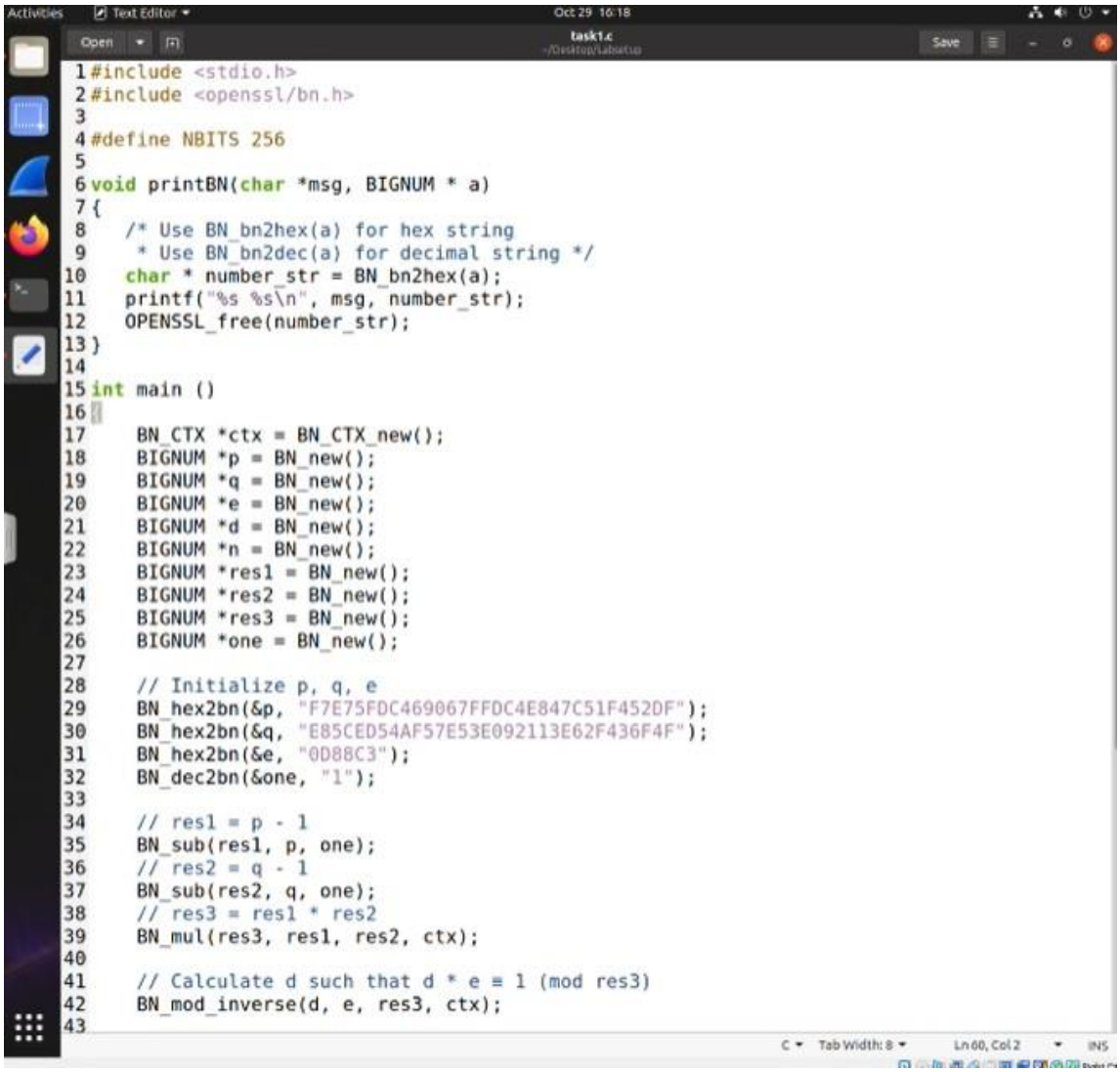
RSA Public Key Encryption and Signature Project

Task 1: Deriving the Private Key

Let p, q, and e be three prime numbers. Let n = p*q. We will use (e, n) as the public key. First, We create a task1.c and use the code showing in the lab instructions.

$ vim task1.c




```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 *d = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *res1 = BN_new();
    BIGNUM *res2 = BN_new();
    BIGNUM *res3 = BN_new();
    BIGNUM *one = BN_new();

    // Initialize p, q, e
    BN_hex2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF");
    BN_hex2bn(&q, "E85CED54AF57E53E092113E62F436F4F");
    BN_hex2bn(&e, "0D88C3");
    BN_dec2bn(&one, "1");

    // res1 = p - 1
    BN_sub(res1, p, one);
    // res2 = q - 1
    BN_sub(res2, q, one);
    // res3 = res1 * res2
    BN_mul(res3, res1, res2, ctx);

    // Calculate d such that d * e ≡ 1 (mod res3)
    BN_mod_inverse(d, e, res3, ctx);

```

```
43
44      // Print d
45      printBN("d =", d);
46
47      // Clean up
48      BN_free(p);
49      BN_free(q);
50      BN_free(e);
51      BN_free(d);
52      BN_free(n);
53      BN_free(res1);
54      BN_free(res2);
55      BN_free(res3);
56      BN_free(one);
57      BN_CTX_free(ctx);
58
59      return 0;
60
```

First print out a big number:

```
 6 void printBN(char *msg, BIGNUM * a)
 7 {
 8     /* Use BN_bn2hex(a) for hex string
 9      * Use BN_bn2dec(a) for decimal string */
10     char * number_str = BN_bn2hex(a);
11     printf("%s %s\n", msg, number_str);
12     OPENSSL_free(number_str);
13 }
```

Then in the main method, create a BN CTX structure to holds BIGNUM temporary variables used by library functions. We need to create such a structure and pass it to the functions that require it. Then we initialize BIGNUM variables: p,q,e,d,res1.res2.res3.one.

