Assignment 2

Course: CISC 870 (Cryptography)
Term: Fall 2022

Submitted by:

Kazi Amit Hasan
ID: 20341105
MSc (research-based) Student, School of Computing
Queen's University

Lab Setup: I set up my lab environment in the previous lab. With the reference to the previous assignment, I installed and used SEED Ubuntu 20.4 on VirtualBox on MacBook Pro (2017 Mid). I have followed the instructions given on the website. [1] [2].

I followed the first approach of using a pre-built SEED VM. Additionally, I have allocated the memory size of 2GB instead of the usual recommendation for better performance of the Virtual machine. Also, I set the video memory to 64MB and increased the number of CPUs to 2 from 1 to increase performance.

Before getting into the main tasks, I explored the 2.1 section of the assignment and tried to understand the command.

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

// Initialize a, b, n
        BN_generate_prime_ex(a, NBITS, 1, NULL, NULL, NULL);
        BN_dec2bn(&b, "273489463796838501848592769467194369268");
        BN_rand(n, NBITS, 0, 0);
        // res = a*b
        BN_mul(res, a, b, ctx);
        printBN("a * b = ", res);
        // res = a^b mod n
        BN_mod_exp(res, a, b, n, ctx);
        printBN("a^c mod n = ", res);
        return 0;
}

```

Figure: Code implementation of example.c (given in the task sheet)

Figure: Output of the given example.

Task 1: Deriving the Private Key

Approach and Explanation: The task was to calculate the private key, d. As per the description, p, q, and e were already given. To complete this task, I have modified the given example that was described in the task sheet. I have initialized big numbers p, q, e, x, y, s, t, d, n. Then, I assigned the given values. To avoid any mistakes, I have copied the given values of p, q and e. And then paste them into the code. The steps were commented on in the code. Figure (1.1 +1.2) and 1.3 represents the implementation of deriving the private key and the private key (in output) respectively.

Open ~/Desktop/Cryptography Lab/Assignment 2 task1.c Save

```c
/* task1.c */

#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 128


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 *x = BN_new();
        BIGNUM *y = BN_new();
        BIGNUM *s = BN_new();
        BIGNUM *t = BN_new();
        BIGNUM *d = BN_new();
        BIGNUM *n = BN_new();

// Assigning the given values of p, q, e


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

```

Terminal

Figure 1.1: Code implementation of task 1

