ARVIND SAI DOODA **A20553046**

RSA Public-Key Encryption and Signature Lab

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
```

```
[10/14/23]seed@VM:~/.../lab8-RSA$ gedit task1.c
^Z
[1]+  Stopped                 gedit task1.c
[10/14/23]seed@VM:~/.../lab8-RSA$ gcc task1.c -o task1 -l crypto
[10/14/23]seed@VM:~/.../lab8-RSA$ ./task1
d=  3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB
[10/14/23]seed@VM:~/.../lab8-RSA$
```

d= 3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB

working Code explanation :

- We have three prime numbers: p, q, and e.
- We want to calculate the private key 'd' using these primes.
- The public key is represented as (e, n), where n is the product of p and q (n = p * q).

  In the main method:

- Create a BN_CTX structure to hold temporary BIGNUM variables used by library functions.
- Initialize BIGNUM variables: p, q, e, d, res1, res2, res3, and one.

  For example:
- Create a BIGNUM structure for p, q, e, d, res1, res2, res3, and one.

Compute the following values:

- res1 = p - 1
- res2 = q - 1
- res3 = res1 * res2

Finally:

Calculate 'd' by finding the modular inverse of 'e' with respect to 'res3', ensuring that d * e mod res3 = 1.

This code will provide the value of 'd', which is the private key.

d= 3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB

Above mentioned is the D value which we need

Task - 1 Code :

Open

*task1.c
~/Desktop/Seedlab/lab8-RSA

```c
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256
void printBN(char *msg, BIGNUM *a){
// Convert the BIGNUM to number string lol #Arvind's
char * number_str = BN_bn2hex(a);
// Print out the number string
printf("%s %s\n", msg, number_str);
// Free the dynamically allocated memory
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();
// here we are goint to initalize p q e lol other wise we will not get output
BN_hex2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF"); // Assign the first large prime
BN_hex2bn(&q, "E85CED54AF57E53E092113E62F436F4F"); // Assign the second large prime
BN_hex2bn(&e, "0D88C3"); // Assign the Modulus
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); //res=a*b mod n
BN_mod_inverse(d, e, res3, ctx);
//print BN
printBN("d= ",d);
return 0;
}
// completed task 1 code
```

Task 2: Encrypting a Message

The public keys are listed in the followings (hexadecimal). We also provide the private key `d` to help ou verify your encryption result.

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

Code explanation :

We are importing libraries

```
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256
```

To print the result

```
//To print a big number
void printBN(char *msg, BIGNUM *a){
// Convert the BIGNUM to number string
char * number_str = BN_bn2hex(a);
// Print out the number string
printf("%s %s\n", msg, number_str);
// Free the dynamically allocated memory
OPENSSL_free(number_str);
}
```

To initialize values and encrypt the message

```
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
    BN_hex2bn(&e,"010001"); //values given in the pdf description
    BN_hex2bn(&n,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&m,"4120746f702073656372657421");//A top secret! code
    BN_hex2bn(&d,"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    //encry = m^e mod n
    BN_mod_exp(enc,m,e,n,ctx);
    printBN("encrypt message = ", enc);
```

To decrypt the message.

```
    //decry = enc^d mod n
    BN_mod_exp(dec,enc,d,n,ctx);
    printBN("The decrypt message = ",dec);
    return 0;
}
//boom! task2 completed successfully.
```

Full code :

*Task2.c
~/Desktop/Seedlab/lab8-RSA

```c
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256
//To print a big number
void printBN(char *msg, BIGNUM *a){
// Convert the BIGNUM to number string
char * number_str = BN_bn2hex(a);
// Print out the number string
printf("%s %s\n", msg, number_str);
// Free the dynamically allocated memory
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
    BN_hex2bn(&e,"010001"); //values given in the pdf description
    BN_hex2bn(&n,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&m,"4120746f702073656372657421");//A top secret! code
    BN_hex2bn(&d,"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    //encry = m^e mod n
    BN_mod_exp(enc,m,e,n,ctx);
    printBN("encrypt message = ", enc);
    //decry = enc^d mod n
    BN_mod_exp(dec,enc,d,n,ctx);
    printBN("The decrypt message = ",dec);
    return 0;
}
//boom! task2 completed successfully.

```

```
[10/14/23]seed@VM:~/.../lab8-RSA$ gedit Task2.c
^Z
[1]+  Stopped                 gedit Task2.c
[10/14/23]seed@VM:~/.../lab8-RSA$ gcc Task2.c -o Task2 -l crypto
[10/14/23]seed@VM:~/.../lab8-RSA$ ./Task2
encrypt message =  6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC
The decrypt message =  4120746F702073656372657421
[10/14/23]seed@VM:~/.../lab8-RSA$
```

In the above snippet we can see the encrypt and decrypt message

—--- **Task 2 done** ---

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!
```

