RSA Public-Key Encryption and Signature Lab

1. BIGNUM

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 datatypes, 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. 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. Task 1: Deriving the Private Key

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

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

The following screenshot shows the C code to derive the private key. We use BIGNUM APIs to do the calculations. As we know from the Euler theorem, we can derive that:

$$e.\ d \bmod (p-1)\ (q-1) = 1$$

Using the above formula, we can calculate the inverse modulo using BIGNUM API and find d.

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

        //res1 = p-1
        BN_sub(res1,p,one);

        //res2 = q-1
        BN_sub(res2,q,one);

        //res3 = res1*res2
        BN_mul(res3,res1,res2,ctx);

        // Modinverse : e.d mod res3 = 1
        BN_mod_inverse(d, e, res3, ctx);
        printBN("d = ", d);
        return 0;
}
```

The below screenshot shows the value of private key d.

```
seed@VM:~/RSA$ gcc -o Task1 Task1.c -lcrypto
seed@VM:~/RSA$ ./Task1
d =  3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB
```

We can also verify the value of d by using the above formula.

3. Task 2: Encrypting a message

The objective of this task is to encrypt a given message. Given are the hexadecimal values of n, e, M. the value of "d" is also given to verify the result.

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

We need to convert the ASCII string to a hex string. We use python command to do that.

```
seed@VM:~/RSA$ python -c 'print("A top secret!".encode("hex"))'
4120746f702073656372657421
```

The below screenshot shows the C code to encrypt the message and then verify it by decrypting it.

```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 *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,"4120746f702073656372657421");
        BN_hex2bn(&e,"010001");
        BN_hex2bn(&n,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
        BN_hex2bn(&d,"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

        // Encryption : m^e 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;
}
```

The below screenshot shows the result of the above code. We get the encrypted message using the Encryption algorithm: M ^ e mod n. We verify it by using the RSA Decryption formula.

```
seed@VM:~/RSA$ gcc -o Task2 Task2.c -lcrypto
seed@VM:~/RSA$ ./Task2
Encrypted message =  6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC
Decrypted message =  4120746F702073656372657421
```

We get the Hex value of the decrypted message. We can convert it to Ascii to see the original message.

```
seed@VM:~/RSA$ python -c 'print("4120746F702073656372657421".decode("hex"))'
A top secret!
```

4. Task 3: 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
```

The below screenshot shows the C code to decrypt the ciphertext.

```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 *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(&d,"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
        BN_hex2bn(&n,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
        BN_hex2bn(&enc,"8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");

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

Using the Decryption algorithm Enc ^ d mod n, we get the decrypted message as shown.

```
seed@VM:~/RSA$ gcc -o Task3 Task3.c -lcrypto
seed@VM:~/RSA$ ./Task3
Decrypted message =  50617373776F72642069732064656573
```

The decrypted message is then decoded to get the ASCII string.

```
seed@VM:~/RSA$ python -c 'print("50617373776F72642069732064656573".decode("hex"))'
Password is dees
```

5. Task 4: Signing a Message

The objective of this task is to generate a signature for the following message. We will use the public/private key set from Task 2.

```
M = I owe you $2000.
```

We generate the Hex of the given string using Python command.

```
seed@VM:~/RSA$ python -c 'print("I owe you $2000.".encode("hex"))'
49206f776520796f752024323030302e
```

The below screenshot gives the C code to generate the signature of the given message.

```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 *m = BN_new();
        BIGNUM *n = BN_new();
        BIGNUM *d = BN_new();
        BIGNUM *sign = BN_new();

        // Initialize p, q, e
        BN_hex2bn(&m,"49206f776520796f752024323030302e");
        BN_hex2bn(&n,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
        BN_hex2bn(&d,"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

        // Signature : m^d mod n
        BN_mod_exp(sign,m,d,n,ctx);
        printBN("Signed Message = ", sign);

        return 0;
}
```

Using the signing algorithm M ^ d mod n, we get the signed message as:

```
seed@VM:~/RSA$ gcc -o Task4 Task4.c -lcrypto
seed@VM:~/RSA$ ./Task4
Signed Message =  55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73CCB35E4CB
```

We will change the given message slightly and check the signed message. We will convert the changed message to Hex form.

```
seed@VM:~/RSA$ python -c 'print("I owe you $3000.".encode("hex"))'
49206f776520796f752024333030302e
```

Using the above code and changing the value of “m”, we get the new signed message as:

```
seed@VM:~/RSA$ gcc -o Task4a Task4a.c -lcrypto
seed@VM:~/RSA$ ./Task4a
Signed Message =  BCC20FB7568E5D48E434C387C06A6025E90D29D848AF9C3EBAC0135D99305822
```

We can see that the signed message is completely different than the previous signed message.

6. Task 5: Verifying a Signature

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

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

The below screenshot shows the C code to get the message from the signature. Bob can run the below and get the original message and check it against the message received by him.

