Applied Information Security

Assignment-3

Submitted by:

Muhammad Osama Khalid

20I-1955 (MS-CNS)

Section: 1

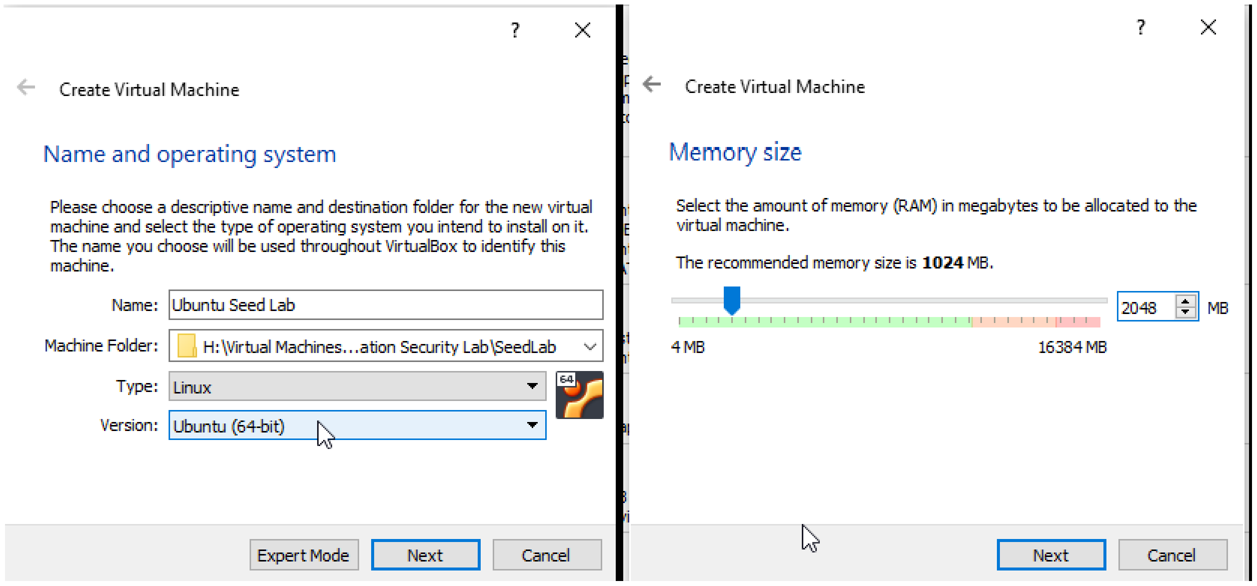
Table of Contents

List of Figures

Part-1 Lab Setup

In this part, I am going to set up the Attack Lab in which I will perform the SQL injection and Cross-Site Scripting Attacks in the next parts.

1. First, I download the Seed Virtual Machine from the SEED Lab Website and then open the virtual box and click on the add button. Then I add details of my VM and click Next. After that, I set the memory size (RAM) to 2 GB and click next. After that from the options, I select "Use an existing virtual hard disk file" and go to the path where I download the Seed VM and select that. After that, I click create to create my VM.



Create Virtual Machine

Name and operating system

Please choose a descriptive name and destination folder for the new virtual machine and select the type of operating system you intend to install on it. The name you choose will be used throughout VirtualBox to identify this machine.

Name: Ubuntu Seed Lab
Machine Folder: H:\Virtual Machines…ation Security Lab\SeedLab
Type: Linux
Version: Ubuntu (64-bit)

Create Virtual Machine

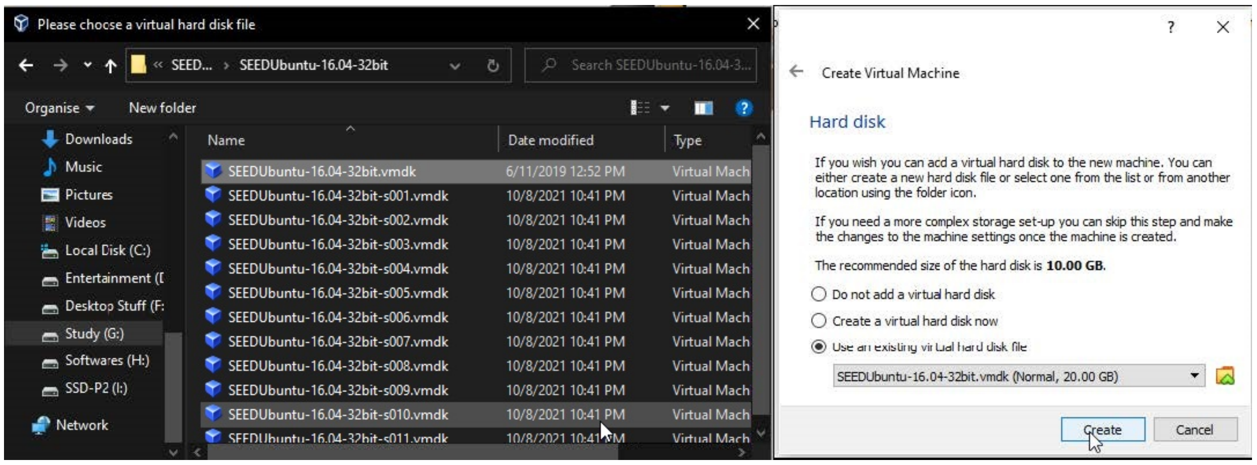
Memory size

Select the amount of memory (RAM) in megabytes to be allocated to the virtual machine.

The recommended memory size is **1024** MB.

2048 MB

Figure 1: Creating the Virtual Machine



Please choose a virtual hard disk file

SEED… › SEEDUbuntu-16.04-32bit

Name	Date modified	Type
SEEDUbuntu-16.04-32bit.vmdk	6/11/2019 12:52 PM	Virtual Mach
SEEDUbuntu-16.04-32bit-s001.vmdk	10/8/2021 10:41 PM	Virtual Mach
SEEDUbuntu-16.04-32bit-s002.vmdk	10/8/2021 10:41 PM	Virtual Mach
SEEDUbuntu-16.04-32bit-s003.vmdk	10/8/2021 10:41 PM	Virtual Mach
SEEDUbuntu-16.04-32bit-s004.vmdk	10/8/2021 10:41 PM	Virtual Mach
SEEDUbuntu-16.04-32bit-s005.vmdk	10/8/2021 10:41 PM	Virtual Mach
SEEDUbuntu-16.04-32bit-s006.vmdk	10/8/2021 10:41 PM	Virtual Mach
SEEDUbuntu-16.04-32bit-s007.vmdk	10/8/2021 10:41 PM	Virtual Mach
SEEDUbuntu-16.04-32bit-s008.vmdk	10/8/2021 10:41 PM	Virtual Mach
SEEDUbuntu-16.04-32bit-s009.vmdk	10/8/2021 10:41 PM	Virtual Mach
SEEDUbuntu-16.04-32bit-s010.vmdk	10/8/2021 10:41 PM	Virtual Mach
SEEDUbuntu-16.04-32bit-s011.vmdk	10/8/2021 10:41 PM	Virtual Mach

Create Virtual Machine

Hard disk

If you wish you can add a virtual hard disk to the new machine. You can either create a new hard disk file or select one from the list or from another location using the folder icon.

If you need a more complex storage set-up you can skip this step and make the changes to the machine settings once the machine is created.

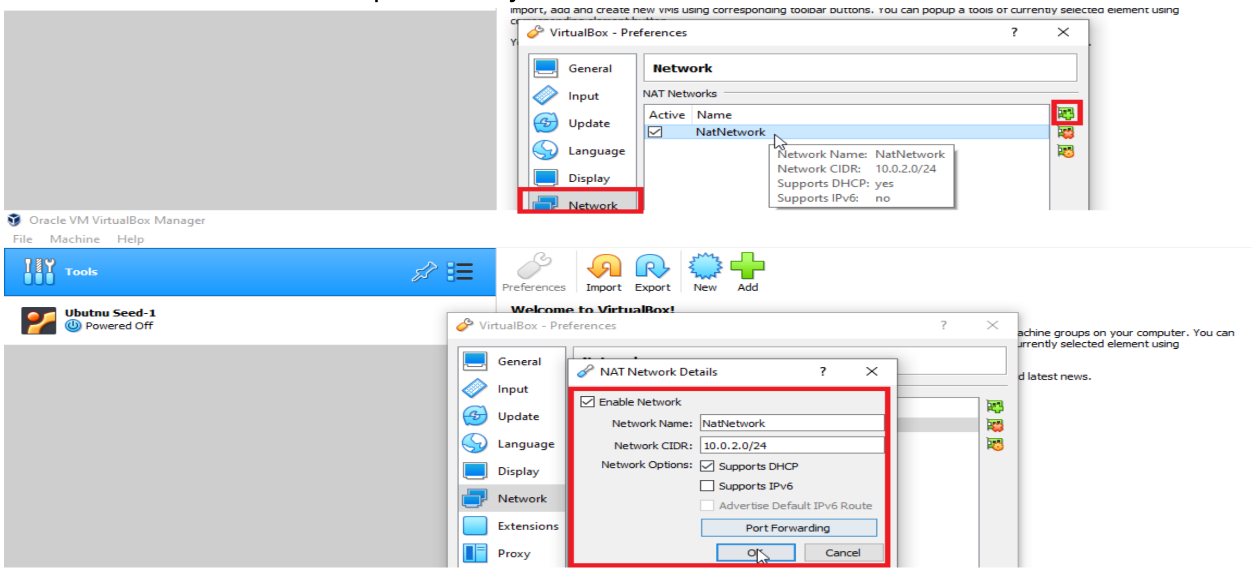
The recommended size of the hard disk is **10.00 GB**.

○ Do not add a virtual hard disk
○ Create a virtual hard disk now
● Use an existing virtual hard disk file

SEEDUbuntu-16.04-32bit.vmdk (Normal, 20.00 GB)

Figure 2: Creating the Virtual Machine

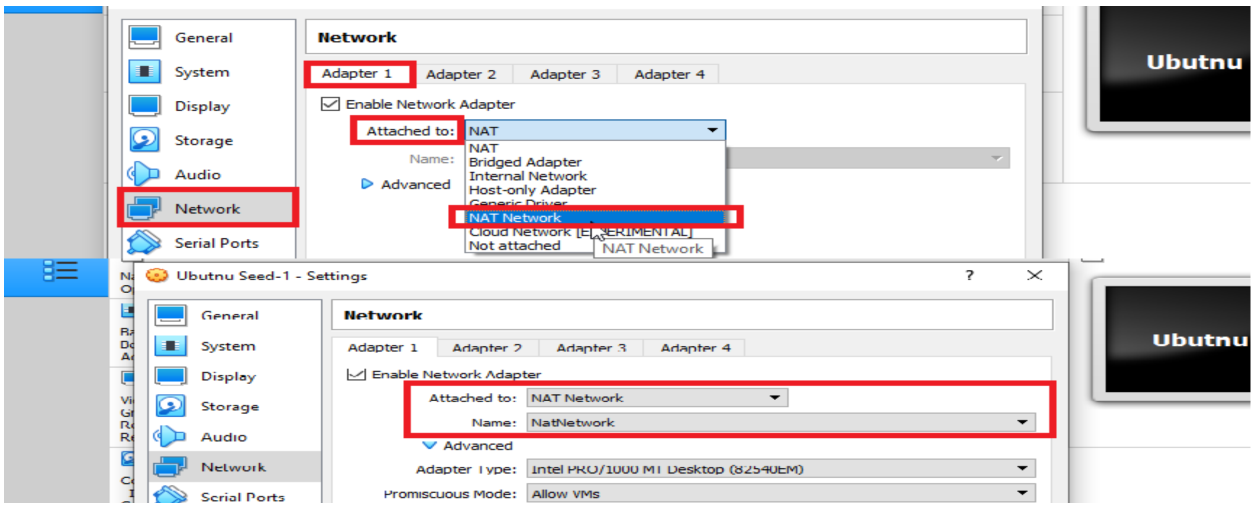
2. After that go to the virtual box setting and select Network from the tab on the left panel. Then I click on the "+" button to create a new NAT Networks adapter. After that, I open the virtual machine setting, select Network from the tab on the left panel, and staying on Adapter 1 and under enable network adapter I click on the "Attah to" drop-down menu and select NAT Network adapter that I just created and click ok.



