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.

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

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.

3 Lab Tasks

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

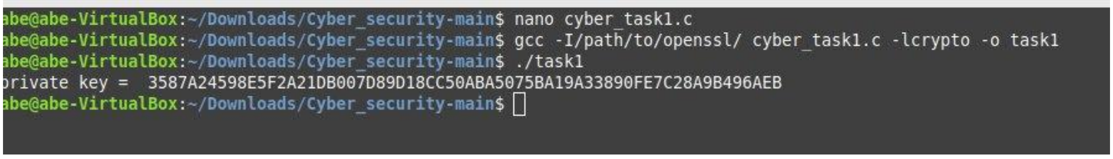
e = 0D88C3

◆ c program to generate the private key

```
// openssl header files and standard libraries
#include <stdio.h>
#include <openssl/bn.h>
#include <openssl/ssl.h>
#include <openssl/rsa.h>
#include <openssl/x509.h>
#include <openssl/evp.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 *p_minus_one = BN_new();
BIGNUM *q = BN_new();
BIGNUM *q_minus_one = BN_new();
BIGNUM *one =BN_new();
BIGNUM *n = BN_new();
BIGNUM *e = BN_new();
BIGNUM *d = BN_new();//private key
// initialize the variables
BN_dec2bn(&one,"1");;
BN_hex2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF");
BN_hex2bn(&q, "E85CED54AF57E53E092113E62F436F4F");
BN_sub(p_minus_one,p,one);
BN_sub(q_minus_one,q,one);
BN_hex2bn(&e, "0D88C3");
BN_mul(n, p_minus_one, q_minus_one, ctx);
// d = BN_mod_inverse(d, e, n, ctx) meaning d is the inverse of e
when d*e mod n = 1
BN_mod_inverse(d, e, n, ctx);
printBN("private key = ", d);
return 0;
```

```
}
```

◆ *Private key output from the program*



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.

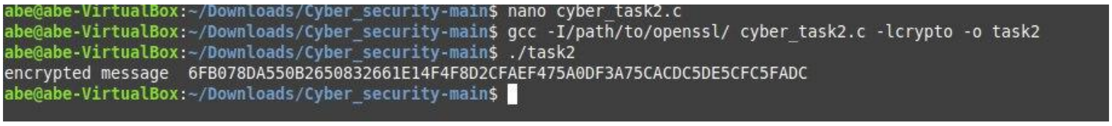
n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
e = 010001 (this hex value equals to decimal 65537)
M = A top secret!
d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30

◆ C program to encrypt the given message

```
// openssl header files and standard libraries
#include <stdio.h>
#include <openssl/bn.h>
#include <openssl/ssl.h>
#include <openssl/rsa.h>
#include <openssl/x509.h>
#include <openssl/evp.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 *message = BN_new();
BIGNUM *e = BN_new();
BIGNUM *n = BN_new();
BIGNUM *encrypted = BN_new();
```

```
BN_hex2bn(&m,"4120746f702073656372657421");
BN_hex2bn(&e, "10001");
BN_hex2bn(&n,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB816
29242FB1A5");
// encrypted = message^e mod n
BN_mod_exp(encrypted, m, e, n, ctx);
printBN("encrypted message ", encrypted);
return 0;
}
```

- *Encrypted message*



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 a plain ASCII string.