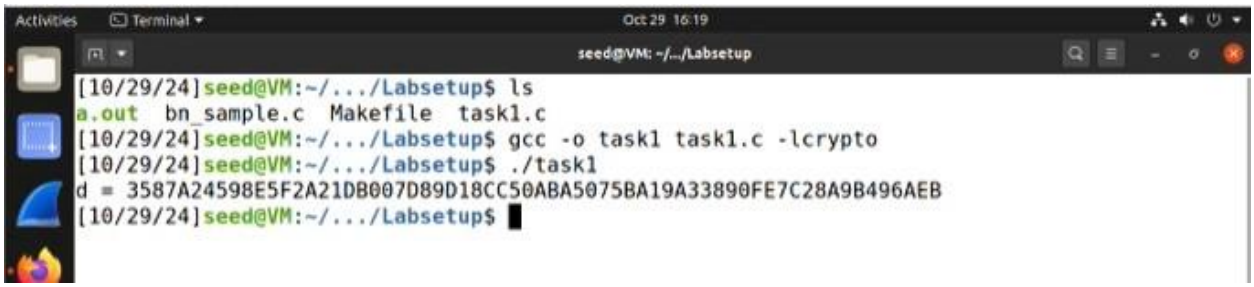
There are a number of ways to assign a value to a BIGNUM variable(p,q,e.one)

```
15 int main ()
16 {
17     BN_CTX *ctx = BN_CTX_new();
18     BIGNUM *p = BN_new();
19     BIGNUM *q = BN_new();
20     BIGNUM *e = BN_new();
21     BIGNUM *d = BN_new();
22     BIGNUM *n = BN_new();
23     BIGNUM *res1 = BN_new();
24     BIGNUM *res2 = BN_new();
25     BIGNUM *res3 = BN_new();
26     BIGNUM *one = BN_new();
27
28     // Initialize p, q, e
29     BN_hex2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF");
30     BN_hex2bn(&q, "E85CED54AF57E53E092113E62F436F4F");
31     BN_hex2bn(&e, "0D88C3");
32     BN_dec2bn(&one, "1");
```

Compute functions: Compute res1 = p-1,res2= q-1, res3 = res1 ∗ res2, d*e mod res3 =1

```
34    // res1 = p - 1
35    BN_sub(res1, p, one);
36    // res2 = q - 1
37    BN_sub(res2, q, one);
38    // res3 = res1 * res2
39    BN_mul(res3, res1, res2, ctx);
40
41    // Calculate d such that d * e = 1 (mod res3)
42    BN_mod_inverse(d, e, res3, ctx);
43
```

Finally, we can get d by following commands:

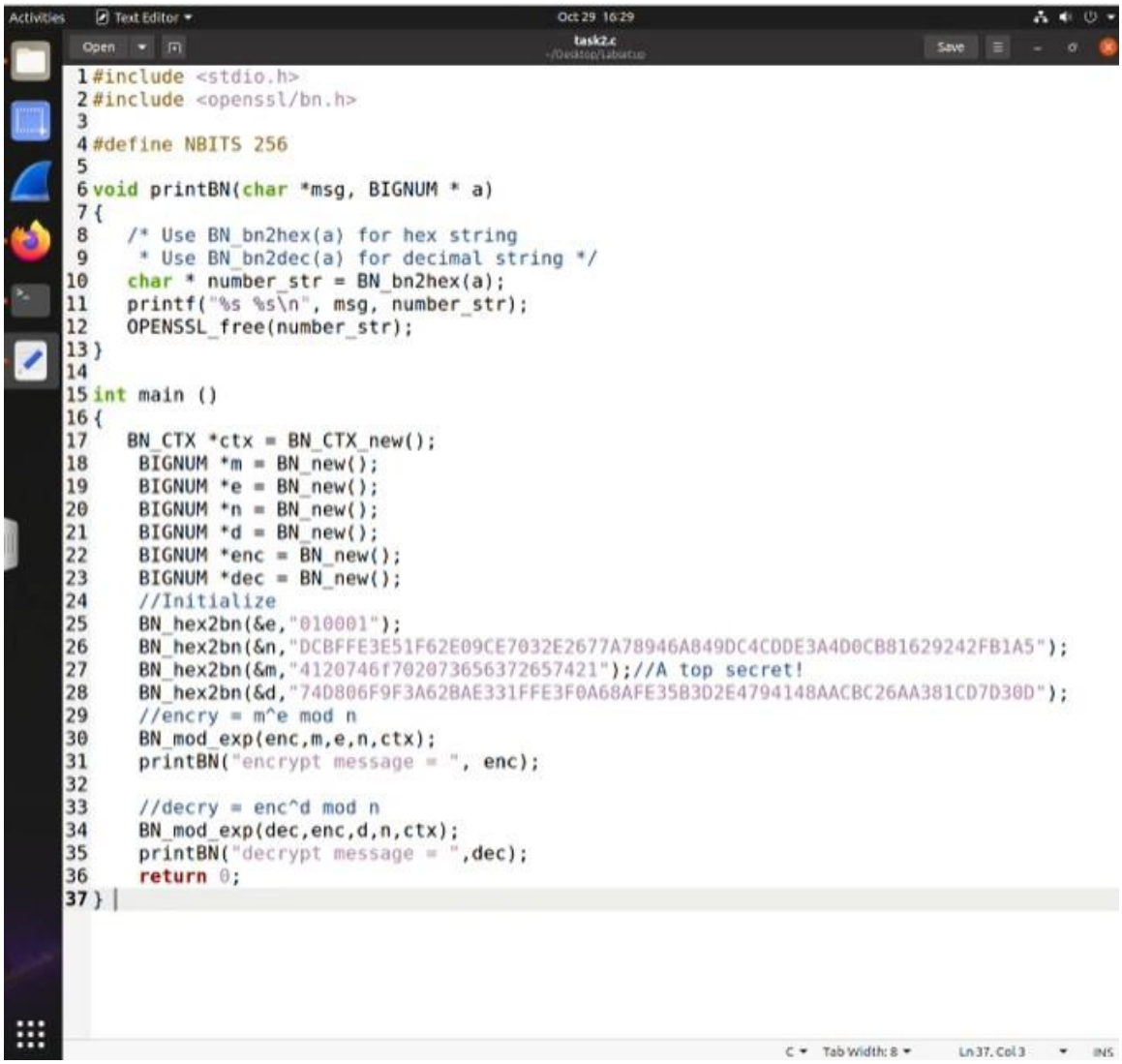


Therefore, we get the private key
d=3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB

Task 2: Encrypting a Message

Let (e, n) be the public key. We need to convert this ASCII string to a hex string, and then convert the hex string to a BIGNUM using the hex-to-bn API BN_hex2bn(). In this task, We create a task2.c.