Network Name: NatNetwork
Network CIDR: 10.0.2.0/24
Supports DHCP: yes
Supports IPv6: no

Enable Network: ✓
Network Name: NatNetwork
Network CIDR: 10.0.2.0/24
Network Options: ✓ Supports DHCP; Supports IPv6; Advertise Default IPv6 Route

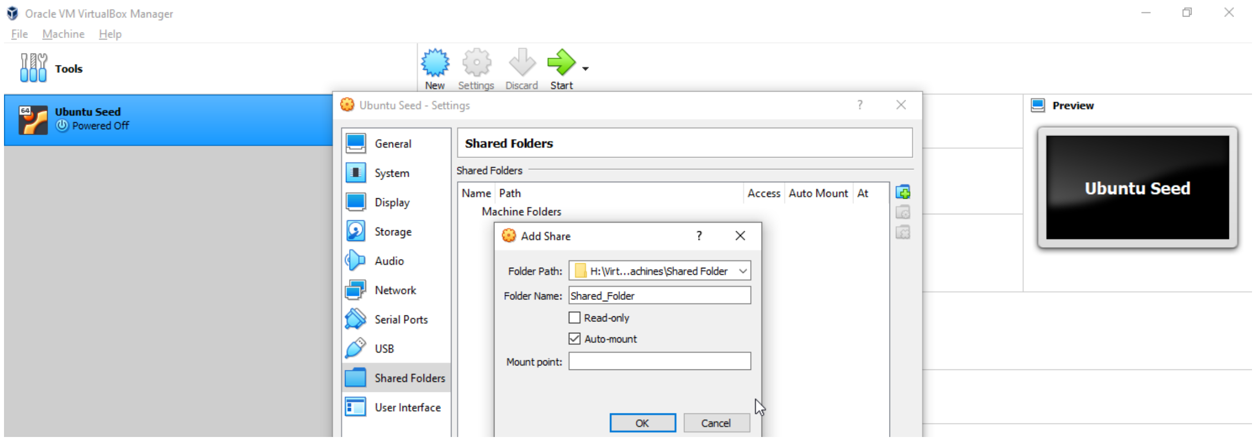
Figure 3: Creating new Nat Adapter from Virtual Box Setting



Enable Network Adapter: ✓
Attached to: NAT Network
Name: NatNetwork
Adapter Type: Intel PRO/1000 MT Desktop (82540EM)
Promiscuous Mode: Allow VMs

Figure 4: Adding Nat adapter in Virtual Machine setting

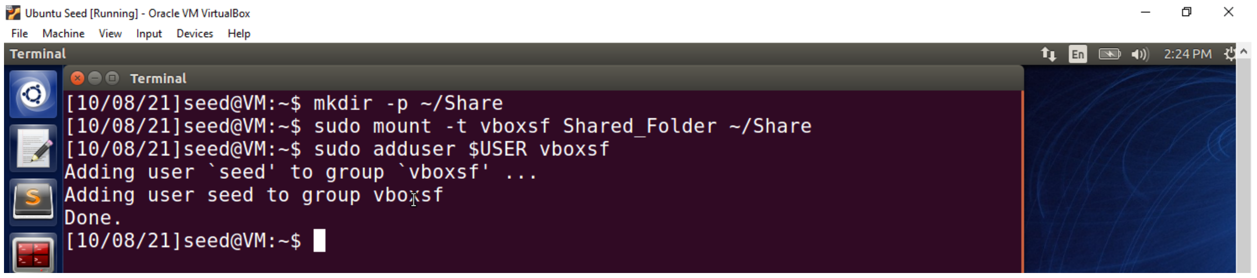
3. After that, I go to the VM settings and click on the shared folder to create the shared folder between the host and the virtual machine so that every time I put something in that folder, it can be accessed by both VM and host machine.



Folder Path: H:\Virt...achines\Shared Folder
Folder Name: Shared_Folder
Read-only: ☐
Auto-mount: ☑
Mount point:

Figure 5: Go to Virtual Machine Shared Folder Setting

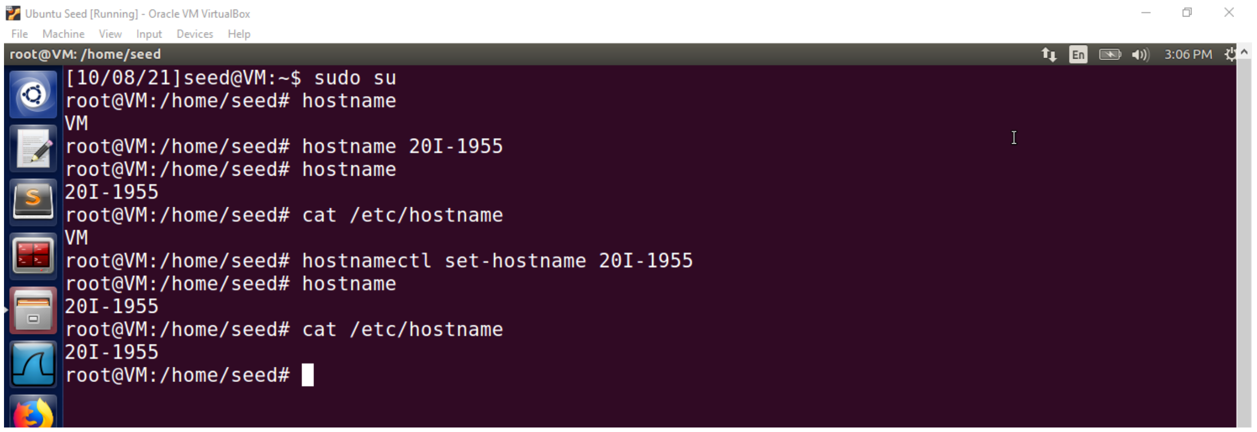
4. After that, I start the virtual machine, and then to mount the shared folder to the home directory of the Virtual machine I create a folder called “Share” in the home directory and then mount the shared folder to this “Share” folder.


Ubuntu Seed [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
2:24 PM

```
[10/08/21]seed@VM:~$ mkdir -p ~/Share
[10/08/21]seed@VM:~$ sudo mount -t vboxsf Shared_Folder ~/Share
[10/08/21]seed@VM:~$ sudo adduser $USER vboxsf
Adding user `seed' to group `vboxsf' ...
Adding user seed to group vboxsf
Done.
[10/08/21]seed@VM:~$
```

Figure 6: Mount the shared folder to the home directory folder

5. After that, I change the hostname of the virtual machine to my registration number to maintain the authenticity of the work.


Ubuntu Seed [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
root@VM: /home/seed
3:06 PM

```
[10/08/21]seed@VM:~$ sudo su
root@VM:/home/seed# hostname
VM
root@VM:/home/seed# hostname 20I-1955
root@VM:/home/seed# hostname
20I-1955
root@VM:/home/seed# cat /etc/hostname
VM
root@VM:/home/seed# hostnamectl set-hostname 20I-1955
root@VM:/home/seed# hostname
20I-1955
root@VM:/home/seed# cat /etc/hostname
20I-1955
root@VM:/home/seed#
```

Figure 7: Changing the hostname of the virtual machine

Part-2 RSA Public-Key Encryption and Signature Lab

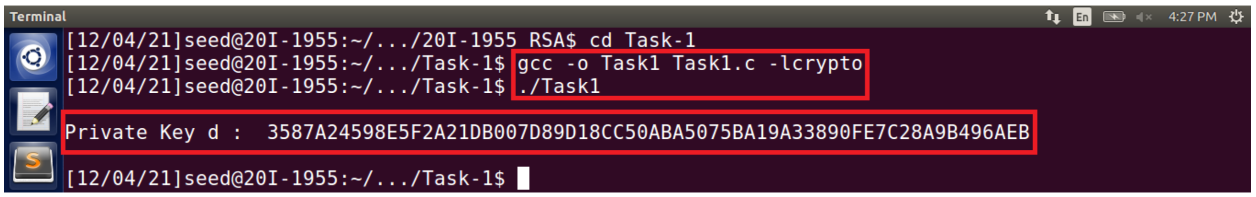
In this lab I have to perform different tasks based on the RSA algorithm that includes driving the private key, encryption, decryption, signing the message, and verifying the signature.

Task-1 Driving the Private key

In this task, I have given the three prime numbers p, q and e with their hexadecimal values and I have to find the private key d. Now to find the private key first I need to find the value of n which is $\boldsymbol{n = p * q}$ and Φn which is $\boldsymbol{\Phi n} = (\boldsymbol{p} - \mathbf{1}) * (\boldsymbol{q} - \mathbf{1})$. Then I need to find that the values of e and Φn are relatively prime or not by using the formula $\mathbf{gcd}(\boldsymbol{\Phi n}, \boldsymbol{e}) = \mathbf{1}$. In last private key can be found using the Extended Euclidian Algorithm which is $\boldsymbol{d} * \boldsymbol{e}\ \boldsymbol{mod}\ \boldsymbol{\Phi n} = \mathbf{1}$. To perform the task, I have to write the code using BIGNUM APIs that will perform all these actions and find the value of d. After writing the code I compile the code using the command "**gcc -o Task1 Task1.c -lcrypto**" and then run it using the command "**./Task1**".

Table 1: Task-1 Code

```c
#include <stdio.h>
#include <openssl/bn.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 *p = BN_new();
    BIGNUM *q = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *phiOfN = BN_new();
    BIGNUM *p_minus_one = BN_new(); //p-1
    BIGNUM *q_minus_one = BN_new(); //q-1
    BIGNUM *e = BN_new();
    BIGNUM *rPrime = BN_new(); //Relatively Prime
    BIGNUM *d = BN_new();
    BIGNUM *one = BN_new(); //1

    // Initializing the values of p,q,e,1
    BN_hex2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF");
    BN_hex2bn(&q, "E85CED54AF57E53E092113E62F436F4F");
    BN_hex2bn(&e, "0D88C3");
    BN_dec2bn(&one, "1");

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

    // Calculate the phiOfN = (p - 1)*(q - 1)
    BN_sub(p_minus_one, p, one);
    BN_sub(q_minus_one, q, one);
    BN_mul(phiOfN, p_minus_one, q_minus_one, ctx);

    // check if gcd(phiOfN,e)=1
    BN_gcd(rPrime, phiOfN, e, ctx);
    if (!BN_is_one(rPrime))
    {
        printf("\nError: e and phiOfN are not relatively prime to each other ");
        exit(0);
    }

    // Calculate the vaule of d : d*e mod phiOfN = 1
    BN_mod_inverse(d, e, phiOfN, ctx);

    printBN("\nPrivate Key d : ", d);
    printf("\n");


    BN_clear_free(p);
    BN_clear_free(q);
    BN_clear_free(n);
    BN_clear_free(phiOfN);
    BN_clear_free(p_minus_one);
    BN_clear_free(q_minus_one);
    BN_clear_free(e);
    BN_clear_free(rPrime);
    BN_clear_free(d);
    BN_clear_free(one);
};
```

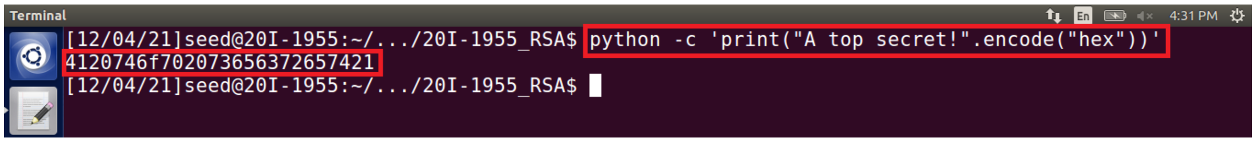


```
[12/04/21]seed@20I-1955:~/.../20I-1955 RSA$ cd Task-1
[12/04/21]seed@20I-1955:~/.../Task-1$ gcc -o Task1 Task1.c -lcrypto
[12/04/21]seed@20I-1955:~/.../Task-1$ ./Task1

Private Key d :  3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB

[12/04/21]seed@20I-1955:~/.../Task-1$
```

Figure 8: Private kay found successfully

Task-2 Encrypting the Message

In this task, I have to encrypt the given message “A top Secret!” and then verify the encryption result by decrypting the message with the help of a private key d. I have been provided with the values of n, e and d. The message can be encrypted by using the formula of $\boldsymbol{C = M^e \ mod \ n}$ and can be decrypted using the formula $\boldsymbol{M = C^d \ mod \ n}$. First I need to convert the message string into hexadecimal by using the command “**python -c ‘print(“A top-secret!”.encode(“hex”))’** ” and then I have to write and compile the code that encrypts the message and displays the result if the message is encrypted and verified successfully by decrypting the message.