The public and private keys employed in this task are identical to those utilized in Task 2. Our objective is to decrypt the provided ciphertext, C, and subsequently convert it back into a plain ASCII string. We achieve this decryption by applying the formula c^d mod n.

```
[10/14/23]seed@VM:~/.../lab8-RSA$ echo "Task3"
Task3
[10/14/23]seed@VM:~/.../lab8-RSA$ gcc task3.c -o task3 -l crypto
[10/14/23]seed@VM:~/.../lab8-RSA$ ./task3
The encrypt message =  50617373776F72642069732064656573
[10/14/23]seed@VM:~/.../lab8-RSA$ python -c 'print("A top secret!".decode("hex"))'
bash: syntax error near unexpected token `('
[10/14/23]seed@VM:~/.../lab8-RSA$ python -c 'print("A top secret!".decode("hex")) '
bash: syntax error near unexpected token `('
[10/14/23]seed@VM:~/.../lab8-RSA$ python -c ' print("A top secret!".decode("hex")) '
bash: syntax error near unexpected token `('
[10/14/23]seed@VM:~/.../lab8-RSA$ python -c 'print("4120746f702073656372657421".decode("hex"))'

Command 'python' not found, did you mean:

  command 'python3' from deb python3
  command 'python' from deb python-is-python3

[10/14/23]seed@VM:~/.../lab8-RSA$ python3 -c 'print("4120746f702073656372657421".decode("hex"))'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
AttributeError: 'str' object has no attribute 'decode'
[10/14/23]seed@VM:~/.../lab8-RSA$ python3 -c 'print(bytes.fromhex("4120746f702073656372657421").decode("utf-8"))'
A top secret!
[10/14/23]seed@VM:~/.../lab8-RSA$
```

Initially, we named the file as **"task-3."** We proceeded to compile the C code, which is explained below. **Following the compilation process, we successfully obtained the desired values without encountering any errors**

Encrypted message also decrypted and below i have attached the screenshot

```
[10/14/23]seed@VM:~/.../lab8-RSA$ python3 -c 'print(bytes.fromhex("50617373776F7
2642069732064656573").decode("utf-8"))'
Password is dees
```

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!
```

The next step involves using a Python command to convert a hex string back into a plain ASCII string. **However, it's important to note that the initial command provided may work only on older Linux versions**, and there were some issues encountered as shown in the previous screenshots. To resolve this, we made some modifications to the Python command, which you can observe in the error messages displayed in the above screenshots. After compiling the code that was written, it became evident that we can successfully obtain the plain ASCII string.

Task 3 - Code :

task3.c
~/Desktop/Seedlab/lab8-RSA

```c
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256

//print a big number
void printBN(char *msg, BIGNUM *a){
// Convert the BIGNUM to number string
char * number_str = BN_bn2hex(a);
// Print out the number string
printf("%s %s\n", msg, number_str);
// Free the dynamically allocated memory
OPENSSL_free(number_str);
}

int main(){
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *n = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *c = BN_new();
    BIGNUM *dec = BN_new();
    //Initialize
    BN_hex2bn(&n,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&c,"8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");
    BN_hex2bn(&d,"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    //encry = m^e mod n
    BN_mod_exp(dec,c,d,n,ctx);
    printBN("The encrypt message = ", dec);

    return 0;
}
```

Upon decryption, we obtain the hexadecimal representation of the message. **Subsequently, we utilize Python to decode this hex value, thereby restoring the original plain ASCII string.**

—-- **Task 3 completed** --------------------------------

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.

First, we should get the hex value of " I owe you $2000."

```
[10/14/23]seed@VM:~/.../lab8-RSA$ gedit task4.c
^Z
[1]+  Stopped                 gedit task4.c
[10/14/23]seed@VM:~/.../lab8-RSA$ python3 -c 'print(("I OWE YOU $2000").encode("hex"))'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
LookupError: 'hex' is not a text encoding; use codecs.encode() to handle arbitrary codecs
[10/14/23]seed@VM:~/.../lab8-RSA$ python3 -c 'print(("I OWE YOU $2000").encode("utf-8").hex())'
49204f574520594f55202432303030
```

We can watch the value of i owe you in the above snap snippet .

The given python command was not working in this version so I modified and runned the new command as marked in the above screenshot.

We run our code to produce the **signature for the message for $2000**

```
[10/14/23]seed@VM:~/.../lab8-RSA$ gedit task4.c
[10/14/23]seed@VM:~/.../lab8-RSA$ gcc task4.c -o task4 -l crypto
[10/14/23]seed@VM:~/.../lab8-RSA$ ./task4
encrypt message =  55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73CCB35E4CB
[10/14/23]seed@VM:~/.../lab8-RSA$
```

Second step is that we should get the hex value of "I owe you $3000." since mentioned in description.

```
[10/14/23]seed@VM:~/.../lab8-RSA$ echo "tsk4 after changing hexvalue"
tsk4 after changing hexvalue
[10/14/23]seed@VM:~/.../lab8-RSA$ python3 -c 'print(("I OWE YOU $3000").encode("utf-8").hex())'
49204f574520594f55202433303030
```

We run our code to produce the **signature for the message for $3000**:

```
[10/14/23]seed@VM:~/.../lab8-RSA$ gcc task4.c -o task4 -l crypto
[10/14/23]seed@VM:~/.../lab8-RSA$ ./task4
encrypt message =  7E3AE53979186F5CDB0BD2DD0385F5069AD564580C3F31C8165F916DF441F4B2
[10/14/23]seed@VM:~/.../lab8-RSA$
```

Code :

Open *task4.c ~/Desktop/Seedlab/lab8-RSA

```c
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256

//print a big number
void printBN(char *msg, BIGNUM *a){
// Convert the BIGNUM to number string
char * number_str = BN_bn2hex(a);
// Print out the number string
printf("%s %s\n", msg, number_str);
// Free the dynamically allocated memory
OPENSSL_free(number_str);
}

int main(){
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *n = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *c = BN_new();
    BIGNUM *dec = BN_new();
    //Initialize
    BN_hex2bn(&n,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&c,"49204f574520594f55202432303030");// HEX value of "I owe you $2000."
  //BN_hex2bn(&c,"49204f574520594f55202433303030");// HEX value of "I owe you $3000."
    BN_hex2bn(&d,"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    //encry = m^e mod n
    BN_mod_exp(dec,c,d,n,ctx);
    printBN("encrypt message = ", dec);

    return 0;
}
```

The above is for both $2000 and $3000 when we run for $2000 we will comment $3000 and while running $3000 we comment $2000.

Observation : It is noticeable that despite a minimal one-byte difference in the messages, their respective signatures are entirely distinct.

—-- **Task 4 completed** —--------------------------------

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.

To begin, **we obtain the hexadecimal representation of the message "Launch a missile."** using Python.

