CPS633 Lab 4: RSA Encryption

TASK 1

> The following changes were made to file bn_sample.c (renamed to task1.c):

```
//Changes start
  BIGNUM *p = BN_new();
  BIGNUM *q = BN_new();
  BIGNUM *e = BN_new();
  BIGNUM *d = BN_new();
  BIGNUM *res = BN_new();
  BIGNUM *phi = BN_new();
  BIGNUM *n = BN_new();
  BIGNUM *p1 = BN_new();
  BIGNUM *q1 = BN_new();

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

```
// n = p*q
BN_mul(n, p, q, ctx);



//phi(m) = (p-1)*(q-1)
BN_sub(p1, p, BN_value_one());
BN_sub(q1, q, BN_value_one());
BN_mul(phi, p1, q1, ctx);

BN_gcd(res, phi, e, ctx);
if (!BN_is_one(res))
{
      printf("Error: GCD of phi and e not 1 \n ");
      exit(0);
  }

//Calculate and print the private key
BN_mod_inverse(d, e, phi, ctx);
printBN("private key", d);
```

> When compiling and running the program with the command

gcc task1.c -lcrypto

the program computes the private key and outputs the following:
3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB

```
[10/28/24]seed@VM:~/.../Labsetup$ gcc bn_task1.c -lcrypto
[10/28/24]seed@VM:~/.../Labsetup$ ./a.out
private key 3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB
```

TASK 2

> First, the command

python3 -c 'print("A top secret!".encode("utf-8").hex())'

is used to turn the secret message into hexadecimal format, resulting in the following:
4120746f702073656372657421

```
[10/28/24]seed@VM:~/.../Labsetup$ python3 -c 'print("A top secret!".encode("utf-8").hex())'
4120746f702073656372657421
```

> The file task1.c was then modified, removing some logic that was implemented just to calculate the private key and adding in logic just for encryption. The relevant code is shown in the following:

```
//Changes start
  BIGNUM *n = BN_new();
  BIGNUM *e = BN_new();
  BIGNUM *plain = BN_new();
  BIGNUM *cipher = BN_new();


  BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
  BN_dec2bn(&e, "65537");
  BN_hex2bn(&plain, "4120746f702073656372657421");


  BN_mod_exp(cipher, plain, e, n, ctx);
  printBN("Encrypted message", cipher);
//Changes end
```

> When compiling and running the program with the command

gcc task2.c -lcrypto

the program encrypts the message, resulting in the following encrypted message:
6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC

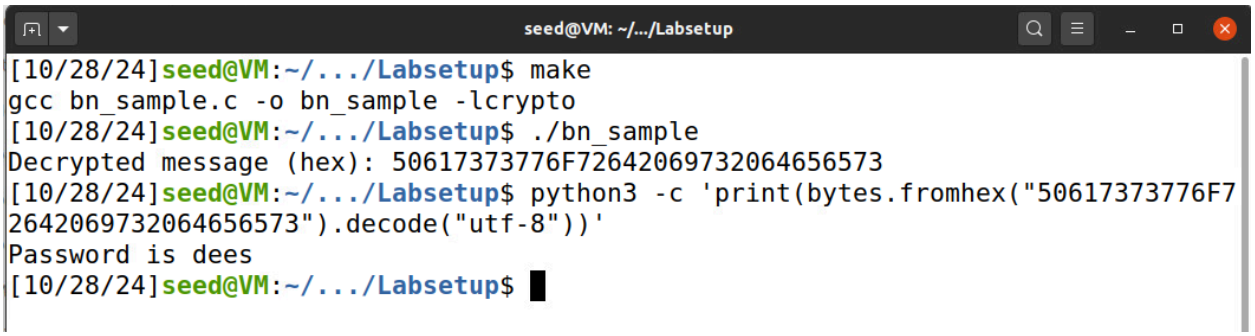
```
[10/28/24]seed@VM:~/.../Labsetup$ gcc task2.c -lcrypto
[10/28/24]seed@VM:~/.../Labsetup$ ./a.out
Encrypted message 6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC
```

TASK 3

Here's the updated code for decrypting the RSA message:

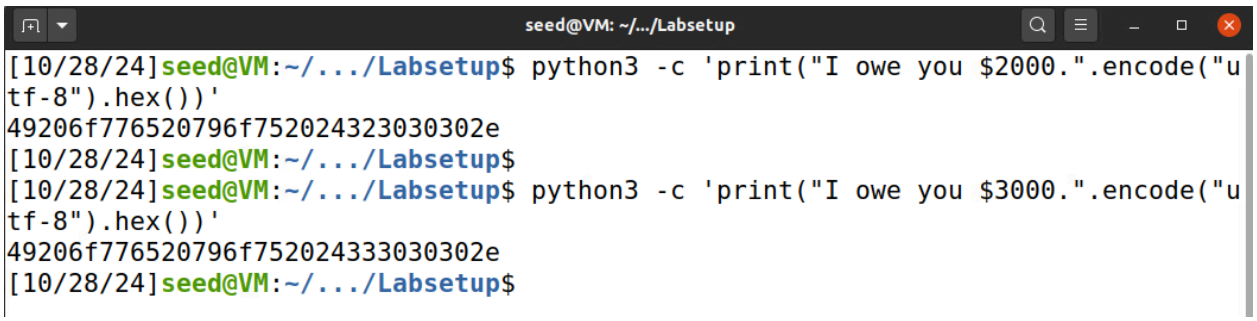
```
int main() {
    BN_CTX *ctx = BN_CTX_new();