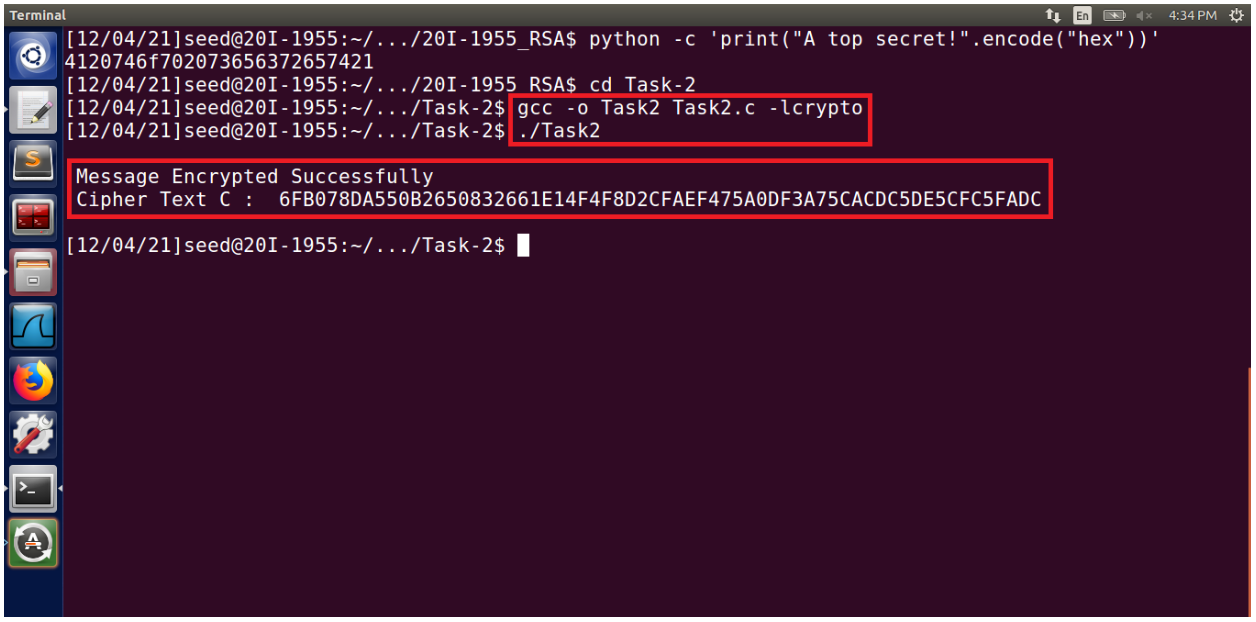


```
[12/04/21]seed@20I-1955:~/.../20I-1955_RSA$ python -c 'print("A top secret!".encode("hex"))'
4120746f702073656372657421
[12/04/21]seed@20I-1955:~/.../20I-1955_RSA$ 
```

Figure 9: Converting the message to hexadecimal

Table 2: Task-2 Code

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

    // Initializing the values of n, e, M, d
    BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&e, "010001");
    BN_hex2bn(&M, "4120746f702073656372657421"); // M = hex(A top secret!)
    BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    // encrypt C= M^e mod n
    BN_mod_exp(C, M, e, n, ctx);

    //decrypt dM= C^d mod n
    BN_mod_exp(dM, C, d, n, ctx);

    //verifying the result
    if (BN_cmp(M,dM) == 0)
    {
        printf("\n Message Encrypted Successfully");
        printBN("\n Cipher Text C : ", C);
        printf("\n");
    }
    else
        printf("\n Encryption Failed");

    BN_clear_free(n);
    BN_clear_free(e);
    BN_clear_free(M);
    BN_clear_free(dM);
    BN_clear_free(C);
    BN_clear_free(d);

    return 0;
}
```



```
[12/04/21]seed@20I-1955:~/.../20I-1955_RSA$ python -c 'print("A top secret!".encode("hex"))'
4120746f702073656372657421
[12/04/21]seed@20I-1955:~/.../20I-1955_RSA$ cd Task-2
[12/04/21]seed@20I-1955:~/.../Task-2$ gcc -o Task2 Task2.c -lcrypto
[12/04/21]seed@20I-1955:~/.../Task-2$ ./Task2

 Message Encrypted Successfully
 Cipher Text C :  6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC

[12/04/21]seed@20I-1955:~/.../Task-2$
```

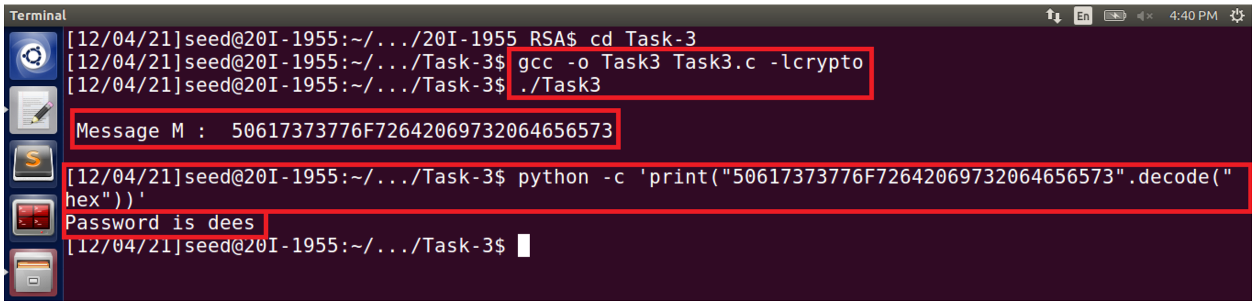
Figure 10: Message Encrypted Successfully

Task-3 Decrypting a Message

In this task, I have to decrypt the ciphertext given to reveal the original message. For this, I have been provided with the value of ciphertext c and the rest of the values n and d are from the previous task. The message can be decrypted by using the formula $\boldsymbol{M = C^d\ mod\ n}$. After decrypting the message we can convert the message from hexadecimal to text by using the command "**python -c 'print("50617373776F72642069732064656573".decode("hex"))**'".

Table 3: Task-3 Code

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

    // Initializing the values of c, d, n
    BN_hex2bn(&C, "8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");
    BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");

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

    printBN("\n Message M : ", M);
    printf("\n");

    BN_clear_free(C);
    BN_clear_free(d);
    BN_clear_free(n);
    BN_clear_free(M);
    return 0;
}
```

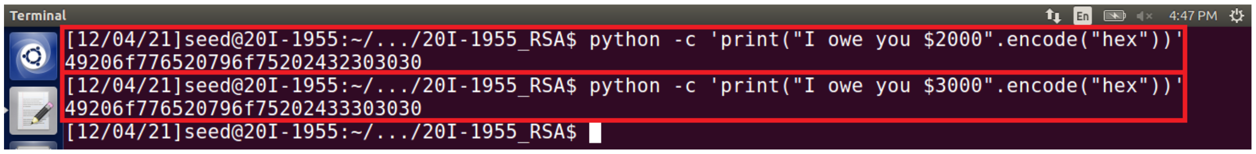


```
[12/04/21]seed@20I-1955:~/.../20I-1955_RSA$ cd Task-3
[12/04/21]seed@20I-1955:~/.../Task-3$ gcc -o Task3 Task3.c -lcrypto
[12/04/21]seed@20I-1955:~/.../Task-3$ ./Task3

 Message M :  50617373776F72642069732064656573

[12/04/21]seed@20I-1955:~/.../Task-3$ python -c 'print("50617373776F72642069732064656573".decode("
hex"))'
Password is dees
[12/04/21]seed@20I-1955:~/.../Task-3$ 
```

Figure 11: Message decrypted Successfully

Task-4 Signing a Message

In this task, I have to sign the message and then verify the result by verifying the signature by regenerating the message. In this task I have been provided with two messages $M1 = I\ owe\ you\ \$2000$, $M2 = I\ owe\ you\ \$3000$, and values of d, e, and n. First I convert the messages to hexadecimal by running the command "**python -c 'print("Message".encode("hex"))'**". The message can be signed by the formula of $\boldsymbol{Sig = M^d\ mod\ n}$ and can be verified by generating the message using the formula $\boldsymbol{VSig = Sig^e\ mod\ n}$.

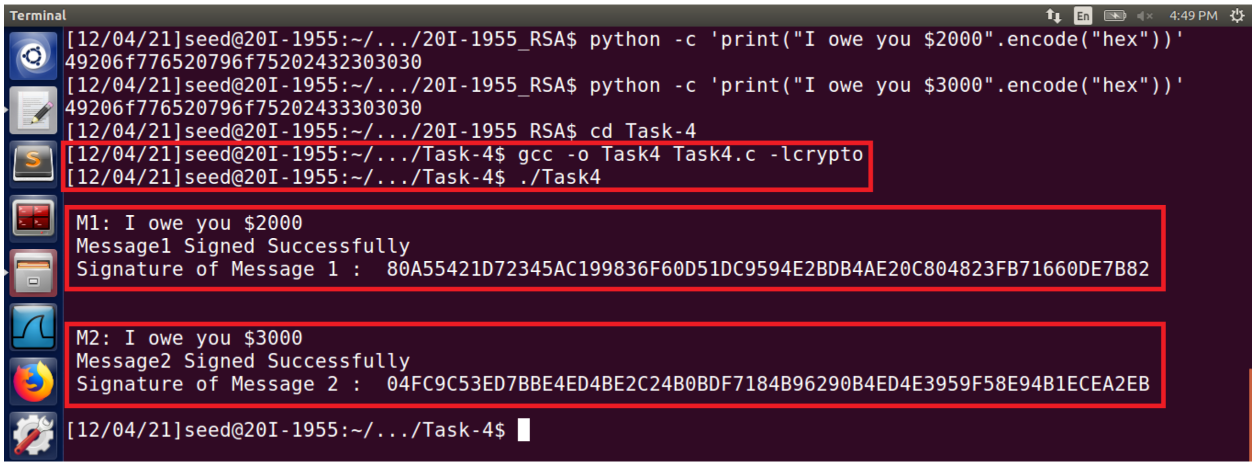


```
[12/04/21]seed@20I-1955:~/.../20I-1955_RSA$ python -c 'print("I owe you $2000".encode("hex"))'
49206f776520796f75202432303030
[12/04/21]seed@20I-1955:~/.../20I-1955_RSA$ python -c 'print("I owe you $3000".encode("hex"))'
49206f776520796f75202433303030
[12/04/21]seed@20I-1955:~/.../20I-1955_RSA$ 
```

Figure 12: Converting the messages to hexadecimal form

Table 4: Task-4 Code

```c
#include <stdio.h>
#include <openssl/bn.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 *M1 = BN_new();
    BIGNUM *M2 = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *Sig1 = BN_new();
    BIGNUM *Sig2 = BN_new();
    BIGNUM *VSig1 = BN_new();
    BIGNUM *VSig2 = BN_new();

    // Initilizing values of M1,M2,d,n,e
    BN_hex2bn(&M1, "49206f776520796f75202432303030"); // M = hex(I owe you $2000)
    BN_hex2bn(&M2, "49206f776520796f75202433303030"); // M = hex(I owe you $3000)
    BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&e, "010001");

    // encrypt Sig1 = M1^d mod n
    BN_mod_exp(Sig1, M1, d, n, ctx);
    printf("\n M1: I owe you $2000");
    // dcrypt VSig1 = Sig1^e mod n
    BN_mod_exp(VSig1, Sig1, e, n, ctx);

    //veryfing the signature of M1
    if (BN_cmp(M1,VSig1) == 0)
    {
        printf("\n Message1 Signed Successfully");
        printBN("\n Signature of Message 1 : ", Sig1);
    printf("\n");
    }
    else
        printf("\n Message Signed Failed");

    // encrypt Sig2= M2^d mod n
    BN_mod_exp(Sig2, M2, d, n, ctx);
    printf("\n M2: I owe you $3000");
    // dcrypt VSig2 = Sig2^e mod n
    BN_mod_exp(VSig2, Sig2, e, n, ctx);

    //veryfing the signature of M2
    if (BN_cmp(M2,VSig2) == 0)
    {
        printf("\n Message2 Signed Successfully");
        printBN("\n Signature of Message 2 : ", Sig2);
    printf("\n");
    }
    else
        printf("\n Message Signed Failed");
    BN_clear_free(M1);
    BN_clear_free(M2);
    BN_clear_free(d);
    BN_clear_free(n);
    BN_clear_free(e);
    BN_clear_free(Sig1);
    BN_clear_free(Sig2);
    BN_clear_free(VSig1);
    BN_clear_free(VSig2);

    return 0;
}
```