```
[10/14/23]seed@VM:~/.../lab8-RSA$ echo "task5"
task5
[10/14/23]seed@VM:~/.../lab8-RSA$ python3 -c 'print(("Launch a missile").encode("utf-8").hex())'
4c61756e63682061206d697373696c65
[10/14/23]seed@VM:~/.../lab8-RSA$
```

We utilize the signature to calculate the value of the **message C**. Subsequently, we employ the BN_cmp API to compare the two messages and determine if the signature belongs to Alice.

```
[10/14/23]seed@VM:~/.../lab8-RSA$ gcc task5.c -o task5 -l crypto
task5.c: In function ‘main’:
task5.c:25:13: error: expected ‘)’ before string constant
   25 | BN_hex2bn(&s "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");
      |              ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      |              )
task5.c:25:1: error: too few arguments to function ‘BN_hex2bn’
   25 | BN_hex2bn(&s "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");
      | ^~~~~~~~~
In file included from task5.c:2:
/usr/local/include/openssl/bn.h:311:5: note: declared here
  311 | int BN_hex2bn(BIGNUM **a, const char *str);
      |     ^~~~~~~~~
task5.c:29:12: error: ‘M’ undeclared (first use in this function)
   29 | BN_mod_exp(M,S,e,n, ctx);
      |            ^
task5.c:29:12: note: each undeclared identifier is reported only once for each function it appears in
task5.c:29:14: error: ‘S’ undeclared (first use in this function)
   29 | BN_mod_exp(M,S,e,n, ctx);
      |              ^
task5.c:31:42: error: expected ‘;’ before ‘BN_free’
   31 | printBN("Verification message (M) = ", M)
      |                                          ^
      |                                          ;
......
   34 | BN_free(n);
      |
[10/14/23]seed@VM:~/.../lab8-RSA$ gcc task5.c -o task5 -l crypto
[10/14/23]seed@VM:~/.../lab8-RSA$ ./task5
Verification message (M) =  4C61756E63682061206D697373696C652E
```

After compiling the code we get the verification message.

Code of both 2F and 3F :

```
*task5.c
~/Desktop/Seedlab/lab8-RSA
#include <stdio.h>
#include <openssl/bn.h>
void printBN(char *msg, BIGNUM *a){
    // Convert the BIGNUM to a number string
    char *number_str = BN_bn2hex(a);
    // Print out the number string
    printf("%s %s\n", msg, number_str);
    // Free the dynamically allocated memory
    OPENSSL_free(number_str);
}
int main(){
    BN_CTX *ctx = BN_CTX_new();

    BIGNUM *n = BN_new();
    BIGNUM *s = BN_new();
    BIGNUM *m = BN_new();
    BIGNUM *e = BN_new();
    // Initialize n, s, and e
    BN_hex2bn(&n, "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");
    BN_hex2bn(&s, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F");
    BN_hex2bn(&e, "010001");
    // Perform the modular exponentiation
    BN_mod_exp(m, s, e, n, ctx);
    // Print the result
    printBN("Verification message (M) = ", m);
    // Free allocated memory
    BN_free(n);
    BN_free(s);
    BN_free(e);
    BN_free(m);
    BN_CTX_free(ctx);
    return 0;
```

At last after in the green after we changed into 3F

Changed 2F TO 3F

```
[10/14/23]seed@VM:~/.../lab8-RSA$ echo"changed 2F to 3F"
echochanged 2F to 3F: command not found
[10/14/23]seed@VM:~/.../lab8-RSA$ gcc task5.c -o task5 -l crypto
[10/14/23]seed@VM:~/.../lab8-RSA$ ./task5
Verification message (M) =  91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294
[10/14/23]seed@VM:~/.../lab8-RSA$
```

We can see it was successfully running and we see the verification message.
If we alter the last byte of the signature from 2F to 3F, the signature becomes:
S = 91471927C80DF _______ so on in the screen shot above. Hehe.

Observation : When we compute the value of the message C using this modified signature and then compare it with the original message, we find that the computed message is entirely different from the original message. This minor change in the signature causes the verification to fail.

—-------------------------------- task 5 completed —-------------------------

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.