$ vim task2.c



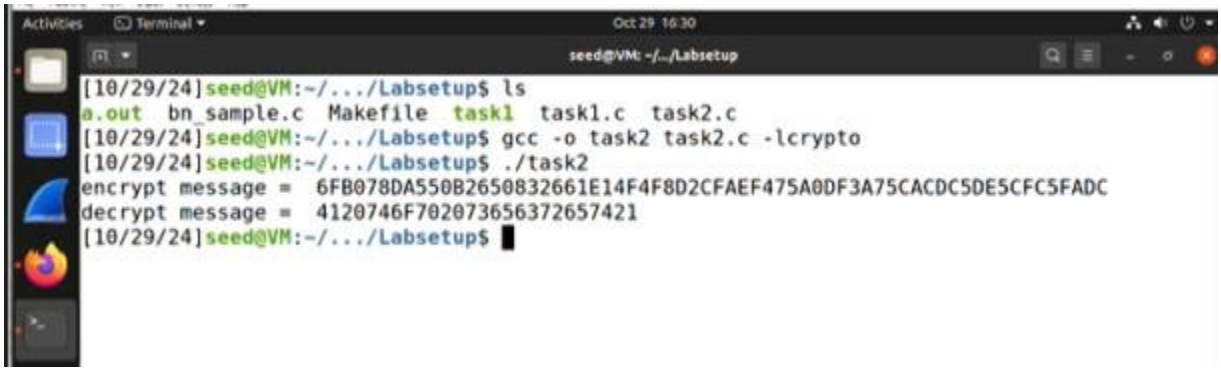
According to the lab, we know

$ python3 -c 'print("A top secret!".encode("utf-8").hex())'
4120746f702073656372657421

Therefore, we can use the hexadecimal of M

```
int main ()
{
   BN_CTX *ctx = BN_CTX_new();
    BIGNUM *m = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *enc = BN_new();
    BIGNUM *dec = BN_new();
    //Initialize
    BN_hex2bn(&e,"010001");
    BN_hex2bn(&n,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&m,"4120746f702073656372657421");//A top secret!
    BN_hex2bn(&d,"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    //encry = m^e mod n
    BN_mod_exp(enc,m,e,n,ctx);
    printBN("encrypt message = ", enc);

    //decry = enc^d mod n
    BN_mod_exp(dec,enc,d,n,ctx);
    printBN("decrypt message = ",dec);
    return 0;
}
```

Then, we run the code in the terminal, the result as shown

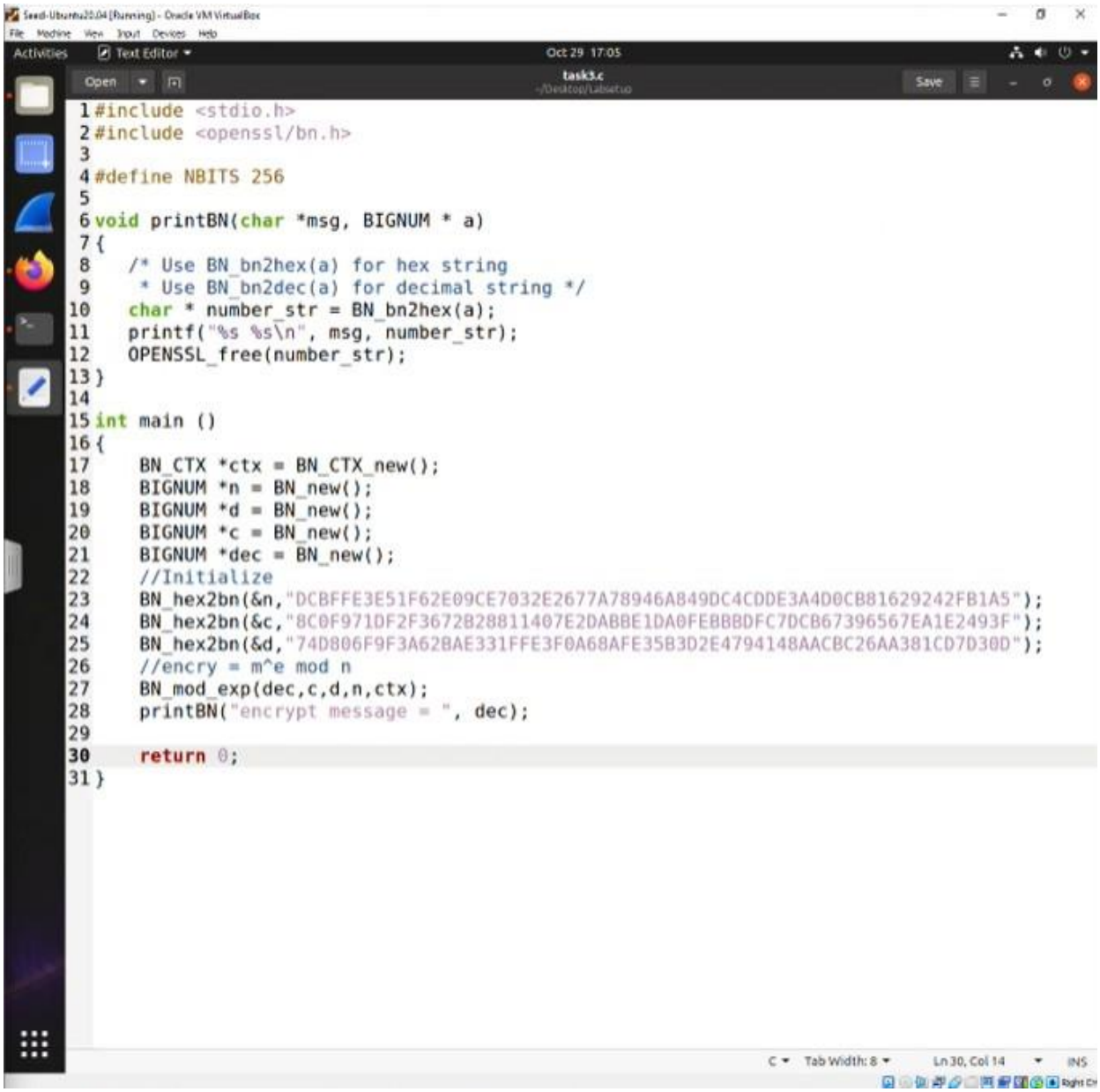


We can see that the decrypted message and the original message are the same.

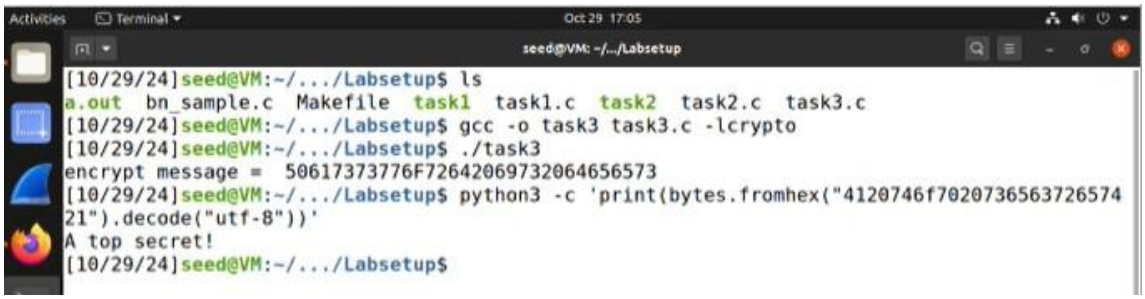
Task 3: Decrypting a Message

The public/private keys used in this task are the same as the ones used in Task 2. Decrypt the following ciphertext C, and convert it back to a plain ASCII string. We created task3.c.