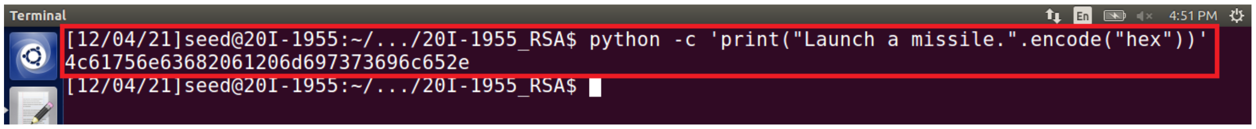
```
[12/04/21]seed@20I-1955:~/.../20I-1955_RSA$ python -c 'print("I owe you $2000".encode("hex"))'
49206f776520796f75202432303030
[12/04/21]seed@20I-1955:~/.../20I-1955_RSA$ python -c 'print("I owe you $3000".encode("hex"))'
49206f776520796f75202433303030
[12/04/21]seed@20I-1955:~/.../20I-1955_RSA$ cd Task-4
[12/04/21]seed@20I-1955:~/.../Task-4$ gcc -o Task4 Task4.c -lcrypto
[12/04/21]seed@20I-1955:~/.../Task-4$ ./Task4

 M1: I owe you $2000
 Message1 Signed Successfully
 Signature of Message 1 :  80A55421D72345AC199836F60D51DC9594E2BDB4AE20C804823FB71660DE7B82

 M2: I owe you $3000
 Message2 Signed Successfully
 Signature of Message 2 :  04FC9C53ED7BBE4ED4BE2C24B0BDF7184B96290B4ED4E3959F58E94B1ECEA2EB

[12/04/21]seed@20I-1955:~/.../Task-4$
```

Figure 13: Message Signed Successfully

After running the code, we can see that a slight change in the message will produce an entirely different signature of the message.

Task-5 Verifying the signature

Bob receives a message from Alice that is “Launch a missile.” and with her signature s. In this task, I have to verify whether the message is from Alice or not. For that, I have been provided with Alice public key e and the value of n. First I convert the messages to hexadecimal by running the command “**python -c ‘print(“Launch the missile.”.encode(“hex”))’**”. Then to verify that the signature is generated from message M, I use the formula $\boldsymbol{VSig = Sig^e \bmod n}$. Now if messages M and $VSig$ are the same then the message is authentic as it is signed by Alice's signature.

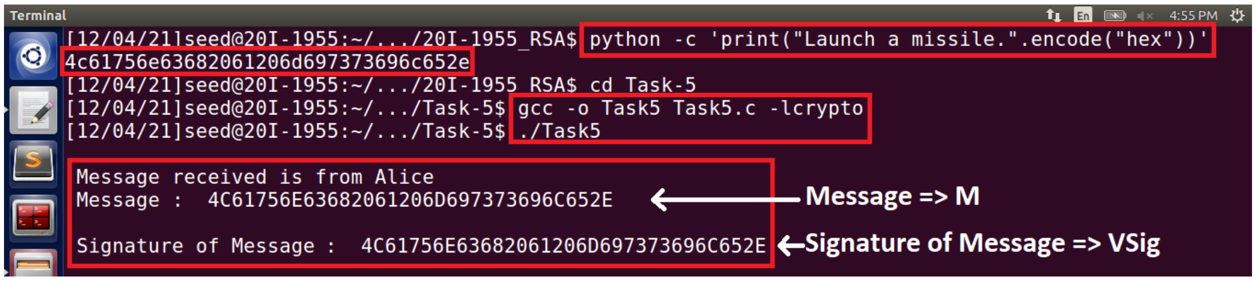


```
[12/04/21]seed@20I-1955:~/.../20I-1955_RSA$ python -c 'print("Launch a missile.".encode("hex"))'
4c61756e63682061206d697373696c652e
[12/04/21]seed@20I-1955:~/.../20I-1955_RSA$
```

Figure 14: Message converted to hexadecimal

Table 5: Task-5 Code

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

    // Initilizing values of M,n,e,1sig
    BN_hex2bn(&M, "4c61756e63682061206d697373696c652e"); //M = hex(Launch a missile.)
    BN_hex2bn(&n, "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");
    BN_hex2bn(&e, "010001");

    //Original Signature
    BN_hex2bn(&Sig, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");

    //Corrupted Signature
    //BN_hex2bn(&Sig, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F");

    // decrypt VSig= Sig^e mod n
    BN_mod_exp(VSig, Sig, e, n, ctx);

    //veryfing the signature of M
    if (BN_cmp(M,VSig) == 0)
    {
        printf("\n Message received is from Alice");

        printBN("\n Message : ", M);
        printBN("\n Signature of Message 1 : ", VSig);
    printf("\n");
    }
    else
    {
        printf("\n Signature Verification Failed");
        printf("\n Message received is not from Alice");
        printBN("\n Message : ", VSig);
        printf("\n");
    }

    BN_clear_free(M);
    BN_clear_free(n);
    BN_clear_free(e);
    BN_clear_free(Sig);
    BN_clear_free(VSig);

    return 0;
}
```



```
[12/04/21]seed@20I-1955:~/.../20I-1955_RSA$ python -c 'print("Launch a missile.".encode("hex"))'
4c61756e63682061206d697373696c652e
[12/04/21]seed@20I-1955:~/.../20I-1955_RSA$ cd Task-5
[12/04/21]seed@20I-1955:~/.../Task-5$ gcc -o Task5 Task5.c -lcrypto
[12/04/21]seed@20I-1955:~/.../Task-5$ ./Task5

Message received is from Alice
Message :  4C61756E63682061206D697373696C652E

Signature of Message :  4C61756E63682061206D697373696C652E
```

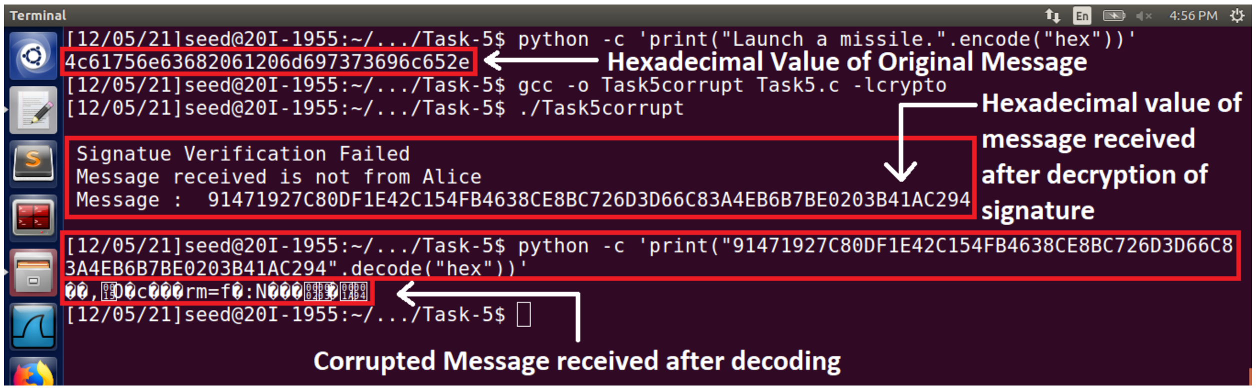
Figure 15: Signature verified Successfully

After running the code, we can see that messages M and $VSig$ are the same so it is verified that the message is from Alice itself.

If we change the last byte of the signature from 2F to 3F and then run the code, we get the hex value that is entirely different from the hex value of the message itself. Now if we decode back the hex value, we get the corrupted characters. This means that the signature received is not from Alice.

Table 6: Changing the last byte of signature in the code

```
  BN_hex2bn(&e, "010001");

     //Original Signature
     //BN_hex2bn(&Sig, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");

     //Corrupted Signature
     BN_hex2bn(&Sig, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F");

    // decrypt VSig= Sig^e mod n
  BN_mod_exp(VSig, Sig, e, n, ctx);
```



```
[12/05/21]seed@20I-1955:~/.../Task-5$ python -c 'print("Launch a missile.".encode("hex"))'
4c61756e63682061206d697373696c652e
[12/05/21]seed@20I-1955:~/.../Task-5$ gcc -o Task5corrupt Task5.c -lcrypto
[12/05/21]seed@20I-1955:~/.../Task-5$ ./Task5corrupt

 Signatue Verification Failed
 Message received is not from Alice
 Message :  91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294

[12/05/21]seed@20I-1955:~/.../Task-5$ python -c 'print("91471927C80DF1E42C154FB4638CE8BC726D3D66C8
3A4EB6B7BE0203B41AC294".decode("hex"))'
��,�D�c���rm=f�:N�����
[12/05/21]seed@20I-1955:~/.../Task-5$
```

Hexadecimal Value of Original Message
Hexadecimal value of message received after decryption of signature
Corrupted Message received after decoding

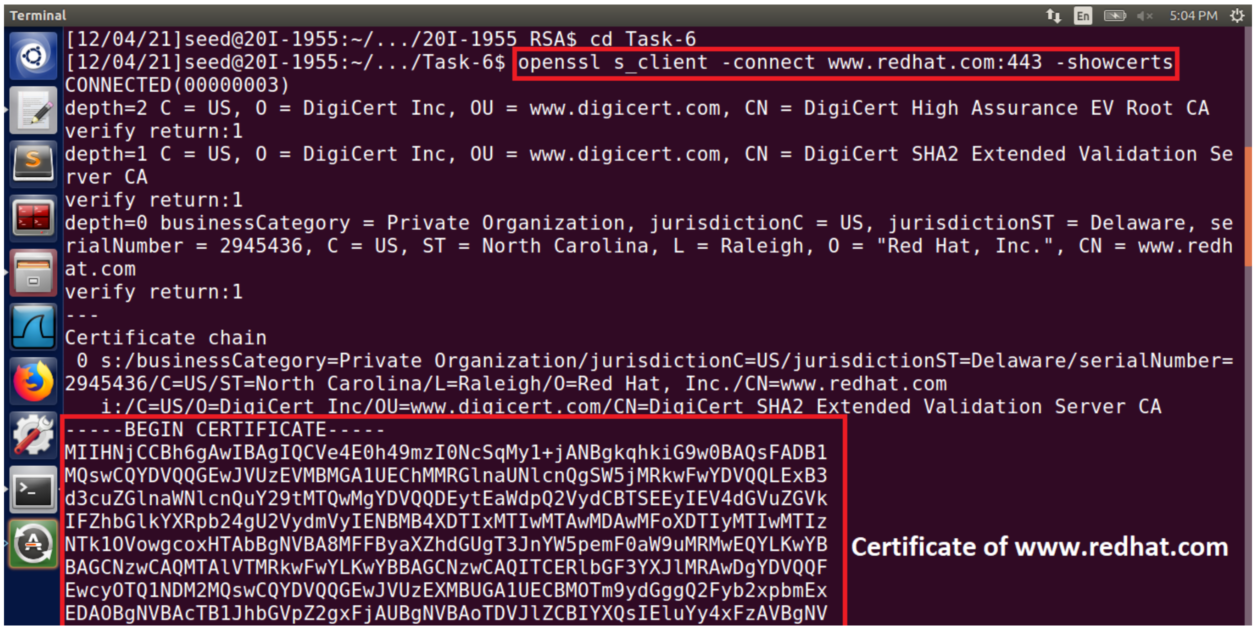
Figure 16: Signature Verification Failed

Task-6 Manually Verifying an X.509 Certificate

In this task, I have to manually verify the X.509 certificate of the website by downloading the certificate from the webserver, getting the issuer's public key, and then using it to verify the signature on the certificate.

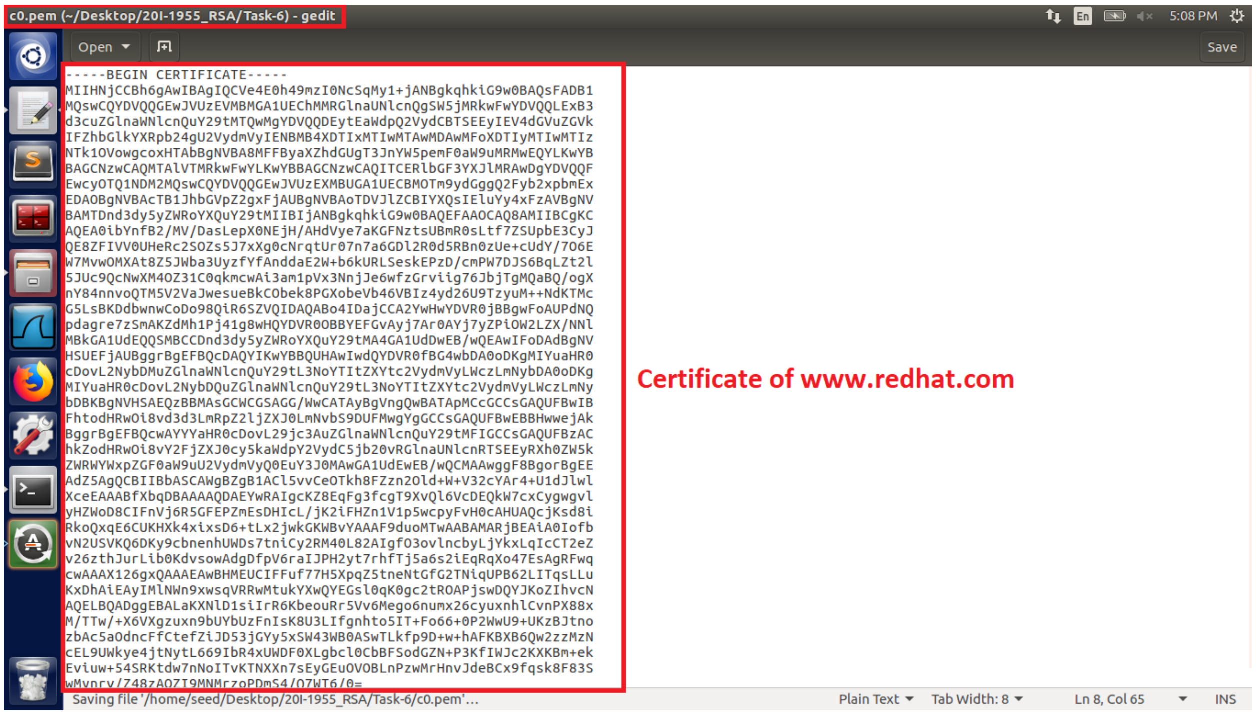
Step-1 Download a certificate from a real webserver

In this task, I will verify the X.509 certificate of www.redhat.com. For that first, I download the certificate from the website using the command "**openssl s_client -connect www.redhat.com:443 -showcerts**". After running the command, I get the results that contain two certificates. The first certificate belongs to www.redhat.com and the second certificate belongs to CA. I copy and paste each certificate from "BEGIN CERTIFICATE" to the line containing "END CERTIFICATE" into the files "c0.pem" and "c1.pem".