```
[10/14/23]seed@VM:~/.../lab8-RSA$ openssl s_client -connect www.iit.edu.org:443 -showcerts
CONNECTED(00000003)
depth=2 C = US, ST = Arizona, L = Scottsdale, O = "GoDaddy.com, Inc.", CN = Go Daddy Root Certificate Authority - G2
verify error:num=19:self signed certificate in certificate chain
verify return:1
depth=2 C = US, ST = Arizona, L = Scottsdale, O = "GoDaddy.com, Inc.", CN = Go Daddy Root Certificate Authority - G2
verify return:1
depth=1 C = US, ST = Arizona, L = Scottsdale, O = "GoDaddy.com, Inc.", OU = http://certs.godaddy.com/repository/, CN = Go Daddy Secure Certificate Authority - G2
verify return:1
depth=0 CN = dan.com
verify return:1
---
Certificate chain
 0 s:CN = dan.com
   i:C = US, ST = Arizona, L = Scottsdale, O = "GoDaddy.com, Inc.", OU = http://certs.godaddy.com/repository/, CN = Go Daddy Secure Certificate Authority - G2
-----BEGIN CERTIFICATE-----
MIIHSzCCBjOgAwIBAgIIDWwD3F2pqZMwDQYJKoZIhvcNAQELBQAwgbQxCzAJBgNV
BAYTAlVTMRAwDgYDVQQIEwdBcml6b25hMRMwEQYDVQQHEwpTY290dHNkYWxlMRow
GAYDVQQKExFHb0RhZGR5LmNvbSwgSW5jLjEtMCsGA1UECxMkaHR0cDovL2NlcnRz
LmdvZGFkZHkuY29tL3JlcG9zaXRvcnkvMTMwMQYDVQQDEypHbyBEYWRkeSBTZWN1
cmUgQ2VydGlmaWNhdGUgQXV0aG9yaXR5IC0gRzIwHhcNMjIxMjIxMjA0MDIwWhcN
MjQwMTIyMjA0MDIwWjASMRAwDgYDVQQDEwdkYW4uY29tMIIBIjANBgkqhkiG9w0B
AQEFAAOCAQ8AMIIBCgKCAQEAwmNgEsS0de7uYDRWOMpZ0Fs8YYA4ftsv4dzxXHR9
QJ1QYjKfTMuD8VHLvSYjG67De307zz0qyoFt61jJD4CvZ5xoLRh7EKmdp3zNbG/w
WCMHP+PkH6TBa3MnUiu/d4tXbN6hGKwQ+wPJC5WiGDw+zOeZSvhhx/Icrh3KzhMI
mAY85Fe7deYCh/cih48MbI6KlJkPS6Ja/bkaK0hzQ5hTlKBz8Y3KNZ26u3q/h+Xa
04V5dpTPI7lJJP51CJsp8O9dHOu8OPJC0TTC7Q1fbkHxGjugzzVt7RyxLobSf5SG
ZuqlL8c7TyIaak0j1RbcQtBTOfJRXA10A3BC+tH32aF7QwIDAQABo4IEADCCA/ww
DAYDVR0TAQH/BAIwADAdBgNVHSUEFjAUBggrBgEFBQcDAQYIKwYBBQUHAwIwDgYD
VR0PAQH/BAQDAgWgMDgGA1UdHwQxMC8wLaAroCmGJ2h0dHA6Ly9jcmwuZ29kYWRk
eS5jb20vZ2RpZzJzMS000TUyLmNybDBdBgNVHSAEVjBUMEgGC2CGSAGG/W0BBxcB
MDkwNwYIKwYBBQUHAgEWK2h0dHA6Ly9jZXJ0aWZpY2F0ZXMuZ29kYWRkeS5jb20v
cmVwb3NpdG9yeS8wCAYGZ4EMAQIBMHYGCCsGAQUFBwEBBGowaDAkBggrBgEFBQcw
AYYYaHR0cDovL29jc3AuZ29kYWRkeS5jb20vMEAGCCsGAQUFBzAChjRodHRwOi8v
Y2VydGlmaWNhdGVzLmdvZGFkZHkuY29tL3JlcG9zaXRvcnkvZ2RpZzIuY3J0MB8G
A1UdIwQYMBaAFEDCvSeOzDSDMKIz1/tss/C0LIDOMIHqBgNVHREEgeIwgd+CD3Vu
ZGV2ZWxvcGVkLmNvbYIOdW5kZXZlbG9wZWQuZXOCDnVuZGV2ZWxvcGVkLmNugg5l
bmRldmVsb3BlZC51a4IOdW5kZXZlbG9wZWQuZXWCDnVuZGV2ZWxvcGVkLnVzgg5l
bmRldmVsb3BlZC5ubIIOdW5kZXZlbG9wZWQuYmWCEXVuZGV2ZWxvcGVkLmNvLnVr
ghN1bmRldmVsb3BlZC5kb21haW5zggdkYW4uY29tggt3d3cuZGFuLmNvbYIOdW5k
ZXZlbG9wZWQuZnKCDnVuZGV2ZWxvcGVkLm51MB0GA1UdDgQWBBQPJd+3++LwLMFp
XGMZRCb5VhhzkjCCAX0GCisGAQQB1nkCBAIEggFtBIIBaQFnAHYA7s3QZNXbGs7F
XLedtM0TojKHRny87N7DUUhZRnEftZsAAAGFNmpnoQAABAMARzBFAiBCPr/UZ7f0
pyOtMYQwkb+UX1DEjuKnyLxD/YtNXiDpSAIhAL+saDN0eyzHZvT01JN29zMr60bI
+el9laFR63ztJf9SAHUASLDja9qmRzQP5WoC+p0w6xxSActW3SyB2bu/qznYhHMA
AAGFNmpokgAABAMARjBEAiBzBTHrMy8koVkc5wAeP6iZuAF4VjazneLcadk58AxN
EAIgKM2xVnnnRqqDl5Ali0FKUlVypf13zXGgRq6aZTfdi8EAdgDatr9rP7W2Ip+b
wrtca+hwkXFsu1GEhTS9pD0wSNf7qwAAAYU2amkVAAAEAwBHMEUCIQDvwFxNSkRG
G6WhoNc8hIamBGEJOXUua5IR/oM/Yn/yfAIgPb/AFXcGqzPusuc7yNre8AxrY1Jp
H93AetSucymjaBYwDQYJKoZIhvcNAQELBQADggEBAC+jR6VVrWbdqGJb3aB91nrd
4cB1+7ZAOOEllycx70eDgP4rJ8KKToAszepWDnJYLJsX4/sGeTbPbw67BTQLXHID
KkUNCbvuwmsTFUJ05lkfNCGbfPIuf7UJ60ceSobD3HIP1WjBVXUrQK+bXQKcMMqA
Rlv9sFWgbtSq3IMP0LB7HfHBC5IYKz/wm+T7zxayA30s3MiGy+md3Ir7XIjFN8G5
DPC6te8bA3J+HlGG/JNfAJ47VUuGqk9QUzg5vkcmNEs828n0y9EkCUszWRWH00R3
2r7IPedHellGJ5m1qJlDBYDrXB3dtl0EobsUyhF1/swIZdKg3p0U+L9UR7rYato=
-----END CERTIFICATE-----
 1 s:C = US, ST = Arizona, L = Scottsdale, O = "GoDaddy.com, Inc.", OU = http://certs.godaddy.com/repository/, CN = Go Daddy Secure Certificate Authority - G2
```

Here I took the website www.iit.edu and downloaded the certificate as explained in the description.