```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 *s = BN_new();
        BIGNUM *n = BN_new();
        BIGNUM *e = BN_new();
        BIGNUM *message = BN_new();

        BN_hex2bn(&s,"643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");
        BN_hex2bn(&n,"AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");
        BN_hex2bn(&e,"10001");

        // Message : S^e mod n
        BN_mod_exp(message,s,e,n,ctx);
        printBN("Message = ", message);

        return 0;
}
```

In the above code, we get the message from the signature using S ^ e mod n. On converting the hex to ascii, we get the actual string. Hence, it is verified that the message was signed by Alice using her private key.

```
seed@VM:~/RSA$ gcc -o Task5 Task5.c -lcrypto
seed@VM:~/RSA$ ./Task5
Message =  4C61756E63682061206D697373696C652E
seed@VM:~/RSA$ python -c 'print("4C61756E63682061206D697373696C652E".decode("hex"))'
Launch a missile.
```

Suppose considering that the signature was corrupted and the last byte changes from 2F to 3F, we will repeat the task and check the verification process.

```
seed@VM:~/RSA$ gcc -o Task5a Task5a.c -lcrypto
seed@VM:~/RSA$ ./Task5a
Message =  91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294
seed@VM:~/RSA$ python -c 'print("91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294".decode("hex"))'
```

We can see that the hex of the message is completely different from the previous one which proves that the signature is incorrect and is not of the message sent by Alice.

7. Task 6: Manually Verifying an X.509 Certificate

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

To verify that a certificate was signed by a specific Certificate Authority, we would need the following:

- Public key of the CA (issuer)
- Signature and Algorithm used to generate signature from the server's certificate

We need to follow steps to get the public key of the issuer and signature from the server's certificate:

Step 1: Download a certificate from a real web server

Download a certificate of any server. For example: www.chase.com using the command :

openssl s_client -connect www.chase.com:443 -showcerts

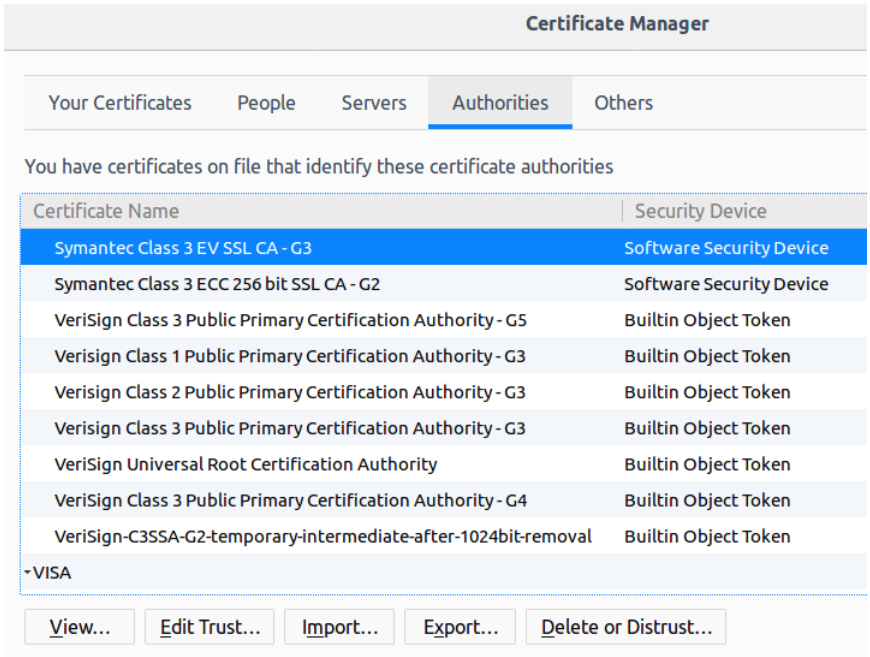
```
---
Server certificate
subject=/jurisdictionC=US/jurisdictionST=Delaware/businessCategory=Private Organization/serialNum
 Ave 38TH FL/O=JPMorgan Chase and Co./OU=GTI GNS/CN=www.chase.com
issuer=/C=US/O=Symantec Corporation/OU=Symantec Trust Network/CN=Symantec Class 3 EV SSL CA - G3
---
No client certificate CA names sent
---
SSL handshake has read 3180 bytes and written 623 bytes
---
New, TLSv1/SSLv3, Cipher is AES128-GCM-SHA256
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : AES128-GCM-SHA256
    Session-ID: C01FF3A9F5C247ED68A734051ECBAF35D901C7B2B18BC7F78E0D7D5B6CDB7EC5
    Session-ID-ctx:
    Master-Key: 2A31DD5C098F307BA707F965AADB1BE0083694BE9426128A246B2EC8790BF7B54C8808C18C8B3D19C
    Key-Arg   : None
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    Start Time: 1524086215
    Timeout   : 300 (sec)
    Verify return code: 0 (ok)
---
```

We copy the contents of the server certificate “BEGIN CERTIFICATE ---------- END CERTIFICATE” and save it to a file as c0.bin.

The above image shows the server certificate. As we can see, it lists the issuer of the certificate. For chase.com, the issuer or the certificate authority is Symantec Class 3 EV SSL CA -G3 as shown.

Once we have the issuer of the certificate we can download it’s certificate from the browser.

In Firefox browser, go *to Edit->Preferences->Privacy and Security->View Certificates*. Search for the name of the issuer and download its certificate. The below image shows the list of certificate authorities. We can export the certificate of the issuer from the list.



We copy the contents of the server certificate "BEGIN CERTIFICATE---------- END CERTIFICATE" and save it to a file as c1.pem.

Step 2: Extract the public key (e, n) from the issuer's certificate.

Now, we need the public key (e, n) of the Certificate Authority. OpenSSL provides many commands to extract certain attributes from the x509 certificates.

The below screenshots show the commands to retrieve modulus and exponent.

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