    // Public and private keys
    BIGNUM *n = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *C = BN_new();
    BIGNUM *M = BN_new();

    // Initialize n, d, and C with the hexadecimal values
    BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BN_hex2bn(&C, "8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");

    // Calculate M = C^d mod n
    BN_mod_exp(M, C, d, n, ctx);

    // Print the decrypted message in hexadecimal
    char *msg = BN_bn2hex(M);
    printf("Decrypted message (hex): %s\n", msg);


    OPENSSL_free(msg);
    BN_free(n);
    BN_free(d);
    BN_free(C);
    BN_free(M);
    BN_CTX_free(ctx);
    return 0;
}
```

And here is the decrypted text : “Password is dees”



TASK 4

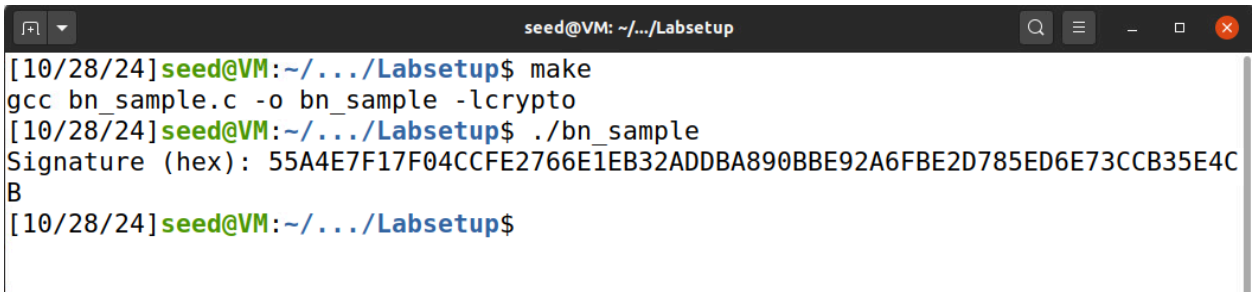
Here is the conversion of the 2 messages into hexadecimal :



Here is the code to sign messages:

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

void sign_message(const char *hex_message, BIGNUM *d, BIGNUM *n) {
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *M = BN_new();
    BIGNUM *S = BN_new();

    // Convert the hex message to a BIGNUM
    BN_hex2bn(&M, hex_message);

    // Calculate S = M^d mod n
    BN_mod_exp(S, M, d, n, ctx);

    // Print the signature in hexadecimal
    char *sig_hex = BN_bn2hex(S);
    printf("Signature (hex): %s\n", sig_hex);

    // Free memory
    OPENSSL_free(sig_hex);
    BN_free(M);
    BN_free(S);
    BN_CTX_free(ctx);
}

int main() {
    // Initialize keys
    BIGNUM *d = BN_new();
    BIGNUM *n = BN_new();

    // Set the private key and modulus
    BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");

    // Original message in hex
    const char *message = "49206f776520796f752024333030302e"; // Hex value of "I owe you $2000."

    // Sign the message
    sign_message(message, d, n);

    // Free keys
    BN_free(d);
    BN_free(n);
    return 0;
}
```