```
-----BEGIN CERTIFICATE-----
MIIE0DCCA7igAwIBAgIBBzANBgkqhkiG9w0BAQsFADCBgzELMAkGA1UEBhMCVVMx
EDAOBgNVBAgTB0FyaXpvbmExEzARBgNVBAcTClNjb3R0c2RhbGUxGjAYBgNVBAoT
EUdvRGFkZHkuY29tLCBJbmMuMTEwLwYDVQQDEyhHbyBEYWRkeSBSb290IENlcnRp
ZmljYXRlIEF1dGhvcml0eSAtIEcyMB4XDTExMDUwMzA3MDAwMFoXDTMxMDUwMzA3
MDAwMFowgbQxCzAJBgNVBAYTAlVTMRAwDgYDVQQIEwdBcml6b25hMRMwEQYDVQQH
EwpTY290dHNkYWxlMRowGAYDVQQKExFHb0RhZGR5LmNvbSwgSW5jLjEtMCsGA1UE
CxMkaHR0cDovL2NlcnRzLmdvZGFkZHkuY29tL3JlcG9zaXRvcnkvMTMwMQYDVQQD
EypHbyBEYWRkeSBTZWN1cmUgQ2VydGlmaWNhdGUgQXV0aG9yaXR5IC0gRzIwggEi
MA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQC54MsQ1K92vdSTYuswZLiBCGzD
BNliF44v/z5lz4/OYuY8UhzaFkVLVat4a2ODYpDOD2lsmcgaFItMzEUz6ojcnqOv
K/6AYZ15V8TPLvQ/MDxdR/yaFrzDN5ZBUY4RS1T4KL7QjL7wMDge87Am+GZHY23e
cSZHjzhHU9FGHbTj3ADqRay9vHHZqm8A29vNMDp5T19MR/gd71vCxJ1gO7GyQ5HY
pDNO6rPWJ0+tJYqlxvTV0KaudAVkV4i1RFXULSo6Pvi4vekyCgKUZMQWOlDxSq7n
eTOvDCAHf+jfBDnCaQJsY1L6d8EbyHSHyLmTGFBUNUtpTrw700kuH9zB0lL7AgMB
AAGjggEaMIIBFjAPBgNVHRMBAf8EBTADAQH/MA4GA1UdDwEB/wQEAwIBBjAdBgNV
HQ4EFgQUQMK9J47MNIMwojPX+2yz8LQsgM4wHwYDVR0jBBgwFoAUOpqFBxBnKLbv
9r0FQW4gwZTaD94wNAYIKwYBBQUHAQEEKDAmMCQGCCsGAQUFBzABhhhodHRwOi8v
b2NzcC5nb2RhZGR5LmNvbS8wNQYDVR0fBC4wLDAqoCigJoYkaHR0cDovL2NybC5n
b2RhZGR5LmNvbS9nZHJvb3QtZzIuY3JsMEYGA1UdIAQ/MD0wOwYEVR0gADAzMDEG
CCsGAQUFBwIBFiVodHRwczovL2NlcnRzLmdvZGFkZHkuY29tL3JlcG9zaXRvcnkv
MA0GCSqGSIb3DQEBCwUAA4IBAQAIfmyTEMg4uJapkEv/oV9PBO9sPpyIBslQj6Zz
91cxG7685C/b+LrTW+C05+Z5Yg4MotdqY3MxtfWoSKQ7CC2iXZDXtHwlTxFWMMS2
RJ17LJ3lXubvDGGqv+QqG+6EnriDfcFDzkSnE3ANkR/0yBOtg2DZ2HKocyQetawi
DsoXiWJYRBuriSUBAA/NxBti21G00w9RKpv0vHP8ds42pM3Z2Czqrpv1KrKQ0U11
GIo/ikGQI31bS/6kA1ibRrLDYGCD+H1QQc7CoZDDu+8CL9IVVO5EFdkKrqeKM+2x
LXY2JtwE65/3YR8V3Idv7kaWKK2hJn0KCacuBKONvPi8BDAB
-----END CERTIFICATE-----
```


Copy and paste each of the certificates (the text between the line containing "Begin CERTIFICATE" and the line containing "END CERTIFICATE", including these two lines) to a file. Saved it as first one c0.pem and the second one c1.pem.

Step 2: Extract the public key (e, n) from the issuer's 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, find the exponent (e):
$ openssl x509 -in c1.pem -text -noout
```