```
seed@VM:~/RSA$ openssl x509 -in c1.pem -noout -modulus
Modulus=D8A1657423E82B64E232D733373D8EF5341648DD4F7F871CF8442313
2EFB2ACEC89A7F87BFD84C041532C9D1CC9571A04E284F84D935FBE3866F9453
900E5FB75DA45172467013BF67F2B6A74D141E6CB953EE231A4E8D48554341B1
8AE1A170070C340F
```

openssl x509 -in c1.pem -text -noout

```
seed@VM:~/RSA$ openssl x509 -in c1.pem -text -noout | grep "Exponent"
            Exponent: 65537 (0x10001)
```

Step 3: Extract the signature from the server's certificate.

The below command outputs the certificate in text format. We copy the Signature Algorithm to a file.

openssl x509 -in c0.pem -text -noout

```
    Signature Algorithm: sha256WithRSAEncryption
         1e:c7:dc:07:dc:24:9a:c4:22:a2:56:85:3d:90:6f:2a:bb:57:
         17:00:b2:66:0e:c8:4f:74:d7:65:b4:47:cf:8e:55:4e:fd:9d:
         25:2b:35:29:11:e2:75:20:8e:34:7e:2b:30:c1:65:5b:99:49:
         f1:a6:9f:12:50:39:95:32:a0:3b:23:ef:75:eb:f0:e9:2c:9e:
         a6:46:29:b1:f1:72:fd:3a:19:06:0d:41:ba:14:0a:e3:9b:8f:
         c9:37:f5:1b:ae:45:5b:12:60:3f:d4:95:45:29:44:18:0d:30:
         b9:8f:c9:ec:56:fb:e0:53:a1:37:b3:41:8c:22:14:b0:35:c9:
         d2:81:d5:42:36:a5:94:d8:38:6a:e1:bd:03:45:ba:e9:ac:c5:
         de:be:b0:67:30:bf:87:57:71:d4:b9:c8:a2:23:d2:90:1a:55:
         f3:ea:7e:06:ec:72:ee:7c:bd:ba:12:70:06:74:43:42:dc:d3:
         e7:9d:18:6b:1e:25:93:b6:89:c2:f7:bd:17:b2:3a:13:d6:52:
         a4:42:fb:31:d8:8a:74:18:fc:01:69:93:34:32:b7:c7:24:0f:
         0e:09:ad:d3:20:c7:ab:a3:5e:c3:23:b1:8c:ce:38:a2:4a:6d:
         8b:a6:d3:97:55:07:e5:20:6c:82:c1:43:78:ff:7d:17:4a:94:
         79:cf:2d:e4
```

Using the below command, we remove the spaces from the signature.

cat signature | tr -d '[:space:]:'

```
seed@VM:~/RSA$ cat signature | tr -d '[:space:]:'
SignatureAlgorithmsha256WithRSAEncryption1ec7dc07dc249ac422a256853d906f2ab
9d252b352911e275208e347e2b30c1655b9949f1a69f1250399532a03b23ef75ebf0e92c9e
937f51bae455b12603fd495452944180d30b98fc9ec56fbe053a137b3418c2214b035c9d28
b06730bf875771d4b9c8a223d2901a55f3ea7e06ec72ee7cbdba127006744342dcd3e79d18
1d88a7418fc0169933432b7c7240f0e09add320c7aba35ec323b18cce38a24a6d8ba6d3975
```

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. The following command parses the certificate which gives the body excluding the Signature.

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

```
seed@VM:~/RSA$ openssl asn1parse -i -in c0.pem -strparse 4 -out c0_body.bin -noout
seed@VM:~/RSA$ sha256sum c0_body.bin
ccad50471e8d1adbd5e560b9ba112fabd416202e771a4ee390eb39013f3ebdaa  c0_body.bin
```

We then generate the hash of the certificate body.

Step 5: Verify the signature

Now we run the code for Task 5 to decrypt the signature by substituting the values of the "Signature" "e" and "n" as shown. Here we are decrypting the Signature of the server's certificate using Certificate Authority's public key.

```
seed@VM:~/RSA$ ./Task6
Decrypted Signature =  01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF003031300D06096
0864801650304020105000420CCAD50471E8D1ADBD5E560B9BA112FABD416202E771A4EE39
0EB39013F3EBDAA
```

The highlighted part shows the decrypted signature. We can see the sha256 of the body of the certificate and the decrypted signature are equal which proves that the server certificate was signed by the Certificate Authority.

What to submit:

- Working version of all code files, properly commented.
- Screen shots of your output for each of the steps/tasks.