$ vim task3.c



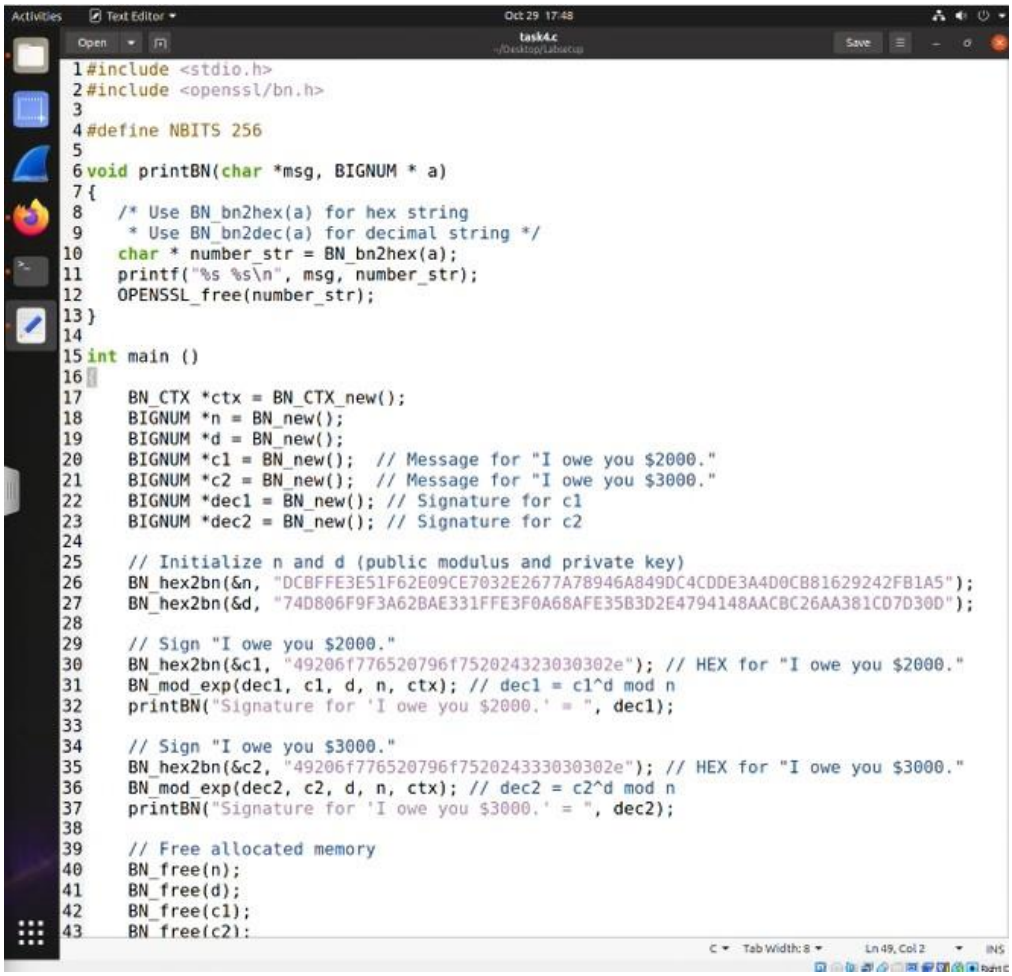
We decrypt the given cipher text, c using the formula: c^d mod n.
By decrypting, we get the hex value of the message.
We then use the python to decode the hex value:



Task 4: Signing a Message

First, we created task4.c.

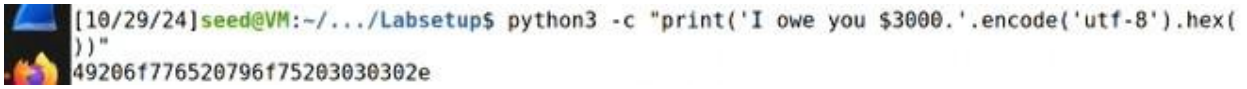
$vim task4.c



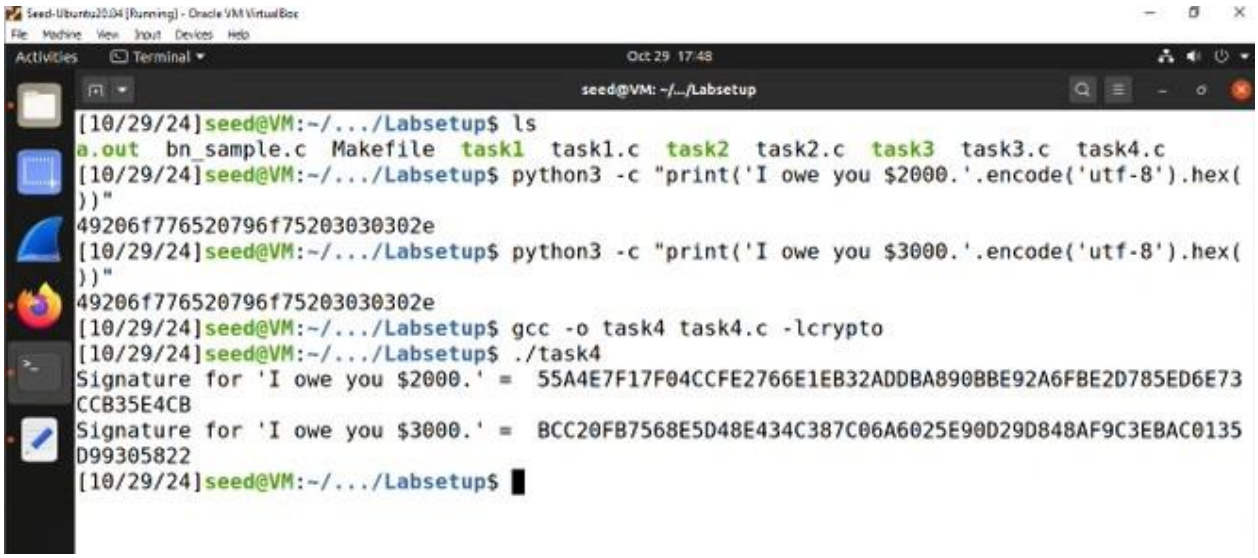
```c
#include <stdio.h>
#include <openssl/bn.h>

#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 *n = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *c1 = BN_new();  // Message for "I owe you $2000."
    BIGNUM *c2 = BN_new();  // Message for "I owe you $3000."
    BIGNUM *dec1 = BN_new(); // Signature for c1
    BIGNUM *dec2 = BN_new(); // Signature for c2

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

    // Sign "I owe you $2000."
    BN_hex2bn(&c1, "49206f776520796f752024323030302e"); // HEX for "I owe you $2000."
    BN_mod_exp(dec1, c1, d, n, ctx); // dec1 = c1^d mod n
    printBN("Signature for 'I owe you $2000.' = ", dec1);

    // Sign "I owe you $3000."
    BN_hex2bn(&c2, "49206f776520796f752024333030302e"); // HEX for "I owe you $3000."
    BN_mod_exp(dec2, c2, d, n, ctx); // dec2 = c2^d mod n
    printBN("Signature for 'I owe you $3000.' = ", dec2);

    // Free allocated memory
    BN_free(n);
    BN_free(d);
    BN_free(c1);
    BN_free(c2);
    BN_free(dec1);
    BN_free(dec2);
    BN_CTX_free(ctx);

    return 0;
}
```