In the screenshot provided below, I've attached the results of two commands. These commands are essential for obtaining the value of 'e,' which will be used later in the process.

```
[10/14/23]seed@VM:~/.../lab8-RSA$ openssl x509 -in c1.pem -noout -modulus
Modulus=C14BB3654770BCDD4F58DBEC9CEDC366E51F311354AD4A66461F2C0AEC6407E52EDCDCB90A20EDDFE3C4D09E9AA97A1D8288E51156DB1E9F58C251E72C340D2ED292E156CBF1795FB3BB87CA25037B9A52416610604F571349F0E83767 83DFE7D34B674C2251A6DF0E9910ED57517426E27DC7CA622E131B7F238825536FC13458008B84FFF8BEA75849227B96ADA2889B15BCA
BF71427D58840380B18101FAF9CA32BBB48E278727C52B74D4A8D697DEC364F9CACE53A256BC78178E490329AEFB494FA415B9CEF25C19576D6B79A72BA2272013B5D03D40D321300793EA99F5
[10/14/23]seed@VM:~/.../lab8-RSA$ openssl x509 -in c1.pem -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            0a:35:08:d5:5c:29:2b:01:7d:f8:ad:65:c0:0f:f7:e4
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, O = DigiCert Inc, OU = www.digicert.com, CN = DigiCert Global Root CA
        Validity
            Not Before: Sep 24 00:00:00 2020 GMT
            Not After : Sep 23 23:59:59 2030 GMT
        Subject: C = US, O = DigiCert Inc, CN = DigiCert TLS RSA SHA256 2020 CA1
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (2048 bit)
                Modulus:
                    00:c1:4b:b3:65:47:70:bc:dd:4f:58:db:ec:9c:ed:
                    c3:66:e5:1f:31:13:54:ad:4a:66:46:1f:2c:0a:ec:
                    64:07:e5:2e:dc:dc:b9:0a:20:ed:df:e3:c4:d0:9e:
                    9a:a9:7a:1d:82:88:e5:11:56:db:1e:9f:58:c2:51:
                    e7:2c:34:0d:2e:d2:92:e1:56:cb:f1:79:5f:b3:bb:
                    87:ca:25:03:7b:9a:52:41:66:10:60:4f:57:13:49:
                    f0:e8:37:67:83:df:e7:d3:4b:67:4c:22:51:a6:df:
                    0e:99:10:ed:57:51:74:26:e2:7d:c7:ca:62:2e:13:
                    1b:7f:23:88:25:53:6f:c1:34:58:00:8b:84:ff:f8:
                    be:a7:58:49:22:7b:96:ad:a2:88:9b:15:bc:a0:7c:
                    df:e9:51:a8:d5:b0:ed:37:e2:36:b4:82:4b:62:b5:
                    49:9a:ec:c7:67:d6:e3:3e:f5:e3:d6:12:5e:44:f1:
                    bf:71:42:7d:58:84:03:80:b1:81:01:fa:f9:ca:32:
                    bb:b4:8e:27:87:27:c5:2b:74:d4:a8:d6:97:de:c3:
                    64:f9:ca:ce:53:a2:56:bc:78:17:8e:49:03:29:ae:
                    fb:49:4f:a4:15:b9:ce:f2:5c:19:57:6d:6b:79:a7:
                    2b:a2:27:20:13:b5:d0:3d:40:d3:21:30:07:93:ea:
                    99:f5
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                B7:6B:A2:EA:A8:AA:84:8C:79:EA:B4:DA:0F:98:B2:C5:95:76:B9:F4
            X509v3 Authority Key Identifier:
                keyid:03:DE:50:35:56:D1:4C:BB:66:F0:A3:E2:1B:1B:C3:97:B2:3D:D1:55

            X509v3 Key Usage: critical
                Digital Signature, Certificate Sign, CRL Sign
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client Authentication
            X509v3 Basic Constraints: critical
                CA:TRUE, pathlen:0
            Authority Information Access:
                OCSP - URI:http://ocsp.digicert.com
                CA Issuers - URI:http://cacerts.digicert.com/DigiCertGlobalRootCA.crt

            X509v3 CRL Distribution Points:

                Full Name:
                  URI:http://crl3.digicert.com/DigiCertGlobalRootCA.crl

                Full Name:
                  URI:http://crl4.digicert.com/DigiCertGlobalRootCA.crl

            X509v3 Certificate Policies:
                Policy: 2.23.140.1.1
                Policy: 2.23.140.1.2.1
                Policy: 2.23.140.1.2.2
                Policy: 2.23.140.1.2.3

    Signature Algorithm: sha256WithRSAEncryption
         77:ab:b7:7a:27:3d:ae:bb:f6:7f:e0:5a:56:c9:84:aa:ca:5b:
         71:17:dd:22:47:fc:4e:9f:ee:d0:c1:a4:04:e1:a3:eb:c5:49:
         c1:fd:d1:c9:df:8c:af:94:45:2c:46:2a:a3:63:39:20:f9:9e:
         4a:24:94:41:c8:a9:d9:e2:9c:54:05:06:cb:5c:1c:be:00:1b:
         0f:a8:5a:ff:19:bb:65:c7:16:af:21:56:dd:61:05:c9:e9:8f:
         98:76:df:6b:1b:d0:72:0c:50:b9:30:29:7a:bf:60:59:10:66:
         13:3a:2d:ac:15:11:6c:2d:23:0c:02:3e:05:3b:fe:e5:a1:9c:
         e2:8a:db:87:d7:4a:e8:5e:e7:48:06:eb:ab:12:9a:f2:af:84:
         c3:5b:83:4a:99:81:83:ab:00:a1:ca:0a:3c:4c:a2:25:89:2a:
```

