. An X.509 certificate contains data about apublic key and herpublisher's signature in the data. You should get an actual X.509 certificate from**one web server**of your choice, export its publisher's public key and to then use this public key to verify the signature on the certificate.

Step 1: Download a certificate from a real oneweb server**of your choice**. You can download certificates using the browser or withopenssl from the terminal, using the following command:

```
$ openssl s_client -connect www.example.org:443 -showcerts
```

The result of the command in this example contains two certificates. His field**subject** (the entry beginning with the indication**s:**) of the certificate is www.example.org, i.e. this is a certificate of www.example.org. His field**publisher** (the entry beginning with the indication**i:**) provides the publisher information. His fieldsubject of the second certificate is the same as its scopepublisher of the first certificate. Basically, the second certificate belongs to an intermediate**CA**(Certification Authority).

```
. . .
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
MQswCQYDFQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMSkwJwYDVQQDEyBE . . .

/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-----
```

```
IIE6jCCA9KgAwIBAgIQCjUI1VwpKwF9+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
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3 . . .

YSEY1QSteDwsOoBrp+uvFRTp2InBuThs4pFsiv9kuXclVzDAGySj4dzp30d8tbQk
CAUw7C29C79Fv1C5qfPrmAESrciIxpg0X40KPMbp1ZWVbd4=
- - - - - END CERTIFICATE-----
```

If using the above command you get**only one certificate**, this means that the certificate you received is signed directly by a**root CA**. The certificate of one**root CA**can be obtained fromFirefox browser installed on the virtual machine. Choose Edit —> Preferences —> Privacy —> and after Security —> View Certificates. Search for the publisher name and download it his certificate.
Copy and paste each of the certificates (the text between the line containing the “Begin CERTIFICATE" and the line containing "End CERTIFICATE", compincluding these two lines). Call it a file **c0.pem**and the other**c1.pem**.

Step 2: Extract the public key**(e, n)**from the issuer's certificate. TheOpenssl provides commands to extract certain attributes from x509 certificates. We can extract the value of n using the option -modulus.

```
$ openssl x509 -in c1.pem -noout -modulus
```

There is no special command to extract it**e**, but we can print all the fields and easily find the value of e.

```
$ openssl x509 -in c1.pem -text -noout
```

Step 3: Extract it**digital signature**from his certificateserver. There is no specific commandopenssl to extract the signature field. However, we can print 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 will need to 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 strip the spaces and colons from the data so that we have a hexadecimal string that we can feed our program with. The following commands can achieve this goal. The command**tr** it is an auxiliary tool ofLinux for string management. In this case, the choice**-d**used to remove ":" and "blanks" from data.

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

5c045564ce9db365fdf68f5e99392115e271aa6a8882
```

Step 4: Extract the body of its certificateserver. A Certificate Authority (CA) creates the signature for a server certificate by first calculating it**hash**of certificate and then signing ithash. To verify her signature, we also need to create thehash from a certificate. Given that the hash is generated before the signature is computed, we must exclude the signature block of a certificate when computing ithash. OR find the part of the certificate used to create ithash is quite difficult without a good understanding of the certificate format.

The certificates**X.509**coded according to the standard**ASN.1**(Abstract Syntax Notation.One), so if we can parse the structure**ASN.1**, we can easily extract any field from a certificate. TheOpenssl has a command that is called**asn1parse**, which can be used to parse a certificateX.509.

```
$ openssl asn1parse -i -in c0.pem
        0:d=0 hl=4 l=1522 cons: SEQUENCE
        4:d=1 hl=4 l=1242 cons: SEQUENCE                  ➊
```

```
    8:d=2 hl=2l= 3cons: cont[0] 10:d=3
              hl=2 l=      1 premium:     INTEGER              :02
    13:d=2    hl=2 l=     16 prims:      INTEGER
    :0E64C5FBC236ADE14B172AEB41C78CB0 . . .

1236:d=4    hl=2 l=     12 cons: SEQUENCE
1238:d=5    hl=2 l=      3 prim: OBJECT          :X509v3 Basic Constraints :255
1243:d=5    hl=2 l=      1 prim: BOOLEAN
1246:d=5    hl=2 l=      2 prims: 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 prime:      NULL
1265:d=1    hl=4 l= 257 prim:          BIT STRING
```

The field starting from❶is the body of the certificate used to create it**hash**. The field starting from❷is the signature block. Their distances (*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 the certificatesX.509, the starting offset is always the same (ie 4), but the end depends on the content length of a certificate. We can use the option**-strparse**to export the field from*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 certificate body, we can compute it**hash**of using the following command:

```
$ sha256sum c0_body.bin
```

(Caution: the above applies if the hash function used for the signature is SHA256. Otherwise, the above command needs adaptation).

Step 5: Verify the signature. Now we have all the information, including him***its public keyCA***, her***her signatureCA***and his ***body of the certificate***ofserver. You should run your own program to verify whether the signature is valid or not.

Submission

You should submit a detailed**laboratory report**(in PDF) which will cover all the actions you took in each activity, your results and observations. In your description, use screenshots and code snippets, according to each activity's instructions. Also, you should submit**to separately archives**the (properly commented) source code you developed.