For the message M = I owe you $2000, the signature is
55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73CCB35E4B



For the message M = I owe you $3000, the signature is
BCC20FB7568E5D48E434C387C06A6025E90D29D848AF9C3EBAC0135D99305822

Conclusion : A very small change in the message changes the entire signature.

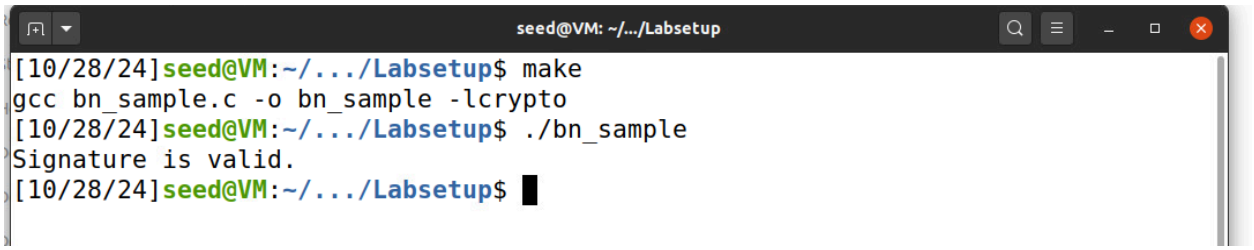
Task 5

Here is the code to check the signatures:

```
int verify_signature(const char *hex_message, const char *hex_signature, BIGNUM *e, BIGNUM *n) {
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *M = BN_new();
    BIGNUM *S = BN_new();
    BIGNUM *M_prime = BN_new();

    // Convert the hex values to BIGNUM
    BN_hex2bn(&M, hex_message);
    BN_hex2bn(&S, hex_signature);

    // Calculate M' = S^e mod n
    BN_mod_exp(M_prime, S, e, n, ctx);

    // Compare M' with M
    if (BN_cmp(M_prime, M) == 0) {
        printf("Signature is valid.\n");
    } else {
        printf("Signature is invalid.\n");
    }

    // Free memory
    BN_free(M);
    BN_free(S);
    BN_free(M_prime);
    BN_CTX_free(ctx);
    return 0;
```

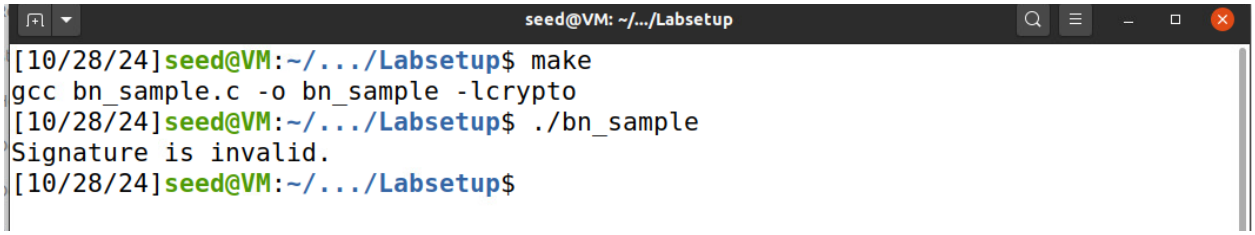
```
int main() {
    // Initialize keys
    BIGNUM *e = BN_new();
    BIGNUM *n = BN_new();

    // Set the public key values
    BN_hex2bn(&e, "010001");
    BN_hex2bn(&n, "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");

    // Original message in hex
    const char *message = "4C61756E63682061206D697373696C652E"; // Hex value of "Launch a missile."
    const char *signature = "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F";

    // Verify the signature
    verify_signature(message, signature, e, n);

    // Free keys
    BN_free(e);
    BN_free(n);
    return 0;
}
```