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

- C program to decrypt the given message

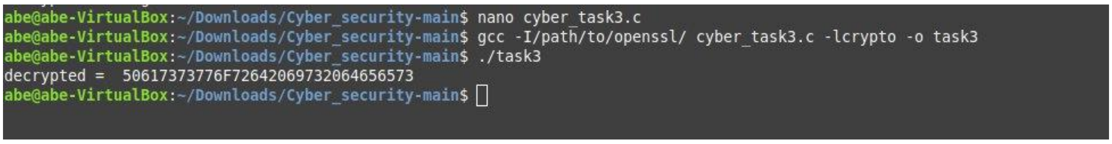
```
#include <stdio.h>
#include <openssl/bn.h>
#include <openssl/ssl.h>
#include <openssl/rsa.h>
#include <openssl/x509.h>
#include <openssl/evp.h>
#define NBITS 256
void printBN(char *msg, BIGNUM * a)
{
/* Use BN_bn2hex(a) for hex string
* Use BN_bn2dec(a) for decimal string */
char * number_str = BN_bn2hex(a);
printf("%s %s\n", msg, number_str);
OPENSSL_free(number_str);
}
int main ()
{
BN_CTX *ctx = BN_CTX_new();
BIGNUM *d = BN_new();
BIGNUM *c = BN_new();
```

```
BIGNUM *n = BN_new();
BIGNUM *decrypted = BN_new();

BN_hex2bn(&d,"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
BN_hex2bn(&c,"8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");
BN_hex2bn(&n,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");

// res = a^b mod n
BN_mod_exp(decrypted, c, d, n, ctx);
printBN("decrypted = ", decrypted);
return 0;
}
```

- *The decrypted message output*



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.

- **C program to signing a message**

```
#include <stdio.h>
#include <openssl/bn.h>
#include <openssl/ssl.h>
#include <openssl/rsa.h>
#include <openssl/x509.h>
#include <openssl/evp.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 *first_message = BN_new();
BIGNUM *second_message =BN_new();
BIGNUM *e = BN_new();
BIGNUM *n = BN_new();
BIGNUM *first_message_encrypted = BN_new();
BIGNUM *second_message_encrypted = BN_new();
//  first message = I owe you $2000 = python -c ' print("I owe you
$2000".encode("hex"))'
// second message = I owe you $3000 = python -c ' print("I owe you
$3000".encode("hex"))'
BN_hex2bn(&first_message, "49206f776520796f752024323030302e");
BN_hex2bn(&second_message, "49206f776520796f752024333030302e");
BN_hex2bn(&e,"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA
381CD7D30D");
BN_hex2bn(&n,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB816
29242FB1A5");
BN_mod_exp(first_message_encrypted, first_message, e, n, ctx);
BN_mod_exp(second_message_encrypted, second_message, e, n, ctx);
printBN("first message encryption =", first_message_encrypted);
printBN("second message encryption=",second_message_encrypted);
return 0;
}
```

- *Encrytped message differences after changing 2000 to 3000*

```
abe@abe-VirtualBox:~/Downloads/Cyber_security-main$ nano cyber_task4.c
abe@abe-VirtualBox:~/Downloads/Cyber_security-main$ gcc -I/path/to/openssl/ cyber_task4.c -lcrypto -o task4
abe@abe-VirtualBox:~/Downloads/Cyber_security-main$ ./task4
first message encryption = 55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73CCB35E4CB
second message encryption= BCC20FB7568E5D48E434C387C06A6025E90D29D848AF9C3EBAC0135D99305822
abe@abe-VirtualBox:~/Downloads/Cyber_security-main$
```

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 in 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

- C program to Verifying a Signature

```
#include <stdio.h>
```

```
#include <openssl/bn.h>
#include <openssl/ssl.h>
#include <openssl/rsa.h>
#include <openssl/x509.h>
#include <openssl/evp.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 *message  = BN_new();
BIGNUM *signature = BN_new();
BIGNUM *changed_signature =BN_new();
BIGNUM *e = BN_new();
BIGNUM *n = BN_new();
BIGNUM *decrytped_signature = BN_new();
BIGNUM *decrypted_change_signature = BN_new();
// first find the hex value of M ="Launch a missile " using python -
c 'print("Launch a missile".encode("hex"))' command that is
"4c61756e63682061206d697373696c65"
// change the last 2F value of the signature to 3f and store as a
changed signature
BN_hex2bn(&message,"4c61756e63682061206d697373696c652e");
BN_hex2bn(&signature,
"643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");
BN_hex2bn(&changed_signature,
"643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F");
BN_hex2bn(&e,"010001");
BN_hex2bn(&n,"AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B45
0F18116115");
BN_mod_exp(decrytped_signature, signature, e, n, ctx);
BN_mod_exp(decrypted_change_signature, changed_signature, e, n, ctx);
printBN("message =",message);
printBN("decrypted signature=",decrytped_signature);
printBN("changed signature decryption =",decrypted_change_signature);
return 0;
}
```