Second, we get the hex value of " I owe you $2000."
Value is 49206f776520796f75203030302e

Third, we get the hex value of "I owe you $3000."
Value is 49206f776520796f75203030302e



We run our code for the complete produce the signature for the message.

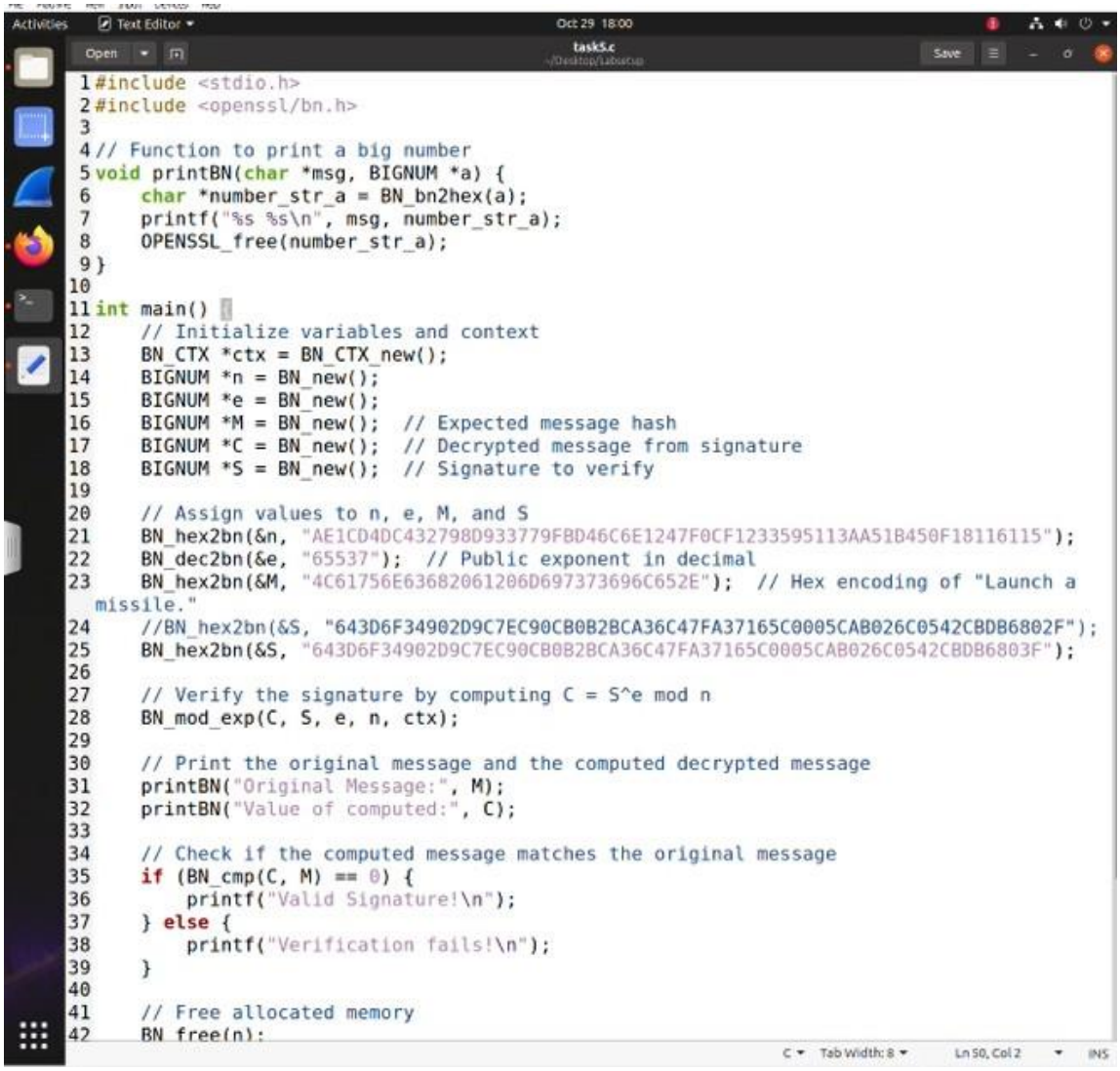


We can observe that, though there is only one byte of difference in the message, their signatures differ completely.

Task 5: Verifying a Signature

First, We create task5.c.