The first signature S = 643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802**F** is valid.



But when you change 2F to 3F: S = 643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803**F** for the second signature, it's no longer valid.

```c
int main() {
    // Initialize keys
    BIGNUM *e = BN_new();
    BIGNUM *n = BN_new();

    // Set the public key values
    BN_hex2bn(&e, "010001");
    BN_hex2bn(&n, "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");

    // Original message in hex
    const char *message = "4C61756E63682061206D697373696C652E"; // Hex value of "Launch a missile."
    const char *signature = "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F";

    // Verify the signature
    verify_signature(message, signature, e, n);

    // Free keys
    BN_free(e);
    BN_free(n);
    return 0;
}
```



Conclusion : The first signature is valid, but all you have to do is change a number in the signature and it's no longer valid.

Task 6

Step 1: Downloading the certificate from a real web server

```
[10/28/24]seed@VM:~$ openssl s_client -connect www.example.org:443 -showcerts
CONNECTED(00000003)
depth=1 C = US, O = DigiCert Inc, CN = DigiCert Global G2 TLS RSA SHA256 2020 CA
1
verify error:num=20:unable to get local issuer certificate
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 Global G2 TLS RSA SHA256 2020 CA1
-----BEGIN CERTIFICATE-----
MIIHbjCCBlagAwIBAgIQB1vO8waJyK3fE+Ua9K/hhzANBgkqhkiG9w0BAQsFADBZ
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMTMwMQYDVQQDEypE
aWdpQ2VydCBHbG9iYWwgRzIgVExTIFJTQSBTSEEyNTYgMjAyMCBDQTEwHhcNMjQw
MTMwMDAwMDAwWhcNMjUwMzAxMjM1OTU5WjCBljELMAkGA1UEBhMCVVMxEzARBgNV
BAgTCkNhbGlmb3JuaWExFDASBgNVBAcTC0xvcyBBbmdlbGVzMUIwQAYDVQQKDDlJ
bnRlcm5ldMKgQ29ycG9yYXRpb27CoGZvcsKgQXNzaWduZWTCoE5hbWVzwqBhbmTC
oE51bWJlcnMxGDAWBgNVBAMTD3d3dy5leGFtcGxlLm9yZzCCASIwDQYJKoZIhvcN
AQEBBQADggEPADCCAQoCggEBAIaFD7sO+cpf2fXgCjIsM9mqDgcpqC8IrXi9wga/
9y0rpqcnPVOmTMNLsid3INbBVEm4CNr5cKlh9rJJnWlX2vttJDRyLkfwBD+dsVvi
```

- Note that the command is continued and generates two certificates

Take the two certificates and put them into two files, c0.pem & c1.pem

Step 2: Extract Public key (e, n) from issuer's certificate

To do this, need to extract modulus and e from the certificate

```
[10/28/24]seed@VM:~$ openssl x509 -in c1.pem -noout -modulus
Modulus=CCF710624FA6BB636FED905256C56D277B7A12568AF1F4F9D6E7E18FBD95ABF260411570
DB1200FA270AB557385B7DB2519371950E6A41945B351BFA7BFABBC5BE2430FE56EFC4F37D97E314
F5144DCBA710F216EAAB22F031221161699026BA78D9971FE37D66AB75449573C8ACFFEF5D0A8A59
43E1ACB23A0FF348FCD76B37C163DCDE46D6DB45FE7D23FD90E851071E51A35FED4946547F2C88C5
F4139C97153C03E8A139DC690C32C1AF16574C9447427CA2C89C7DE6D44D54AF4299A8C104C2779C
D648E4CE11E02A8099F04370CF3F766BD14C49AB245EC20D82FD46A8AB6C93CC6252427592F89AFA
5E5EB2B061E51F1FB97F0998E83DFA837F4769A1
```