C program output

```
abe@abe-VirtualBox:~/cyber$ ./task5
message = 4C61756E63682061206D697373696C652E
decrypted signature= 4C61756E63682061206D697373696C652E
changed signature decryption = 91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294
abe@abe-VirtualBox:~/cyber$
```

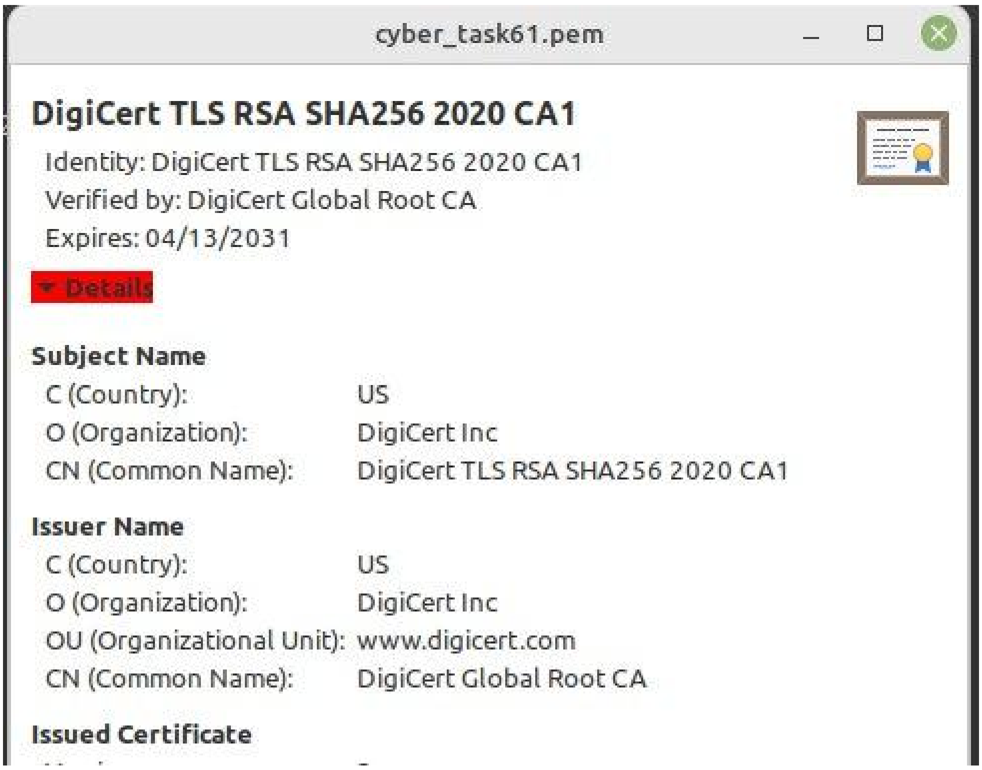
- *The hex value of the message* M = *Launch a missile is the same as the decoded hex value of the signature after decrypting the signature by Alice's public key. So the message was from her.*
- *When I changed the last byte of the signature from 2F to 3F and decrypt both the signature and the changed signature, I got different hex values. The signature says:* Launch a missile. *And the changed signature says*: � � ,0� c� � rm=f� :N� � � � �

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

```
abe@abe-VirtualBox:~/cyber$ openssl s_client -connect www.example.org:443 -showcerts
CONNECTED(00000003)
depth=2 C = US, O = DigiCert Inc, OU = www.digicert.com, CN = DigiCert Global Root CA
verify return:1
depth=1 C = US, O = DigiCert Inc, CN = DigiCert TLS RSA SHA256 2020 CA1
verify return:1
depth=0 C = US, ST = California, L = Los Angeles, O = Internet\C2\A0Corporation\C2\A0for\C2\A0Assigned\C2\A0Names\C2\A0and\C2\A0Numbers, CN = www.example.org
verify return:1
---
Certificate chain
 0 s:C = US, ST = California, L = Los Angeles, O = Internet\C2\A0Corporation\C2\A0for\C2\A0Assigned\C2\A0Names\C2\A0and\C2\A0Numbers, CN = www.example.org
   i:C = US, O = DigiCert Inc, CN = DigiCert TLS RSA SHA256 2020 CA1
-----BEGIN CERTIFICATE-----
MIIHRzCCBi+gAwIBAgIQD6pjEJMHvD1BSJJkDM1NmjANBgkqhkiG9w0BAQsFADBP
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMSkwJwYDVQQDEyBE
aWdpQ2VydCBUTFMgUlNBIFNIQTI1NiAyMDIwIENBMTAeFw0yMjAzMTQwMDAwMDBa
Fw0yMzAzMTQyMzU5NTlaMIGWMQswCQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWZv
cm5pYTEUMBIGA1UEBxMLTG9zIEFuZ2VsZXMxQjBABgNVBAoMOUludGVybmV0wqBD
b3Jwb3JhdGlvbsKgZm9ywqBBc3NpZ25lZMKgTmFtZXPCoGFuZMKgTnVtYmVyczEY
MBYGA1UEAxMPd3d3LmV4YW1wbGUub3JnMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8A
MIIBCgKCAQEAlV2WY5rlGn1fpwvuBhj0nVBcNxCxkHUG/pJG4HvaJen7YIZ1mLc7
/P4snOJZiEfwWFTikHNbcUCcYiKG8JkFebZOYMc1U9PiEtVWGU4kuYuxiXpD8oMP
```


cyber_task60.pem
www.example.org
Identity: www.example.org
Verified by: DigiCert TLS RSA SHA256 2020 CA1
Expires: 03/14/2023
Details
Subject Name
C (Country): US
ST (State): California
L (Locality): Los Angeles
O (Organization): Internet Corporation for Assigned Names and Numbers
CN (Common Name): www.example.org
Issuer Name
C (Country): US
O (Organization): DigiCert Inc


cyber_task61.pem
DigiCert TLS RSA SHA256 2020 CA1
Identity: DigiCert TLS RSA SHA256 2020 CA1
Verified by: DigiCert Global Root CA
Expires: 04/13/2031
Details
Subject Name
C (Country): US
O (Organization): DigiCert Inc
CN (Common Name): DigiCert TLS RSA SHA256 2020 CA1
Issuer Name
C (Country): US
O (Organization): DigiCert Inc
OU (Organizational Unit): www.digicert.com
CN (Common Name): DigiCert Global Root CA
Issued Certificate

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,

```
abe@abe-VirtualBox:~$ openssl x509 -in cyber_task61.pem -noout -modulus
Modulus=C14BB3654770BCDD4F58DBEC9CEDC366E51F311354AD4A66461F2C0AEC6407E52EDCDCB9
0A20EDDFE3C4D09E9AA97A1D8288E51156DB1E9F58C251E72C340D2ED292E156CBF1795FB3BB87CA
25037B9A52416610604F571349F0E8376783DFE7D34B674C2251A6DF0E9910ED57517426E27DC7CA
622E131B7F238825536FC13458008B84FFF8BEA75849227B96ADA2889B15BCA07CDFE951A8D5B0ED
37E236B4824B62B5499AECC767D6E33EF5E3D6125E44F1BF71427D58840380B18101FAF9CA32BBB4
8E278727C52B74D4A8D697DEC364F9CACE53A256BC78178E490329AEFB494FA415B9CEF25C19576D
6B79A72BA2272013B5D03D40D321300793EA99F5
abe@abe-VirtualBox:~$
```

find the exponent (e): $ `openssl x509 -in c1.pem -text -noout`

```
Exponent: 65537 (0x10001)
```