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

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
```

```
[10/14/23]seed@VM:~/.../lab8-RSA$ openssl x509 -in c0.pem -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            07:2c:f5:0a:82:b6:25:c8:f6:73:91:93:8d:c2:eb:5a
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, O = DigiCert Inc, CN = DigiCert TLS RSA SHA256 2020 CA1
        Validity
            Not Before: Jan  5 00:00:00 2023 GMT
            Not After : Jan 24 23:59:59 2024 GMT
        Subject: C = US, ST = Illinois, L = Chicago, O = Illinois Institute of Technology, CN = *.iit.edu
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (2048 bit)
                Modulus:
                    00:98:80:47:34:55:e1:b4:92:9c:2b:55:09:06:25:
                    d2:69:96:37:82:18:5d:e4:61:25:3c:40:a2:b0:e5:
                    dd:0a:39:d0:c3:1b:6a:53:ef:b9:77:1f:d9:f4:87:
                    b3:fd:cd:d1:d7:02:25:73:78:7d:07:04:c4:b9:63:
                    2e:f1:5f:04:ad:97:a5:0d:f1:2b:6a:c6:bd:55:c5:
                    3c:20:5b:20:e1:b3:98:00:a5:bd:e8:0a:53:d7:e7:
                    05:e4:d3:40:08:22:a7:d8:d0:71:f1:f6:6b:b5:6e:
                    8c:a8:7a:e9:ff:47:ab:cb:1c:59:4a:dc:b8:ac:27:
                    2c:58:88:ca:6b:ae:a4:78:47:72:bb:ba:16:d6:6b:
                    fa:2f:87:bc:50:30:34:e6:6c:76:9a:00:e3:94:e8:
                    8a:5d:c8:9c:3e:4f:55:5a:74:d9:9e:ac:01:b4:1d:
                    53:e2:cc:18:8c:bb:37:0e:7e:20:e4:c3:fb:e1:33:
                    2b:71:df:68:62:8d:52:33:44:06:aa:c0:93:af:21:
                    9c:c3:c0:31:f0:15:2e:ae:e6:8d:4b:85:c6:df:03:
                    65:28:c1:cb:4b:97:54:c3:65:82:e7:94:b9:53:13:
                    09:27:46:c6:3f:e9:22:52:6f:71:b2:b6:06:57:2a:
                    33:04:e2:bb:fa:87:f9:81:fb:37:53:60:5d:be:34:
                    f5:bd
                Exponent: 65537 (0x10001)

                              B7:C4:82:6F:0C:3A:4B
 Signature Algorithm: sha256WithRSAEncryption
      11:39:f4:0b:3b:91:b8:25:81:66:7c:ee:b9:da:33:fc:cf:c3:
      24:e1:e2:09:4d:12:66:66:ce:8b:93:c6:eb:42:3e:ab:7e:0d:
      96:eb:0a:a6:46:2c:c9:85:81:e7:f5:02:dc:2b:60:72:b8:f7:
      2d:5c:08:71:12:43:f0:f9:8d:9d:52:d0:67:95:4d:cd:a2:20:
      53:29:c7:95:4b:4d:5c:5e:9f:ec:1f:65:b7:06:c5:1c:5b:ff:
      9a:69:3f:0e:d1:6a:9b:c9:99:dd:7b:fb:2d:df:42:75:35:c2:
      1c:69:4a:97:84:40:a1:c6:4d:a6:d5:ac:c5:e3:a7:75:8c:44:
      12:0e:45:2a:e8:ac:3f:2b:4d:3e:76:39:10:4b:1f:47:90:5f:
      ba:f4:ea:44:b6:c4:60:53:7e:48:4a:20:9e:b8:38:70:14:5d:
      4d:87:07:02:cd:d8:21:6d:09:18:00:16:d1:a9:e0:e0:61:66:
      d8:7e:da:0a:33:e0:9e:56:f1:a3:f2:14:5d:37:22:87:05:38:
      4b:f7:8a:47:75:c3:9c:05:8a:7c:79:77:b4:a9:ca:a5:1c:d2:
      5b:10:03:3c:1a:68:8a:2c:8e:39:c8:56:a5:51:f3:89:80:51:
      fa:db:dc:08:50:65:dc:e9:25:e6:45:e3:5c:a5:26:a3:0a:e2:
      70:de:1a:22
```

From the step 3 we took the values and variables of signature block into the file

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
```