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

Step 3: Extract signature from the server's certificate

```
Signature Algorithm: sha256WithRSAEncryption
     04:e1:6e:02:3e:0d:e3:23:46:f4:e3:96:35:05:93:35:22:02:
     0b:84:5d:e2:73:86:d4:74:4f:fc:1b:27:af:3e:ca:ad:c3:ce:
     46:d6:fa:0f:e2:71:f9:0d:1a:9a:13:b7:d5:08:48:bd:50:58:
     b3:5e:20:63:86:29:ca:3e:cc:cc:78:26:e1:59:8f:5d:ca:8b:
     bc:49:31:6f:61:bd:42:ff:61:62:e1:22:35:24:26:9b:57:eb:
     e5:00:0d:ff:40:33:6c:46:c2:33:77:08:98:b2:7a:f6:43:f9:
     6d:48:df:bf:fe:fa:28:1e:7b:8a:cf:2d:61:ff:6c:87:98:a4:
     2c:62:9a:bb:10:8c:ff:34:48:70:66:b7:6d:72:c3:69:f9:39:
     4b:68:39:56:bd:a1:b3:6d:f4:77:f3:46:5b:5c:19:ac:4f:b3:
     74:6b:8c:c5:f1:89:cc:93:fe:0c:01:6f:88:17:dc:42:71:60:
     e3:ed:73:30:42:9c:a9:2f:3b:a2:78:8e:c8:6f:ba:d1:13:0c:
     d0:c7:5e:8c:10:fb:01:2e:37:9b:db:ac:f7:a1:ac:ba:7f:f8:
     92:e7:cb:41:44:c8:15:f9:f3:c4:bb:ad:51:5f:be:de:c7:ac:
     86:07:9f:40:ec:b9:0b:f6:b2:8b:cc:b5:55:33:66:ba:33:c2:
     c4:f0:a2:e9
```

Need to truncate the signature algorithm using tr to remove all colons and spaces

```
[10/28/24]seed@VM:~$ cat signature.txt | tr -d '[:space:]:'
04e16e023e0de32346f4e3963505933522020b845de27386d4744ffc1b27af3ecaadc3ce46d6fa0f
e271f90d1a9a13b7d50848bd5058b35e20638629ca3ecccc7826e1598f5dca8bbc49316f61bd42ff
6162e1223524269b57ebe5000dff40336c46c233770898b27af643f96d48dfbffefa281e7b8acf2d
61ff6c8798a42c629abb108cff34487066b76d72c369f9394b683956bda1b36df477f3465b5c19ac
4fb3746b8cc5f189cc93fe0c016f8817dc427160e3ed7330429ca92f3ba2788ec86fbad1130cd0c7
5e8c10fb012e379bdbacf7a1acba7ff892e7cb4144c815f9f3c4bbad515fbedec7ac86079f40ecb9
0bf6b28bccb5553366ba33c2c4f0a2e9[10/28/24]seed@VM:~$
```

- Note that signature.txt is the file that contains the signature algorithm for easier processing

Step 4: Extracting the body of the server's certificate

```
[10/28/24]seed@VM:~$ openssl asn1parse -i -in c0.pem
    0:d=0  hl=4 l=1902 cons: SEQUENCE
    4:d=1  hl=4 l=1622 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           :075BCEF30689C8ADDF13E51AF4AFE1
87
   31:d=2  hl=2 l=  13 cons:   SEQUENCE

 1630:d=1  hl=2 l=  13 cons:  SEQUENCE
 1632:d=2  hl=2 l=   9 prim:   OBJECT            :sha256WithRSAEncryption
 1643:d=2  hl=2 l=   0 prim:   NULL
 1645:d=1  hl=4 l= 257 prim:  BIT STRING
```