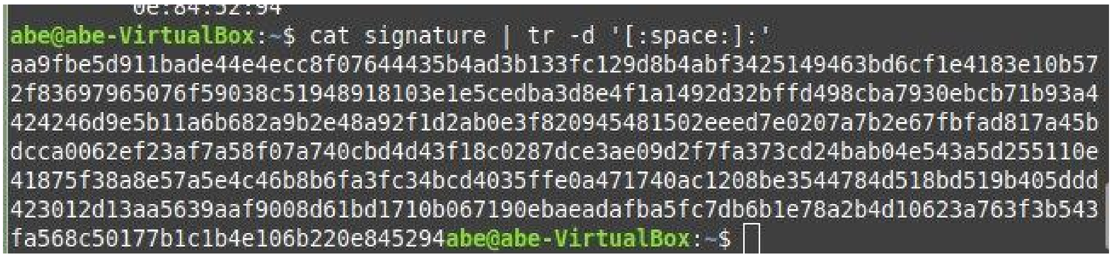
Step 3: Extract the signature from the server's certificate.
There is no specific opensslcommand to extract the signature field. However, we can print out all the fields and then copy and paste the signature block into a file (note: if the signature algorithm used in the certificate is not based on RSA, 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

```
                              16:F6:95:1E:97:96:D7
    Signature Algorithm: sha256WithRSAEncryption
         aa:9f:be:5d:91:1b:ad:e4:4e:4e:cc:8f:07:64:44:35:b4:ad:
         3b:13:3f:c1:29:d8:b4:ab:f3:42:51:49:46:3b:d6:cf:1e:41:
         83:e1:0b:57:2f:83:69:79:65:07:6f:59:03:8c:51:94:89:18:
         10:3e:1e:5c:ed:ba:3d:8e:4f:1a:14:92:d3:2b:ff:d4:98:cb:
         a7:93:0e:bc:b7:1b:93:a4:42:42:46:d9:e5:b1:1a:6b:68:2a:
         9b:2e:48:a9:2f:1d:2a:b0:e3:f8:20:94:54:81:50:2e:ee:d7:
         e0:20:7a:7b:2e:67:fb:fa:d8:17:a4:5b:dc:ca:00:62:ef:23:
         af:7a:58:f0:7a:74:0c:bd:4d:43:f1:8c:02:87:dc:e3:ae:09:
         d2:f7:fa:37:3c:d2:4b:ab:04:e5:43:a5:d2:55:11:0e:41:87:
         5f:38:a8:e5:7a:5e:4c:46:b8:b6:fa:3f:c3:4b:cd:40:35:ff:
         e0:a4:71:74:0a:c1:20:8b:e3:54:47:84:d5:18:bd:51:9b:40:
         5d:dd:42:30:12:d1:3a:a5:63:9a:af:90:08:d6:1b:d1:71:0b:
         06:71:90:eb:ae:ad:af:ba:5f:c7:db:6b:1e:78:a2:b4:d1:06:
         23:a7:63:f3:b5:43:fa:56:8c:50:17:7b:1c:1b:4e:10:6b:22:
         0e:84:52:94
abe@abe-VirtualBox:~$
```

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, which can be used to parse a 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

```
abe@abe-VirtualBox:~$  openssl asn1parse -i -in cyber_task60.pem
    0:d=0  hl=4 l=1863 cons: SEQUENCE
    4:d=1  hl=4 l=1583 cons:  SEQUENCE
    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           :0FAA63109307BC3D414892640
   31:d=2  hl=2 l=  13 cons:   SEQUENCE
   33:d=3  hl=2 l=   9 prim:    OBJECT            :sha256WithRSAEncryption
   44:d=3  hl=2 l=   0 prim:    NULL
   46:d=2  hl=2 l=  79 cons:   SEQUENCE
   48:d=3  hl=2 l=  11 cons:    SET
   50:d=4  hl=2 l=   9 cons:     SEQUENCE
   52:d=5  hl=2 l=   3 prim:      OBJECT            :countryName
   57:d=5  hl=2 l=   2 prim:      PRINTABLESTRING   :US
   61:d=3  hl=2 l=  21 cons:    SET
   63:d=4  hl=2 l=  19 cons:     SEQUENCE
   65:d=5  hl=2 l=   3 prim:      OBJECT            :organizationName
   70:d=5  hl=2 l=  12 prim:      PRINTABLESTRING   :DigiCert Inc
   84:d=3  hl=2 l=  41 cons:    SET
   86:d=4  hl=2 l=  39 cons:     SEQUENCE
```