```
[12/04/21]seed@20I-1955:~/.../20I-1955 RSA$ cd Task-6
[12/04/21]seed@20I-1955:~/.../Task-6$ openssl s_client -connect www.redhat.com:443 -showcerts
CONNECTED(00000003)
depth=2 C = US, O = DigiCert Inc, OU = www.digicert.com, CN = DigiCert High Assurance EV Root CA
verify return:1
depth=1 C = US, O = DigiCert Inc, OU = www.digicert.com, CN = DigiCert SHA2 Extended Validation Se
rver CA
verify return:1
depth=0 businessCategory = Private Organization, jurisdictionC = US, jurisdictionST = Delaware, se
rialNumber = 2945436, C = US, ST = North Carolina, L = Raleigh, O = "Red Hat, Inc.", CN = www.redh
at.com
verify return:1
---
Certificate chain
 0 s:/businessCategory=Private Organization/jurisdictionC=US/jurisdictionST=Delaware/serialNumber=
2945436/C=US/ST=North Carolina/L=Raleigh/O=Red Hat, Inc./CN=www.redhat.com
   i:/C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert SHA2 Extended Validation Server CA
-----BEGIN CERTIFICATE-----
MIIHNjCCBh6gAwIBAgIQCVe4E0h49mzI0NcSqMy1+jANBgkqhkiG9w0BAQsFADB1
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
d3cuZGlnaWNlcnQuY29tMTQwMgYDVQQDEytEaWdpQ2VydCBTSEEyIEV4dGVuZGVk
IFZhbGlkYXRpb24gU2VydmVyIENBMB4XDTIxMTIwMTAwMDAwMFoXDTIyMTIwMTIz
NTk1OVowgcoxHTAbBgNVBA8MFFByaXZhdGUgT3JnYW5pemF0aW9uMRMwEQYLKwYB
BAGCNzwCAQMTAlVTMRkwFwYLKwYBBAGCNzwCAQITCERlbGF3YXJlMRAwDgYDVQQF
EwcyOTQ1NDM2MQswCQYDVQQGEwJVUzEXMBUGA1UECBMOTm9ydGggQ2Fyb2xpbmEx
EDAOBgNVBAcTB1JhbGVpZ2gxFjAUBgNVBAoTDVJlZCBIYXQsIEluYy4xFzAVBgNV
```

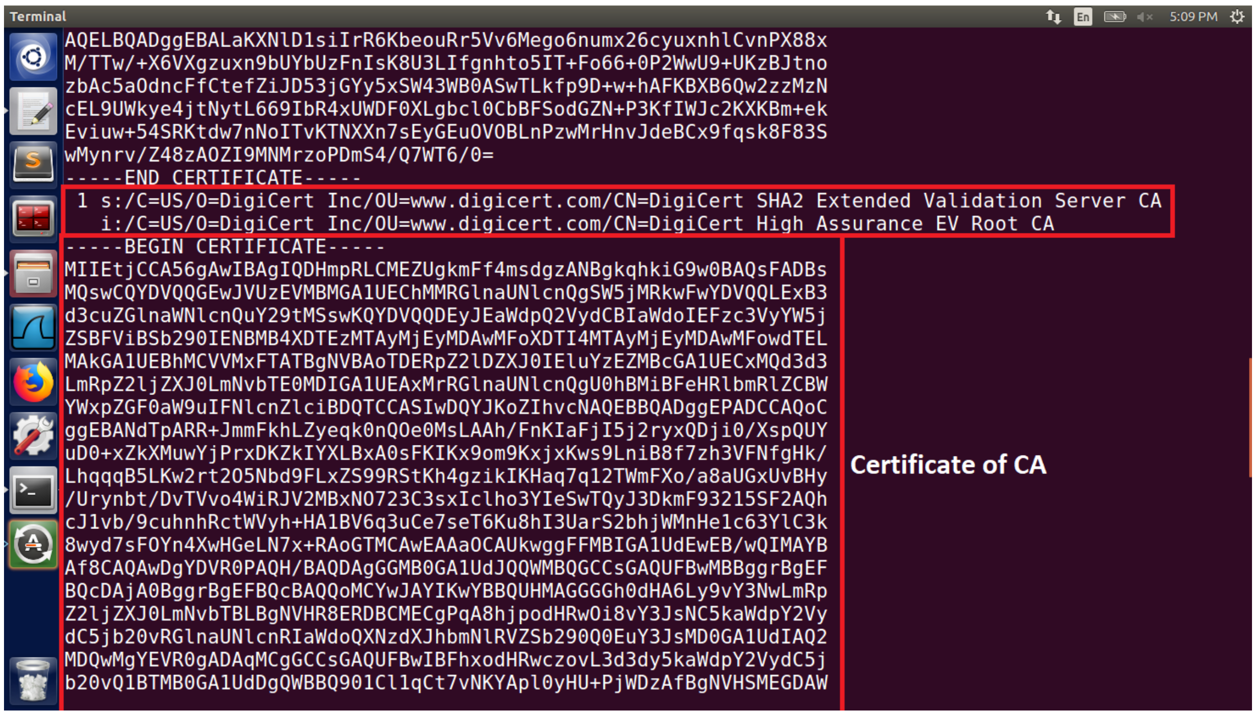
Figure 17: Certificate of www.redhat.com


c0.pem (~/Desktop/20I-1955_RSA/Task-6) - gedit
Open
Save
Certificate of www.redhat.com
Plain Text
Tab Width: 8
Ln 8, Col 65
INS
5:08 PM

```
-----BEGIN CERTIFICATE-----
MIIHNjCCBh6gAwIBAgIQCVe4E0h49mzI0NcSqMy1+jANBgkqhkiG9w0BAQsFADB1
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
d3cuZGlnaWNlcnQuY29tMTQwMgYDVQQDEytEaWdpQ2VydCBTSEEyIEV4dGVuZGVk
IFZhbGlkYXRpb24gU2VydmVyIENBMB4XDTIxMTIwMTAwMDAwMFoXDTIyMTIwMTIz
NTk1OVowgcoxHTAbBgNVBA8MFFByaXZhdGUgT3JnYW5pemF0aW9uMRMwEQYLKwYB
BAGCNzwCAQMTAlVTMRkwFwYLKwYBBAGCNzwCAQITCERlbGF3YXJlMRAwDgYDVQQF
EwcyOTQ1NDM2MQswCQYDVQQGEwJVUzEXMBUGA1UECBMOTm9ydGggQ2Fyb2xpbmEx
EDAOBgNVBAcTB1JhbGVpZ2gxFjAUBgNVBAoTDVJlZCBIYXQsIEluYy4xFzAVBgNV
BAMTDnd3dy5yZWRoYXQuY29tMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKC
AQEA0ibYnfB2/MV/DasLepX0NEjH/AHdVye7aKGFNztsUBmR0sLtf7ZSUpbE3CyJ
QE8ZFIVV0UHeRc2SOZs5J7xXg0cNrqtUr07n7a6GDl2R0d5RBn0zUe+cUdY/7O6E
W7MvwOMXAt8Z5JWba3UyzfYfAnddaE2W+b6kURLSeskEPzD/cmPW7DJS6BqLZt2l
5JUc9QcNwXM4OZ31C0qkmcwAi3am1pVx3NnjJe6wfzGrviig76JbjTgMQaBQ/ogX
nY84nnvoQTM5V2VaJwesueBkCObek8PGXobeVb46VBIz4yd26U9TzyuM++NdKTMc
G5LsBKDdbwnwCoDo98QiR6SZVQIDAQABo4IDajCCA2YwHwYDVR0jBBgwFoAUPdNQ
pdagre7zSmAKZdMh1Pj41g8wHQYDVR0OBBYEFGvAyj7Ar0AYj7yZPiOW2LZX/NNl
MBkGA1UdEQQSMBCCDnd3dy5yZWRoYXQuY29tMA4GA1UdDwEB/wQEAwIFoDAdBgNV
HSUEFjAUBggrBgEFBQcDAQYIKwYBBQUHAwIwdQYDVR0fBG4wbDA0oDKgMIYuaHR0
cDovL2NybDMuZGlnaWNlcnQuY29tL3NoYTItZXYtc2VydmVyLWczLmNybDA0oDKg
MIYuaHR0cDovL2NybDQuZGlnaWNlcnQuY29tL3NoYTItZXYtc2VydmVyLWczLmNy
bDBKBgNVHSAEQzBBMAsGCWCGSAGG/WwCATAyBgVngQwBATApMCcGCCsGAQUFBwIB
FhtodHRwOi8vd3d3LmRpZ2ljZXJ0LmNvbS9DUFMwgYgGCCsGAQUFBwEBBHwwejAk
BggrBgEFBQcwAYYYaHR0cDovL29jc3AuZGlnaWNlcnQuY29tMFIGCCsGAQUFBzAC
hkZodHRwOi8vY2FjZXJ0cy5kaWdpY2VydC5jb20vRGlnaUNlcnRTSEEyRXh0ZW5k
ZWRWYWxpZGF0aW9uU2VydmVyQ0EuY3J0MAwGA1UdEwEB/wQCMAAwggF8BgorBgEE
AdZ5AgQCBIIBbASCAWgBZgB1ACl5vvCeOTkh8FZzn2Old+W+V32cYAr4+U1dJlwl
XceEAAABfXbqDBAAAAQDAEYwRAIgcKZ8EqFg3fcgT9XvQl6VcDEQkW7cxCygwgvl
yHZWoD8CIFnVj6R5GFEPZmEsDHIcL/jK2iFHZn1V1p5wcpyFvH0cAHUAQcjKsd8i
RkoQxqE6CUKHXk4xixsD6+tLx2jwkGKWBvYAAAF9duoMTwAABAMARjBEAiA0Iofb
vN2USVKQ6DKy9cbnenhUWDs7tniCy2RM40L82AIgfO3ovlncbyLjYkxLqIcCT2eZ
v26zthJurLib0KdvsowAdgDfpV6raIJPH2yt7rhfTj5a6s2iEqRqXo47EsAgRFwq
cwAAAX126gxQAAAEAwBHMEUCIFFuf77H5XpqZ5tneNtGfG2TNiqUPB62LITqsLLu
KxDhAiEAyIMlNWn9xwsqVRRwMtukYXwQYEGsl0qK0gc2tROAPjswDQYJKoZIhvcN
AQELBQADggEBALaKXNlD1siIrR6KbeouRr5Vv6Mego6numx26cyuxnhlCvnPX88x
M/TTw/+X6VXgzuxn9bUYbUzFnIsK8U3LIfgnhto5IT+Fo66+0P2WwU9+UKzBJtno
zbAc5aOdncFfCtefZiJD53jGYy5xSW43WB0ASwTLkfp9D+w+hAFKBXB6Qw2zzMzN
cEL9UWkye4jtNytL669IbR4xUWDF0XLgbcl0CbBFSodGZN+P3KfIWJc2KXKBm+ek
Eviuw+54SRKtdw7nNoITvKTNXXn7sEyGEuOVOBLnPzwMrHnvJdeBCx9fqsk8F83S
wMynrv/Z48zAOZI9MNMrzoPDmS4/Q7WT6/0=
```