```
32
33
34
35          BN_hex2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF");
36          BN_hex2bn(&q, "E85CED54AF57E53E092113E62F436F4F");
37          BN_hex2bn(&e, "0D88C3");
38
39          //Assigning s as 1
40          BN_dec2bn(&s, "1");
41
42          // Calculating (p-1)
43          BN_sub(x, p, s);
44
45          // Calculating (q-1)
46          BN_sub(y, q, s);
47
48          //Calculate (p-1)*(q-1)
49          BN_mul(t, x, y, ctx);
50
51          //Calculate private key using the below function
52          BN_mod_inverse(d, e, t, ctx);
53
54
55          printBN("P is",p);
56          printBN("Q is",q);
57          printBN("Private key is",d);
58          return 0;
59
60 }
```

C ▾ Tab Width: 8 ▾ Ln 57, Col 27 ▾ INS

Figure 1.2: Code implementation of task 1

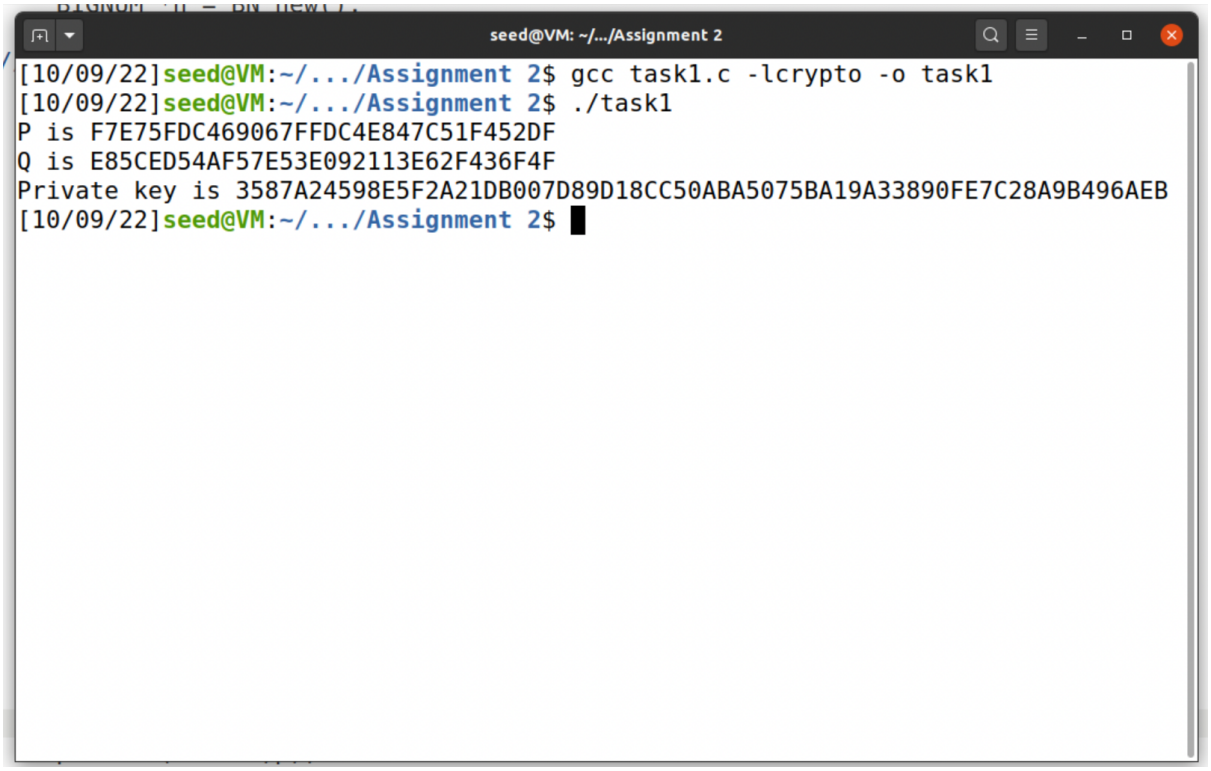

Figure 1.3: Output of task 1 (deriving the private key)

Task 2: Encrypting a Message

Approach and Explanation: The task was to encrypt the given text, "A top secret!". To get the hex string, I used the given command on the task sheet. (screenshot attached in figure 2.1) As per the description, n, e, M and d were already given.

To complete this task, I have modified the code of task 1. I have initialized big numbers d, n, e, M, C. Then, I assigned the given values. To avoid any mistakes, I have copied the given values of n, e, M and d. And then paste them into the code. The steps were commented on in the code. Figures 2.2 and 2.3 are representing the codes and the output of this task respectively. As I am using version 3 of python. That's why I had to modify the code in Figure 2.1.

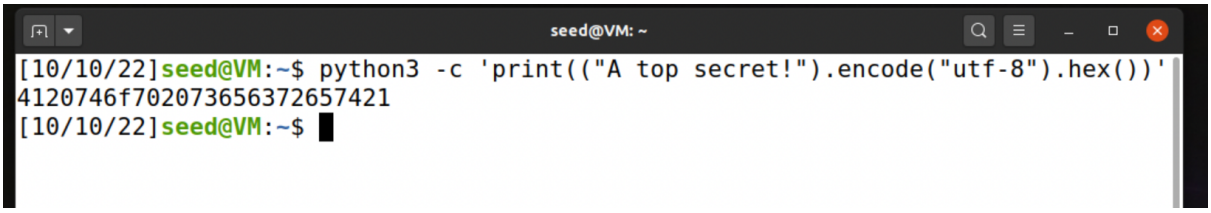

Figure 2.1: Generating hex string.

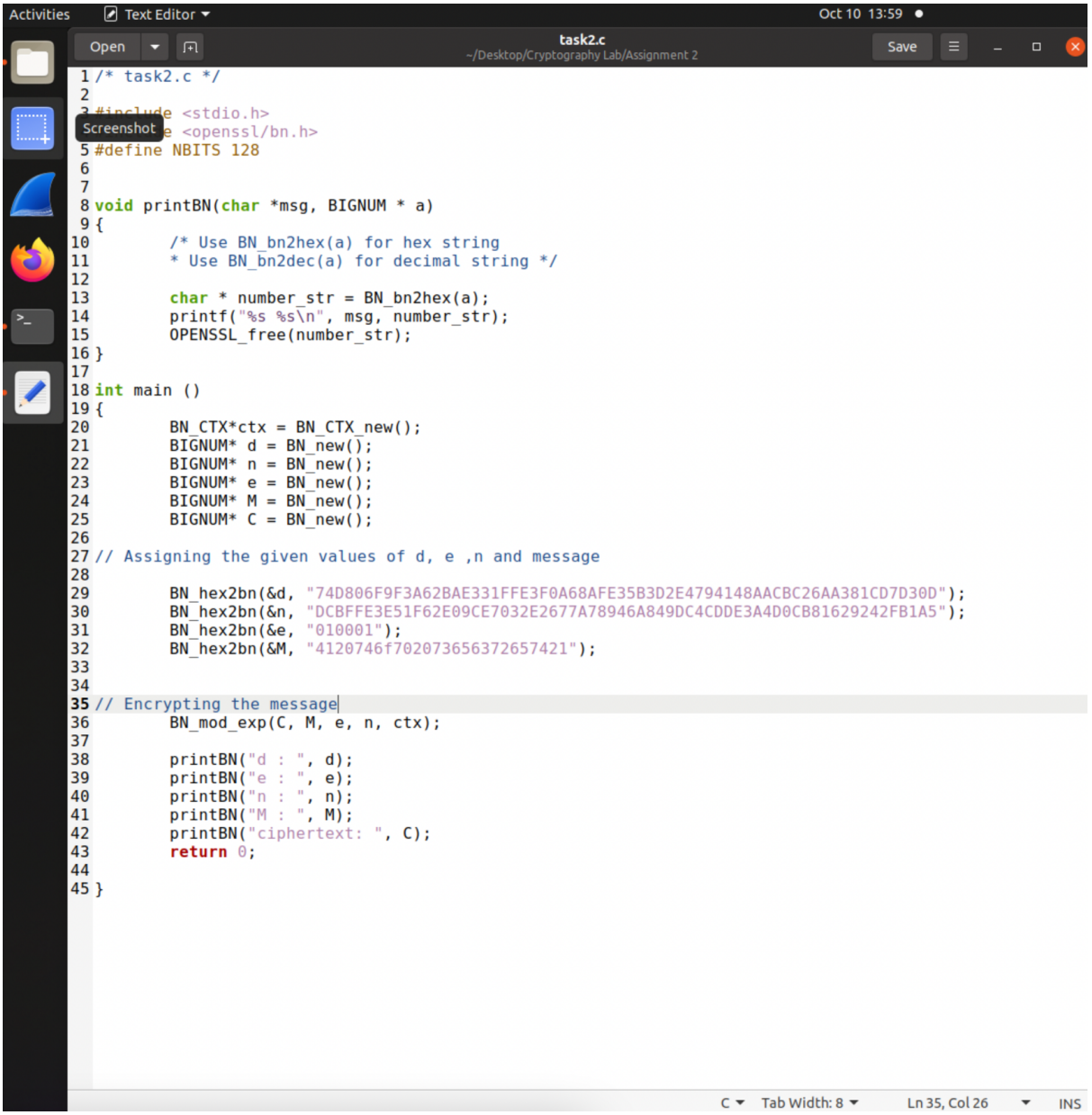

Fig 2.2: Code implementation of task 2

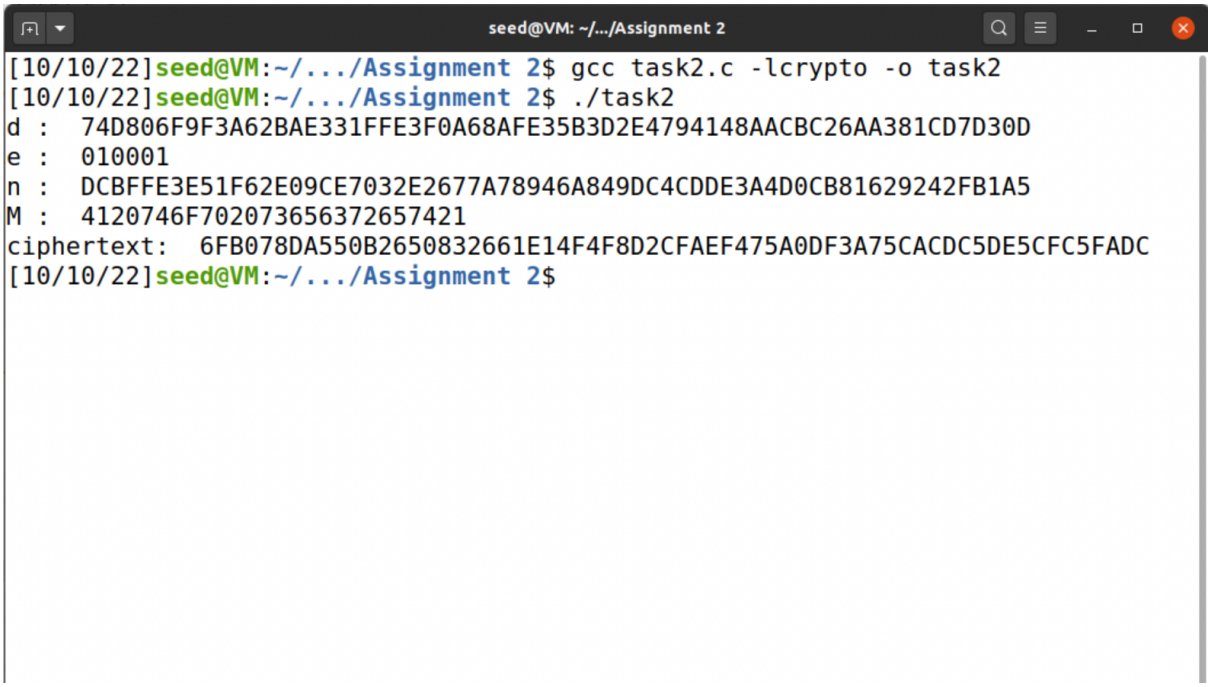

Fig 2.3: Output of task 2 (encrypting a Message)

Task 3: Decrypting a Message

Approach and Explanation: The task was to decrypt a cipher text. To solve this problem, I need to take help from task 2. The public/private keys will remain the same for this. I have modified the source code of task 2 to perform this task. After executing the task, I got the plaintext in hex string.

To get an ASCII string, I used the command given in the task sheet. As I am using version 3 of python, I had to use a modified command to get the ASCII message. [3] After executing that, I got the message that '**Password is dees'**. Figures 3.1 and 3.2 are representing the implementation of the task and the output respectively.

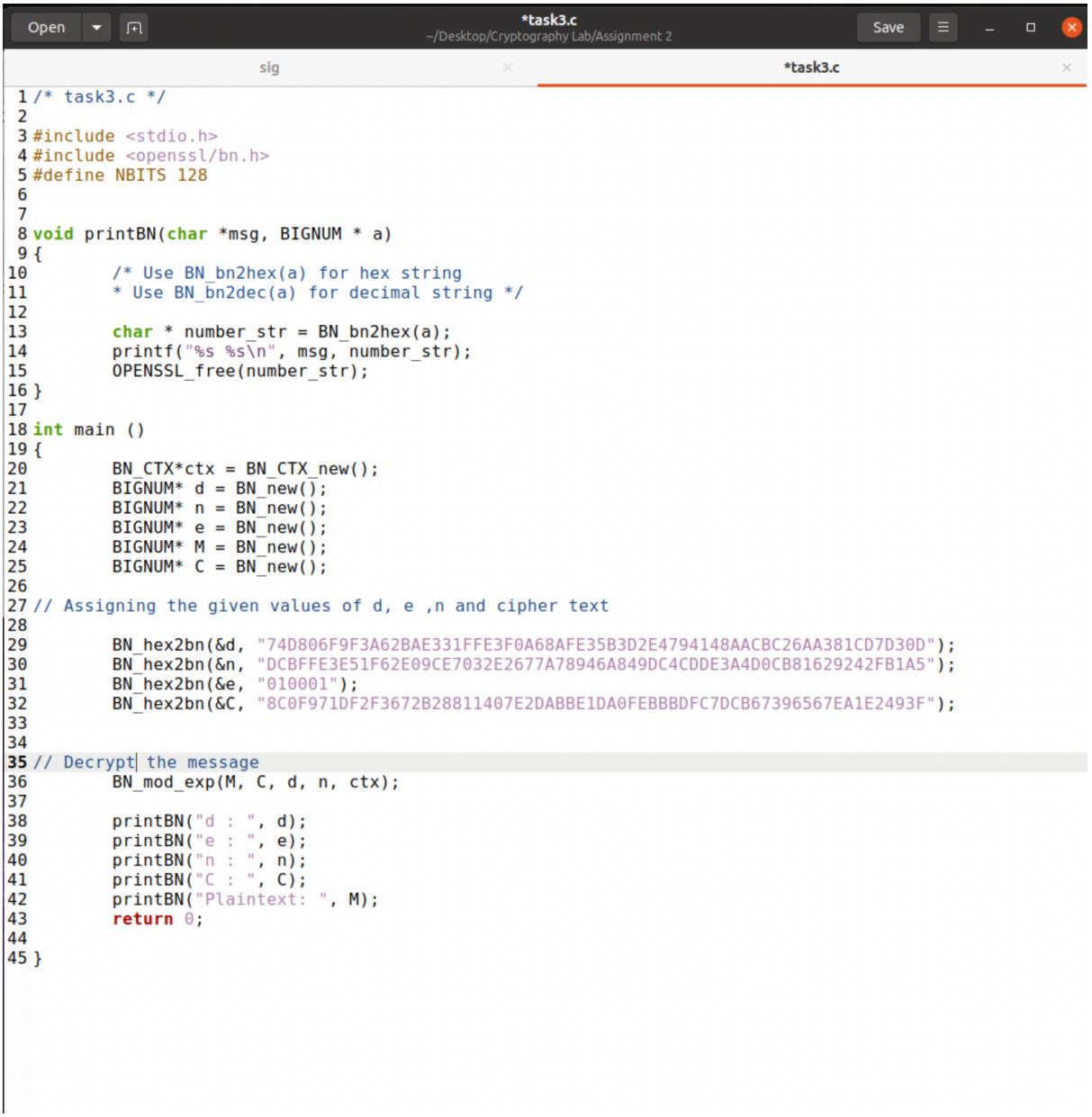

Figure 3.2: Code implementation of task 3

```
[10/10/22]seed@VM:~/.../Assignment 2$ gcc task3.c -lcrypto -o task3
[10/10/22]seed@VM:~/.../Assignment 2$ ./task3
d :  74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D
e :  010001
n :  DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
C :  8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F
Plaintext:  50617373776F726420697320064656573
[10/10/22]seed@VM:~/.../Assignment 2$ python3 -c 'print(bytes.fromhex("506173737
76F726420697320064656573").decode("utf-8"))'
Password is dees
[10/10/22]seed@VM:~/.../Assignment 2$
```

Figure 3.3: Output of task 3.

Task 4: Signing a Message

Approach and Explanation: The task was to sign a message. Like task 3, to solve this problem, I need to take the help of task 2 and task 3. The public/private keys will remain the same for this. I have modified the source code of task 2 to perform this task.
Firstly, I had to generate a hex string from the given strings. I kept the message the same as the task sheet. After that, I generated the signatures of each message.
Figure 4.1 represents the process of generating the hex strings. In this assignment, I have used version 3 of python.

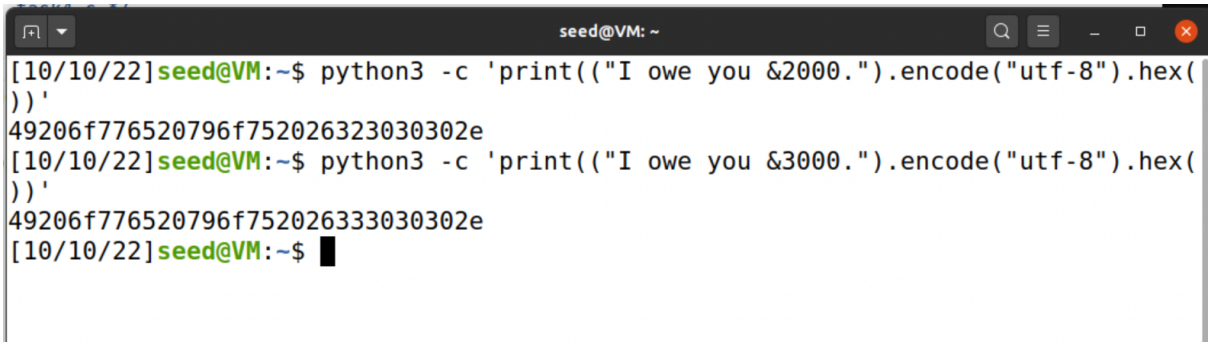

Figure 4.1: Generating hex string from given strings.

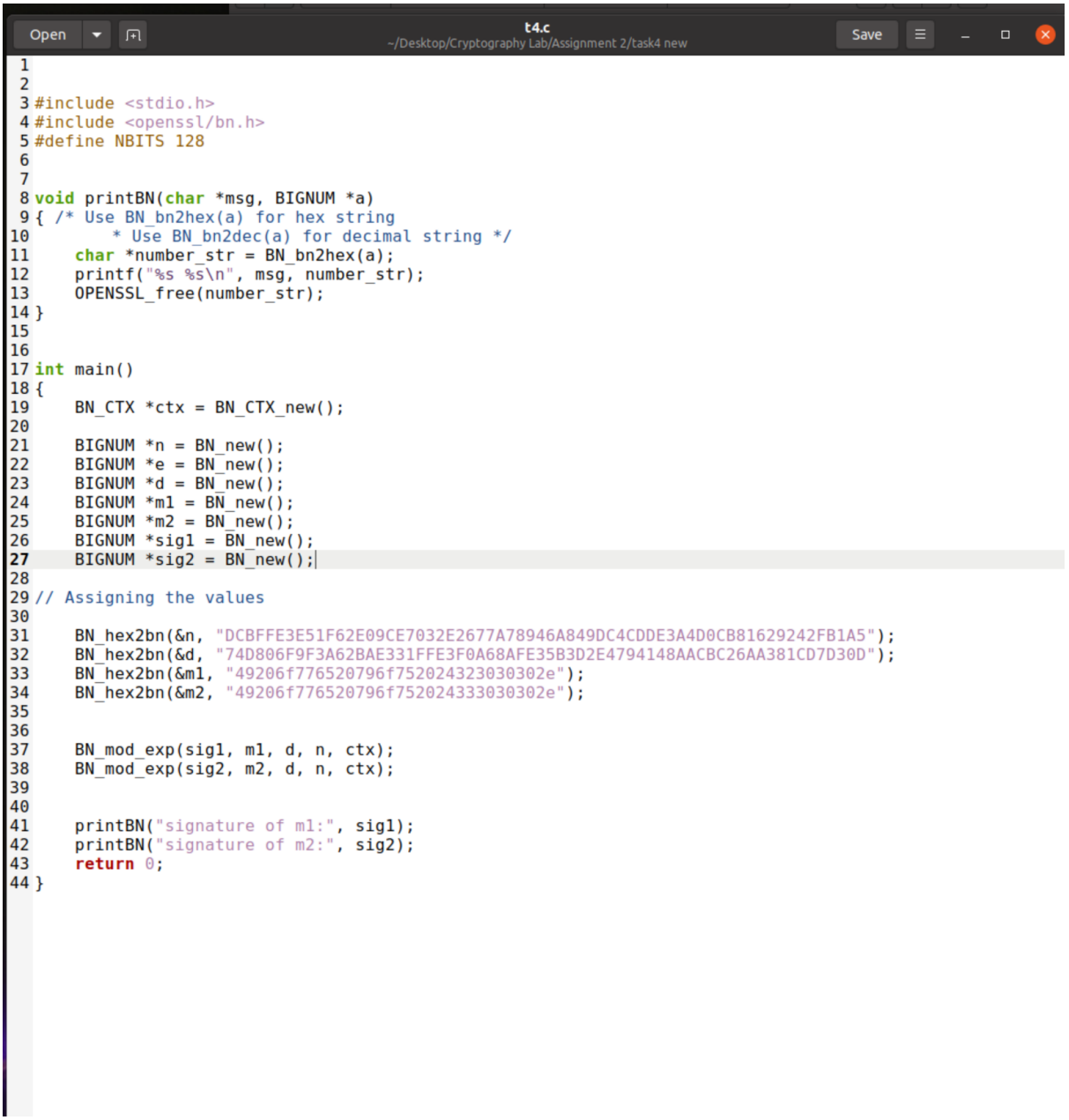

Figure 4.1: Code implementation of task 4

Figure 4.3: Output of task 4.

Comparing both signatures and describing the observation: I modified the message from $2000 to $3000 and then performed the signature operation. As we can see in Figure 4.3, the signatures of each message are entirely different. The small change in string resulted in a small change in hex string too (screenshot in Figure 4.1). But after observing the signatures, I can see that there's a big change in the signatures of both of the messages (screenshot in Figure 4.3). So, we can conclude that even a small change can create a big change in the signature.

Task 5: Verifying a Signature

Approach and Explanation: In order to perform this task, I have followed several steps. Also, I changed the signature from 2F to 3F (last byte) in the code to perform the additional task in the assignment. The steps are described below:

1. Firstly, I had to generate the hex string from the given message. Figure 5.1 represents the process of generating the hex string.
2. As per the task sheet, we already know the public key. Now, I have to check and verify whether the signature is valid or not. Here, valid refers to Alice's signature or not. The task sheet already provides several pieces of information like M, S, e, and n. Figure 5.2 and figure 5.3 represent the whole process of implementing the task.
3. In Figure 5.2, I have initialized the corrupted signature s_2 with a change from 2F to 3F.
4. In figure 5.3, I have implemented the process of verifying signatures.
5. The figure 5.4 represents the output of the task.

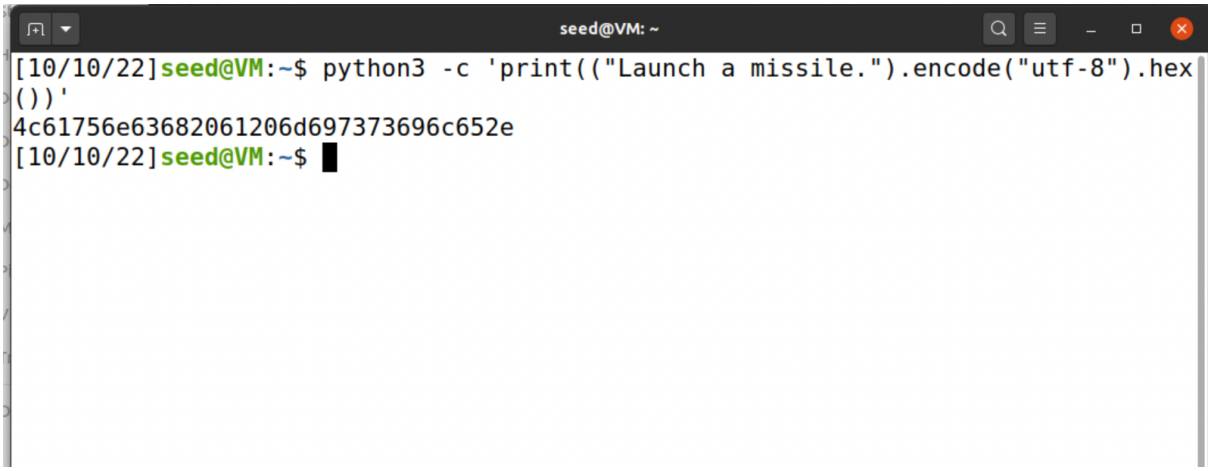

Figure 5.1: Generating hex string from given message

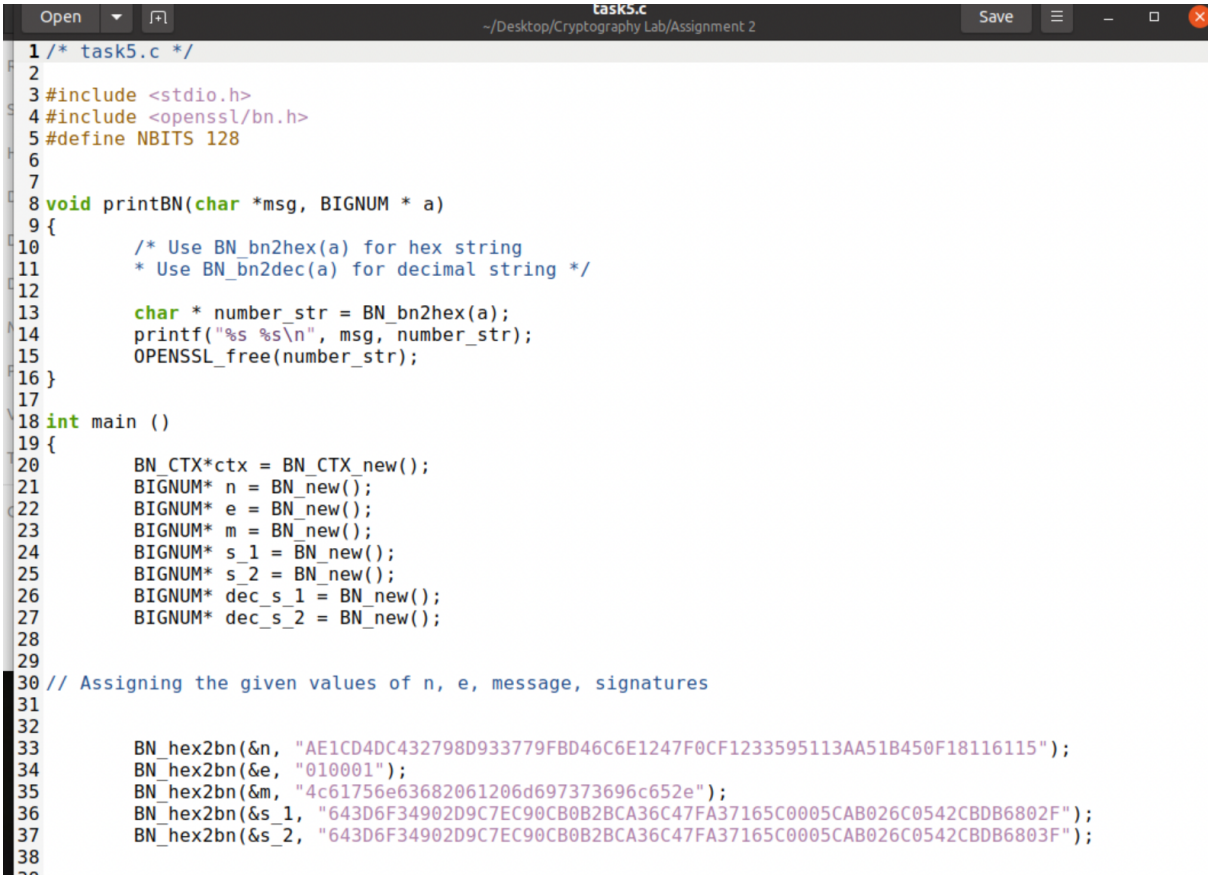

Figure 5.2: Code implementation of task 5

```
39
40          BN_mod_exp(dec_s_1, s_1, e, n, ctx);
41
42
43          BN_mod_exp(dec_s_2, s_2, e, n, ctx);
44
45
46          printBN("e : ", e);
47          printBN("n : ", n);
48          printBN("original signature : ", dec_s_1);
49          printBN("corrupted signature : ", dec_s_2);
50
51          if (BN_cmp(dec_s_1, m) == 0)
52          {
53          printf("valid!\n");
54          }
55          else
56          {
57          printf("invalid!\n");
58          }
59          if (BN_cmp(dec_s_2, m) == 0)
60          {
61          printf("valid!\n");
62          }
63          else
64          {
65          printf("invalid!\n");
66          }
67
68
69          return 0;
70
71 }
```

Figure 5.3: Code implementation of task 5

```
seed@VM: ~/.../Assignment 2
[10/10/22]seed@VM:~/.../Assignment 2$ gcc task5.c -lcrypto -o task5
[10/10/22]seed@VM:~/.../Assignment 2$ ./task5
e :  010001
n :  AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115
original signature :  4C61756E63682061206D697373696C652E
corrupted signature :  91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B
41AC294
valid!
invalid!
[10/10/22]seed@VM:~/.../Assignment 2$
```

Figure 5.4: Output of task 5.

Describing what will happen to the verification process:
In simpler words, the verification will be invalid with a corrupted signature.
Explanation: After changing the signature according to the task sheet (from 2F to 3F), it is shown that the corrupted signature is completely different from the original signature (figure 5.4 contains the screenshot). Also, the verification was successful when I used the original signature. So, it means that that's the message from Alice. The verification was invalid when

I used the corrupted signature. Here, in figure 5.4, the 'valid' term represents the verification of the original signature and the 'invalid' term represents the corrupted signature. I observed that if any attacker changes the signature slightly, it will show the message will be invalid.

Task 6: Manually Verifying an X.509 Certificate

Approach and Explanation: I have followed the mentioned steps described in the task sheet.

1. I have used the official website of PyTorch (https://pytorch.org/) for this task. I downloaded the certificate of the PyTorch website using the mentioned command in the task sheet and saved them in a text file. Figure 6.1 represents the command. After that, I created two files named c0.pem and c1.pem and copy-pasted the certificates according to the task sheet.
2. Then, I extracted the public keys. For this, I followed the given instructions on the task sheet. Figure 6.2 and figure 6.3 represents the extraction of the public keys (modulus and exponent).
3. After that, by following the given instructions in the task sheet, I extracted the signature from the server's certificate. In order to remove the space and colons, I again executed the given command on the task sheet. Then save the values in another file. Figure 6.4.1 and figure 6.4.2 represent the screenshots of the output.
4. Then, I extracted the body of the server's certificate. Figure 6.5.1 and figure 6.5.2 represent the output. As per the output, the field starts from '4:d=1 hl =4 l=1027 cons: SEQUENCE'. This will be used to generate the hash value. Also, '1035:d=1 hl=2 l =13 cons: SEQUENCE' is the starting of the signature block.
5. Then, I generated the hash value. Figure 6.6 represents the hash value.
6. At this stage, I have all the information to verify the signature. Figure 6.7 represents the program to verify the certificate. I generated the result and then checked the validity of the signature. Figure 6.8 and figure 6.9 represent the steps for checking the validity. The output 'true' shown in Figure 6.9 represents the verification of the signature.

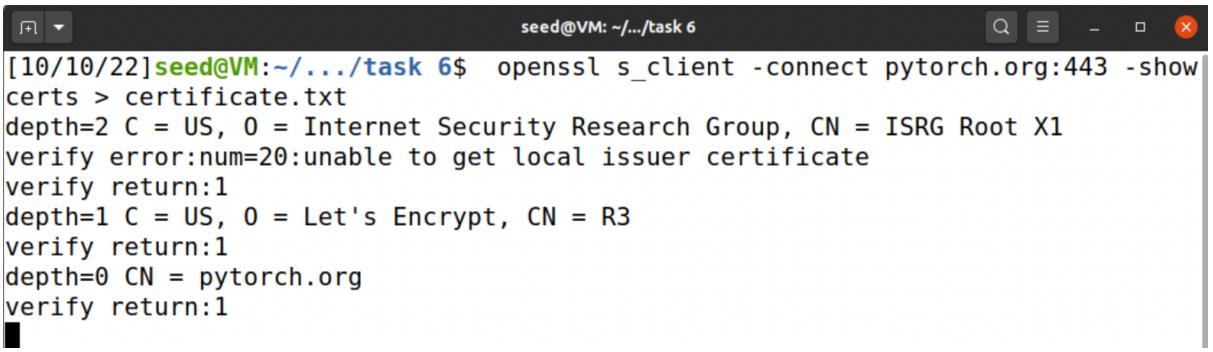

Figure 6.1: Downloading certificate

```
[10/10/22]seed@VM:~/.../task 6$ openssl x509 -in c1.pem -noout -modulus
Modulus=BB021528CCF6A094D30F12EC8D5592C3F882F199A67A4288A75D26AAB52BB9C54CB1AF8E
6BF975C8A3D70F4794145535578C9EA8A23919F5823C42A94E6EF53BC32EDB8DC0B05CF35938E7ED
CF69F05A0B1BBEC094242587FA3771B313E71CACE19BEFDBE43B45524596A9C153CE34C852EEB5AE
ED8FDE6070E2A554ABB66D0E97A540346B2BD3BC66EB66347CFA6B8B8F572999F830175DBA726FFB
81C5ADD286583D17C7E709BBF12BF786DCC1DA715DD446E3CCAD25C188BC60677566B3F118F7A25C
E653FF3A88B647A5FF1318EA9809773F9D53F9CF01E5F5A6701714AF63A4FF99B3939DDC53A706FE
48851DA169AE2575BB13CC5203F5ED51A18BDB15
[10/10/22]seed@VM:~/.../task 6$
```

Figure 6.2: Extracting modular (n)

```
[10/10/22]seed@VM:~/.../task 6$ openssl x509 -in c1.pem -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            91:2b:08:4a:cf:0c:18:a7:53:f6:d6:2e:25:a7:5f:5a
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, O = Internet Security Research Group, CN = ISRG Root X1
        Validity
            Not Before: Sep  4 00:00:00 2020 GMT
            Not After : Sep 15 16:00:00 2025 GMT
        Subject: C = US, O = Let's Encrypt, CN = R3
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (2048 bit)
                Modulus:
                    00:bb:02:15:28:cc:f6:a0:94:d3:0f:12:ec:8d:55:
                    92:c3:f8:82:f1:99:a6:7a:42:88:a7:5d:26:aa:b5:
                    2b:b9:c5:4c:b1:af:8e:6b:f9:75:c8:a3:d7:0f:47:
```

Figure 6.3: Extracting the exponent

```
                                    49:6B:34:E5:C4:CB:DB
    Signature Algorithm: sha256WithRSAEncryption
         0e:26:7c:ab:0f:7e:ca:f5:68:b9:03:05:c0:15:d0:b7:b7:24:
         2b:10:b0:ea:4f:9b:77:85:56:e5:c7:45:f4:be:6f:04:af:23:
         b3:5b:2c:2d:e5:69:b8:63:53:ea:6b:54:39:e9:a1:00:0d:37:
         91:dd:dc:8d:7a:d5:49:28:09:5f:91:9a:43:e6:1e:00:1c:18:
         fd:07:2f:54:3c:20:31:d4:b3:d9:a8:e1:c4:cb:88:68:cc:02:
         55:04:aa:8f:b3:7b:e7:71:b1:83:e7:3e:77:01:d6:c5:db:1a:
         82:0c:83:cc:a3:59:5e:0b:82:4d:f4:82:76:c3:0e:06:cd:1d:
         d7:6b:92:64:5d:de:38:2b:13:01:a5:31:c6:8e:c8:24:b6:35:
         b4:21:f2:2f:5c:e3:90:c7:9f:3c:5c:30:fa:02:76:4f:73:c4:
         e0:01:51:d7:90:2d:b0:09:ea:a6:33:dd:a8:64:22:bb:ea:af:
         10:4c:a1:01:34:9b:ec:b9:b4:25:99:02:2b:34:e6:ba:13:01:
         52:bd:ad:c3:dc:77:53:52:70:e3:17:99:3a:3d:83:01:e8:7c:
         79:28:5d:0d:59:cb:0c:0e:b9:cf:e1:a7:c9:e5:1d:21:e8:6d:
         50:e0:0d:99:9a:2b:5a:1a:2c:05:3b:ad:3a:01:a5:8a:20:d5:
         90:36:5e:db
[10/10/22]seed@VM:~/.../task 6$
```

Figure 6.4.1: Extracting the signature

```
        90:36:5e:db
[10/10/22]seed@VM:~/.../task 6$  cat signature | tr -d '[:space:]:'
0e267cab0f7ecaf568b90305c015d0b7b7242b10b0ea4f9b778556e5c745f4be6f04af23b35b2c2d
e569b86353ea6b5439e9a1000d3791dddc8d7ad54928095f919a43e61e001c18fd072f543c2031d4
b3d9a8e1c4cb8868cc025504aa8fb37be771b183e73e7701d6c5db1a820c83cca3595e0b824df482
76c30e06cd1dd76b92645dde382b1301a531c68ec824b635b421f22f5ce390c79f3c5c30fa02764f
73c4e00151d7902db009eaa633dda86422bbeaaf104ca101349becb9b42599022b34e6ba130152bd
adc3dc77535270e317993a3d8301e87c79285d0d59cb0c0eb9cfe1a7c9e51d21e86d50e00d999a2b
5a1a2c053bad3a01a58a20d590365edb[10/10/22]seed@VM:~/.../task 6$
```

Figure 6.4.2: Extracting the signature

```
[10/10/22]seed@VM:~/.../task 6$ openssl asn1parse -i -in c0.pem
    0:d=0  hl=4 l=1307 cons: SEQUENCE
    4:d=1  hl=4 l=1027 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=  18 prim:   INTEGER           :033DAEB1B112E0D11072808F4A0E85
```

Figure 6.5.1: Extracting the body of the server's certificate

```
ED0DA3EF5063552E75728BC896BC903D3CBD1116BECEB69E1777D6D06BD6E00000183B7B29354000
0040300473045022070BCBE343BE329F4FB8E404C25ADBA0003D89A00D641D252D023A472C358D7
3022100A9EDF45CCF160649EC7158DDDF2198F45D444F1C9F7282798A496B34E5C4CBDB
 1035:d=1  hl=2 l=  13 cons:  SEQUENCE
 1037:d=2  hl=2 l=   9 prim:   OBJECT            :sha256WithRSAEncryption
 1048:d=2  hl=2 l=   0 prim:   NULL
 1050:d=1  hl=4 l= 257 prim:  BIT STRING
[10/10/22]seed@VM:~/.../task 6$ openssl asn1parse -i -in c0.pem -strparse 4 -out
 c0_body.bin -noout
```

Figure 6.5.2: Extracting the body of the server's certificate

```
[10/10/22]seed@VM:~/.../task 6$ openssl asn1parse -i -in c0.pem -strparse 4 -out
 c0_body.bin -noout
[10/10/22]seed@VM:~/.../task 6$ sha256sum c0_body.bin
f9c30edf19c4364a3a643201064cb87ef5f4105433a426bfcd4e225cee666b24  c0_body.bin
```

Figure 6.6: Generating hash value.

```c
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 128

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 *n = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *sig = BN_new();
    BIGNUM *m = BN_new();

    BN_hex2bn(&n,
"BB021528CCF6A094D30F12EC8D5592C3F882F199A67A4288A75D26AAB52BB9C54CB1AF8E6BF975C8A3D70F4794145535578C9E
    BN_hex2bn(&e, "010001");
    BN_hex2bn(&sig,
"0e267cab0f7ecaf568b90305c015d0b7b7242b10b0ea4f9b778556e5c745f4be6f04af23b35b2c2de569b86353ea6b5439e9a1

    BN_mod_exp(m, sig, e, n, ctx);

    printBN("message:", m);

    return 0;
}
```

Figure 6.7: Code implementation for task 6.

```
[10/10/22]seed@VM:~/.../task 6$ touch task6.c
[10/10/22]seed@VM:~/.../task 6$ gcc task6.c -lcrypto -o task6
[10/10/22]seed@VM:~/.../task 6$ ./task6
message: 01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFF003031300D060960864801650304020105000420F9C30EDF19C4364A3A6432010
64CB87EF5F4105433A426BFCD4E225CEE666B24
```

Figure 6.8: Generating the message of task 6

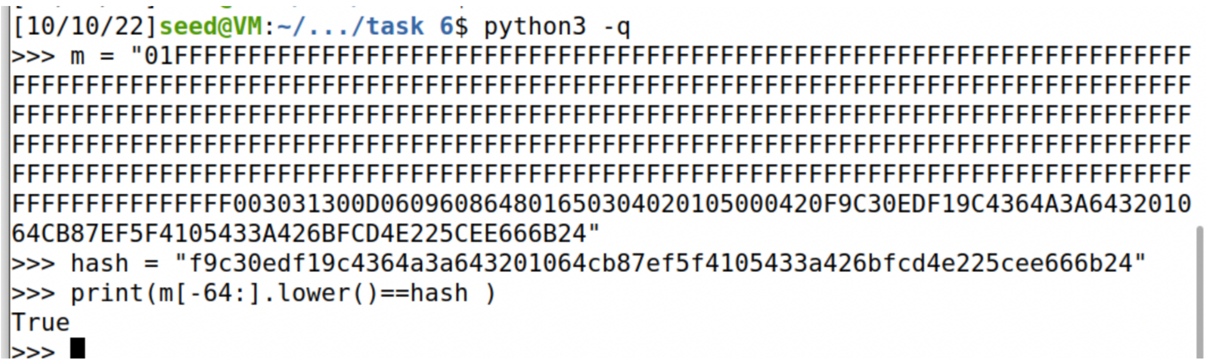

Figure 6.9: Checking the verification of the signature

Conclusion:

In this assignment, I learned about public key encryption, key generation, digital signatures, encryption and decryption using RSA and X-509 certificates. Some outputs were really interesting and were fun to explore and observe. I used Ubuntu 20.04 on VM with some extra cores and allocated some more space. Still, the machine felt a bit slow while doing the assignments. I struggled a lot in this assignment while doing task 6. My VM was constantly crashing. However, this assignment was really informative and interesting.

References:

1. (Available online) https://seedsecuritylabs.org/labsetup.html
2. (Available online) https://github.com/seed-labs/seed-labs/blob/master/manuals/vm/seedvm-manual.md
3. (Available online) https://stackoverflow.com/questions/28583565/str-object-has-no-attribute-decode-python-3-error