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
```

```
abe@abe-VirtualBox:~$ sha256sum cy60.bin
7061df0a50b8f2ba3367ecfabab273a16f3bb1378dbe1fe524e6dfd90dfa3b91  cy60.bin
abe@abe-VirtualBox:~$
```

Step 5: Verify the signature.

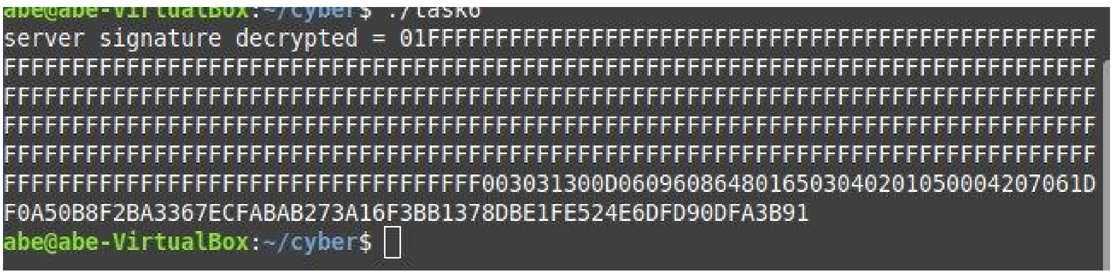
Now we have all the information, including the CA's public key, the CA's signature, and the body of the server's certificate. We can run our own program to verify whether the signature is valid or not.

◆ **C program to verify the signature**

```c
#include <stdio.h>
#include <openssl/bn.h>
#include <openssl/ssl.h>
#include <openssl/rsa.h>
#include <openssl/x509.h>
```

```c
#include <openssl/evp.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 *modulo = BN_new();
BIGNUM *exponent = BN_new();
BIGNUM *server_signature = BN_new();
BIGNUM *decrypted = BN_new();

BN_hex2bn(&modulo,
"C14BB3654770BCDD4F58DBEC9CEDC366E51F311354AD4A66461F2C0AEC6407E52ED
CDCB90A20EDDFE3C4D09E9AA97A1D8288E51156DB1E9F58C251E72C340D2ED292E15
6CBF1795FB3BB87CA25037B9A52416610604F571349F0E8376783DFE7D34B674C225
1A6DF0E9910ED575174>
BN_hex2bn(&server_signature,
"aa9fbe5d911bade44e4ecc8f07644435b4ad3b133fc129d8b4abf3425149463bd6c
f1e4183e10b572f83697965076f59038c51948918103e1e5cedba3d8e4f1a1492d32
bffd498cba7930ebcb71b93a4424246d9e5b11a6b682a9b2e48a92f1d2ab0e3f8209
45481502eeed7e0207a>
BN_hex2bn(&exponent,"010001");
//7061df0a50b8f2ba3367ecfabab273a16f3bb1378dbe1fe524e6dfd90dfa3b91 ..
.body the serve certificate
BN_mod_exp(decrypted,server_signature, exponent, modulo, ctx);
BN_CTX_free(ctx);
printBN("server signature decrypted =",decrypted);
return 0;
}
```



```
abe@abe-VirtualBox:~/cyber$ ./task6
server signature decrypted = 01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF003031300D0609608648016503040201050004207061D
F0A50B8F2BA3367ECFABAB273A16F3BB1378DBE1FE524E6DFD90DFA3B91
abe@abe-VirtualBox:~/cyber$
```

```
abe@abe-VirtualBox:~/cyber$  python -c 'print("7061df0a50b8f2ba3367ecfabab273a16
f3bb1378dbe1fe524e6dfd90dfa3b91".decode("hex"))'
pa�
�;�3g����s�o;�7���$���
abe@abe-VirtualBox:~/cyber$ python -c 'print("01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF003031300D060960864801650304
0201050004207061DF0A50B8F2BA3367ECFABAB273A16F3BB1378DBE1FE524E6DFD90DFA3B91".de
code("hex"))'
��������������������������������������������������������������������������������
��������������������������������������������������������������������������������
��������`�He pa���������������������������010
�;�3g����s�o;�7���$���
```

- *In the above data the body of the server certificate is a little different after decrypting the signature of the server*