$vim task5.c




```
41    // Free allocated memory
42    BN_free(n);
43    BN_free(e);
44    BN_free(M);
45    BN_free(C);
46    BN_free(S);
47    BN_CTX_free(ctx);
48
49    return 0;
50 }
```

Second, We get the hex value of the message M, “Launch a missile.” using python

```
[10/29/24]seed@VM:~/.../Labsetup$ python3 -c "print('Launch a missle.'.encode('utf-8').hex(
))"
4c61756e63682061206d6973736c652e
```

We use the signature to compute the value of the message C.
We then use the BN_cmp API in order to compare the two messages and conclude whether the signature is Alice’s or not:

```
[10/29/24]seed@VM:~/.../Labsetup$ gcc -o task5 task5.c -lcrypto
[10/29/24]seed@VM:~/.../Labsetup$ ./task5
Original Message: 4C61756E63682061206D697373696C652E
Value of computed: 4C61756E63682061206D697373696C652E
Valid Signature!
```

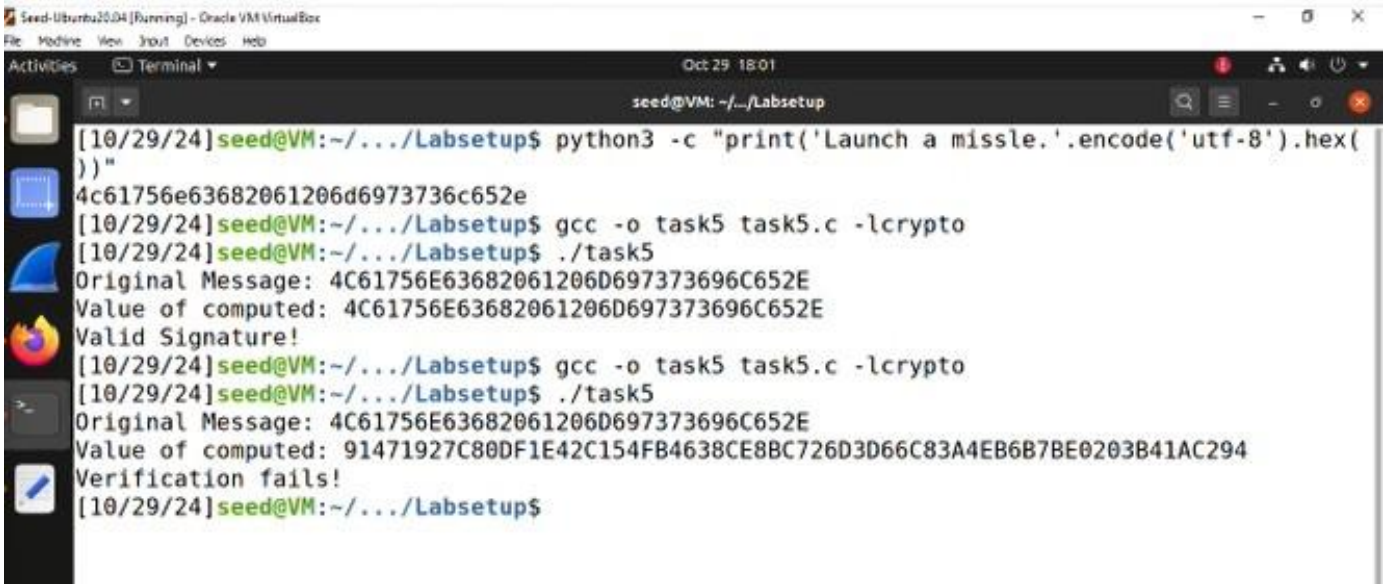
From the result, we know the same message value, therefore, it’s Alice’s signature.

Suppose that the signature in is corrupted, such that the last byte of the signature changes from 2F to 3F,
S = 643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F
We use the signature to compute the value of the message C.
We then use the BN_cmp API in order to compare the two messages and conclude whether the signature is Alice’s or not:

```
[10/29/24]seed@VM:~/.../Labsetup$ gcc -o task5 task5.c -lcrypto
[10/29/24]seed@VM:~/.../Labsetup$ ./task5
Original Message: 4C61756E63682061206D697373696C652E
Value of computed: 91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294
Verification fails!
[10/29/24]seed@VM:~/.../Labsetup$
```

Therefore, we get the value of computed message is entirely different from original message, though only 1 byte of the signature is changed. It causes the verification fails.

This is the whole commands for task 5:



Task 6: Manually Verifying an X.509 Certificate

Step 1: Download a certificate from a real web server.
We use the www.google.com server in this document, and We save these three certificates in the files c0.pem, c1.pem and c2.pem respectively.

