



Faculty of Engineering & Technology
Electrical & Computer Engineering Department
Applied Cryptography ENCS4320
RSA public key encryption and signature lab

Prepared by:

Hasan Hamed 1190496

Shorooq Ngjar 1192415

Instructor: Dr. Ahmad Alsadeh

Section: 1

Date: 27/8/2022

1. Abstract

The aim of this lab is to learn the RSA (Rivest-Shamir-Adleman) public-key encryption which is used widely for secure communications and to learn how to generate both the private and public keys and use them for encryption and decrypting messages and finally to learn the generation of digital signatures and digital signature verification and manually verifying an X.509 certificate.

Table of Contents

1. Abstract	1
2. Theory.....	1
2.1. Introduction.....	1
2.2. Operation.....	1
2.2.1. Key Generation	2
2.2.2. Key Distribution	2
2.2.3. Encryption.....	2
2.2.4. Decryption	3
2.3. Digital Signature	3
2.4. X.509 Certificate	4
3. Procedure and Results.....	5
3.1. Creating BN Functions	5
3.2. Task 1: Deriving the Private Key	6
3.2.1. Code.....	6
3.2.2. Code Output	7
3.3. Task 2: Encrypting a Message.....	8
3.3.1. Code.....	8
3.3.2. Output.....	8
3.4. Task 3: Decrypting a Message	9
3.4.1. Code.....	9
3.4.2. Output.....	10
3.5. Task 4: Signing a Message	11
3.5.1. Code.....	11
3.5.2. Output.....	12
3.5.3. Code.....	13
3.5.4. Output.....	13
3.6. Task 5: Verifying a Signature	14
3.6.1. Code.....	14
3.6.3. Code.....	15
3.7. Task 6: Manually Verifying an X.509 Certificate.....	16
3.7.1. Download a certificate from a real web server	16

3.7.2. Extract the public key (e, n) from the issuer's certificate.....	18
3.7.3. Extract the signature from the server's certificate.....	19
3.7.4. Extract the body of the server's certificate.	21
3.7.5. Verify the signature	22
4. Conclusion	24
5. References	25

Table of Figures

Figure 2.1: Signing Messages.....	3
Figure 3.1: printBN, Private Key Generation, Encryption Functions Codes.	5
Figure 3.2: Hex2int, Hex2Ascii and printHX Functions Codes	5
Figure 3.3: Deriving the Private Key Code	6
Figure 3.4: Task 1 Output	7
Figure 3.5: Encrypting a Message Code.....	8
Figure 3.6: Encrypting a Message Output	8
Figure 3.7: Decrypting a Message Code	9
Figure 3.8: Decrypting a Message Output.....	10
Figure 3.9: Signing a Message Code	11
Figure 3.10: Signing a Message Output.....	12
Figure 3.11: Modifying One Bit of M	13
Figure 3.12: Signature for Modified Message	13
Figure 3.13: Verifying a Signature Code	14
Figure 3.14: Verifying a Signature Output.....	14
Figure 3.15: Verifying a Modified Signature Code.....	15
Figure 3.16: Verifying a Modified Signature Output	16
Figure 3.17: First Certificate of www.blank.com.....	17
Figure 3.18: Second Certificate of www.blank.com	17
Figure 3.19: Finding Modulus (n).....	18
Figure 3.20: Finding Exponent	19
Figure 3.21: Extracting Signature Value	20
Figure 3.22: Removing Colons and Spaces From Signature File.....	20
Figure 3.23: Extracting the Body of the Certificate	21
Figure 3.24: Calculating the Hash of the Certificate.....	22
Figure 3.25: Verifying the Signature Code.....	23
Figure 3.26: Verifying the Signature Output	23

2. Theory

2.1. Introduction

RSA (Rivest–Shamir–Adleman) is a public-key cryptosystem that is widely used for secure data transmission. It is also one of the oldest. The acronym "RSA" comes from the surnames of Ron Rivest, Adi Shamir and Leonard Adleman, who publicly described the algorithm in 1977. An equivalent system was developed secretly in 1973 at GCHQ (the British signals intelligence agency) by the English mathematician Clifford Cocks. That system was declassified in 1997. In a public-key cryptosystem, the encryption key is public and distinct from the decryption key, which is kept secret (private). An RSA user creates and publishes a public key based on two large prime numbers, along with an auxiliary value. The prime numbers are kept secret. Messages can be encrypted by anyone, via the public key, but can only be decoded by someone who knows the prime numbers. The security of RSA relies on the practical difficulty of factoring the product of two large prime numbers, the "factoring problem". Breaking RSA encryption is known as the RSA problem. Whether it is as difficult as the factoring problem is an open question. There are no published methods to defeat the system if a large enough key is used. RSA is a relatively slow algorithm. Because of this, it is not commonly used to directly encrypt user data. More often, RSA is used to transmit shared keys for symmetric-key cryptography, which are then used for bulk encryption–decryption.

2.2. Operation

The RSA algorithm involves four steps: key generation, key distribution, encryption, and decryption.

A basic principle behind RSA is the observation that it is practical to find three very large positive integers e , d , and n , such that with modular exponentiation for all integers m (with $0 \leq m < n$):

$$(m^e)^d \equiv m \pmod{n}$$

and that knowing e and n , or even m , it can be extremely difficult to find d . The triple bar (\equiv) here denotes modular congruence. (In simple terms, modular congruence means that when you divide $(m^e)^d$ by n and divide m by n , each has the same remainder.)

In addition, for some operations it is convenient that the order of the two exponentiations can be changed and that this relation also implies

$$(m^d)^e \equiv m \pmod{n}$$

RSA involves a public key and a private key. The public key can be known by everyone and is used for encrypting messages. The intention is that messages encrypted with the public key can only be decrypted in a reasonable amount of time by using the private key. The public key is represented by the integers n and e ,

and the private key by the integer d (although n is also used during the decryption process, so it might be considered to be a part of the private key too). m represents the message (previously prepared with a certain technique explained below).

2.2.1. Key Generation

- Need to generate: modulus n , public key exponent e , private key exponent d
- Approach
 - Choose p, q (large random prime numbers)
 - $n = pq$ (should be large)
 - Choose e , $1 < e < \phi(n)$ and e is relatively prime to $\phi(n)$
 - Find d , $ed \bmod \phi(n) = 1$
- Result
 - (e, n) is public key
 - d is private key

2.2.2. Key Distribution

Suppose that Bob wants to send information to Alice. If they decide to use RSA, Bob must know Alice's public key to encrypt the message, and Alice must use her private key to decrypt the message.

To enable Bob to send his encrypted messages, Alice transmits her public key (n, e) to Bob via a reliable, but not necessarily secret, route. Alice's private key (d) is never distributed.

2.2.3. Encryption

After Bob obtains Alice's public key, he can send a message M to Alice. To do it, he first turns M (strictly speaking, the un-padded plaintext) into an integer m (strictly speaking, the padded plaintext), such that $0 \leq m < n$ by using an agreed-upon reversible protocol known as a padding scheme. He then computes the ciphertext c , using Alice's public key e , corresponding to

$$c \equiv m^e \pmod{n}$$

This can be done reasonably quickly, even for very large numbers, using modular exponentiation. Bob then transmits c to Alice. Note that at least nine values of m will yield a ciphertext c equal to m , but this is very unlikely to occur in practice.

2.2.4. Decryption

Alice can recover m from c by using her private key exponent d by computing

$$c^d \equiv (m^e)^d \equiv m$$

Given m , she can recover the original message M by reversing the padding scheme.

2.3. Digital Signature

A digital signature is a mathematical scheme for verifying the authenticity of digital messages or documents. A valid digital signature, where the prerequisites are satisfied, gives a recipient very high confidence that the message was created by a known sender (authenticity), and that the message was not altered in transit (integrity).

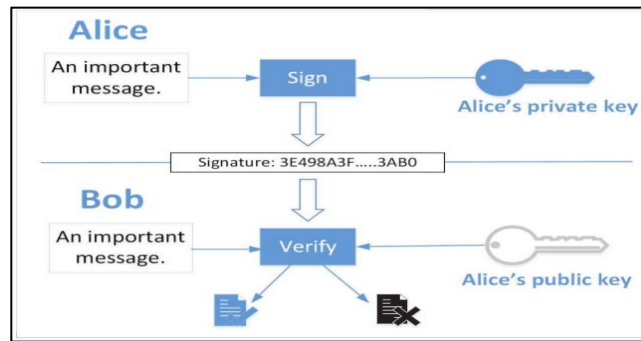


Figure 2.1: Signing Messages

For a message m that needs to be signed:

$$\text{Digital signature} = m^d \text{ mod } n$$

In practice, message may be long resulting in long signature and more computing time. Instead, we generate a cryptographic hash value from the original message, and only sign the hash.

Attackers cannot generate a valid signature from a modified message because they do not know the private key. If attackers modify the message, the hash will change and it will not be able to match with the hash produced from the signature verification as will be noticed in the lab.

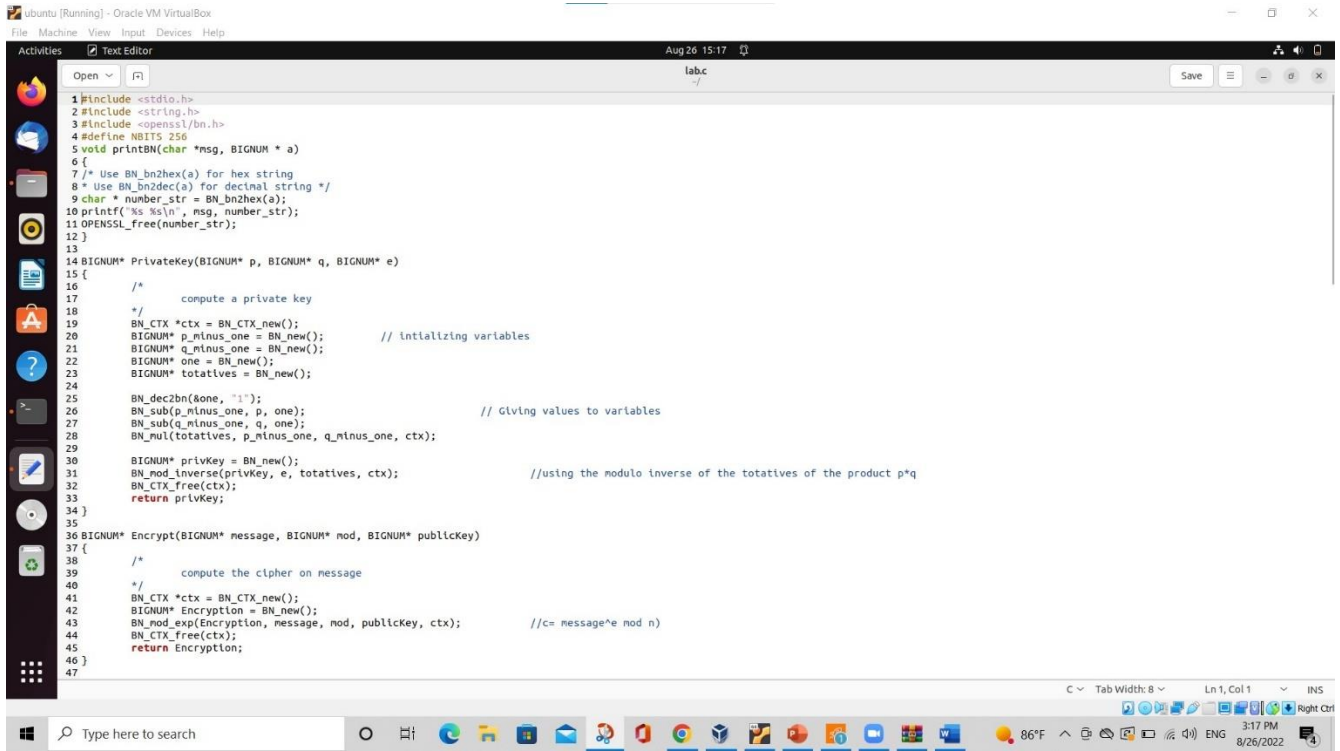
2.4. X.509 Certificate

An X.509 certificate is a digital certificate based on the widely accepted International Telecommunications Union (ITU) X.509 standard, which defines the format of public key infrastructure (PKI) certificates. They are used to manage identity and security in internet communications and computer networking. They are unobtrusive and ubiquitous, and we encounter them every day when using websites, mobile apps, online documents, and connected devices.

One of the structural strengths of the X.509 certificate is that it is architected using a key pair consisting of a related public key and a private key. Applied to cryptography, the public and private key pair is used to encrypt and decrypt a message, ensuring both the identity of the sender and the security of the message itself. The most common use case of X.509-based PKI is Transport Layer Security (TLS)/Secure Socket Layer (SSL), which is the basis of the HTTPS protocol, which enables secure web browsing. But the X.509 protocol is also applied to code signing for application security, digital signatures, and other critical internet protocols. In this lab, we will manually verify an X.509 certificate using our program

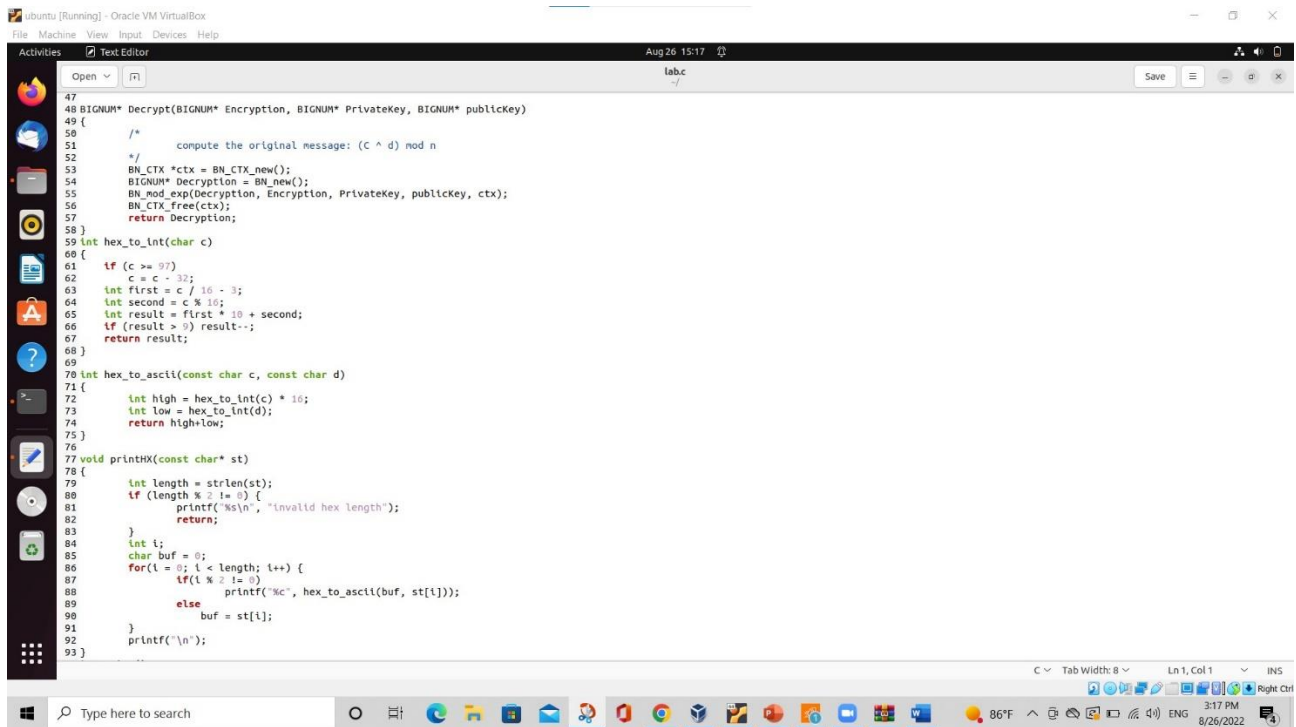
3. Procedure and Results

3.1. Creating BN Functions



```
1#include <stdio.h>
2#include <string.h>
3#include <openssl/bn.h>
4#define NBITS 256
5void printBN(char *msg, BIGNUM * a)
6{
7    /* Use BN_bn2hex(a) for hex string
8    * Use BN_bn2dec(a) for decimal string */
9    char * number_str = BN_bn2hex(a);
10   printf("%s %s\n", msg, number_str);
11   OPENSSL_free(number_str);
12}
13
14BIGNUM* PrivateKey(BIGNUM* p, BIGNUM* q, BIGNUM* e)
15{
16    /*
17    * compute a private key
18    */
19    BN_CTX *ctx = BN_CTX_new();
20    BIGNUM* p_minus_one = BN_new();
21    BIGNUM* q_minus_one = BN_new();
22    BIGNUM* one = BN_new();
23    BIGNUM* totatives = BN_new();
24
25    BN_dec2bn(&one, "1");
26    BN_sub(p_minus_one, p, one);
27    BN_sub(q_minus_one, q, one);
28    BN_mul(totatives, p_minus_one, q_minus_one, ctx);
29
30    BIGNUM* privKey = BN_new();
31    BN_mod_inverse(privKey, e, totatives, ctx);
32    BN_CTX_free(ctx);
33    return privKey;
34}
35
36BIGNUM* Encrypt(BIGNUM* message, BIGNUM* mod, BIGNUM* publicKey)
37{
38    /*
39    * compute the cipher on message
40    */
41    BN_CTX *ctx = BN_CTX_new();
42    BIGNUM* Encryption = BN_new();
43    BN_mod_exp(Encryption, message, mod, publicKey, ctx);
44    BN_CTX_free(ctx);
45    return Encryption;
46}
47
```

Figure 3.1: printBN, Private Key Generation, Encryption Functions Codes.



```
47
48BIGNUM* Decrypt(BIGNUM* Encryption, BIGNUM* PrivateKey, BIGNUM* publicKey)
49{
50    /*
51    * compute the original message: (C ^ d) mod n
52    */
53    BN_CTX *ctx = BN_CTX_new();
54    BIGNUM* Decryption = BN_new();
55    BN_mod_exp(Decryption, Encryption, PrivateKey, publicKey, ctx);
56    BN_CTX_free(ctx);
57    return Decryption;
58}
59int hex_to_int(char c)
60{
61    if (c >= '0')
62        c = c - '0';
63    int first = c / 16 - 3;
64    int second = c % 16;
65    int result = first * 16 + second;
66    if (result > 0) result--;
67    return result;
68}
69
70int hex_to_ascii(const char c, const char d)
71{
72    int high = hex_to_int(c) * 16;
73    int low = hex_to_int(d);
74    return high+low;
75}
76
77void printHX(const char* st)
78{
79    int length = strlen(st);
80    if (length % 2 != 0) {
81        printf("%s\n", "invalid hex length");
82        return;
83    }
84    int i;
85    char buf = 0;
86    for(i = 0; i < length; i++) {
87        if(i % 2 != 0)
88            printf("%c", hex_to_ascii(buf, st[i]));
89        else
90            buf = st[i];
91    }
92    printf("\n");
93}

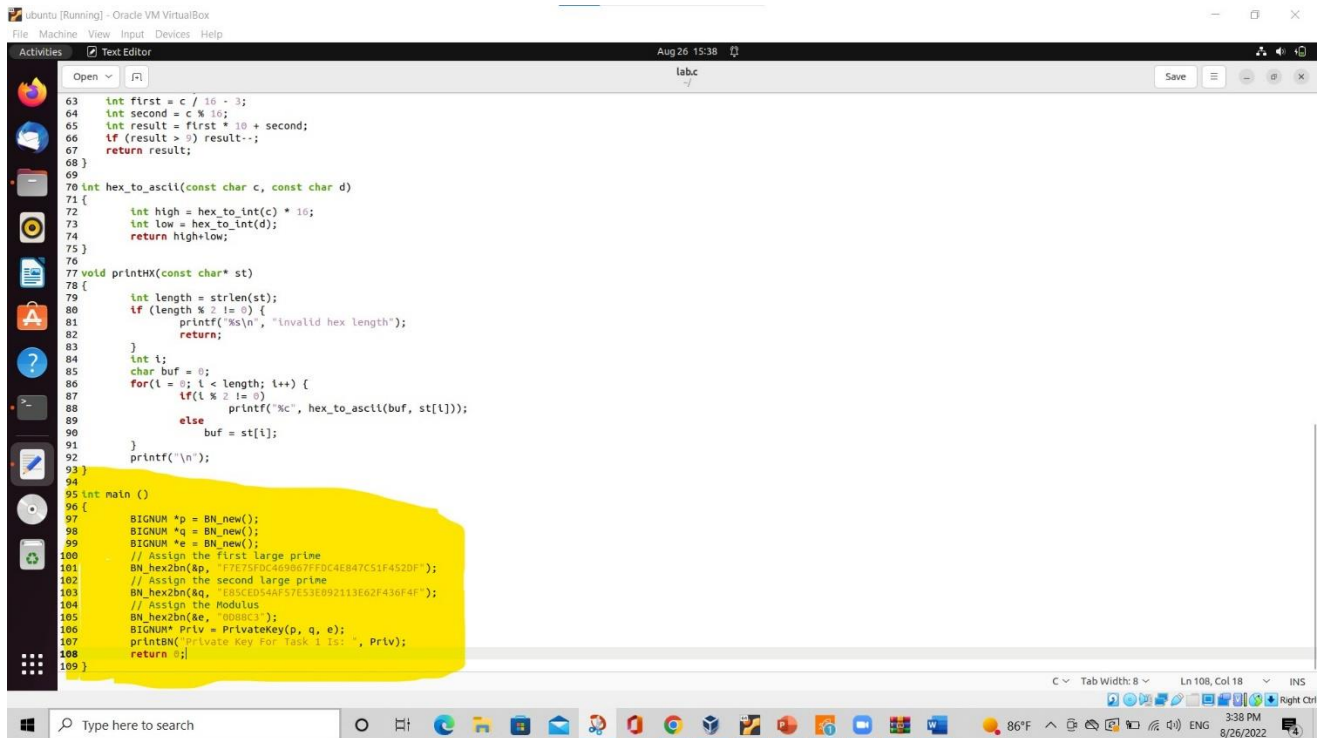
```

Figure 3.2: Hex2int, Hex2Ascii and printHX Functions Codes

printBN function was built to print the big numbers values using dynamic allocating, Hex2int was used to convert Hexadecimal values to integers and Hex2Ascii was used to change Hexadecimal values to Ascii which both were used in printHX function the change Hexadecimal values into characters for decryption. Private Key function was used to generate the private key for the RSA using the inverse totatives method and the encryption and decryption functions were built using the methods discussed in the theoretical part.

3.2. Task 1: Deriving the Private Key

3.2.1. Code



```
63 int first = c / 10 - 3;
64 int second = c % 10;
65 int result = first * 10 + second;
66 if (result > 9) result--;
67 return result;
68 }
69
70 int hex_to_ascii(const char c, const char d)
71 {
72     int high = hex_to_int(c) * 16;
73     int low = hex_to_int(d);
74     return high+low;
75 }
76
77 void printHX(const char* st)
78 {
79     int length = strlen(st);
80     if (length % 2 != 0) {
81         printf("%s\n", "invalid hex length");
82         return;
83     }
84     int i;
85     char buf = 0;
86     for(i = 0; i < length; i++) {
87         if(i % 2 != 0)
88             printf("%c", hex_to_ascii(buf, st[i]));
89         else
90             buf = st[i];
91     }
92     printf("\n");
93 }
94
95 int main ()
96 {
97     BIGNUM *p = BN_new();
98     BIGNUM *q = BN_new();
99     BIGNUM *e = BN_new();
100    // Assign the first large prime
101    BN_hex2bn(&p, "F7E75F0C46967FFDC4E847C31F452DF");
102    // Assign the second large prime
103    BN_hex2bn(&q, "E83ED944F5FES3E092113E62F430F4F");
104    // Assign the Modulus
105    BN_hex2bn(&e, "0DBBC3");
106    BIGNUM* Prlv = PrivateKey(p, q, e);
107    printBN("Private Key For Task 1 is: ", Prlv);
108    return 0;
109 }
```

Figure 3.3: Deriving the Private Key Code

3.2.2. Code Output

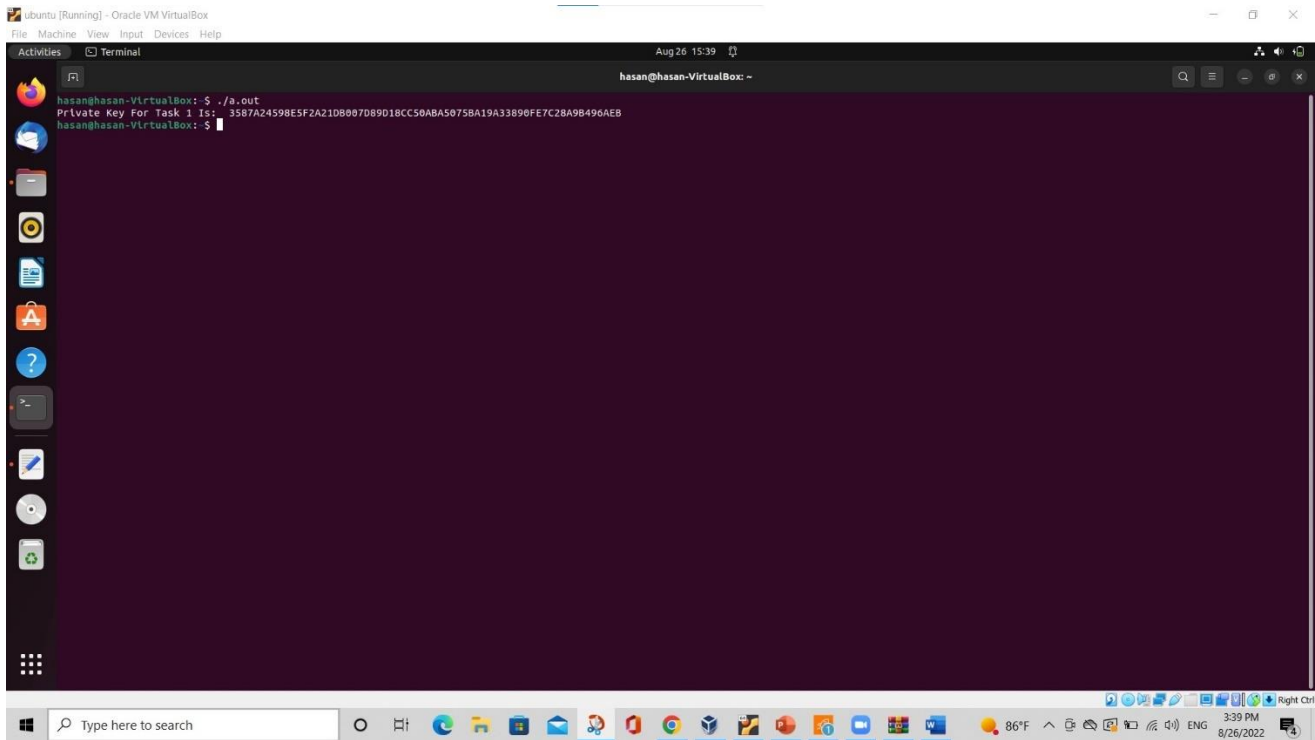
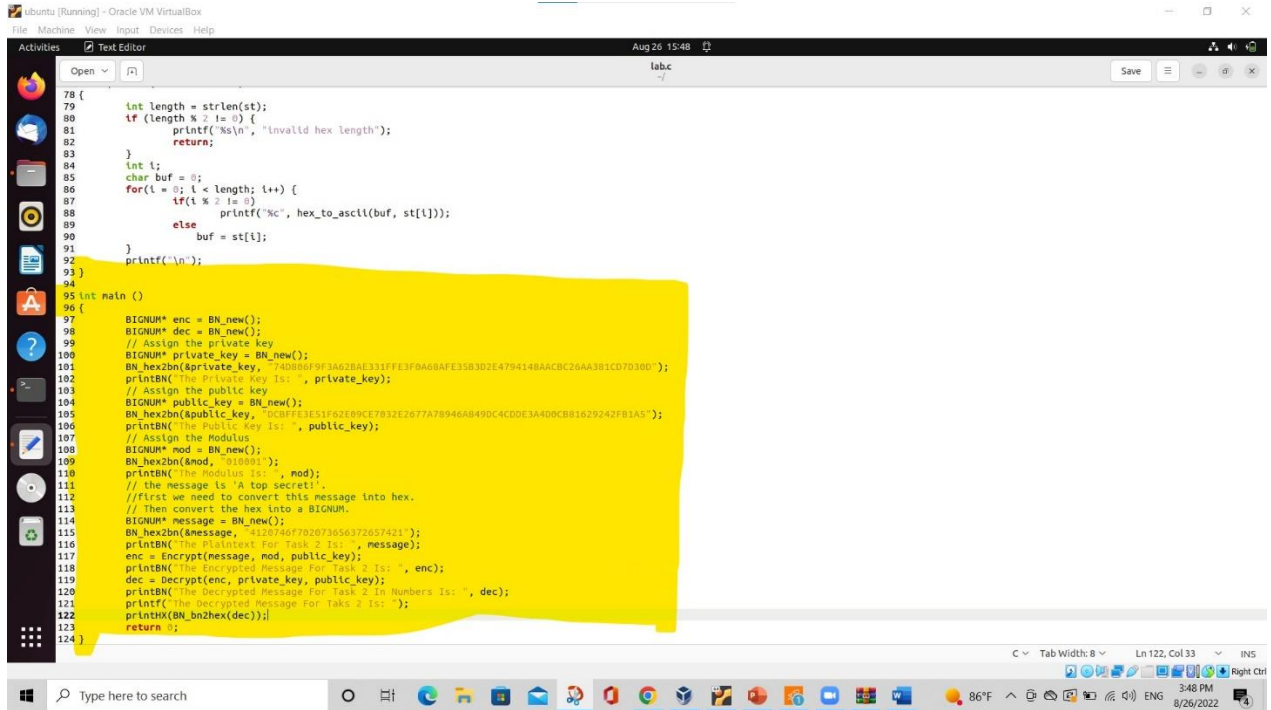


Figure 3.4: Task 1 Output

Given the values of p, q and e the value of private key, d was calculated after changing all of the values to Bignum as shown in Figure 3.3. The output of the code after it gets executed will be the private key as shown in Figure 3.4.

3.3. Task 2: Encrypting a Message

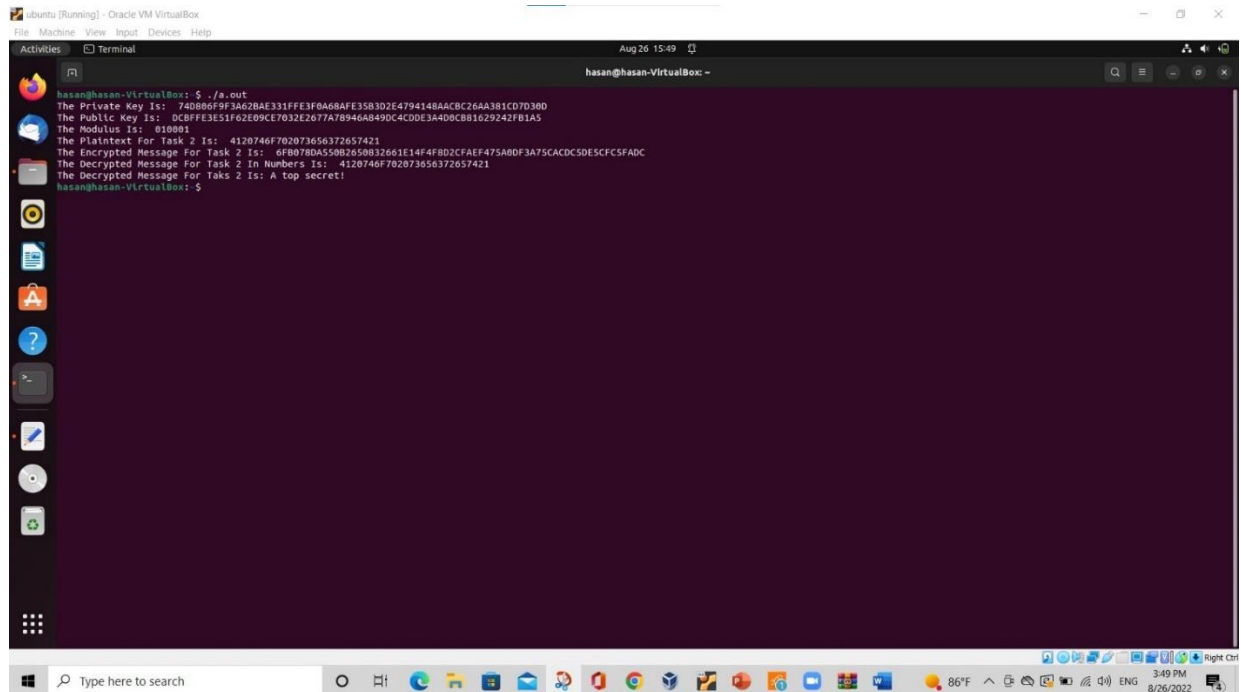
3.3.1. Code



```
78 {
79     int length = strlen(st);
80     if (length % 2 != 0) {
81         printf("%s\n", "Invalid hex length");
82         return;
83     }
84     int i;
85     char buf = 0;
86     for(i = 0; i < length; i++) {
87         if(i % 2 == 0)
88             printf("%c", hex_to_ascii(buf, st[i]));
89         else
90             buf = st[i];
91     }
92     printf("\n");
93 }
94
95 int main ()
96 {
97     BIGNUM* enc = BN_new();
98     BIGNUM* dec = BN_new();
99     // Assign the private key
100     BIGNUM* private_key = BN_new();
101     BN_hex2bn(&private_key, "740866f9f3a62bae331ffe3f0a68afe35b3d2e4794148aacbc26aa381cd70300");
102     printf("The Private Key Is: ", private_key);
103     // Assign the public key
104     BIGNUM* public_key = BN_new();
105     BN_hex2bn(&public_key, "dc0ffe3e51f62e99ce7832e2677a78946a8490c4c0de3a400cb81629242fb1a5");
106     printf("The Public Key Is: ", public_key);
107     // Assign the Modulus
108     BIGNUM* mod = BN_new();
109     BN_hex2bn(&mod, "010901");
110     printf("The Modulus Is: ", mod);
111     // the message is "A top secret!"
112     // first we need to convert this message into hex.
113     // Then convert the hex into a BIGNUM.
114     BIGNUM* message = BN_new();
115     BN_hex2bn(&message, "4120746f702073656372657421");
116     printf("The Plaintext For Task 2 Is: ", message);
117     enc = Encrypt(message, mod, public_key);
118     printf("The Encrypted Message For Task 2 Is: ", enc);
119     dec = Decrypt(enc, private_key, public_key);
120     printf("The Decrypted Message For Task 2 In Numbers Is: ", dec);
121     printf("The Decrypted Message For Taks 2 Is: ");
122     printf(BN_bn2hex(dec));
123     return 0;
124 }
```

Figure 3.5: Encrypting a Message Code

3.3.2. Output



```
hasan@hasan-VirtualBox: ~$ ./a.out
The Private Key Is: 740866f9f3a62bae331ffe3f0a68afe35b3d2e4794148aacbc26aa381cd70300
The Public Key Is: dc0ffe3e51f62e99ce7832e2677a78946a8490c4c0de3a400cb81629242fb1a5
The Modulus Is: 010901
The Plaintext For Task 2 Is: 4120746f702073656372657421
The Encrypted Message For Task 2 Is: 6f8078da5508260832661e14f4f8d2cfaef475a0df3a753acd5desfcfcfadc
The Decrypted message For Task 2 In Numbers Is: 4120746f702073656372657421
The Decrypted Message For Taks 2 Is: A top secret!
```

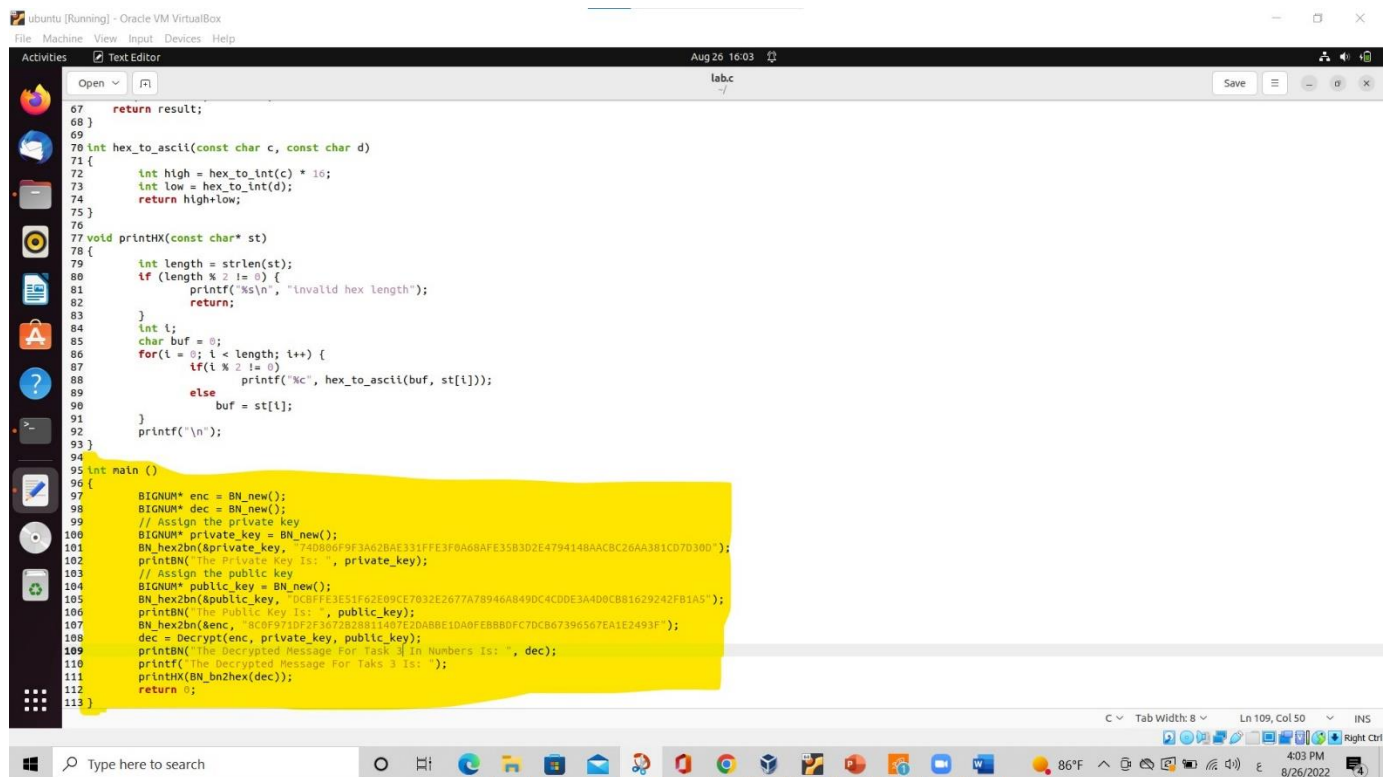
Figure 3.6: Encrypting a Message Output

In this task, the message "A top secret!" was encrypted after converting the text into hexadecimal using any converter and using the encryption function the message was encrypted using the given public keys, and after encrypting the message the given private key was used to decrypt the message to assure that it was encrypted correctly as shown in Figure 3.5.

And as shown in the output shown in Figure 3.6 after excuting the code that both the decrypted message and the original message were identical which means that the encryption was correct and it also shows the encryption of the message.

3.4. Task 3: Decrypting a Message

3.4.1. Code



```
67 return result;
68 }
69
70 int hex_to_ascii(const char c, const char d)
71 {
72     int high = hex_to_int(c) * 16;
73     int low = hex_to_int(d);
74     return high+low;
75 }
76
77 void printHX(const char* st)
78 {
79     int length = strlen(st);
80     if (length % 2 != 0) {
81         printf("%s\n", "invalid hex length");
82         return;
83     }
84     int i;
85     char buf = 0;
86     for(i = 0; i < length; i++) {
87         if(i % 2 != 0)
88             printf("%c", hex_to_ascii(buf, st[i]));
89         else
90             buf = st[i];
91     }
92     printf("\n");
93 }
94
95 int main ()
96 {
97     BIGNUM* enc = BN_new();
98     BIGNUM* dec = BN_new();
99     // Assign the private key
100    BIGNUM* private_key = BN_new();
101    BN_hex2bn(&private_key, "740806f9f3a02bae31ffe3f0a60afe35b302e4794148aacbc26aa381cd70300");
102    printBN("The Private Key Is: ", private_key);
103    // Assign the public key
104    BIGNUM* public_key = BN_new();
105    BN_hex2bn(&public_key, "dcbffe3e51f62e99ce7932e2677a78946a849dc4c0de3a400cb81629242fb1a5");
106    printBN("The Public Key Is: ", public_key);
107    BN_hex2bn(&enc, "8cbf971df2f3672828811407e2dabbe1da0feb8bdfc7dcb67396567ea1e2493f");
108    dec = Decrypt(enc, private_key, public_key);
109    printBN("The Decrypted Message For Task 3 In Numbers Is: ", dec);
110    printf("The Decrypted Message For Task 3 Is: ");
111    printHX(BN_bn2hex(dec));
112    return 0;
113 }
```

Figure 3.7: Decrypting a Message Code

3.4.2. Output

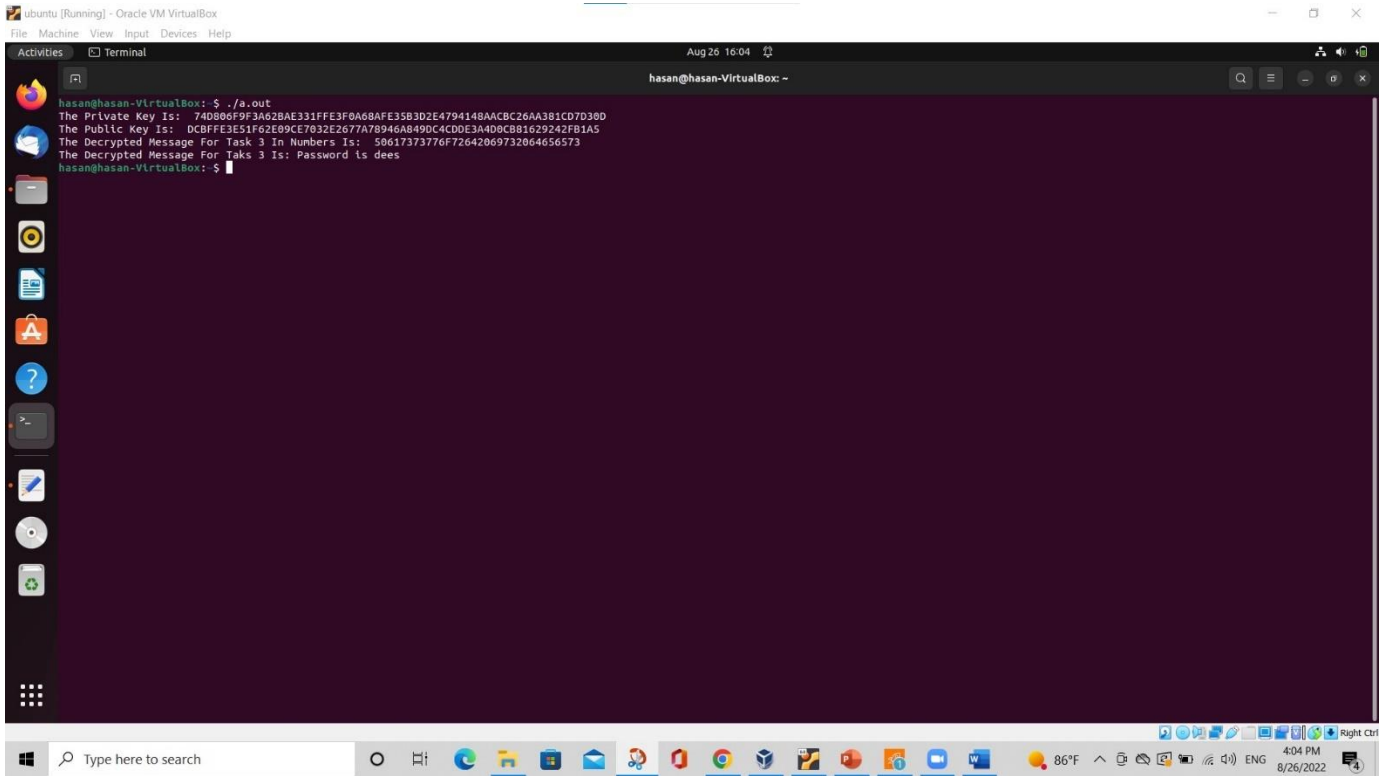
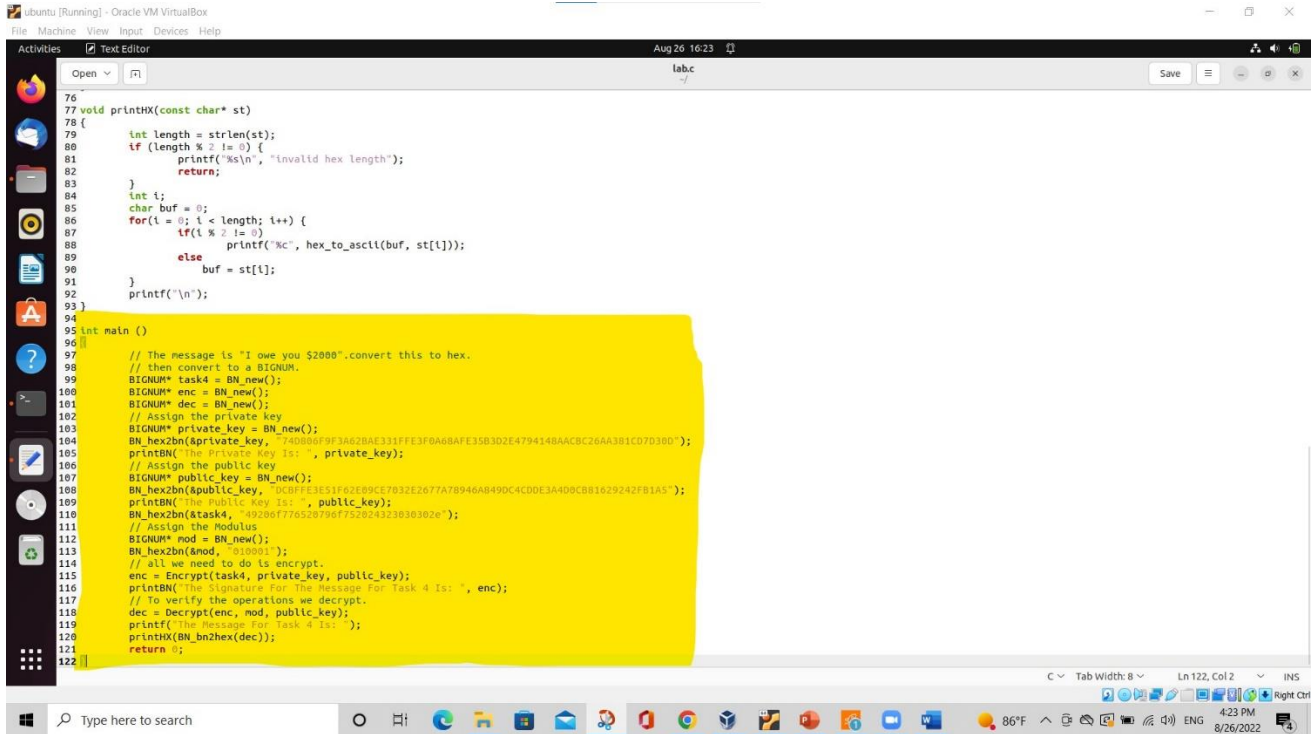


Figure 3.8: Decrypting a Message Output

The public and private keys used in this task are the same as the ones used in Task 2. The cipher c given was decrypted using the decryption function shown in the code in Figure 3.7 and then the output of the decryption was converted from hex to text as shown in the output shown in Figure 3.8 after the code execution.

3.5. Task 4: Signing a Message

3.5.1. Code



```
76
77 void printHX(const char* st)
78 {
79     int length = strlen(st);
80     if (length % 2 != 0) {
81         printf("%s\n", "invalid hex length");
82         return;
83     }
84     int i;
85     char buf = 0;
86     for(i = 0; i < length; i++) {
87         if(i % 2 != 0)
88             printf("%c", hex_to_ascii(buf, st[i]));
89         else
90             buf = st[i];
91     }
92     printf("\n");
93 }
94
95 int main ()
96 {
97     // The message is "I owe you $2000".convert this to hex.
98     // then convert to a BIGNUM.
99     BIGNUM* task4 = BN_new();
100    BIGNUM* enc = BN_new();
101    BIGNUM* dec = BN_new();
102    // Assgn the private key
103    BIGNUM* private_key = BN_new();
104    BN_hex2bn(&private_key, "740806f9f3a62bae331ffe3f0a68afe35b3d2e4794148aacbc26aa381cd07d300");
105    printf("The private key is: ", private_key);
106    // Assgn the public key
107    BIGNUM* public_key = BN_new();
108    BN_hex2bn(&public_key, "dcbffe3e51f62e89ce7032e2677a78946a8490c4c0de3a400c8b1629242fb1a5");
109    printf("The public key is: ", public_key);
110    BN_hex2bn(&task4, "49200f776329f96f752624323830302e");
111    // Assgn the Modulus
112    BIGNUM* mod = BN_new();
113    BN_hex2bn(&mod, "010001");
114    // all we need to do is encrypt.
115    enc = Encrypt(task4, private_key, public_key);
116    printf("The Signature For The Message For Task 4 is: ", enc);
117    // To verify the operations we decrypt.
118    dec = Decrypt(enc, mod, public_key);
119    printf("The Message For Task 4 is: ");
120    printHX(BN_bn2hex(dec));
121    return 0;
122 }
```

Figure 3.9: Signing a Message Code

3.5.2. Output

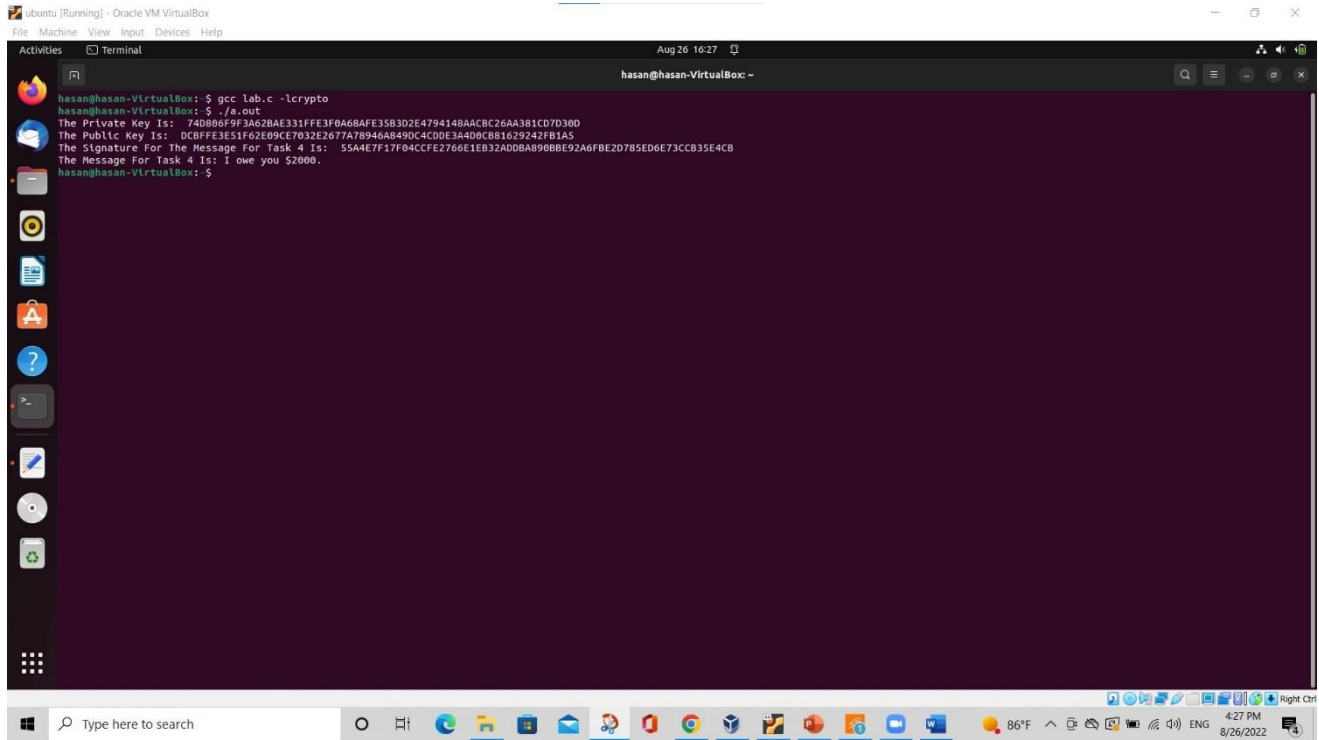


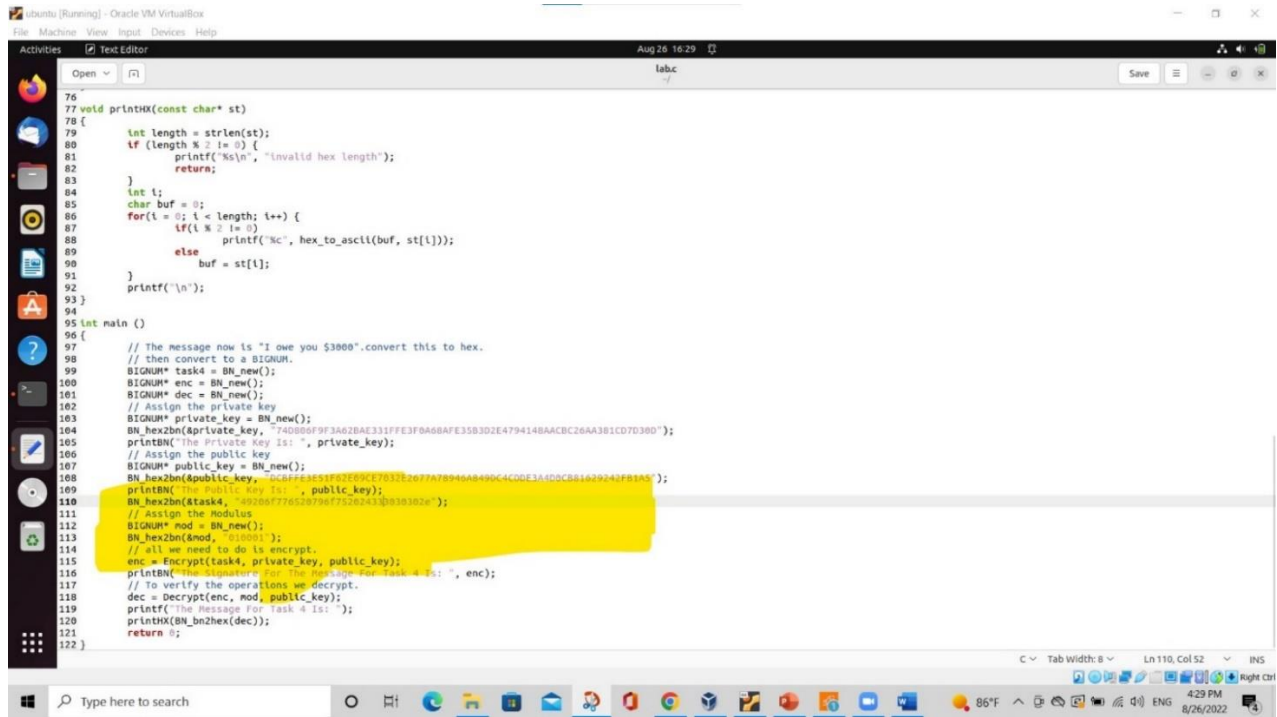
Figure 3.10: Signing a Message Output

The public and private keys used in this task are the same as the ones used in Task 2. A signature for the following message M = “I owe you \$2000” was generated.

The hex value of the message “I owe you \$2000.” was observed using websites. The code shown in Figure 3.9 was executed and its output is shown in Figure 3.10.

Now, the hex value of the message “I owe you \$3000.” was observed and the message was modified as shown in Figure 3.11, and after running the code the output was observed as shown in Figure 3.12.

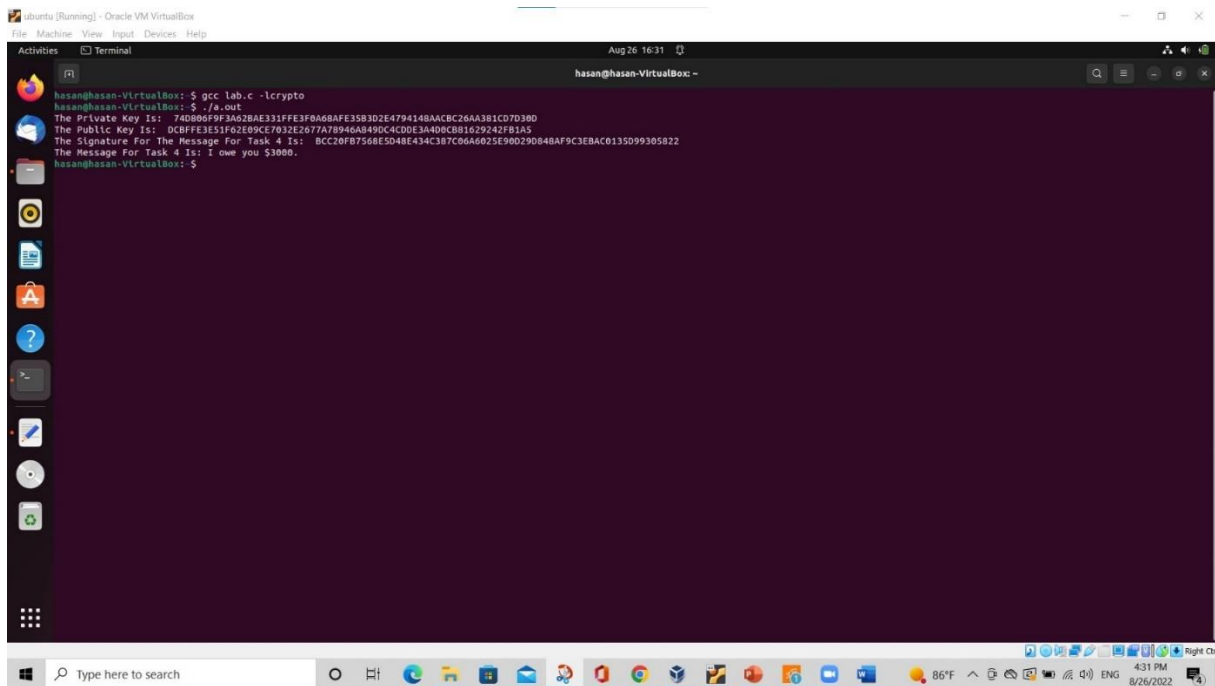
3.5.3. Code



```
76
77 void printHex(const char* st)
78 {
79     int length = strlen(st);
80     if (length % 2 != 0) {
81         printf("%s\n", "Invalid hex length");
82         return;
83     }
84     int i;
85     char buf = 0;
86     for(i = 0; i < length; i++) {
87         if(i % 2 != 0)
88             printf("%c", hex_to_ascii(buf, st[i]));
89         else
90             buf = st[i];
91     }
92     printf("\n");
93 }
94
95 int main ()
96 {
97     // The message now is "I owe you $3000".convert this to hex.
98     // then convert to a BIGNUM.
99     BIGNUM* task4 = BN_new();
100    BIGNUM* enc = BN_new();
101    BIGNUM* dec = BN_new();
102    // Assign the private key
103    BIGNUM* private_key = BN_new();
104    BN_hex2bn(&private_key, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E479414BAACBC26AA381CD7D30D");
105    printf("The Private Key Is: ", private_key);
106    // Assign the public key
107    BIGNUM* public_key = BN_new();
108    BN_hex2bn(&public_key, "DCBFFE3E51F02E09CE7032E2677A78946A849DC4DDE3A40BC81029242FB1A5");
109    printf("The Public Key Is: ", public_key);
110    BN_hex2bn(&task4, "B028F776529798F75263434018302e");
111    // Assign the Modulus
112    BIGNUM* mod = BN_new();
113    BN_hex2bn(mod, "B028F776529798F75263434018302e");
114    // all we need to do is encrypt.
115    enc = Encrypt(task4, private_key, public_key);
116    printf("The signature for the message for task 4 is: ", enc);
117    // To verify the operations we decrypt.
118    dec = Decrypt(enc, mod, public_key);
119    printf("The Message For Task 4 Is: ");
120    printf(BN_bn2hex(dec));
121    return 0;
122 }
```

Figure 3.11: Modifying One Bit of M

3.5.4. Output



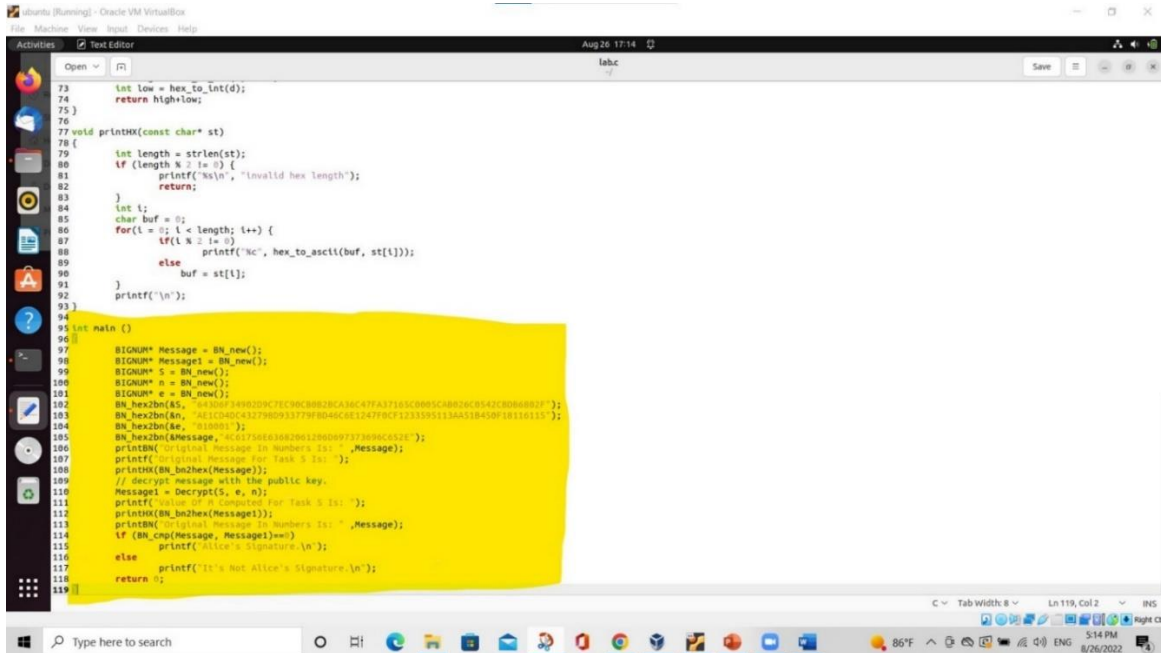
```
hasan@hasan-VirtualBox:~$ gcc lab.c -lcrypto
hasan@hasan-VirtualBox:~$ ./a.out
The Private Key Is: 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E479414BAACBC26AA381CD7D30D
The Public Key Is: DCBFFE3E51F02E09CE7032E2677A78946A849DC4DDE3A40BC81029242FB1A5
The Signature For The Message For Task 4 Is: BC28F87568E5D4BE434C387C6A6623E900290848AF9C3EBAC0135D99305822
The Message For Task 4 Is: I owe you $3000.
```

Figure 3.12: Signature for Modified Message

It was surprising that only one byte of difference in the message their signatures were completely different.

3.6. Task 5: Verifying a Signature

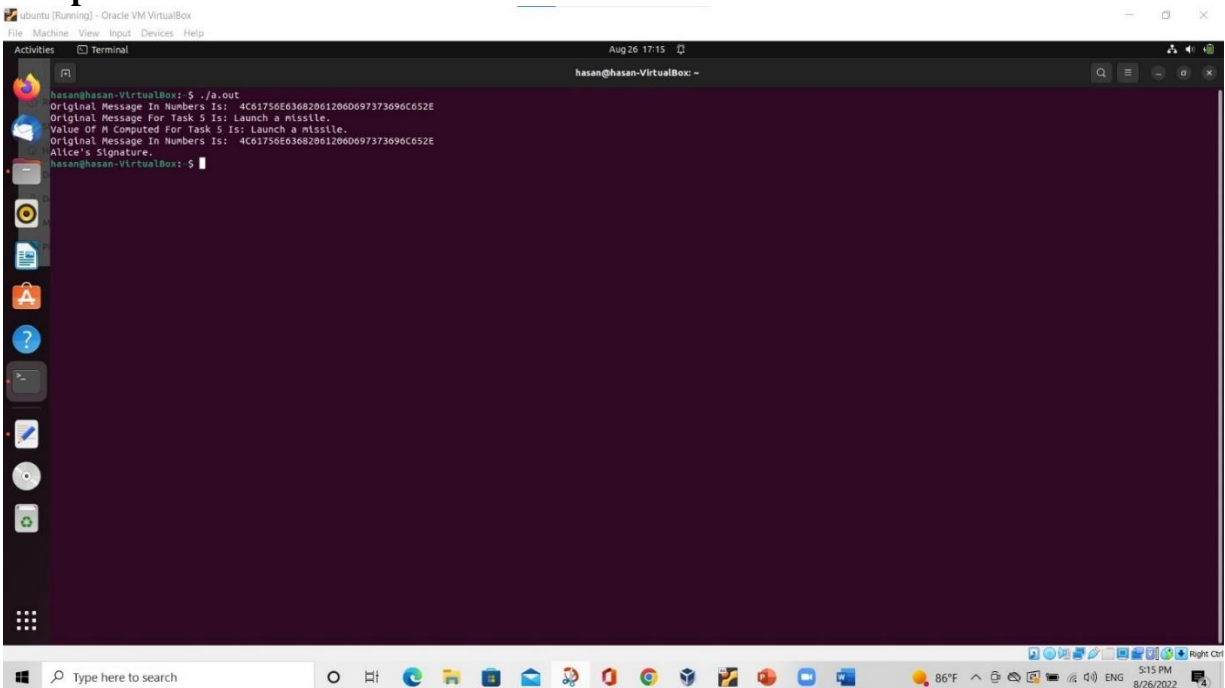
3.6.1. Code



```
73 int low = hex_to_int(d);
74 return high+low;
75 }
76
77 void printX(const char* st)
78 {
79     int length = strlen(st);
80     if (length % 2 != 0) {
81         printf("Ns\n", "Invalid hex length");
82         return;
83     }
84     int i;
85     char buf = 0;
86     for(i = 0; i < length; i++) {
87         if(i % 2 != 0)
88             printf("%c", hex_to_ascll(buf, st[i]));
89         else
90             buf = st[i];
91     }
92     printf("\n");
93 }
94
95 int main ()
96 {
97     BIGNUM* Message = BN_new();
98     BIGNUM* Message2 = BN_new();
99     BIGNUM* S = BN_new();
100     BIGNUM* n = BN_new();
101     BIGNUM* e = BN_new();
102     BN_hex2bn(&n, "40154003490209C7E39038028CA36C47A37165C0095C4003C8542C8064802F");
103     BN_hex2bn(&e, "10001");
104     BN_hex2bn(&S, "4E1CD40C432790D933779F8D46C8E124770CF1233595113AA51B450F1816115");
105     BN_hex2bn(&Message, "4C6175E63682061206D6973736996C652");
106     printf("Original Message For Task 5 Is: ", Message);
107     printf("Original Message Is: ");
108     printX(BN_bn2hex(Message));
109     // decrypt message with the public key.
110     Message2 = Decrypt(S, e, n);
111     printf("Original Message For Task 5 Is: ");
112     printX(BN_bn2hex(Message2));
113     printf("Original Message In Numbers Is: ", Message);
114     if (BN_cmp(Message, Message2) == 0)
115         printf("Alice's Signature.\n");
116     else
117         printf("It's Not Alice's Signature.\n");
118     return 0;
119 }
```

Figure 3.13: Verifying a Signature Code

3.6.2. Output



```
hasan@hasan-VirtualBox: ~$ ./a.out
Original Message In Numbers Is: 4C6175E63682061206D6973736996C652E
Original Message For Task 5 Is: Launch a missile.
Value Of M Computed For Task 5 Is: Launch a missile.
Original Message In Numbers Is: 4C6175E63682061206D6973736996C652E
Alice's Signature.
hasan@hasan-VirtualBox: ~$
```

Figure 3.14: Verifying a Signature Output

Bob receives a message $M = \text{"Launch a missile."}$ from Alice, with her signature S . Alice's public key is (e, n) . The signature was verified using the decryption function by decrypting the signature using the public key as shown in the code shown in Figure 3.13, and by comparing the decrypted message with the original message if they were equal which means it's Alice signature else it's not. The output of the code after executing it is shown in Figure 3.14.

The last byte of the signature was changed from 2F to 3F and the code was modified as shown in Figure 3.15. The output was observed as shown in Figure 3.16.

3.6.3. Code

```

74     return high+low;
75 }
76
77 void printHX(const char* st)
78 {
79     int length = strlen(st);
80     if (length % 2 != 0) {
81         printf("%s\n", "Invalid hex length");
82         return;
83     }
84     int i;
85     char buf = 0;
86     for (i = 0; i < length; i++) {
87         if (i % 2 != 0)
88             printf("%c", hex_to_ascii(buf, st[i]));
89         else
90             buf = st[i];
91     }
92     printf("\n");
93 }
94
95 int main ()
96 {
97     BIGNUM* Message = BN_new();
98     BIGNUM* Message1 = BN_new();
99     BIGNUM* S = BN_new();
100    BIGNUM* n = BN_new();
101    BIGNUM* e = BN_new();
102    // Now Changing Signature from 2F to 3F
103    BN_hex2bn(&S, "64306f34902d9c7ec90c80b28ca36c47fa37165c0005cabb26c0542c80b6803f");
104    BN_hex2bn(&n, "a1cd40c43279b0933779f8d46c6e1247f8cf123359513aas1b450f18116415");
105    BN_hex2bn(&e, "010001");
106    BN_hex2bn(&Message, "4c01750e630820612060697373696c652e");
107    printBN("Original Message In Numbers Is: ", Message);
108    printf("Original Message For Task 5 Is: ");
109    printHX(BN_bn2hex(Message));
110    // decrypt message with the public key.
111    Message1 = Decrypt(S, e, n);
112    printf("Value Of M Computed For Task 5 Is: ");
113    printHX(BN_bn2hex(Message1));
114    printBN("Original Message In Numbers Is: ", Message);
115    if (BN_cmp(Message, Message1) == 0)
116        printf("Alice's Signature.\n");
117    else
118        printf("It's Not Alice's Signature.\n");
119    return 0;
120 }

```

Figure 3.15: Verifying a Modified Signature Code

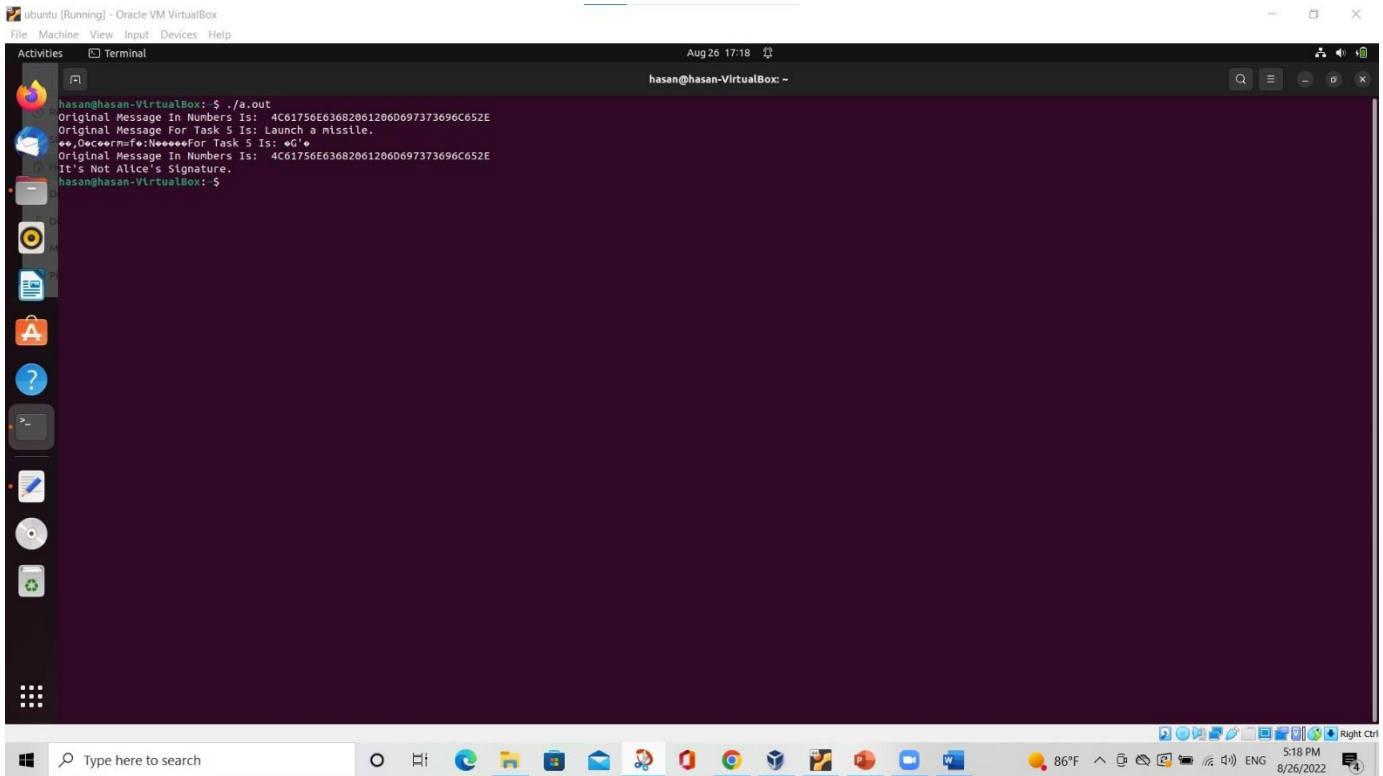


Figure 3.16: Verifying a Modified Signature Output

It was surprising that changing one byte of the signatures results changing the whole decrypted message completely which results a failed verification.

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

3.7.1. Download a certificate from a real web server

The certificate of www.blank.com was observed using the following command as shown in Figure 3.17 and 3.18:

```
$ openssl s_client -connect www.example.org:443 -showcerts
```

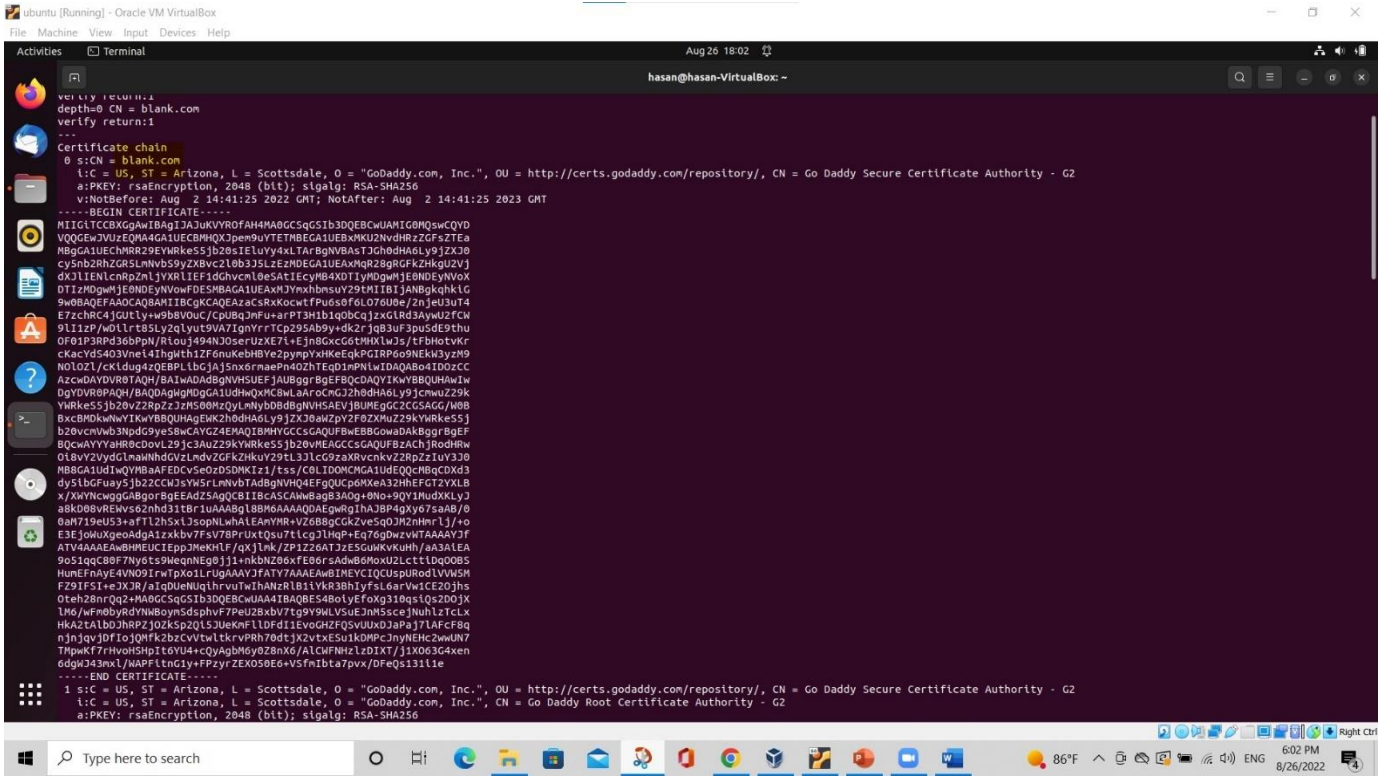


Figure 3.17: First Certificate of www.blank.com

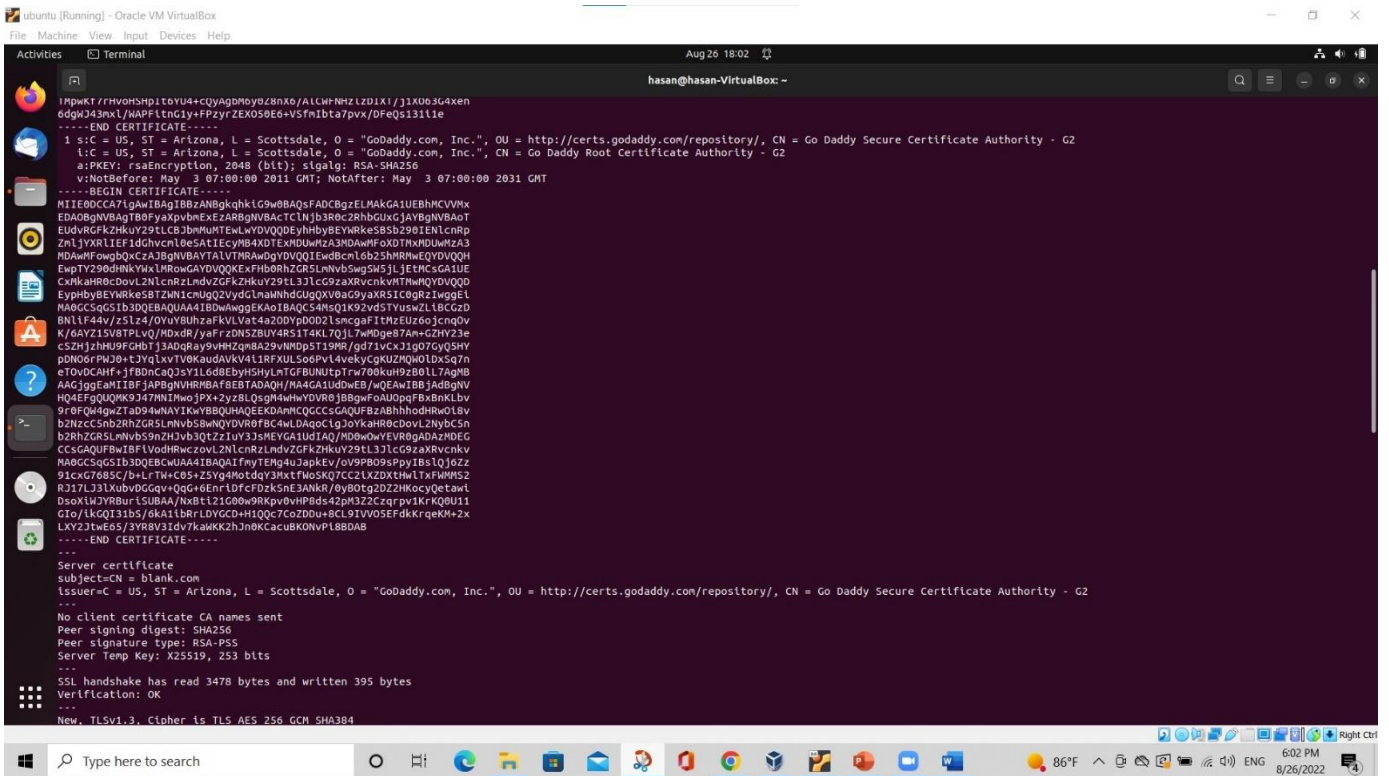


Figure 3.18: Second Certificate of www.blank.com

Each of the certificate (the text between the line containing "Begin CERTIFICATE" and the line containing "END CERTIFICATE", including these two lines) were copied and pasted to a file. c0.pem and c1.pem.

3.7.2. Extract the public key (e, n) from the issuer's certificate

For modulus (n) the following command was used:

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

For the exponent, all the fields were printed out using the following command:

```
$ openssl x509 -in c1.pem -text -noout
```

After printing all the fields, e was observed and n was observed using the first command both results are shown in figure 3.19 and 3.20.

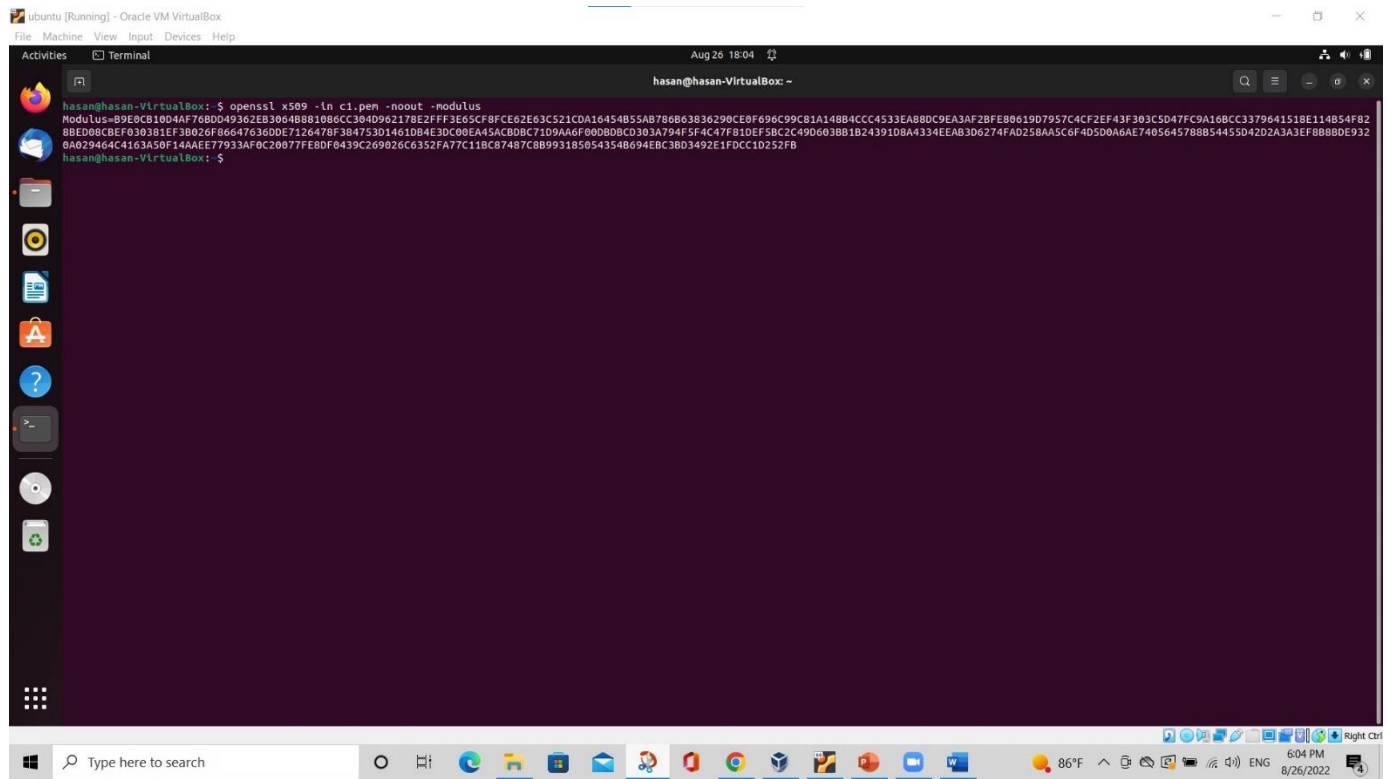


Figure 3.19: Finding Modulus (n)

```

libuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal
hasan@hasan-VirtualBox: -
hasan@hasan-VirtualBox:~$ openssl x509 -in c1.pem -text -noout
Certificate:
Data:
  Version: 3 (0x2)
  Serial Number: 7 (0x7)
  Signature Algorithm: sha256WithRSAEncryption
  Issuer: C = US, ST = Arizona, L = Scottsdale, O = "GoDaddy.com, Inc.", CN = Go Daddy Root Certificate Authority - G2
  Validity
    Not Before: May  3 07:00:00 2011 GMT
    Not After : May  3 07:00:00 2031 GMT
  Subject: C = US, ST = Arizona, L = Scottsdale, O = "GoDaddy.com, Inc.", OU = http://certs.godaddy.com/repository/, CN = Go Daddy Secure Certificate Authority - G2
  Subject Public Key Info:
    Public Key Algorithm: rsaEncryption
    Public-Key: (2048 bit)
    Modulus:
      00:b9:e0:cb:10:d4:af:76:bd:d4:93:62:eb:30:64:
      b8:81:08:6c:c3:04:d9:62:17:8e:2f:ff:3e:65:cf:
      8f:ce:62:e6:3c:52:1c:da:16:45:4b:55:ab:78:0b:
      69:83:62:90:cc:0f:f6:9d:99:c8:1a:14:0b:4c:cc:
      45:33:ea:88:dc:9e:a3:af:2b:fe:80:01:9d:79:57:
      c4:cf:2e:f4:3f:30:3c:5d:47:fc:9a:16:bc:c3:37:
      96:41:91:8e:11:4b:54:f8:28:be:d0:8c:be:f8:30:
      30:1e:f3:90:26:f8:66:4f:63:6d:0e:71:26:47:8f:
      38:47:53:d1:46:1d:b4:e3:dc:80:ea:45:ac:bd:bc:
      71:d9:aa:6f:00:db:db:cd:30:3a:79:4f:5f:4c:47:
      f8:1d:ef:5b:c2:c4:9d:d0:3b:b1:b2:43:91:0b:a4:
      33:4e:ea:1b:06:27:f4:ad:25:8a:a5:c6:f4:d5:d8:
      a6:ae:74:05:e4:57:08:b5:44:55:d4:2d:2a:3a:3e:
      f8:b8:bd:e9:32:0a:02:94:64:c4:16:3a:50:f1:4a:
      aee:7:79:33:af:0c:20:07:7f:e8:df:04:39:c2:69:
      02:6c:d3:52:fa:77:c1:1b:cb:74:07:c8:b9:93:18:
      50:54:35:4b:69:4e:bc:3b:d3:49:2e:1f:dc:c1:d2:
      52:fb
    Exponent: 65537 (0x10001)
  X509v3 extensions:
    X509v3 Basic Constraints: critical
      CA:TRUE
    X509v3 Key Usage: critical
      Certificate Sign, CRL Sign
    X509v3 Subject Key Identifier:
      40:C2:8D:27:8E:CC:34:83:30:A2:33:D7:FB:6C:B3:F0:B4:2C:80:CE
    X509v3 Authority Key Identifier:
      3A:9A:85:07:1B:67:2B:86:EF:F6:8D:05:41:6E:20:C1:94:DA:0F:DE
    Authority Information Access:
      OCSP - URI:http://ocsp.godaddy.com/
    X509v3 CRL Distribution Points:
      Full Names
      URI:http://crl.godaddy.com/gdroot-g2.crl
      X509v3 Certificate Policies

```

Figure 3.20: Finding Exponent

3.7.3. Extract the signature from the server's certificate

The following command was used to print out all the fields:

openssl x509 -in c0.pem -text -noout

And then the signature block was copied and pasted into a file called signature as shown in Figure 3.21.

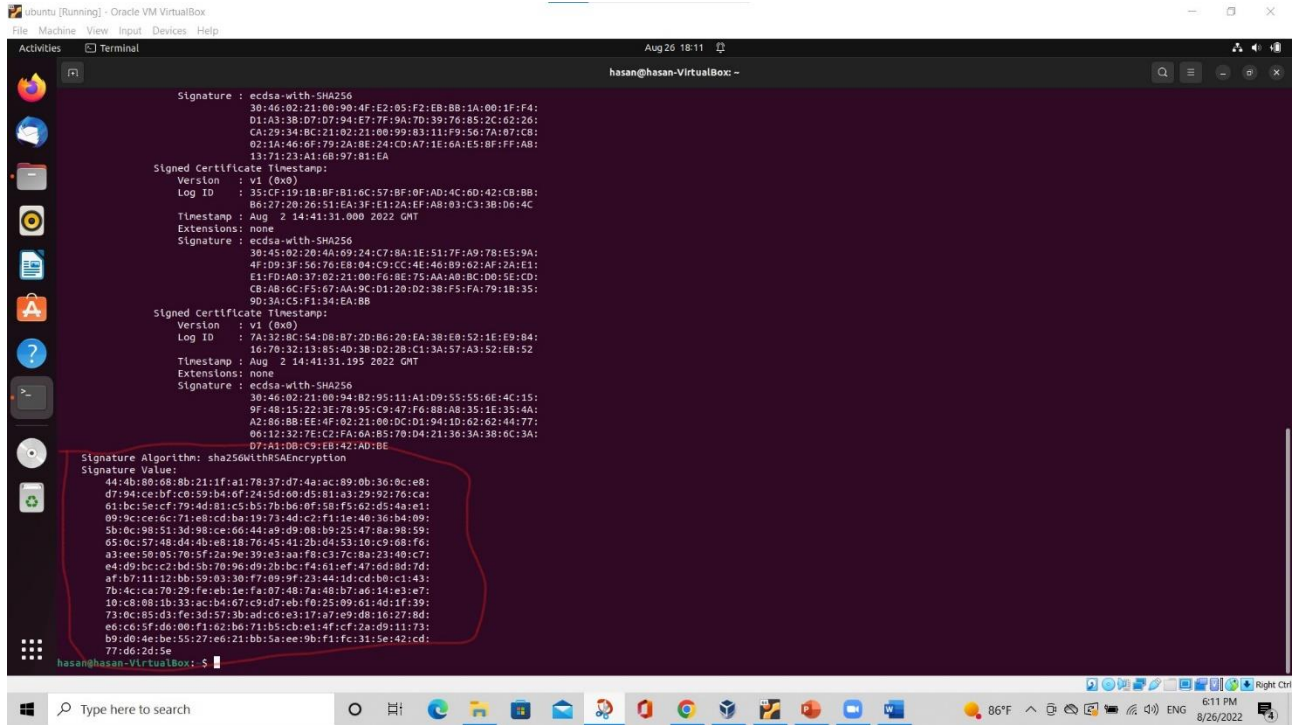


Figure 3.21: Extracting Signature Value

All the colons and spaces were removed from the signature using the tr command as shown in Figure 3.22.

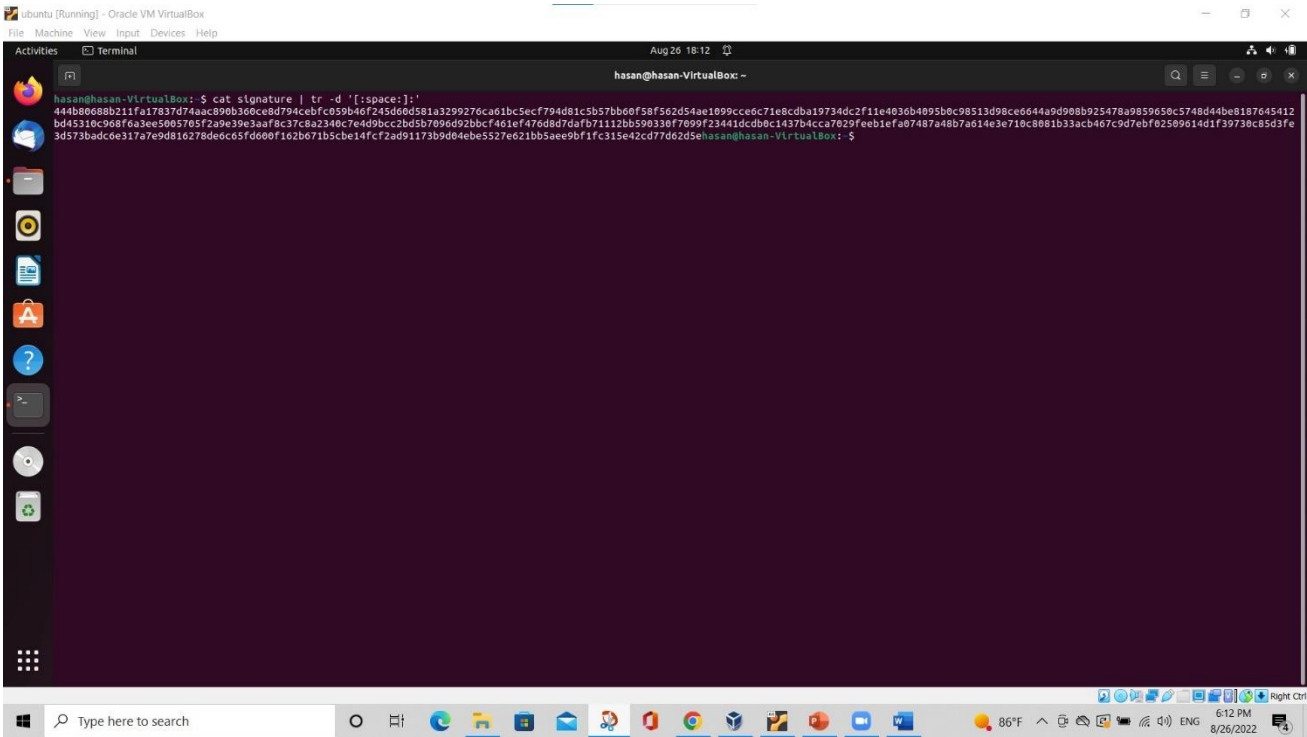


Figure 3.22: Removing Colons and Spaces From Signature File.

3.7.4. Extract the body of the server's certificate.

The following command to extract the body of the certificate as shown in Figure 3.23:

```
openssl asn1parse -i -in c0.pem
```

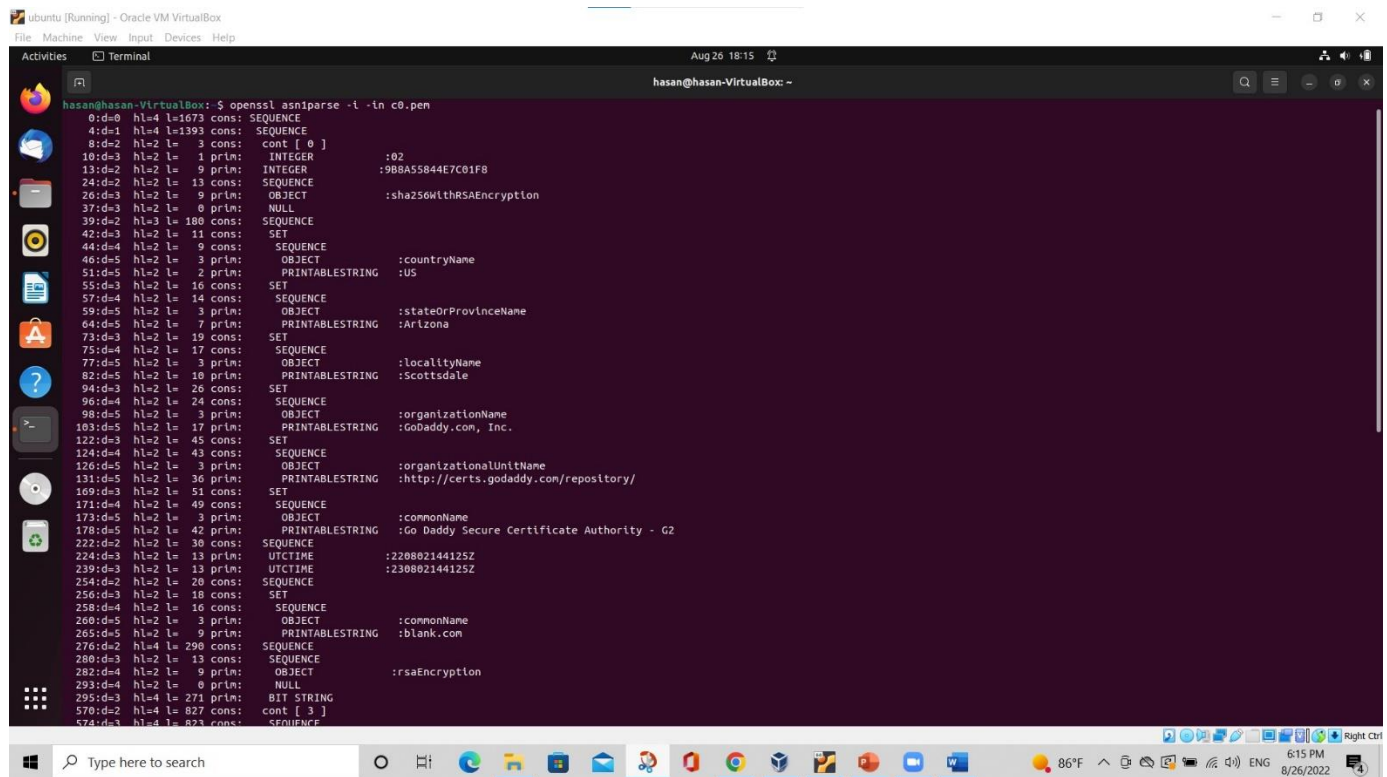


Figure 3.23: Extracting the Body of the Certificate

The `-strparse` option was used 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 the body of the certificate is observed, its hash was calculated using the following command as shown in Figure 3.24:

```
$ sha256sum c0_body.bin
```

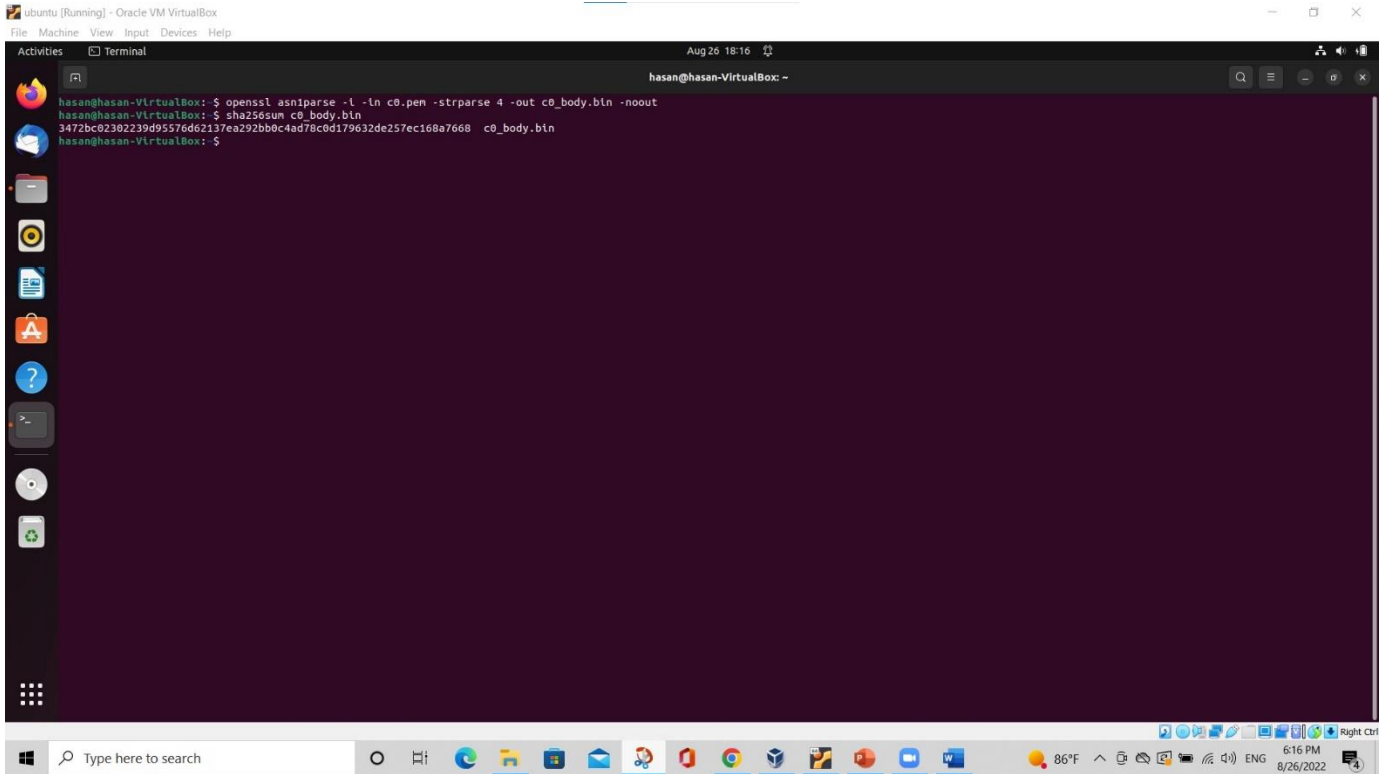


Figure 3.24: Calculating the Hash of the Certificate

3.7.5. Verify the signature

Code:

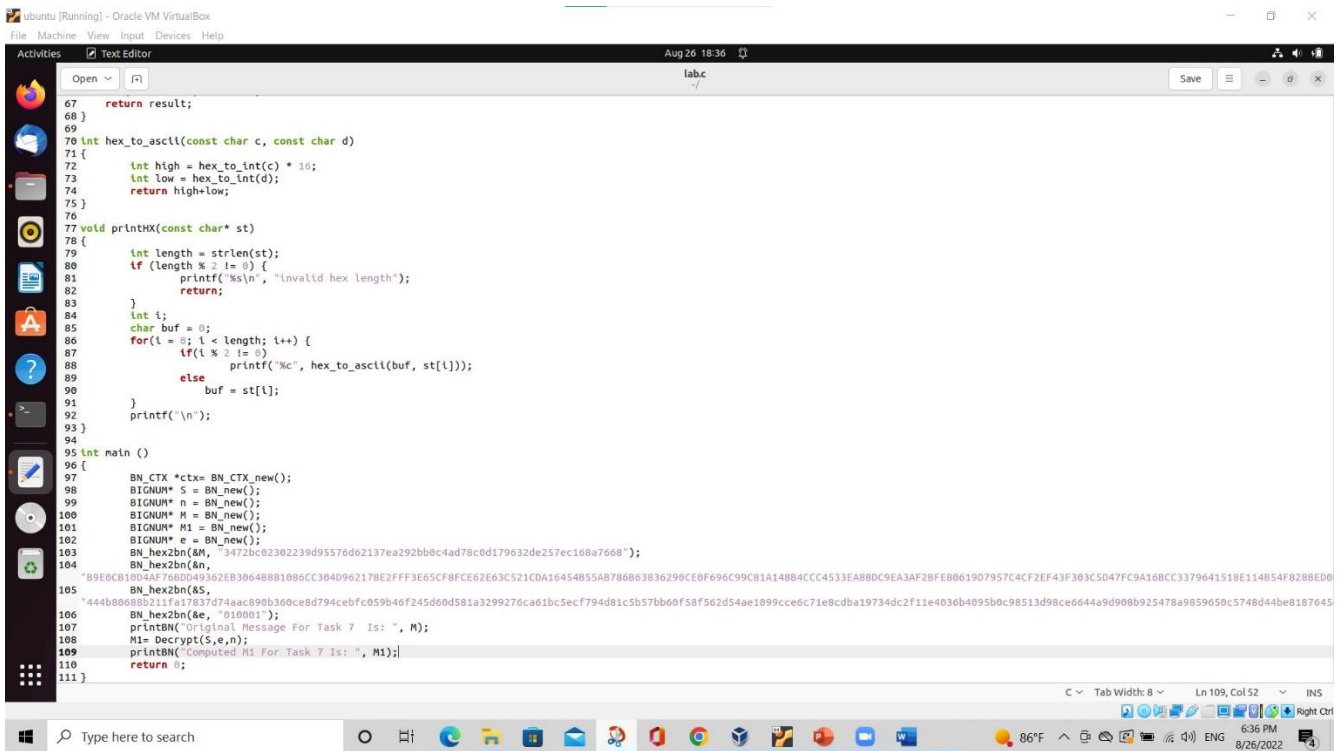


Figure 3.25: Verifying the Signature Code

The values obtained from the previous steps were used. The signature was obtained and the signature obtained was verified with the original signature as shown in Figure 3.26.

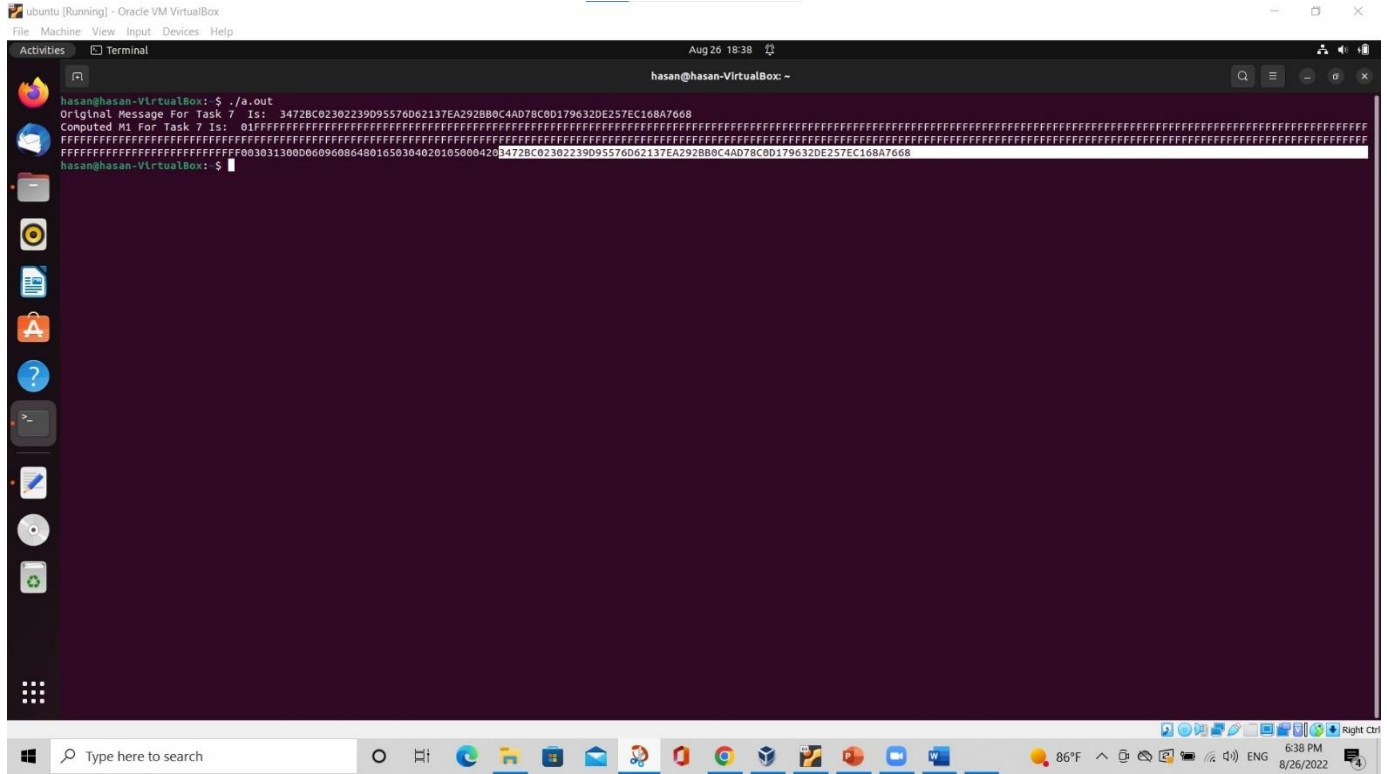


Figure 3.26: Verifying the Signature Output

The computed message's hash value and the original message have the same value. Therefore, we can say that the www.blank.com certificate has been confirmed to be right.

4. Conclusion

In conclusion, we understand the RSA public key cryptosystem, and we understand the methods: key generation, encryption and decryption using Linux and C language, and we understand how to deal with big numbers, and we learned how to generate and verify digital signatures and finally, an X.509 Certificate was manually verified using openssl.

5. References

[1] [https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem))

Accessed 27 August 2022

[2] https://en.wikipedia.org/wiki/Digital_signature

Accessed 27 August 2022

[3] <https://sectigo.com/resource-library/what-is-x509-certificate>

Accessed 27 August 2022

[4] Seed Labs RSA Encryption and Signature Lab PDF

Accessed 27 August 2022

[5] C03_Public_Key_Encryption.pptx Giving by Dr. Ahmad Alsadeh

Accessed 27 August 2022