Below is the screenshot of the above executed command.

```
[10/14/23]seed@VM:~/.../lab8-RSA$ cat signature | tr -d '[:space:]:'
1139f40b3b91b82581667ceeb9da33fccfc324e1e2094d126666ce8b93c6eb423eab7e0d96eb0aa6462cc98581e7f502dc2b6072b8f72d5c08711243f0f98d9d52d067954dcda2205329c7954b4d5c5e9fec1f65b706c51c5bff9a693f0ed16a9bc999dd7bfb2ddf427535c21c694a978440
a1c64da6d5acc5e3a7758c44120e452ae8ac3f2b4d3e7639104b1f47905fbaf4ea44b6c460537e484a209eb83870145d4d870702cdd8216d09180016d1a9e0e06166d87eda0a33e09e56f1a3f2145d37228705384bf78a4775c39c058a7c7977b4a9caa51cd25b10033c1a688a2c8e39c856
a551f3898051fadbdc085065dce925e645e35ca526a30ae270de1a22[10/14/23]seed@VM:~/.../lab8-RSA$
```

Step 4: Extract the body of the server's 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
```

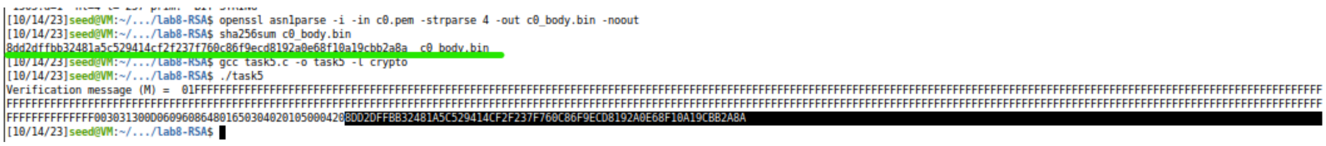
The field starting from ❶ is the body of the certificate that is used to generate the hash; the field starting

```
[10/14/23]seed@VM:~/.../lab8-RSA$ openssl asn1parse -i -in c0.pem
    0:d=0  hl=4 l=1762 cons: SEQUENCE
    4:d=1  hl=4 l=1482 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           :072CF50A82B625C8F67391938DC2EB5A
   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
   88:d=5  hl=2 l=   3 prim:      OBJECT            :commonName
   93:d=5  hl=2 l=  32 prim:      PRINTABLESTRING   :DigiCert TLS RSA SHA256 2020 CA1
  127:d=2  hl=2 l=  30 cons:   SEQUENCE
  129:d=3  hl=2 l=  13 prim:    UTCTIME           :230105000000Z
  144:d=3  hl=2 l=  13 prim:    UTCTIME           :240124235959Z
  159:d=2  hl=2 l= 113 cons:   SEQUENCE
  161:d=3  hl=2 l=  11 cons:    SET
  163:d=4  hl=2 l=   9 cons:     SEQUENCE
  165:d=5  hl=2 l=   3 prim:      OBJECT            :countryName
  170:d=5  hl=2 l=   2 prim:      PRINTABLESTRING   :US
```

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

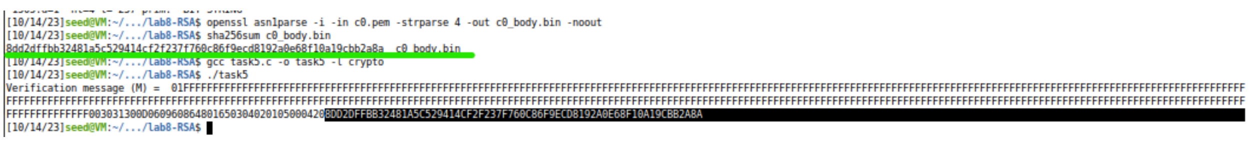
Once we get the body of the certificate, we can calculate its hash using the following command:

```
$ sha256sum c0_body.bin
```



In the above screen shot after using the command we can see the sha256 sum

Step 5 : Signature verification



We can verify the correctness of our Task 5 code (modified code) by comparing the verification message marked in green with the final message marked in dark. If these messages match, it indicates that we have successfully completed the lab.

It's worth noting that the original message and the hash value of the computed message are identical.

Signature verification done .

Below I have attached the code snippet of the task ….

Task 6 Code :

```
#include <stdio.h>
#include <openssl/bn.h>
void printBN(char *msg, BIGNUM *a){
    // Convert the BIGNUM to a number string
    char *number_str = BN_bn2hex(a);
    // Print out the number string
    printf("%s %s\n", msg, number_str);
    // Free the dynamically allocated memory
    OPENSSL_free(number_str);
}
int main(){
    BN_CTX *ctx = BN_CTX_new();

    BIGNUM *n = BN_new();
    BIGNUM *s = BN_new();
    BIGNUM *m = BN_new();
    BIGNUM *e = BN_new();
    // Initialize n, s, and e

BN_hex2bn(&n,"C14BB3654770BCDD4F58DBEC9CEDC366E51F311354AD4A66461F2C0AEC6407E52EDCDCB90A20EDDFE3C4D09E9AA97A1D8288E51156DB1E9F58C251E72C340

BN_hex2bn(&s,"1139f40b3b91b82581667ceeb9da33fccfc324e1e2094d126666ce8b93c6eb423eab7e0d96eb0aa6462cc98581e7f502dc2b6072b8f72d5c08711243f0f98
    BN_hex2bn(&e, "010001");
    // Perform the modular exponentiation
    BN_mod_exp(m, s, e, n, ctx);
    // Print the result
    printBN("Verification message (M) = ", m);
    // Free allocated memory
    BN_free(n);
    BN_free(s);
    BN_free(e);
    BN_free(m);
    BN_CTX_free(ctx);
    return 0;
}
```

In this step, we have incorporated the values of n, s, and e, which were extracted from the preceding steps explained in the description.

—-------- Task 6 done successfully and verified done hehe ----------------------