$ openssl s_client -connect www.google.com:443 -showcerts

```
CONNECTED(00000003)
---
Certificate chain
 0 s:CN = www.google.com
   i:C = US, O = Google Trust Services, CN = WR2
-----BEGIN CERTIFICATE-----
MIIEVzCCAz+gAwIBAgIQDUOxSrucFZYQ4T1VI58lTjANBgkqhkiG9w0BAQsFADA7
MQswCQYDVQQGEwJVUzEeMBwGA1UEChMVR29vZ2xlIFRydXN0IFNlcnZpY2VzMQww
CgYDVQQDEwNXUjIwHhcNMjQxMDA3MDgyNjM2WhcNMjQxMjMwMDgyNjM1WjAZMRcw
FQYDVQQDEw53d3cuZ29vZ2xlLmNvbTBZMBMGByqGSM49AgEGCCqGSM49AwEHA0IA
BO4/mHOK7c0x6J5PAwAQYK5a9FoKEEKo6yIebzVGmjl3TdY5CwIpoxNER+tsQH/B
ijis3XTXpooymvgmKu0ohbqjggJCMIICPjAOBgNVHQ8BAf8EBAMCB4AwEwYDVR0l
BAwwCgYIKwYBBQUHAwEwDAYDVR0TAQH/BAIwADAdBgNVHQ4EFgQUk9u5lkdRRete
dUnhEnK9kbOpfC8wHwYDVR0jBBgwFoAU3hse7XkV1D43JMMhu+w0OW1CsjAwWAYI
KwYBBQUHAQEETDBKMCEGCCsGAQUFBzABhhVodHRwOi8vby5wa2kuZ29vZy93cjIw
JQYIKwYBBQUHMAKGGWh0dHA6Ly9pLnBraS5nb29nL3dyMi5jcnQwGQYDVR0RBBIw
EIIOd3d3Lmdvb2dsZS5jb20wEwYDVR0gBAwwCjAIBgZngQwBAgEwNgYDVR0fBC8w
LTAroCmgJ4YlaHR0cDovL2MucGtpLmdvb2cvd3IyLzlVVmJOMHc1RTZZLmNybDCC
AQUGCisGAQQB1nkCBAIEgfYEgfMA8QB3AO7N0GTV2xrOxVy3nbTNE6Iyh0Z8vOze
w1FIWUZxH7WbAAABkmZNV8MAAAQDAEgwRgIhAPP9CVkB0GXiS2b7kTXWPPHFk2Ty
Nd87dMf9OB4rNBAxAiEAufW+murZkpZ7q/RQEV2zTK/9o67r/y8GB68lxEZ03dAA
dgBIsONr2qZHNA/lagL6nTDrHFIBy1bdLIHZu7+rOdiEcwAAAZJmTVfnAAAEAwBH
MEUCIGTyK07CmC5fWrNvdbOyqwUJy5pHLY+5cUfnPQbtv1KxAiEAwrIZKO79rAlR
IU+heWW8IE8KeQGXntqAxDjmkPfM9aAwDQYJKoZIhvcNAQELBQADggEBAJ39sqzq
Vv8185didLQhMoJncKSntbcNCDNgiCpTEw/oDv+Gy56ngWnayGeZWpYPk3epPSDC
iJTqPL/SkZMyhaOQOYXGFFhP3JfmNeYEj9oCEJVdZMuu+n0eOkEaoo9GXqRnI1VR
wF2OI1ngxbN08ijGf1jV401aYQa3Rp4rtGbKe4ARyFtGKWAExgBMsiZd33rH+K7J
GUG0VUqVdr42KU/ImNM4N66WyYOZemIjwtu6gD6I0+5REgYPBXY7av+QiLSB8Z+U
ZEEh3LiCFTAmc4RfgHS5/gbzKourAHJOJubV0iWpivBkn1QAyAllsm2VwuRqjt+m
0jNzprolWaZAh4Y=
-----END CERTIFICATE-----
 1 s:C = US, O = Google Trust Services, CN = WR2
   i:C = US, O = Google Trust Services LLC, CN = GTS Root R1
-----BEGIN CERTIFICATE-----
MIIFCzCCAvOgAwIBAgIQf/AFoHxM3tEArZ1mpRB7mDANBgkqhkiG9w0BAQsFADBH
MQswCQYDVQQGEwJVUzEiMCAGA1UEChMZR29vZ2xlIFRydXN0IFNlcnZpY2VzIExM
QzEUMBIGA1UEAxMLR1RTIFJvb3QgUjEwHhcNMjMxMjEzMDkwMDAwWhcNMjkwMjIw
MTQwMDAwWjA7MQswCQYDVQQGEwJVUzEeMBwGA1UEChMVR29vZ2xlIFRydXN0IFNl
cnZpY2VzMQwwCgYDVQQDEwNXUjIwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEK
```

```
AoIBAQCp/5x/RR5wqFOfytnlDd5GV1d9vI+aWqxG8YSau5HbyfsvAfuSCQAWXqAc
+MGr+XgvSszYhaLYWTwO0xj7sfUkDSbutltkdnwUxy96zqhMt/TZCPzfhyM1IKji
aeKMTj+xWfpgoh6zySBTGYLKNlNtYE3pAJH8do1cCA8Kwtzxc2vFE24KT3rC8gIc
LrRjg9ox9i11MLL7q8Ju26nADrn5Z9TDJVd06wW06Y613ijNzHoU5HEDy01hLmFX
xRmpC5iEGuh5KdmyjS//V2pm4M6rlagplmNwEmceOuHbsCFx13ye/aoXbv4r+zgX
FNFmp6+atXDMyGOBOozAKql2N87jAgMBAAGjgf4wgfswDgYDVR0PAQH/BAQDAgGG
MB0GA1UdJQQWMBQGCCsGAQUFBwMBBggrBgEFBQcDAjASBgNVHRMBAf8ECDAGAQH/
AgEAMB0GA1UdDgQWBBTeGx7teRXUPjckwyG77DQ5bUKyMDAfBgNVHSMEGDAWgBTk
rysmcRorSCeFL1JmLO/wiRNxPjA0BggrBgEFBQcBAQQoMCYwJAYIKwYBBQUHMAKG
GGh0dHA6Ly9pLnBraS5nb29nL3IxLmNydDArBgNVHR8EJDAiMCCgHqAchhpodHRw
Oi8vYy5wa2kuZ29vZy9yL3IxLmNybDATBgNVHSAEDDAKMAgGBmeBDAECATANBgkq
hkiG9w0BAQsFAAOCAgEARXWL5R87RBOWGqtY8TXJbz3S0DNKhjO6V1FP7sQ02hYS
TL8Tnw3UVOlIecAwPJQl8hr0ujKUtjNyC4XuCRElNJThb0Lbgpt7fyqaqf9/qdLe
SiDLs/sDA7j4BwXaWZIvGEaYzq9yviQmsR4ATb0IrZNBRAq7x9UBhb+TV+PfdBJT
DhEl05vc3ssnbrPCuTNiOcLgNeFbpwkuGcuRKnZc8d/KI4RApW//mkHgte8y0YWu
ryUJ8GLFbsLIbjL9uNrizkqRSvOFVU6xddZIMy9vhNkSXJ/UcZhjJY1pXAprffJB
vei7j+Qi151lRehMCofa6WBmiA4fx+FOVsV2/7R6V2nyAiIJJkEd2nSi5SnzxJrl
XdaqEv3htytmOPvoKWa676ATL/hzfvDaQBEcXd2Ppvy+275W+DKcH0FBbX62xevG
iza3F4ydzxl6NJ8hk8R+dDXSqv1MbRT1ybB5W0k8878XSOjvmiYTDIfyc9acxVJr
Y/cykHipa+te1pOhv7wYPYtZ9orGBV5SGOJm4NrB3K1aJar0RfzxC3ikr7Dyc6Qw
qDTBU39CluVIQeuQRgwG3MuSxl7zRERDRilGoKb8uY45JzmxWuKxrfwT/478JuHU
/oTxUFqOl2stKnn7QGTq8z29W+GgBLCXSBxC9epaHM0myFH/FJlniXJfHeytWt0=
-----END CERTIFICATE-----
 2 s:C = US, O = Google Trust Services LLC, CN = GTS Root R1
   i:C = BE, O = GlobalSign nv-sa, OU = Root CA, CN = GlobalSign Root CA
-----BEGIN CERTIFICATE-----
MIIFYjCCBEqgAwIBAgIQd70NbNs2+RrqIQ/E8FjTDTANBgkqhkiG9w0BAQsFADBX
MQswCQYDVQQGEwJCRTEZMBcGA1UEChMQR2xvYmFsU2lnbiBudi1zYTEQMA4GA1UE
CxMHUm9vdCBDQTEbMBkGA1UEAxMSR2xvYmFsU2lnbiBSb290IENBMB4XDTIwMDYx
OTAwMDA0MloXDTI4MDEyODAwMDA0MlowRzELMAkGA1UEBhMCVVMxIjAgBgNVBAoT
GUdvb2dsZSBUcnVzdCBTZXJ2aWNlcyBMTEMxFDASBgNVBAMTC0dUUyBSb290IFIx
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAthECix7joXebO9y/lD63
ladAPKH9gvl9MgaCcfb2jH/76Nu8ai6Xl6OMS/kr9rH5zoQdsfnFl97vufKj6bwS
iV6nqlKr+CMny6SxnGPb15l+8Ape62im9MZaRw1NEDPjTrETo8gYbEvs/AmQ351k
KSUjB6G00j0uYODP0gmHu81I8E3CwnqIiru6z1kZ1q+PsAewnjHxgsHA3y6mbWwZ
DrXYfiYaRQM9sHmklCitD38m5agI/pboPGiUU+6DOogrFZYJsuB6jC511pzrp1Zk
j5ZPaK49l8KEj8C8QMALXL32h7M1bKwYUH+E4EzNktMg6TO8UpmvMrUpsyUqtEj5
cuHKZPfmghCN6J3Cioj6OGaK/GP5Afl4/Xtcd/p2h/rs37EOeZVXtL0m79YB0esW
CruOC7XFxYpVq9Os6pFLKcwZpDIlTirxZUTQAs6qzkm06p98g7BAe+dDq6dso499
iYH6TKX/1Y7DzkvgtdizjkXPdsDtQCv9Uw+wp9U7DbGKogPeMa3Md+pvez7W35Ei
Eua++tgy/BBjFFFy3l3WFpO9KWgz7zpm7AeKJt8T11dleCfeXkkUAKIAf5qoIbap
sZWwpbkNFhHax2xIPEDgfg1azVY80ZcFuctL7TlLnMQ/0lUTbiSw1nH69MG6zO0b
9f6BQdgAmD06yK56mDcYBZUCAwEAAaOCATgwggE0MA4GA1UdDwEB/wQEAwIBhjAP
BgNVHRMBAf8EBTADAQH/MB0GA1UdDgQWBBTkrysmcRorSCeFL1JmLO/wiRNxPjAf
BgNVHSMEGDAWgBRge2YaRQ2XyolQL30EzTSo//z9SzBgBggrBgEFBQcBAQRUMFIw
JQYIKwYBBQUHMAGGGWh0dHA6Ly9vY3NwLnBraS5nb29nL2dzcjEwKQYIKwYBBQUH
MAKGHWh0dHA6Ly9wa2kuZ29vZy9nc3IxL2dzcjEuY3J0MDIGA1UdHwQrMCkwJ6Al
```

```
oCOGIWh0dHA6Ly9jcmwucGtpLmdvb2cvZ3NyMS9nc3IxLmNybDA7BgNVHSAENDAy
MAgGBmeBDAECATAIBgZngQwBAgIwDQYLKwYBBAHWeQIFAwIwDQYLKwYBBAHWeQIF
AwMwDQYJKoZIhvcNAQELBQADggEBADSkHrEoo9C0dhemMXoh6dFSPsjbdBZBiLg9
NR3t5P+T4Vxfq7vqfM/b5A3Ri1fyJm9bvhdGaJQ3b2t6yMAYN/olUazsaL+yyEn9
WprKASOshIArAoyZl+tJaox118fessmXn1hIVw41oeQa1v1vg4Fv74zPl6/AhSrw
9U5pCZEt4Wi4wStz6dTZ/CLANx8LZh1J7QJVj2fhMtfTJr9w4z30Z209fOU0iOMy
+qduBmpvvYuR7hZL6Dupszfnw0Skfths18dG9ZKb59UhvmaSGZRVbNQpsg3BZlvi
d0lIKO2d1xozclOzgjXPYovJJIultzkMu34qQb9Sz/yilrbCgj8=
-----END CERTIFICATE-----
---
Server certificate
subject=CN = www.google.com

issuer=C = US, O = Google Trust Services, CN = WR2

---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: ECDSA
Server Temp Key: X25519, 253 bits
---
SSL handshake has read 4107 bytes and written 396 bytes
Verification error: unable to get local issuer certificate
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server public key is 256 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 20 (unable to get local issuer certificate)
---
HTTP/1.0 400 Bad Request
Content-Length: 54
Content-Type: text/html; charset=UTF-8
Date: Wed, 30 Oct 2024 01:47:47 GMT

<html><title>Error 400 (Bad Request)!!1</title></html>---
Post-Handshake New Session Ticket arrived:
SSL-Session:
    Protocol  : TLSv1.3
    Cipher    : TLS_AES_256_GCM_SHA384
    Session-ID:
71BDAD9ADE008D9AB73C191EC574891ED56FB6C7B52DDCB0143956A353DB6455
    Session-ID-ctx:
```