- We can see that the body of the certificate ranges from 4-1630.

```
[10/28/24]seed@VM:~$ openssl asn1parse -i -in c0.pem -strparse 4 -out c0_body.bi
n -noout
```

- Sending the body of the certificate into c0_body.bin

```
[10/28/24]seed@VM:~$ sha256sum c0_body.bin
b2825cb7d71ec7093e7ff7026c562a29122de3b4900ed13dad63d1be73706e0d  c0_body.bin
```

- Calculating the hash

Step 5: Verifying the Signature

- In order to do this, need to pad the hash so that it matches the length of M and N.
- Create padHash.c

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

int main() {
   // Define prefix, hash, and A as strings
   const char *prefix = "0001";
   const char *hash =
"b2825cb7d71ec7093e7ff7026c562a29122de3b4900ed13dad63d1be73706e0d";
   const char *A = "3031300D060960864801650304020105000420";

   // Define the total length and calculate pad length
   int total_len = 256;
   int pad_len = total_len - 1 - (strlen(A) + strlen(prefix) + strlen(hash)) / 2;

   // Create buffer for the final result string
   char result[total_len * 2 + 1]; // +1 for null terminator
   strcpy(result, prefix); // Start with the prefix

   // Add the "FF" padding
   for (int i = 0; i < pad_len; i++) {
      strcat(result, "FF");
   }

   // Append "00", A, and hash
strcat(result, "00");
   strcat(result, A);
   strcat(result, hash);

   // Output the result
   printf("%s\n", result);

   return 0;
}
```

- Take result of this, and replace the values of the program from task 4

```
    // MODIFICATION BY ROHAN MANOHARAN
 BN_hex2bn(&n, "CCF710624FA6BB636FED905256C56D277B7A12568AF1F4F9D6E7E18FBD95ABF260411570DB1200FA270AB557385B7DB251937
1950E6A41945B351BFA7BFABBC5BE2430FE56EFC4F37D97E314F5144DCBA710F216EAAB22F031221161699026BA78D9971FE37D66AB75449573C8
ACFFEF5D0A8A5943E1ACB23A0FF348FCD76B37C163DCDE46D6DB45FE7D23FD90E851071E51A35FED4946547F2C88C5F4139C97153C03E8A139DC6
90C32C1AF16574C9447427CA2C89C7DE6D44D54AF4299A8C104C2779CD648E4CE11E02A8099F04370CF3F766BD14C49AB245EC20D82FD46A8AB6C
93CC6252427592F89AFA5E5EB2B061E51F1FB97F0998E83DFA837F4769A1");
 BN_dec2bn(&e, "65537");
    BN_hex2bn(&M, "0001FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF003031300D060960864801650304020105000420b
2825cb7d71ec7093e7ff7026c562a29122de3b4900ed13dad63d1be73706e0d"); //hash of the certificate body
    BN_hex2bn(&S,"04e16e023e0de32346f4e3963505933522020b845de27386d4744ffc1b27af3ecaadc3ce46d6fa0fe271f90d1a9a13b7d50
848bd5058b35e20638629ca3ecccc7826e1598f5dca8bbc49316f61bd42ff6162e1223524269b57ebe5000dff40336c46c233770898b27af643f9
6d48dfbffefa281e7b8acf2d61ff6c8798a42c629abb108cff34487066b76d72c369f9394b683956bda1b36df477f3465b5c19ac4fb3746b8cc5f
189cc93fe0c016f8817dc427160e3ed7330429ca92f3ba2788ec86fbad1130cd0c75e8c10fb012e379bdbacf7a1acba7ff892e7cb4144c815f9f3
c4bbad515fbedec7ac86079f40ecb90bf6b28bccb5553366ba33c2c4f0a2e9");
```

- Running the program: