Team: Abhijeet Amitava Banerjee G01349260 and Joel Sadanand Samson G01352483

Mini Project 2

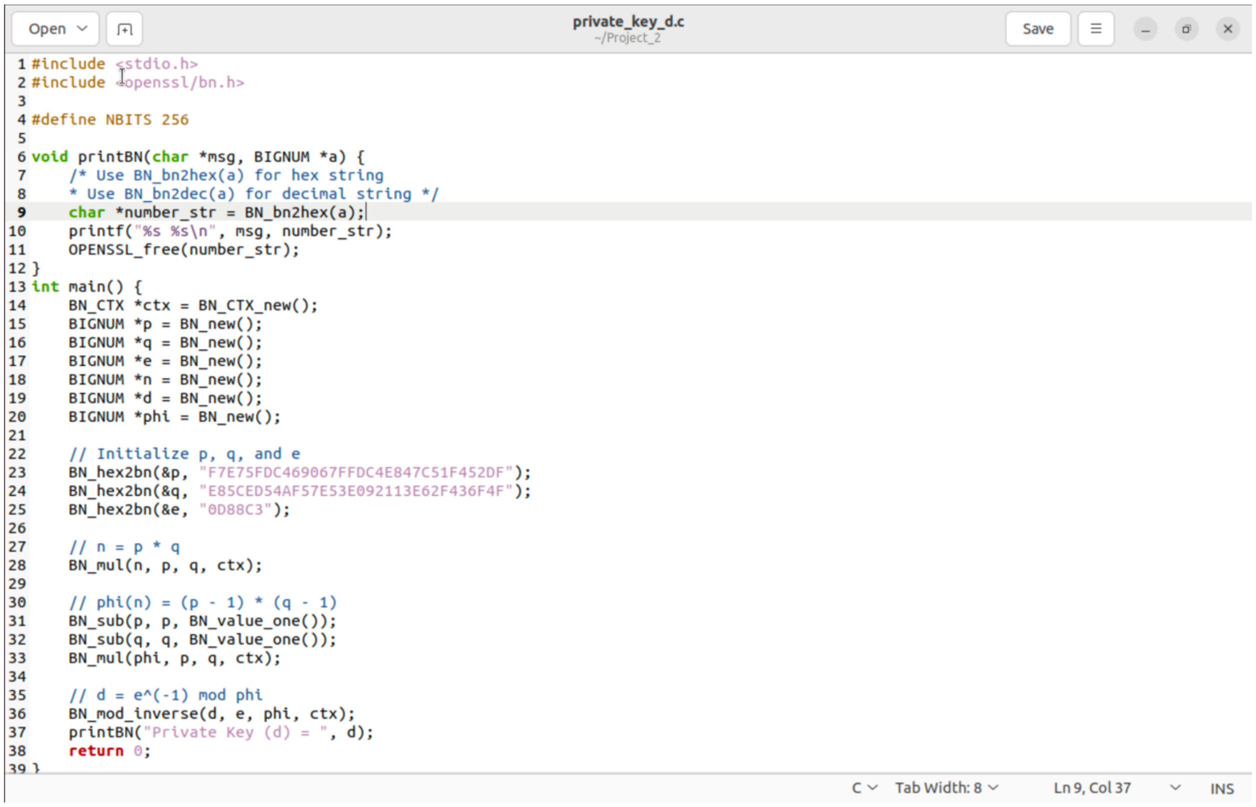
Task 1: Deriving the Private Key

In this task, we are given three prime numbers p, q, and e. We will use the public key (e, n); where n = p * q. Our aim is to derive the Private Key 'd' using the given values.

Given values:
p = F7E75FDC469067FFDC4E847C51F452DF
q = E85CED54AF57E53E092113E62F436F4F
e = 0D88C3

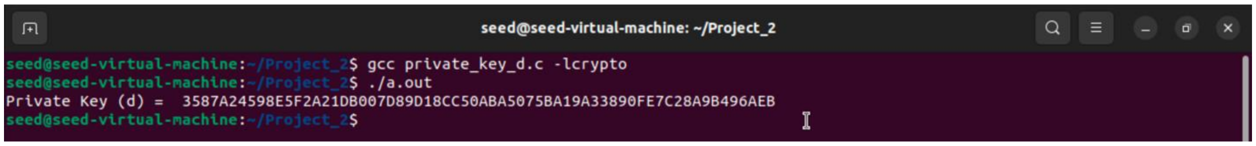
We know that d = e^{-1} mod phi(n)
Where, phi(n) = (p - 1) * (q - 1)

The following code is used to derive the Private Key using BIGNUM from the OpenSSL library.



```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 *p = BN_new();
    BIGNUM *q = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *phi = BN_new();

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

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

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

    // d = e^(-1) mod phi
    BN_mod_inverse(d, e, phi, ctx);
    printBN("Private Key (d) = ", d);
    return 0;
}
```

Execution and output of the above code can be seen below:



```
seed@seed-virtual-machine:~/Project_2$ gcc private_key_d.c -lcrypto
seed@seed-virtual-machine:~/Project_2$ ./a.out
Private Key (d) =  3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB
seed@seed-virtual-machine:~/Project_2$
```

So, the derived Private Key 'd' is:
d = 3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB

Conclusion:

Now that we have completed this task, we can see that we can compute the Private Key 'd' using the supplied prime numbers p and q as well as the public exponent e. We cannot further encrypt and decrypt in an RSA cryptosystem, due to the lack of a specific message 'M'. But, to make a note on the security of the RSA encryption, one would generally use significantly bigger prime numbers (for example, 2048 bits or 3072 bits), as opposed to the comparatively tiny prime values p and q utilized in this task. It is computationally more challenging for an attacker to factor n and retrieve the private key for larger prime values.

Task 2: Encrypting a Message

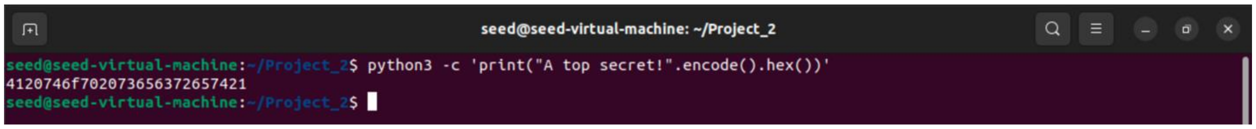
For this task we are given a Public Key (e, n) along with a message 'M'. The aim of this task is to encrypt the message M using the given information.

Given information:
n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
e = 010001 (this hex value equals to decimal 65537)
M = A top secret!

Our first step would be to convert the message from ASCII string to hex string and this is done with the help of the following python command.

```
$ python3 -c 'print("A top secret!".encode().hex())'
This is the output: 4120746f702073656372657421
```

Execution and output of the above python command can be seen below:



```
seed@seed-virtual-machine:~/Project_2$ python3 -c 'print("A top secret!".encode().hex())'
4120746f702073656372657421
seed@seed-virtual-machine:~/Project_2$
```

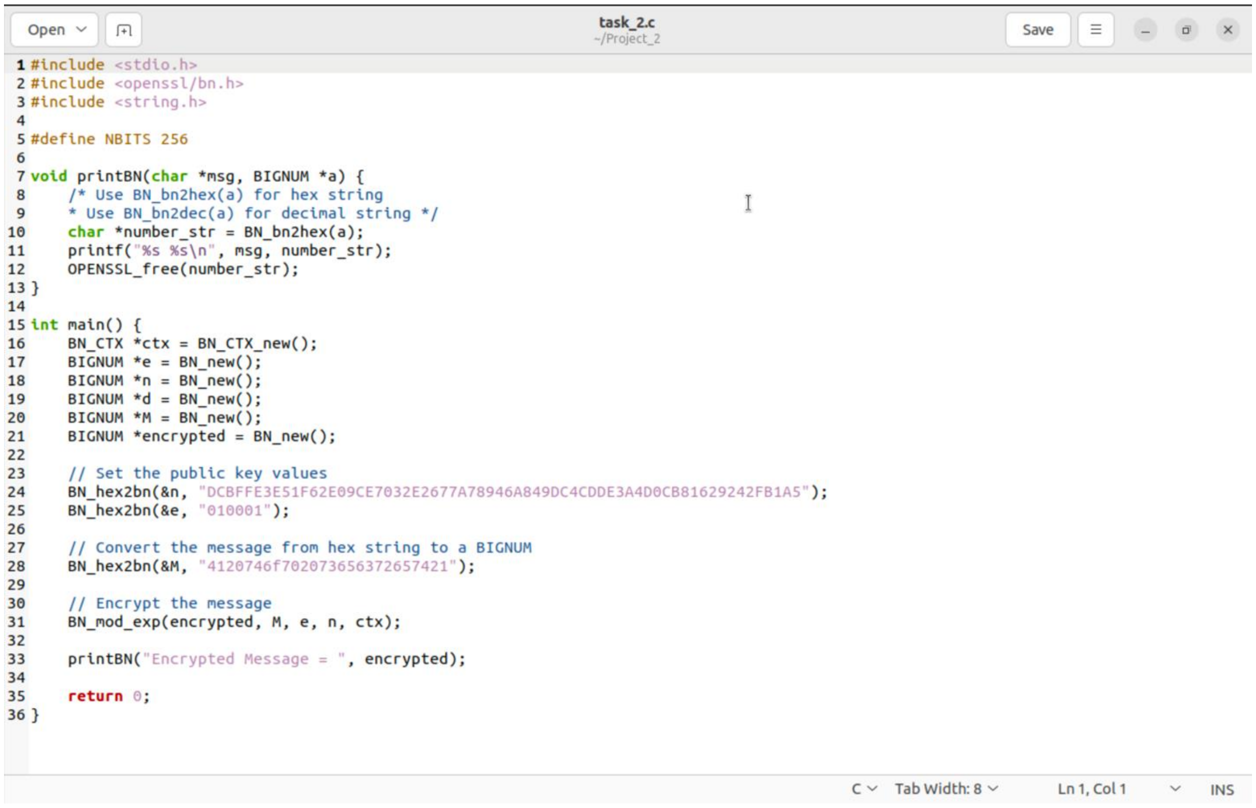
Then we convert the hex string to a BIGNUM using the hex-to-bn API BN hex2bn() and store it in the variable M along with converting the hexadecimal values of n and e to the same.
Now, that we have M, e, and n we can proceed with our encryption process.

In RSA, to encrypt a message M using the public key (e, n) we use the following formula:

$$C = M^e \bmod n$$

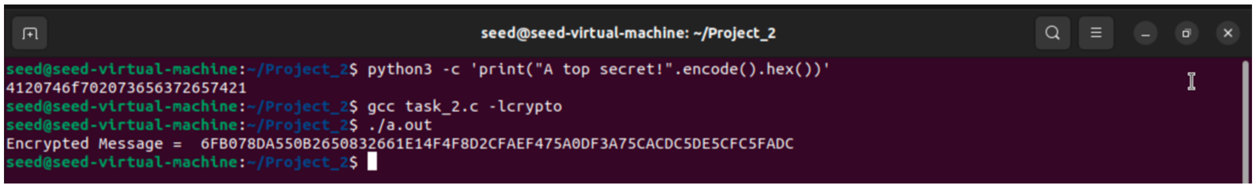
Where, C stands for Ciphertext.

The following code is used to encrypt the message "A top secret!" using BIGNUM from the OpenSSL library.



```c
#include <stdio.h>
#include <openssl/bn.h>
#include <string.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 *e = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *M = BN_new();
    BIGNUM *encrypted = BN_new();

    // Set the public key values
    BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&e, "010001");

    // Convert the message from hex string to a BIGNUM
    BN_hex2bn(&M, "4120746f702073656372657421");

    // Encrypt the message
    BN_mod_exp(encrypted, M, e, n, ctx);

    printBN("Encrypted Message = ", encrypted);

    return 0;
}
```

Execution and output of the above code can be seen below:

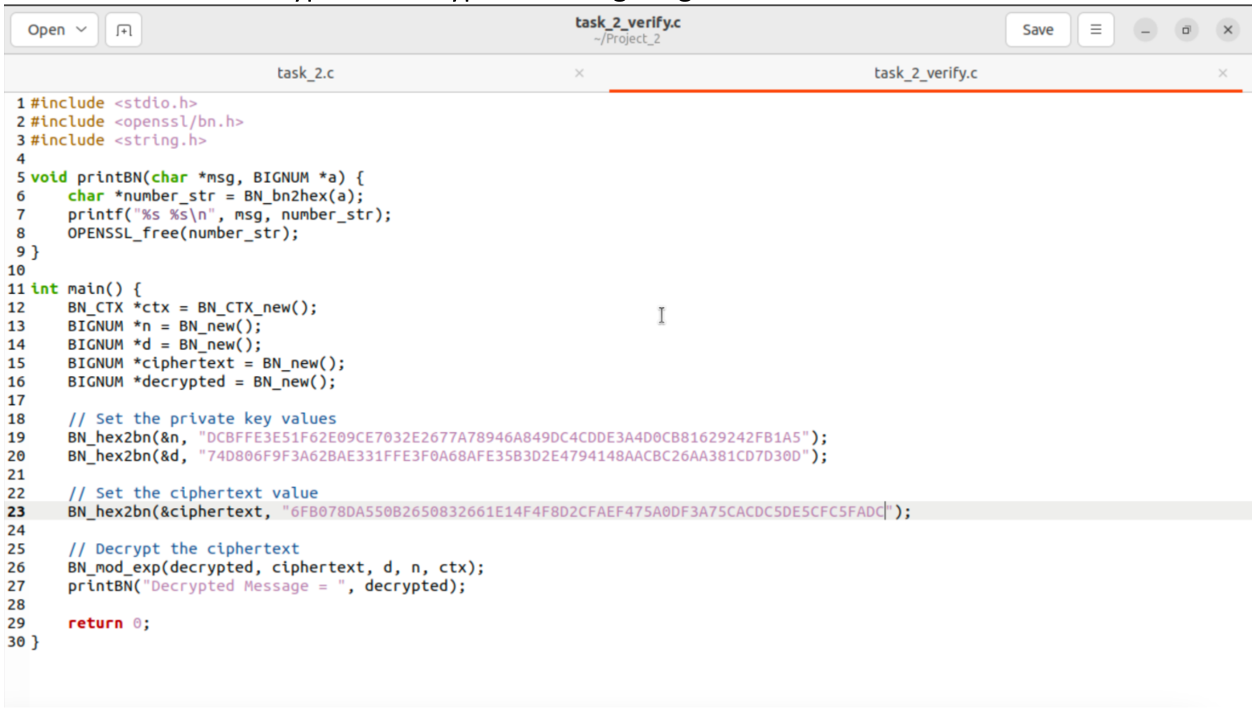


```
seed@seed-virtual-machine:~/Project_2$ python3 -c 'print("A top secret!".encode().hex())'
4120746f702073656372657421
seed@seed-virtual-machine:~/Project_2$ gcc task_2.c -lcrypto
seed@seed-virtual-machine:~/Project_2$ ./a.out
Encrypted Message =  6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC
seed@seed-virtual-machine:~/Project_2$ 
```

So, now we have the value of the ciphertext 'c' which is the encrypted form of the given message in hexadecimal form.
c = 6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC

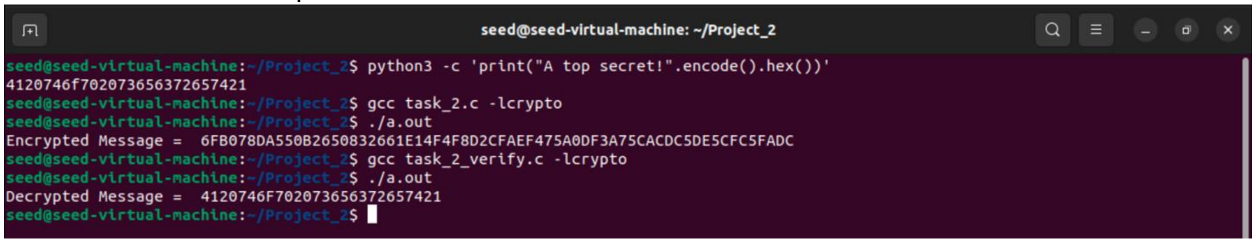
The task also provides us the private key d to help us verify our encryption result. So now, we will decrypt the encrypted message to see if we get the original message again. If the decryption output is same as the original message, then we can say that our encryption is correct.

The code used to decrypt the encrypted message is given below:



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

void printBN(char *msg, BIGNUM *a) {
    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 *d = BN_new();
    BIGNUM *ciphertext = BN_new();
    BIGNUM *decrypted = BN_new();

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

    // Set the ciphertext value
    BN_hex2bn(&ciphertext, "6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC");

    // Decrypt the ciphertext
    BN_mod_exp(decrypted, ciphertext, d, n, ctx);
    printBN("Decrypted Message = ", decrypted);

    return 0;
}
```

The execution and output can be seen below:



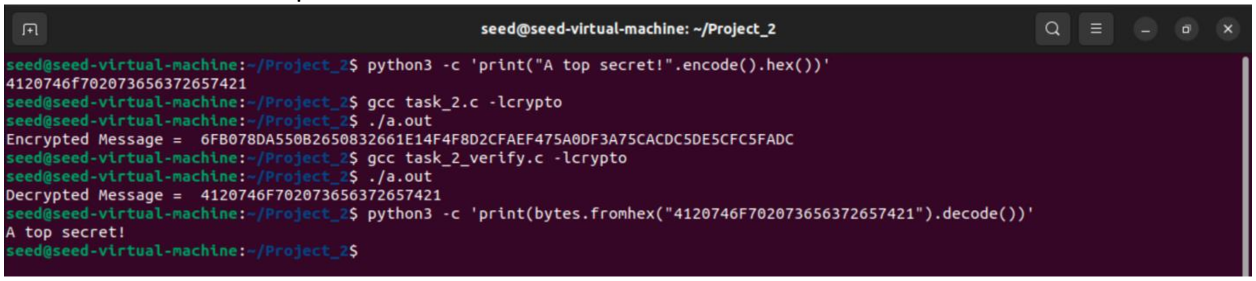
```
seed@seed-virtual-machine:~/Project_2$ python3 -c 'print("A top secret!".encode().hex())'
4120746f702073656372657421
seed@seed-virtual-machine:~/Project_2$ gcc task_2.c -lcrypto
seed@seed-virtual-machine:~/Project_2$ ./a.out
Encrypted Message =  6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC
seed@seed-virtual-machine:~/Project_2$ gcc task_2_verify.c -lcrypto
seed@seed-virtual-machine:~/Project_2$ ./a.out
Decrypted Message =  4120746F702073656372657421
seed@seed-virtual-machine:~/Project_2$ 
```

Here we can see that the decrypted message is in hex format to convert it into ASCII string we use the following python3 command.

```
$ python3 -c ‘print(bytes.fromhex(“4120746f702073656372657421”).decode())’
```

This is the output: A top secret! which is same as the original message.

The execution and output can be seen below:



```
seed@seed-virtual-machine: ~/Project_2
seed@seed-virtual-machine:~/Project_2$ python3 -c 'print("A top secret!".encode().hex())'
4120746f702073656372657421
seed@seed-virtual-machine:~/Project_2$ gcc task_2.c -lcrypto
seed@seed-virtual-machine:~/Project_2$ ./a.out
Encrypted Message =  6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC
seed@seed-virtual-machine:~/Project_2$ gcc task_2_verify.c -lcrypto
seed@seed-virtual-machine:~/Project_2$ ./a.out
Decrypted Message =  4120746F702073656372657421
seed@seed-virtual-machine:~/Project_2$ python3 -c 'print(bytes.fromhex("4120746F702073656372657421").decode())'
A top secret!
seed@seed-virtual-machine:~/Project_2$
```

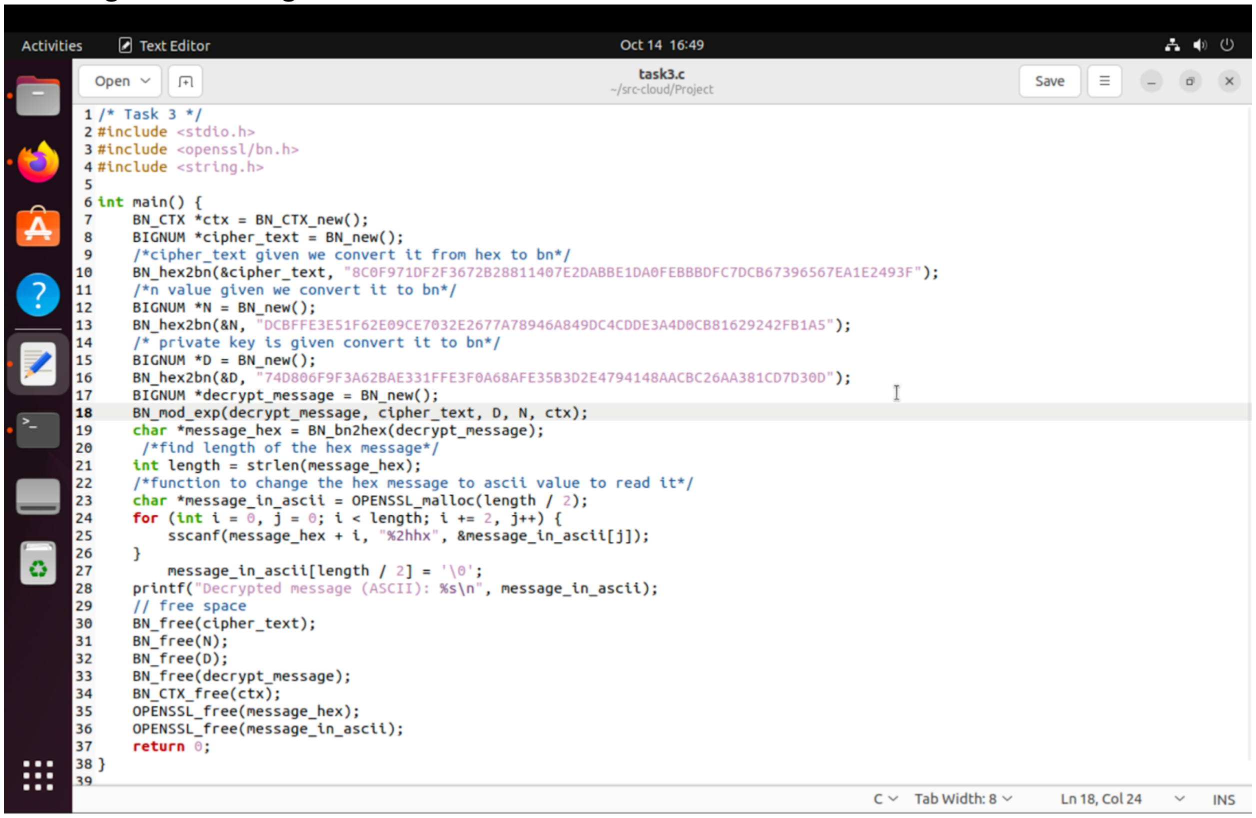
This shows that our encryption was correct.

Conclusion:

We have successfully encrypted the message "A top secret!" into a ciphertext using the RSA public key (e, n). RSA encryption is a public-key encryption algorithm, and the security of the encryption depends on the difficulty of factoring n. It's worth noting that the task uses small values for simplicity. We can observe the process of encryption and how the public key (e, n) is used to secure the message. To decrypt and recover the original message, we used the private key d. The task also involves converting an ASCII string to a hexadecimal string and then converting it to a BIGNUM for cryptographic operations. This conversion demonstrates the need for consistent data representation when working with cryptographic libraries. The provided private key d is used to verify that the encryption process is correct. We have decrypted the ciphertext using d and compare the result with the original message to ensure that the encryption and decryption are consistent.

Task 3: Decrypting a Message

The Program for doing this task is as follows:



```c
/* Task 3 */
#include <stdio.h>
#include <openssl/bn.h>
#include <string.h>

int main() {
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *cipher_text = BN_new();
    /*cipher_text given we convert it from hex to bn*/
    BN_hex2bn(&cipher_text, "8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");
    /*n value given we convert it to bn*/
    BIGNUM *N = BN_new();
    BN_hex2bn(&N, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    /* private key is given convert it to bn*/
    BIGNUM *D = BN_new();
    BN_hex2bn(&D, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BIGNUM *decrypt_message = BN_new();
    BN_mod_exp(decrypt_message, cipher_text, D, N, ctx);
    char *message_hex = BN_bn2hex(decrypt_message);
     /*find length of the hex message*/
    int length = strlen(message_hex);
    /*function to change the hex message to ascii value to read it*/
    char *message_in_ascii = OPENSSL_malloc(length / 2);
    for (int i = 0, j = 0; i < length; i += 2, j++) {
        sscanf(message_hex + i, "%2hhx", &message_in_ascii[j]);
    }
        message_in_ascii[length / 2] = '\0';
    printf("Decrypted message (ASCII): %s\n", message_in_ascii);
    // free space
    BN_free(cipher_text);
    BN_free(N);
    BN_free(D);
    BN_free(decrypt_message);
    BN_CTX_free(ctx);
    OPENSSL_free(message_hex);
    OPENSSL_free(message_in_ascii);
    return 0;
}

```

Here we are given the cipher text in hexadecimal format ie.
C = 8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F
We are also given the value of n and the private key gotten from the Task 2 and the public key
n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
e = 010001 (this hex value equals to decimal 65537)
d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D
All these are in hexadecimal.

For our computation purpose we have to change all these values into BIGNUM.
That's what is done in the first part of the program, we are initializing values to store the BIGNUM counterpart which is gotten with the help of the function BN_hex2bn(variable,value).

According to rsa encryption and decryption: we must know the public key this is found in Task 2.
Using the formulae

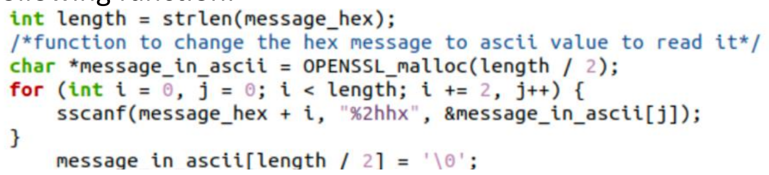
- e.d=1 mod ø(n) and 0≤d≤n

So we have to compute d using e^-1 mod ø(n)
Now using this value of the private key we make use of the formulae to find the cipher text given below:

- to decrypt the ciphertext C the owner:
  - uses their private key PR={d,n}
  - computes: M = C^d mod n

So using D in our program as private Key and N as n we take the ciphertext all values which are converted to BIGNUM and perform the function M = C^d mod n
This can be done using the program function
BN_mod_exp(target_value,cipher_text,private_key,n,ctx)
The value of this operation is stored in the target_value variable and this is the decrypted message in BIGNUM
So we do an operation to turn it into hex again using BN_bn2hex(variable,value).

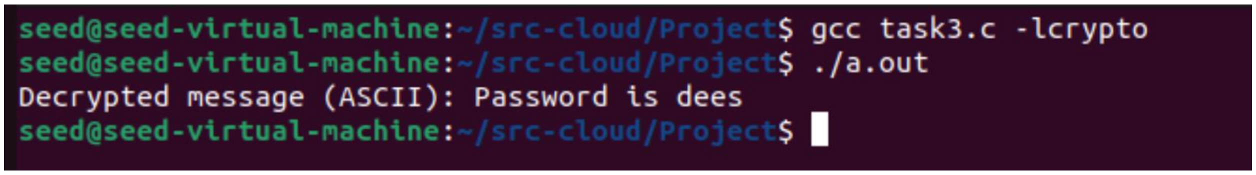
Now according to our question we have to display it as a ascii format so we have to convert the Hex representation of the decrypted message into ascii format this can be done using the following function.



```
int length = strlen(message_hex);
/*function to change the hex message to ascii value to read it*/
char *message_in_ascii = OPENSSL_malloc(length / 2);
for (int i = 0, j = 0; i < length; i += 2, j++) {
    sscanf(message_hex + i, "%2hhx", &message_in_ascii[j]);
}
    message_in_ascii[length / 2] = '\0';
```

Using this we get the ascii representation of our encrypted message

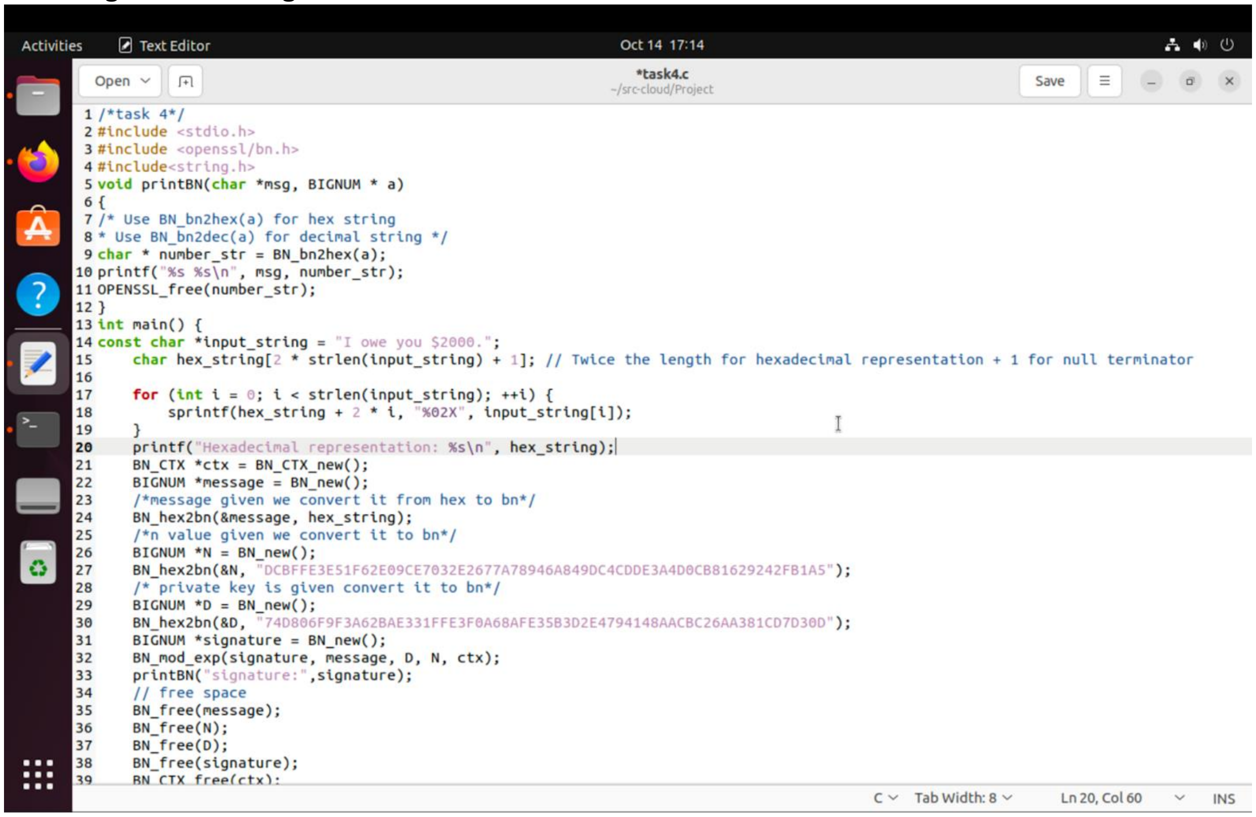
Running the program we get the result:



```
seed@seed-virtual-machine:~/src-cloud/Project$ gcc task3.c -lcrypto
seed@seed-virtual-machine:~/src-cloud/Project$ ./a.out
Decrypted message (ASCII): Password is dees
seed@seed-virtual-machine:~/src-cloud/Project$
```

Task 4: Signing a Message

The Program for doing this task is as follows:

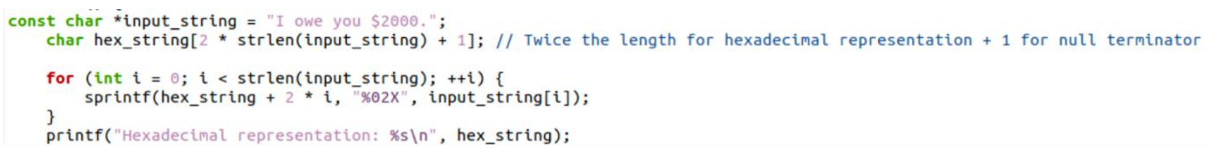


```
/*task 4*/
#include <stdio.h>
#include <openssl/bn.h>
#include<string.h>
void printBN(char *msg, BIGNUM * a)
{
/* Use BN_bn2hex(a) for hex string
* Use BN_bn2dec(a) for decimal string */
char * number_str = BN_bn2hex(a);
printf("%s %s\n", msg, number_str);
OPENSSL_free(number_str);
}
int main() {
const char *input_string = "I owe you $2000.";
    char hex_string[2 * strlen(input_string) + 1]; // Twice the length for hexadecimal representation + 1 for null terminator

    for (int i = 0; i < strlen(input_string); ++i) {
        sprintf(hex_string + 2 * i, "%02X", input_string[i]);
    }
    printf("Hexadecimal representation: %s\n", hex_string);
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *message = BN_new();
    /*message given we convert it from hex to bn*/
    BN_hex2bn(&message, hex_string);
    /*n value given we convert it to bn*/
    BIGNUM *N = BN_new();
    BN_hex2bn(&N, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    /* private key is given convert it to bn*/
    BIGNUM *D = BN_new();
    BN_hex2bn(&D, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BIGNUM *signature = BN_new();
    BN_mod_exp(signature, message, D, N, ctx);
    printBN("signature:",signature);
    // free space
    BN_free(message);
    BN_free(N);
    BN_free(D);
    BN_free(signature);
    BN_CTX_free(ctx);
```

Here we are given the Message in ASCII format for which we have to find the signature for i.e.
M = I owe you $2000.

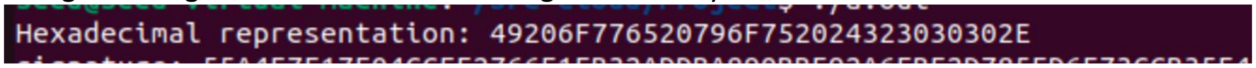
We are also given the value of n and the private key gotten from the Task 2 and the public key.
n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
e = 010001 (this hex value equals to decimal 65537)
d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D
All these are in hexadecimal.

For our computation purpose we have to convert our Message given in ASCII format into a hexadecimal version for computation. We have done that using the following function:



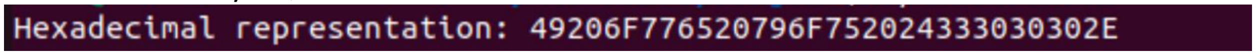
```
const char *input_string = "I owe you $2000.";
    char hex_string[2 * strlen(input_string) + 1]; // Twice the length for hexadecimal representation + 1 for null terminator

    for (int i = 0; i < strlen(input_string); ++i) {
        sprintf(hex_string + 2 * i, "%02X", input_string[i]);
    }
    printf("Hexadecimal representation: %s\n", hex_string);
```

Using this we get the value for the Message: M: I owe you $2000. As



```
Hexadecimal representation: 49206F776520796F752024323030302E
```

And for M: I owe you $3000. As



```
Hexadecimal representation: 49206F776520796F752024333030302E
```

Here we can see the hexadecimal value of the 2 messages differ slightly only in one block difference is in the value of 2 and 3 where it is 2332 for 2 and 2433 for 4 in the hexadecimal format

I.e.

For M: I owe you $2000.

49206F776520796F752024323030302E

For M: I owe you $3000.

49206F776520796F752024333030302E

Now for digital signature we are going to obtain the signatures for the above messages Using the private key of the owner of the message

Private key and n value have to be converted into BIGNUM to do the process.

Done in the program using BN_hex2bn(variable,hex_value)

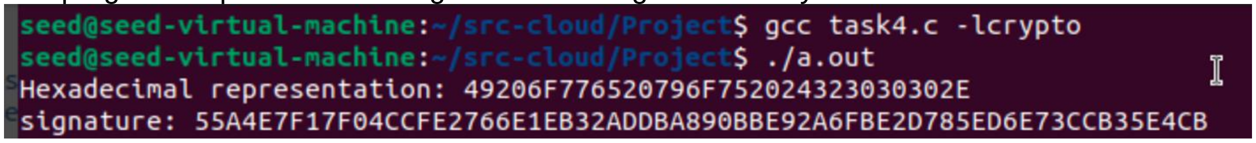
Now according to RSA the method of signing is done using the private key of who is sending it and n using the following function

- Signature = M^d mod n

So reciprocating the function in our program we do it using:

BN_mod_exp(signature,message_in_BIGNUM,private_key,n,ctx)

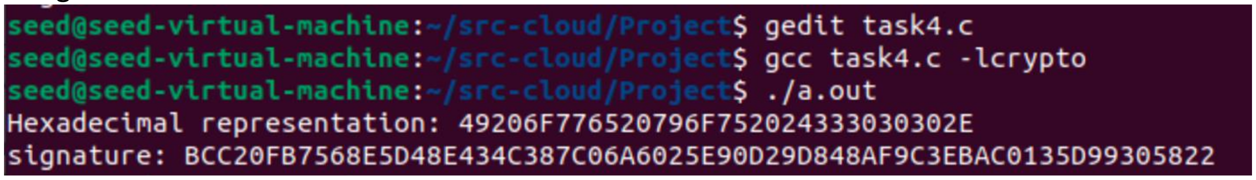
Our program outputs the following for the message M: I owe you $2000.



```
seed@seed-virtual-machine:~/src-cloud/Project$ gcc task4.c -lcrypto
seed@seed-virtual-machine:~/src-cloud/Project$ ./a.out
Hexadecimal representation: 49206F776520796F752024323030302E
signature: 55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73CCB35E4CB
```

And for M: I owe you $3000.

We get:



```
seed@seed-virtual-machine:~/src-cloud/Project$ gedit task4.c
seed@seed-virtual-machine:~/src-cloud/Project$ gcc task4.c -lcrypto
seed@seed-virtual-machine:~/src-cloud/Project$ ./a.out
Hexadecimal representation: 49206F776520796F752024333030302E
signature: BCC20FB7568E5D48E434C387C06A6025E90D29D848AF9C3EBAC0135D99305822
```

Here we can see that just changing one value in the message generates a new signature altogether. Hence this is a really good way to provide authentication to the user's messages.

This demonstrates the sensitivity of the signature to the message content.

Task 5: Verifying a Signature

Bob receives a message M = "Launch a missile." from Alice, with her signature S. Alice's public key is (e, n), given. The task is to verify whether the signature is indeed Alice's or not.

The public key and signature (hexadecimal) are listed in the following:
S = 643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F
e = 010001 (this hex value equals to decimal 65537)
n = AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115

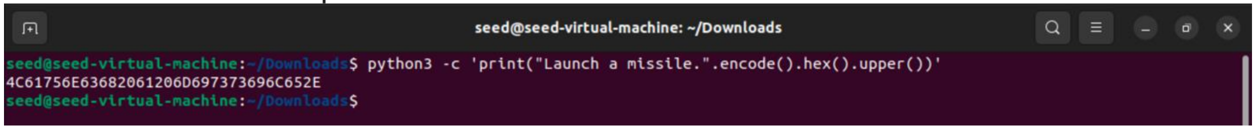
The values of S, e, and n are in hexadecimal format so, we convert them to BIGNUM.

We also convert M which is in ASCII string format to hexadecimal string using the following python command:

$ python3 -c 'print("Launch a missile.".encode().hex.upper())'
The output is 4C61756E63682061206D697373696C652E

We use .upper() function to make the letter in the hex string capital because it makes the comparison to the hex string given by the program easy in later part.

The execution and output is shown below:



```
seed@seed-virtual-machine:~/Downloads$ python3 -c 'print("Launch a missile.".encode().hex().upper())'
4C61756E63682061206D697373696C652E
seed@seed-virtual-machine:~/Downloads$
```

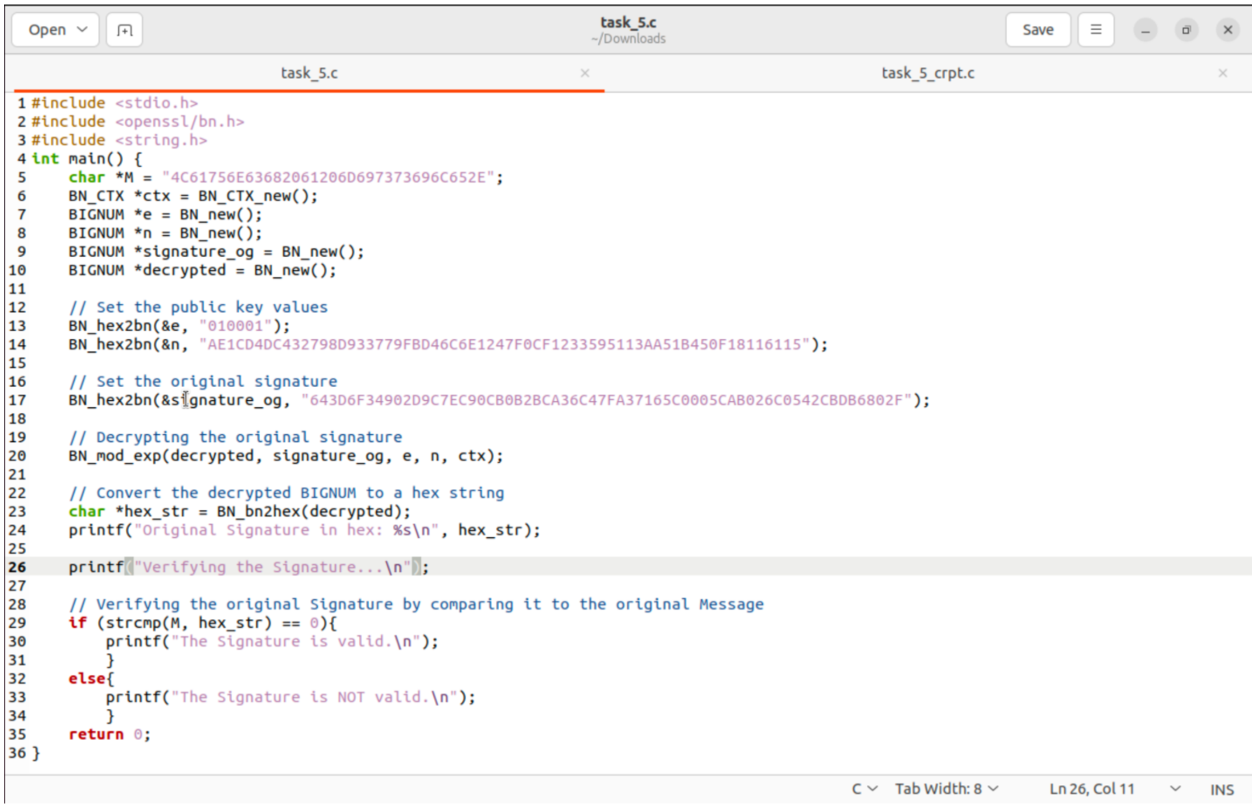
To verify that the signature is Alice's, we calculate the plaintext message (recovered message) from the signature using Alice's public key (e, n):

Recovered Message = S^e mod n

Now, the Recovered Message is in BIGNUM format so we convert it to hex string so that we can compare it to the hex string representation of the original message M.

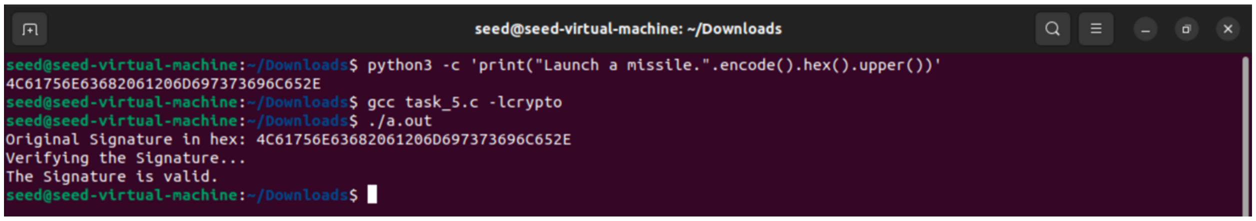
Now, we know that if the hexadecimal value of the message recovered from the signature using Alice's public key matches the hexadecimal value of the Original Message, then we can say that the signature was created using Alice's private key. Hence, verifying that the signature belongs to Alice.

The following code is used to compute the hex string of the message recovered from the given signature using Alice's public key and comparing the value to the hex string of the Original Message M, if the strings match then the signature will come out as Valid:



```c
#include <stdio.h>
#include <openssl/bn.h>
#include <string.h>
int main() {
    char *M = "4C61756E63682061206D697373696C652E";
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *e = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *signature_og = BN_new();
    BIGNUM *decrypted = BN_new();

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

    // Set the original signature
    BN_hex2bn(&signature_og, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");

    // Decrypting the original signature
    BN_mod_exp(decrypted, signature_og, e, n, ctx);

    // Convert the decrypted BIGNUM to a hex string
    char *hex_str = BN_bn2hex(decrypted);
    printf("Original Signature in hex: %s\n", hex_str);

    printf("Verifying the Signature...\n");

    // Verifying the original Signature by comparing it to the original Message
    if (strcmp(M, hex_str) == 0){
        printf("The Signature is valid.\n");
        }
    else{
        printf("The Signature is NOT valid.\n");
        }
    return 0;
}
```

The execution and output of the above code is shown below:

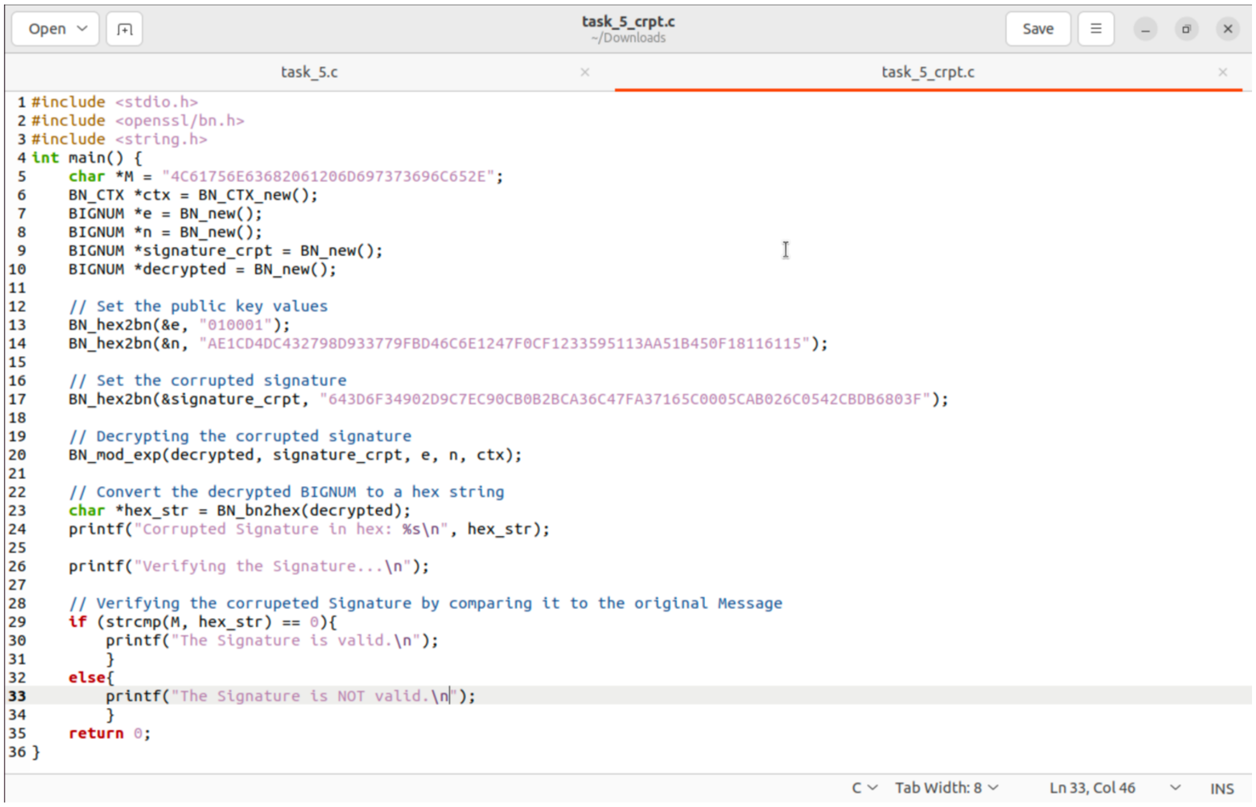


```
seed@seed-virtual-machine:~/Downloads$ python3 -c 'print("Launch a missile.".encode().hex().upper())'
4C61756E63682061206D697373696C652E
seed@seed-virtual-machine:~/Downloads$ gcc task_5.c -lcrypto
seed@seed-virtual-machine:~/Downloads$ ./a.out
Original Signature in hex: 4C61756E63682061206D697373696C652E
Verifying the Signature...
The Signature is valid.
seed@seed-virtual-machine:~/Downloads$
```

Here we can see that the strings match and hence it is Valid.

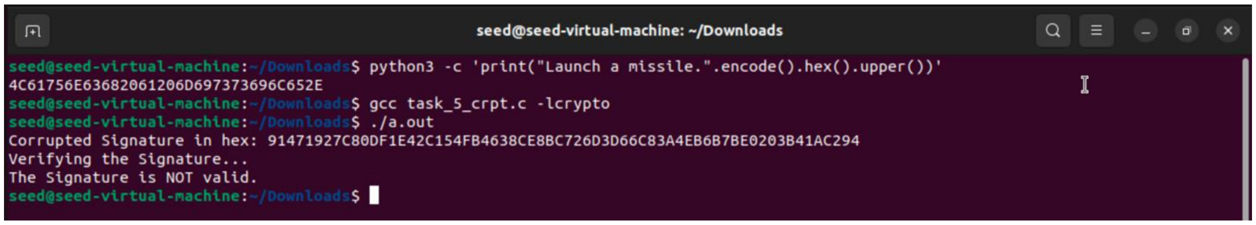
Suppose that the signature is corrupted, such that the last byte of the signature changes from 2F to 3F, i.e., there is only one bit of change. To see what happens we repeat the task.

The code to verify the corrupted signature is shown below:



```c
#include <stdio.h>
#include <openssl/bn.h>
#include <string.h>
int main() {
    char *M = "4C61756E63682061206D697373696C652E";
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *e = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *signature_crpt = BN_new();
    BIGNUM *decrypted = BN_new();

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

    // Set the corrupted signature
    BN_hex2bn(&signature_crpt, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F");

    // Decrypting the corrupted signature
    BN_mod_exp(decrypted, signature_crpt, e, n, ctx);

    // Convert the decrypted BIGNUM to a hex string
    char *hex_str = BN_bn2hex(decrypted);
    printf("Corrupted Signature in hex: %s\n", hex_str);

    printf("Verifying the Signature...\n");

    // Verifying the correpted Signature by comparing it to the original Message
    if (strcmp(M, hex_str) == 0){
        printf("The Signature is valid.\n");
        }
    else{
        printf("The Signature is NOT valid.\n");
        }
    return 0;
}
```

The execution and output of the above code is show below:



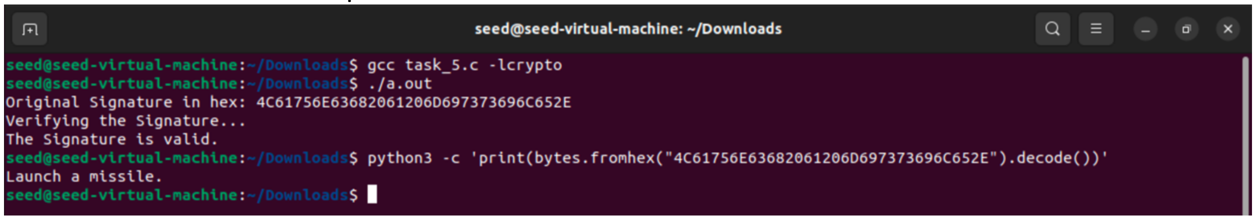
```
seed@seed-virtual-machine:~/Downloads$ python3 -c 'print("Launch a missile.".encode().hex().upper())'
4C61756E63682061206D697373696C652E
seed@seed-virtual-machine:~/Downloads$ gcc task_5_crpt.c -lcrypto
seed@seed-virtual-machine:~/Downloads$ ./a.out
Corrupted Signature in hex: 91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294
Verifying the Signature...
The Signature is NOT valid.
seed@seed-virtual-machine:~/Downloads$ 
```

We can see that the hex string message recovered from the corrupted signature does not match the hex string of the original message. Hence the corrupted signature is NOT Valid.

Now, to take it a step further in our observation, we will convert the hex string of the validated signature to ASCII string using the following python command:

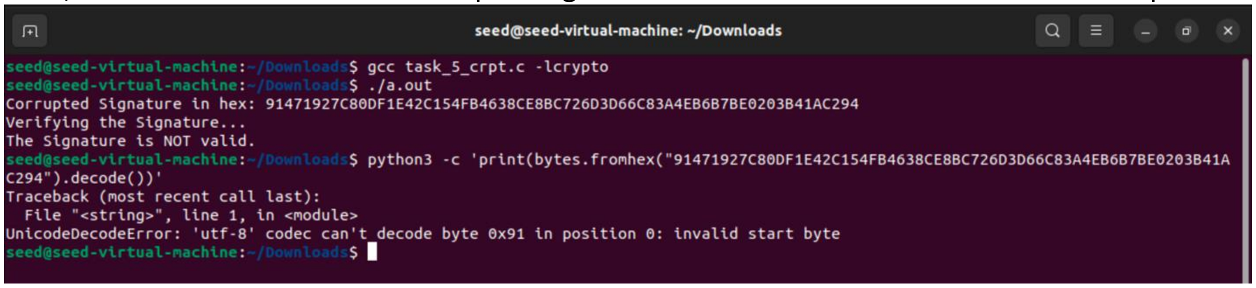
$ python3 -c
‘print(bytes.fromhex(“4C61756E63682061206D697373696C652E”).decode())’
The output is Launch a missile. Which is the same as the message M.

The execution and the output of the command is shown below:



```
seed@seed-virtual-machine:~/Downloads$ gcc task_5.c -lcrypto
seed@seed-virtual-machine:~/Downloads$ ./a.out
Original Signature in hex: 4C61756E63682061206D697373696C652E
Verifying the Signature...
The Signature is valid.
seed@seed-virtual-machine:~/Downloads$ python3 -c 'print(bytes.fromhex("4C61756E63682061206D697373696C652E").decode())'
Launch a missile.
seed@seed-virtual-machine:~/Downloads$ 
```

Now, let’s do the same for the corrupted signature which is not valid and look at the output:



```
seed@seed-virtual-machine:~/Downloads$ gcc task_5_crpt.c -lcrypto
seed@seed-virtual-machine:~/Downloads$ ./a.out
Corrupted Signature in hex: 91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294
Verifying the Signature...
The Signature is NOT valid.
seed@seed-virtual-machine:~/Downloads$ python3 -c 'print(bytes.fromhex("91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41A
C294").decode())'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x91 in position 0: invalid start byte
seed@seed-virtual-machine:~/Downloads$ 
```

Here we can see the error displayed which says that the ‘utf-8’ codec can’t decode byte 0x91 in position 0: invalid start byte. This means that the corrupted signature cannot be converted to ASCII string using the same python command used earlier. So, there won’t be any match with the original message.

Conclusion:

When verifying the original signature with the public key, we found that the signature is valid, and it matches the message "Launch a missile." This is because the original signature, S, was generated correctly using the private key corresponding to Alice's public key.
When verifying the corrupted signature (changing the last byte from 2F to 3F) with the same public key, we found that the verification fails. The decrypted message from the corrupted signature will not match the original message.
This task demonstrates the sensitivity of digital signatures to even minor changes in the signed data. A single-bit change in the signature results in a completely different message after decryption, leading to a failed verification. This property of digital signatures ensures that they are tamper-evident; any alteration to the signed data, no matter how small, will result in a mismatch during verification.

Team Contribution:
Abhijeet Banerjee Tasks 1, 2, and 5.
Joel Samson Tasks 3 and 4.