```
Saving file '/home/seed/Desktop/20I-1955_RSA/Task-6/c0.pem'...
```

Figure 18: Certificate of www.redhat.com

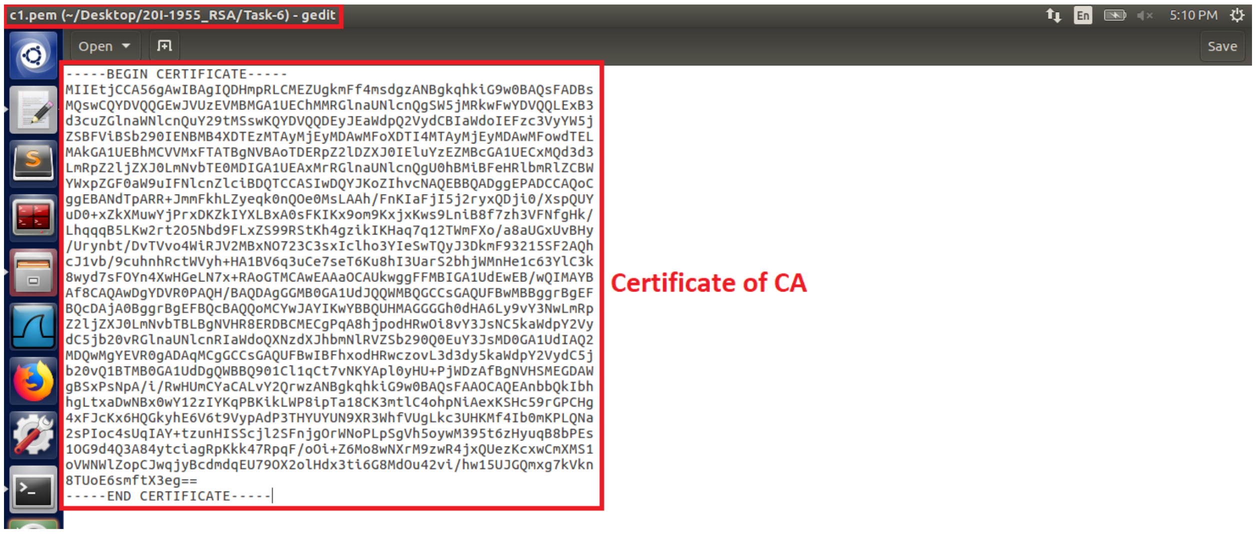


```
AQELBQADggEBALaKXNlD1siIrR6KbeouRr5Vv6Mego6numx26cyuxnhlCvnPX88x
M/TTw/+X6VXgzuxn9bUYbUzFnIsK8U3LIfgnhto5IT+Fo66+0P2WwU9+UKzBJtno
zbAc5aOdncFfCtefZiJD53jGYy5xSW43WB0ASwTLkfp9D+w+hAFKBXB6Qw2zzMzN
cEL9UWkye4jtNytL669IbR4xUWDF0XLgbcl0CbBFSodGZN+P3KfIWJc2KXKBm+ek
Eviuw+54SRKtdw7nNoITvKTNXXn7sEyGEuOVOBLnPzwMrHnvJdeBCx9fqsk8F83S
wMynrv/Z48zAOZI9MNMrzoPDmS4/Q7WT6/0=
-----END CERTIFICATE-----
```

```
 1 s:/C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert SHA2 Extended Validation Server CA
   i:/C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert High Assurance EV Root CA
```

```
-----BEGIN CERTIFICATE-----
MIIEtjCCA56gAwIBAgIQDHmpRLCMEZUgkmFf4msdgzANBgkqhkiG9w0BAQsFADBs
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
d3cuZGlnaWNlcnQuY29tMSswKQYDVQQDEyJEaWdpQ2VydCBIaWdoIEFzc3VyYW5j
ZSBFViBSb290IENBMB4XDTEzMTAyMjEyMDAwMFoXDTI4MTAyMjEyMDAwMFowdTEL
MAkGA1UEBhMCVVMxFTATBgNVBAoTDERpZ2lDZXJ0IEluYzEZMBcGA1UECxMQd3d3
LmRpZ2ljZXJ0LmNvbTE0MDIGA1UEAxMrRGlnaUNlcnQgU0hBMiBFeHRlbmRlZCBW
YWxpZGF0aW9uIFNlcnZlciBDQTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoC
ggEBANdTpARR+JmmFkhLZyeqk0nQOe0MsLAAh/FnKIaFjI5j2ryxQDji0/XspQUY
uD0+xZkXMuwYjPrxDKZkIYXLBxA0sFKIKx9om9KxjxKws9LniB8f7zh3VFNfgHk/
LhqqqB5LKw2rt2O5Nbd9FLxZS99RStKh4gzikIKHaq7q12TWmFXo/a8aUGxUvBHy
/Urynbt/DvTVvo4WiRJV2MBxNO723C3sxIclho3YIeSwTQyJ3DkmF93215SF2AQh
cJ1vb/9cuhnhRctWVyh+HA1BV6q3uCe7seT6Ku8hI3UarS2bhjWMnHe1c63YlC3k
8wyd7sFOYn4XwHGeLN7x+RAoGTMCAwEAAaOCAUkwggFFMBIGA1UdEwEB/wQIMAYB
Af8CAQAwDgYDVR0PAQH/BAQDAgGGMB0GA1UdJQQWMBQGCCsGAQUFBwMBBggrBgEF
BQcDAjA0BggrBgEFBQcBAQQoMCYwJAYIKwYBBQUHMAGGGGh0dHA6Ly9vY3NwLmRp
Z2ljZXJ0LmNvbTBLBgNVHR8ERDBCMECgPqA8hjpodHRwOi8vY3JsNC5kaWdpY2Vy
dC5jb20vRGlnaUNlcnRIaWdoQXNzdXJhbmNlRVZSb290Q0EuY3JsMD0GA1UdIAQ2
MDQwMgYEVR0gADAqMCgGCCsGAQUFBwIBFhxodHRwczovL3d3dy5kaWdpY2VydC5j
b20vQ1BTMB0GA1UdDgQWBBQ901Cl1qCt7vNKYApl0yHU+PjWDzAfBgNVHSMEGDAW
```

Figure 19: Certificate of CA

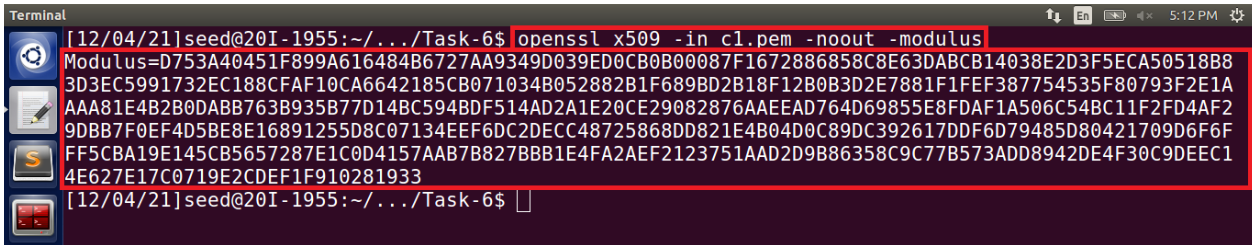


```
-----BEGIN CERTIFICATE-----
MIIEtjCCA56gAwIBAgIQDHmpRLCMEZUgkmFf4msdgzANBgkqhkiG9w0BAQsFADBs
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
d3cuZGlnaWNlcnQuY29tMSswKQYDVQQDEyJEaWdpQ2VydCBIaWdoIEFzc3VyYW5j
ZSBFViBSb290IENBMB4XDTEzMTAyMjEyMDAwMFoXDTI4MTAyMjEyMDAwMFowdTEL
MAkGA1UEBhMCVVMxFTATBgNVBAoTDERpZ2lDZXJ0IEluYzEZMBcGA1UECxMQd3d3
LmRpZ2ljZXJ0LmNvbTE0MDIGA1UEAxMrRGlnaUNlcnQgU0hBMiBFeHRlbmRlZCBW
YWxpZGF0aW9uIFNlcnZlciBDQTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoC
ggEBANdTpARR+JmmFkhLZyeqk0nQOe0MsLAAh/FnKIaFjI5j2ryxQDji0/XspQUY
uD0+xZkXMuwYjPrxDKZkIYXLBxA0sFKIKx9om9KxjxKws9LniB8f7zh3VFNfgHk/
LhqqqB5LKw2rt2O5Nbd9FLxZS99RStKh4gzikIKHaq7q12TWmFXo/a8aUGxUvBHy
/Urynbt/DvTVvo4WiRJV2MBxNO723C3sxIclho3YIeSwTQyJ3DkmF93215SF2AQh
cJ1vb/9cuhnhRctWVyh+HA1BV6q3uCe7seT6Ku8hI3UarS2bhjWMnHe1c63YlC3k
8wyd7sFOYn4XwHGeLN7x+RAoGTMCAwEAAaOCAUkwggFFMBIGA1UdEwEB/wQIMAYB
Af8CAQAwDgYDVR0PAQH/BAQDAgGGMB0GA1UdJQQWMBQGCCsGAQUFBwMBBggrBgEF
BQcDAjA0BggrBgEFBQcBAQQoMCYwJAYIKwYBBQUHMAGGGGh0dHA6Ly9vY3NwLmRp
Z2ljZXJ0LmNvbTBLBgNVHR8ERDBCMECgPqA8hjpodHRwOi8vY3JsNC5kaWdpY2Vy
dC5jb20vRGlnaUNlcnRIaWdoQXNzdXJhbmNlRVZSb290Q0EuY3JsMD0GA1UdIAQ2
MDQwMgYEVR0gADAqMCgGCCsGAQUFBwIBFhxodHRwczovL3d3dy5kaWdpY2VydC5j
b20vQ1BTMB0GA1UdDgQWBBQ901Cl1qCt7vNKYApl0yHU+PjWDzAfBgNVHSMEGDAW
gBSxPsNpA/i/RwHUmCYaCALvY2QrwzANBgkqhkiG9w0BAQsFAAOCAQEAnbbQkIbh
hgLtxaDwNBx0wY12zIYKqPBKikLWP8ipTa18CK3mtlC4ohpNiAexKSHc59rGPCHg
4xFJcKx6HQGkyhE6V6t9VypAdP3THYUYUN9XR3WhfVUgLkc3UHKMf4Ib0mKPLQNa
2sPIoc4sUqIAY+tzunHISScjl2SFnjgOrWNoPLpSgVh5oywM395t6zHyuqB8bPEs
1OG9d4Q3A84ytciagRpKkk47RpqF/oOi+Z6Mo8wNXrM9zwR4jxQUezKcxwCmXMS1
oVWNWlZopCJwqjyBcdmdqEU79OX2olHdx3ti6G8MdOu42vi/hw15UJGQmxg7kVkn
8TUoE6smftX3eg==
-----END CERTIFICATE-----
```

Figure 20: Certificate of CA

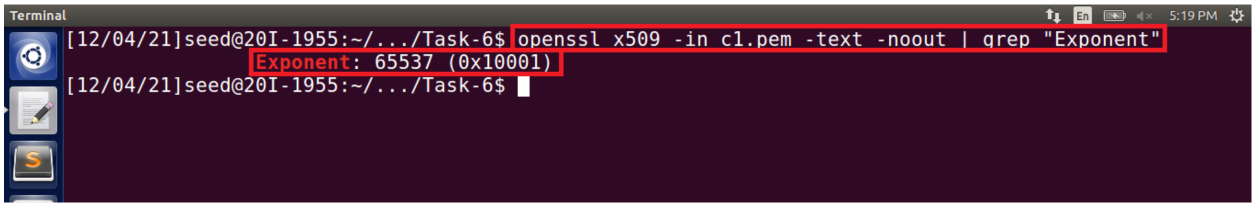
Step-2 Extract the public key (e, n) from the issuer's certificate.

In this step, I have to extract the values of modulus n and public key e from the CA's certificate. Openssl provides the commands to extract this information from the CA's certificate. To extract the value of modulus n, I use the command "**openssl x509 -in c1.pem -noout -modulus**" and to extract the value of exponent e using the command "**openssl x509 -in c1.pem -text -noout | grep "Exponent"**".



```
[12/04/21]seed@20I-1955:~/.../Task-6$ openssl x509 -in c1.pem -noout -modulus
Modulus=D753A40451F899A616484B6727AA9349D039ED0CB0B00087F1672886858C8E63DABCB14038E2D3F5ECA50518B8
3D3EC5991732EC188CFAF10CA6642185CB071034B052882B1F689BD2B18F12B0B3D2E7881F1FEF387754535F80793F2E1A
AAA81E4B2B0DABB763B935B77D14BC594BDF514AD2A1E20CE29082876AAEEAD764D69855E8FDAF1A506C54BC11F2FD4AF2
9DBB7F0EF4D5BE8E16891255D8C07134EEF6DC2DECC48725868DD821E4B04D0C89DC392617DDF6D79485D80421709D6F6F
FF5CBA19E145CB5657287E1C0D4157AAB7B827BBB1E4FA2AEF2123751AAD2D9B86358C9C77B573ADD8942DE4F30C9DEEC1
4E627E17C0719E2CDEF1F910281933
[12/04/21]seed@20I-1955:~/.../Task-6$
```

Figure 21: Extracting the value of modulus n from CA Certificate

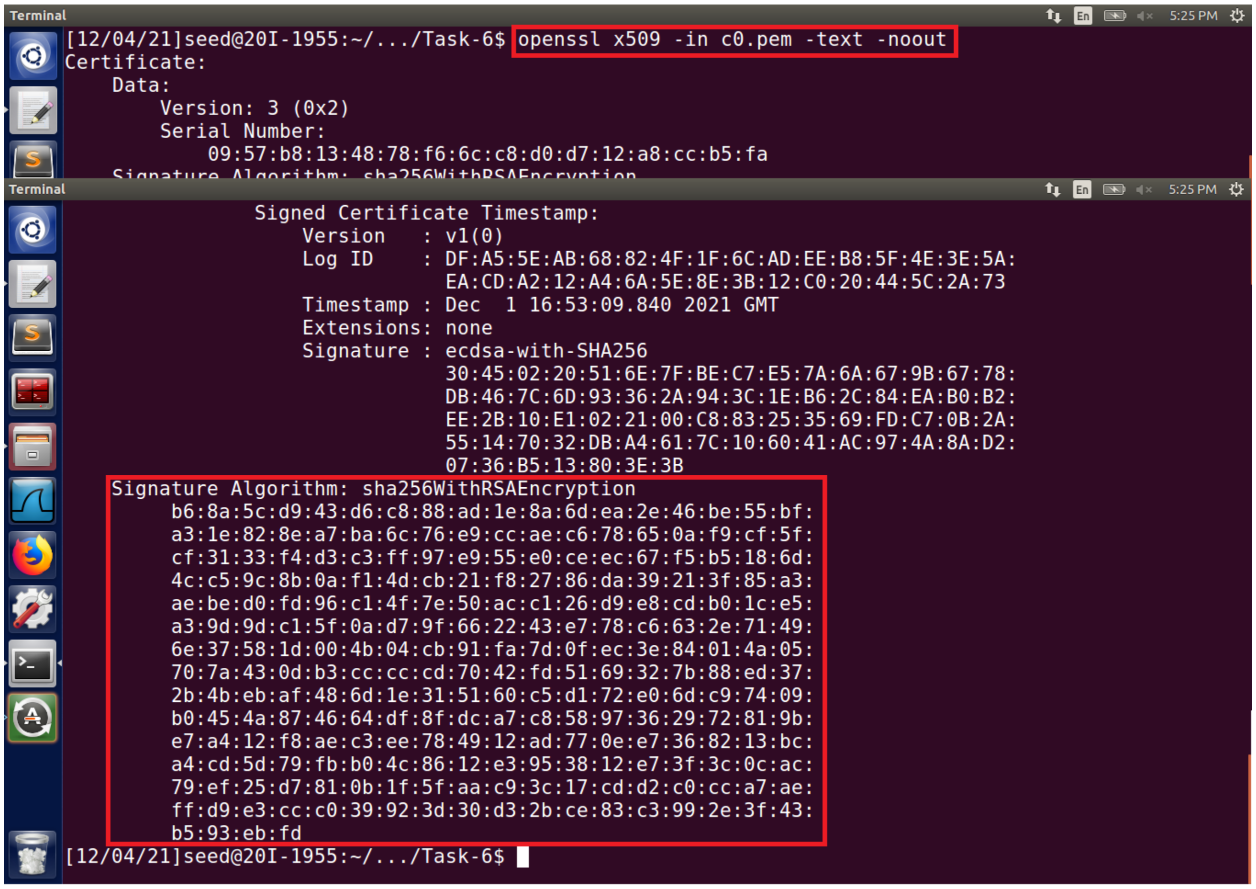


```
[12/04/21]seed@20I-1955:~/.../Task-6$ openssl x509 -in c1.pem -text -noout | grep "Exponent"
                Exponent: 65537 (0x10001)
[12/04/21]seed@20I-1955:~/.../Task-6$
```

Figure 22: Extracting the value of exponent e from CA Certificate

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

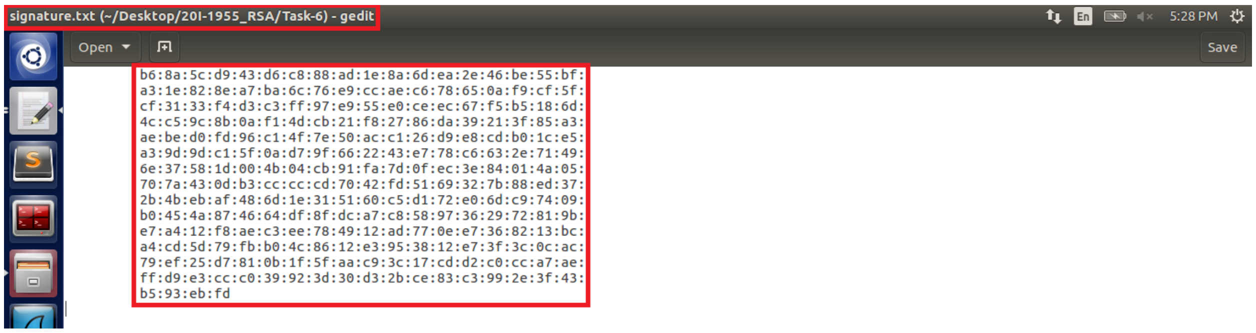
In this step, I have to extract the signature from the server's certificate. Openssl provides the command to display the content of the server's certificate from which we can extract the signature. To display the content of the server's certificate I use the command "**openssl x509 -in c0.pem -text -noout**". After getting the information of the signature I copy the signature into the file "signature.txt". The signature we found from the certificate contains spaces and colons, now to remove these spaces and colons I use the command "**cat signature.txt | tr -d '[:space:]:'**".



```
[12/04/21]seed@20I-1955:~/.../Task-6$ openssl x509 -in c0.pem -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            09:57:b8:13:48:78:f6:6c:c8:d0:d7:12:a8:cc:b5:fa
```

```
                Signed Certificate Timestamp:
                    Version   : v1(0)
                    Log ID    : DF:A5:5E:AB:68:82:4F:1F:6C:AD:EE:B8:5F:4E:3E:5A:
                                EA:CD:A2:12:A4:6A:5E:8E:3B:12:C0:20:44:5C:2A:73
                    Timestamp : Dec  1 16:53:09.840 2021 GMT
                    Extensions: none
                    Signature : ecdsa-with-SHA256
                                30:45:02:20:51:6E:7F:BE:C7:E5:7A:6A:67:9B:67:78:
                                DB:46:7C:6D:93:36:2A:94:3C:1E:B6:2C:84:EA:B0:B2:
                                EE:2B:10:E1:02:21:00:C8:83:25:35:69:FD:C7:0B:2A:
                                55:14:70:32:DB:A4:61:7C:10:60:41:AC:97:4A:8A:D2:
                                07:36:B5:13:80:3E:3B
    Signature Algorithm: sha256WithRSAEncryption
         b6:8a:5c:d9:43:d6:c8:88:ad:1e:8a:6d:ea:2e:46:be:55:bf:
         a3:1e:82:8e:a7:ba:6c:76:e9:cc:ae:c6:78:65:0a:f9:cf:5f:
         cf:31:33:f4:d3:c3:ff:97:e9:55:e0:ce:ec:67:f5:b5:18:6d:
         4c:c5:9c:8b:0a:f1:4d:cb:21:f8:27:86:da:39:21:3f:85:a3:
         ae:be:d0:fd:96:c1:4f:7e:50:ac:c1:26:d9:e8:cd:b0:1c:e5:
         a3:9d:9d:c1:5f:0a:d7:9f:66:22:43:e7:78:c6:63:2e:71:49:
         6e:37:58:1d:00:4b:04:cb:91:fa:7d:0f:ec:3e:84:01:4a:05:
         70:7a:43:0d:b3:cc:cc:cd:70:42:fd:51:69:32:7b:88:ed:37:
         2b:4b:eb:af:48:6d:1e:31:51:60:c5:d1:72:e0:6d:c9:74:09:
         b0:45:4a:87:46:64:df:8f:dc:a7:c8:58:97:36:29:72:81:9b:
         e7:a4:12:f8:ae:c3:ee:78:49:12:ad:77:0e:e7:36:82:13:bc:
         a4:cd:5d:79:fb:b0:4c:86:12:e3:95:38:12:e7:3f:3c:0c:ac:
         79:ef:25:d7:81:0b:1f:5f:aa:c9:3c:17:cd:d2:c0:cc:a7:ae:
         ff:d9:e3:cc:c0:39:92:3d:30:d3:2b:ce:83:c3:99:2e:3f:43:
         b5:93:eb:fd
[12/04/21]seed@20I-1955:~/.../Task-6$
```

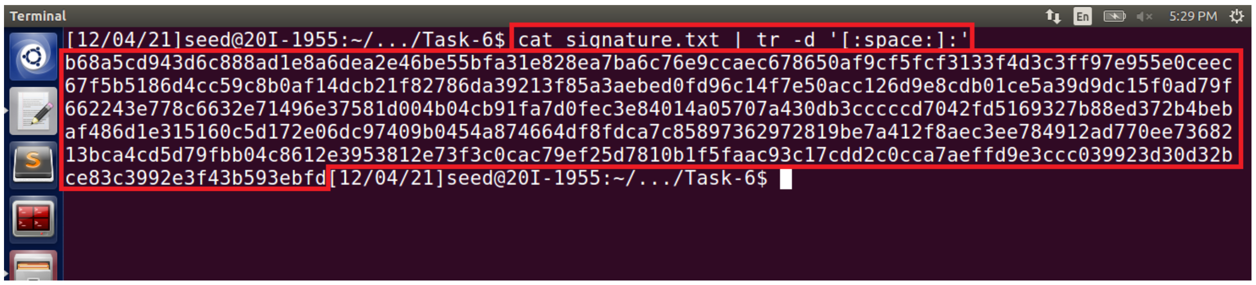
Figure 23: Extracting the signature from the server's certificate



signature.txt (~/Desktop/20I-1955_RSA/Task-6) - gedit

```
b6:8a:5c:d9:43:d6:c8:88:ad:1e:8a:6d:ea:2e:46:be:55:bf:
a3:1e:82:8e:a7:ba:6c:76:e9:cc:ae:c6:78:65:0a:f9:cf:5f:
cf:31:33:f4:d3:c3:ff:97:e9:55:e0:ce:ec:67:f5:b5:18:6d:
4c:c5:9c:8b:0a:f1:4d:cb:21:f8:27:86:da:39:21:3f:85:a3:
ae:be:d0:fd:96:c1:4f:7e:50:ac:c1:26:d9:e8:cd:b0:1c:e5:
a3:9d:9d:c1:5f:0a:d7:9f:66:22:43:e7:78:c6:63:2e:71:49:
6e:37:58:1d:00:4b:04:cb:91:fa:7d:0f:ec:3e:84:01:4a:05:
70:7a:43:0d:b3:cc:cc:cd:70:42:fd:51:69:32:7b:88:ed:37:
2b:4b:eb:af:48:6d:1e:31:51:60:c5:d1:72:e0:6d:c9:74:09:
b0:45:4a:87:46:64:df:8f:dc:a7:c8:58:97:36:29:72:81:9b:
e7:a4:12:f8:ae:c3:ee:78:49:12:ad:77:0e:e7:36:82:13:bc:
a4:cd:5d:79:fb:b0:4c:86:12:e3:95:38:12:e7:3f:3c:0c:ac:
79:ef:25:d7:81:0b:1f:5f:aa:c9:3c:17:cd:d2:c0:cc:a7:ae:
ff:d9:e3:cc:c0:39:92:3d:30:d3:2b:ce:83:c3:99:2e:3f:43:
b5:93:eb:fd
```

Figure 24: Extracting the signature from the server's certificate

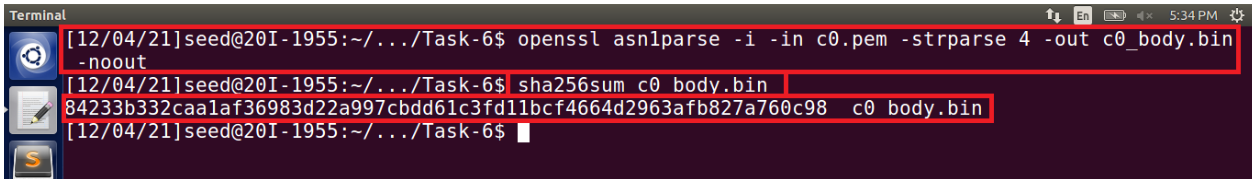


```
[12/04/21]seed@20I-1955:~/.../Task-6$ cat signature.txt | tr -d '[:space:]:'
b68a5cd943d6c888ad1e8a6dea2e46be55bfa31e828ea7ba6c76e9ccaec678650af9cf5fcf3133f4d3c3ff97e955e0ceec
67f5b5186d4cc59c8b0af14dcb21f82786da39213f85a3aebed0fd96c14f7e50acc126d9e8cdb01ce5a39d9dc15f0ad79f
662243e778c6632e71496e37581d004b04cb91fa7d0fec3e84014a05707a430db3ccccd7042fd5169327b88ed372b4beb
af486d1e315160c5d172e06dc97409b0454a874664df8fdca7c85897362972819be7a412f8aec3ee784912ad770ee73682
13bca4cd5d79fbb04c8612e3953812e73f3c0cac79ef25d7810b1f5faac93c17cdd2c0cca7aeffd9e3ccc039923d30d32b
ce83c3992e3f43b593ebfd[12/04/21]seed@20I-1955:~/.../Task-6$
```

Figure 25: Removing the spaces and colons from signature

Step-4 Extract the body of the server's certificate

In this step, I have to extract the body of the certificate from the server's certificate using the OpenSSL "**-strparse**" option. This option will provide me with the body of the certificate by excluding the signature block. Now to extract the body I use the command "**openssl asn1parse -I -in c0.pem -strparse 4 -out c0_body.bin -noout**". After extracting the body of the certificate, I calculate its hash using the command "**sha256sum c0_body.bin**".



```
[12/04/21]seed@20I-1955:~/.../Task-6$ openssl asn1parse -i -in c0.pem -strparse 4 -out c0_body.bin
 -noout
[12/04/21]seed@20I-1955:~/.../Task-6$ sha256sum c0_body.bin
84233b332caa1af36983d22a997cbdd61c3fd11bcf4664d2963afb827a760c98  c0_body.bin
[12/04/21]seed@20I-1955:~/.../Task-6$ 
```

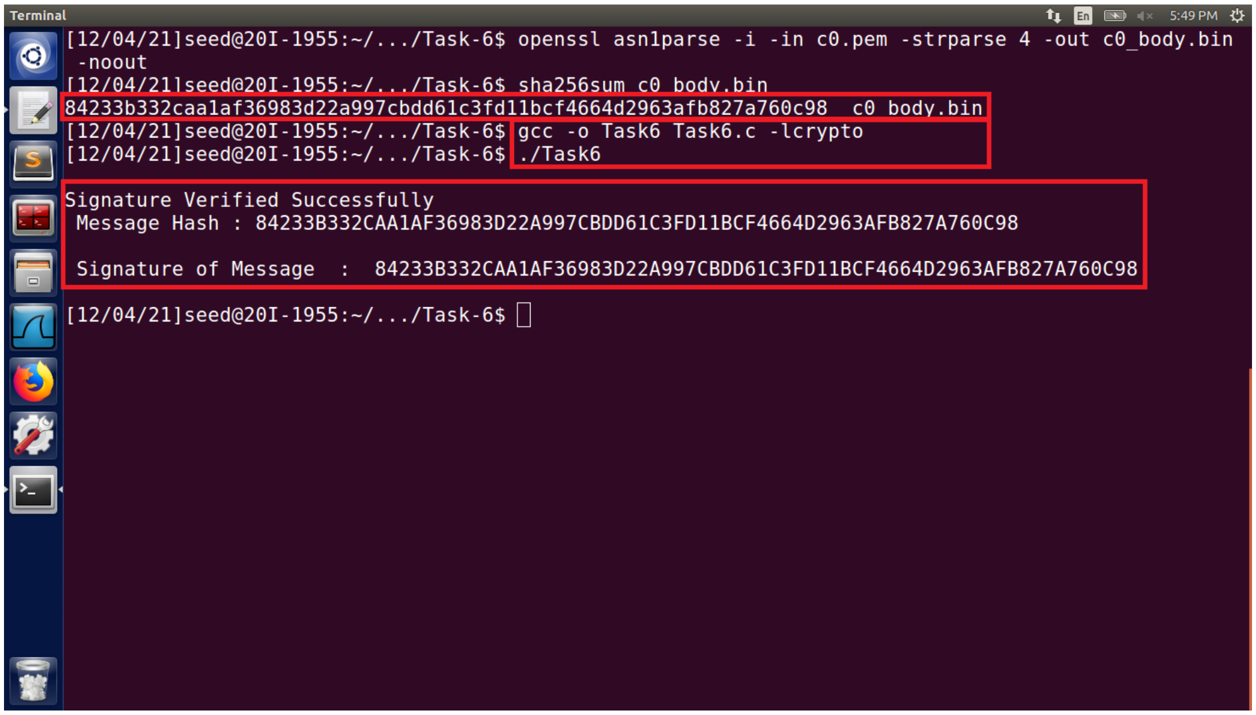
Figure 26: Extracting the body from the certificate and calculating its hash

Step-5 Verify the signature

Now as I have all the information, including CA's public key, CA's signature, and the body of the server certificate, in this task I have to write the code and use this information in the code to verify the signature to check whether it is valid or not. To verify the signature, I first decrypt the signature to get the message by using the formula $\boldsymbol{VSig = Sig^e \bmod n.}$ Now to truncate the value of VSig to 256 bits I use the BIGNUM function BN_mask_bits. After that to verify the signature, I compare the hexadecimal value of message body M received in the previous step with $VSig$, If the M ==$VSig$ then the signature is valid otherwise the signature is invalid.

Table 7: Task-6 Step-5 Code

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

    //Initilizing values of n,e,m,sig
    BN_hex2bn(&n,
"D753A40451F899A616484B6727AA9349D039ED0CB0B00087F1672886858C8E63DABCB14038E2D3F5ECA50
518B83D3EC5991732EC188CFAF10CA6642185CB071034B052882B1F689BD2B18F12B0B3D2E7881F1FEF387
754535F80793F2E1AAAA81E4B2B0DABB763B935B77D14BC594BDF514AD2A1E20CE29082876AAEEAD764D69
855E8FDAF1A506C54BC11F2FD4AF29DBB7F0EF4D5BE8E16891255D8C07134EEF6DC2DECC48725868DD821E
4B04D0C89DC392617DDF6D79485D80421709D6F6FFF5CBA19E145CB5657287E1C0D4157AAB7B827BBB1E4F
A2AEF2123751AAD2D9B86358C9C77B573ADD8942DE4F30C9DEEC14E627E17C0719E2CDEF1F910281933");

    BN_hex2bn(&e, "010001");

    BN_hex2bn(&M, "84233b332caa1af36983d22a997cbdd61c3fd11bcf4664d2963afb827a760c98");

    BN_hex2bn(&Sig,
"b68a5cd943d6c888ad1e8a6dea2e46be55bfa31e828ea7ba6c76e9ccaec678650af9cf5fcf3133f4d3c3f
f97e955e0ceec67f5b5186d4cc59c8b0af14dcb21f82786da39213f85a3aebed0fd96c14f7e50acc126d9e
8cdb01ce5a39d9dc15f0ad79f662243e778c6632e71496e37581d004b04cb91fa7d0fec3e84014a05707a4
30db3ccccd7042fd5169327b88ed372b4bebaf486d1e315160c5d172e06dc97409b0454a874664df8fdca
7c85897362972819be7a412f8aec3ee784912ad770ee7368213bca4cd5d79fbb04c8612e3953812e73f3c0
cac79ef25d7810b1f5faac93c17cdd2c0cca7aeffd9e3ccc039923d30d32bce83c3992e3f43b593ebfd");
    // decrypt VSig = Sig^e mod n
    BN_mod_exp(VSig, Sig, e, n, ctx);

    // Truncate hash value to 256 bits
    BN_mask_bits(VSig, 256);

     //veryfing the signature
    if (BN_cmp(M,VSig) == 0)
    {
        printf("\nSignature Verified Successfully");
        printBN("\n Message Hash :", M);
        printBN("\n Signature of Message  : ", VSig);
        printf("\n");
    }
    else
        printf("\n Sigature is Invalid");

    BN_clear_free(Sig);
    BN_clear_free(e);
    BN_clear_free(n);
    BN_clear_free(M);
    BN_clear_free(VSig);

    return 0;
}
```



```
[12/04/21]seed@20I-1955:~/.../Task-6$ openssl asn1parse -i -in c0.pem -strparse 4 -out c0_body.bin
 -noout
[12/04/21]seed@20I-1955:~/.../Task-6$ sha256sum c0_body.bin
84233b332caa1af36983d22a997cbdd61c3fd11bcf4664d2963afb827a760c98  c0_body.bin
[12/04/21]seed@20I-1955:~/.../Task-6$ gcc -o Task6 Task6.c -lcrypto
[12/04/21]seed@20I-1955:~/.../Task-6$ ./Task6

Signature Verified Successfully
 Message Hash : 84233B332CAA1AF36983D22A997CBDD61C3FD11BCF4664D2963AFB827A760C98

 Signature of Message  :  84233B332CAA1AF36983D22A997CBDD61C3FD11BCF4664D2963AFB827A760C98

[12/04/21]seed@20I-1955:~/.../Task-6$ 
```

Figure 27: Signature verified successfully