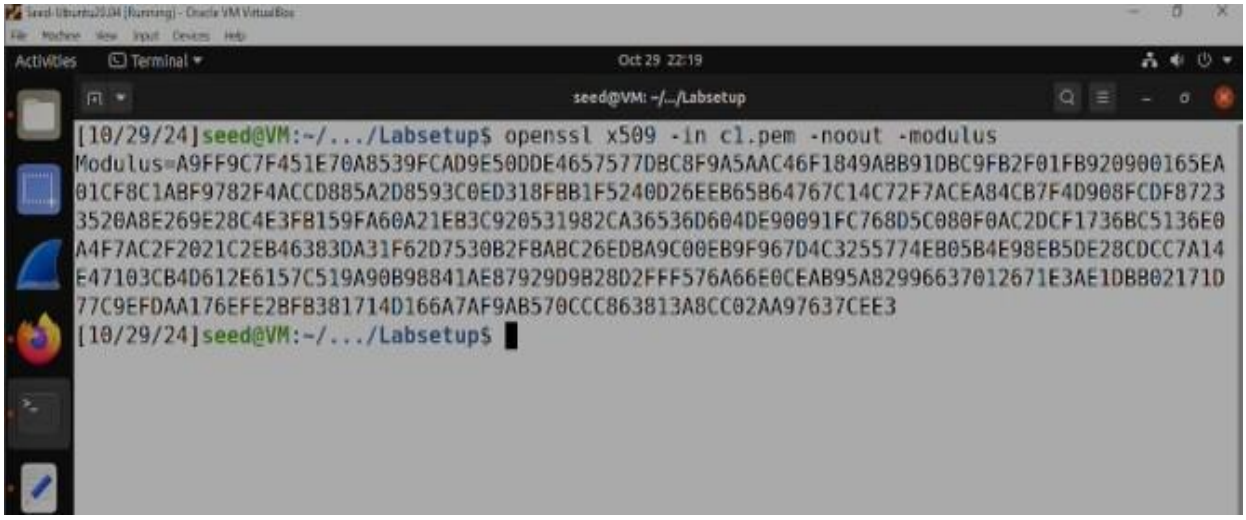
```
    Resumption PSK:
30CCD029DF4BB99380898802A275102211E41FC89DD82C53642B2F521E7A6E22A40D9325FA
4AD88787DAD51AE73FEDA0
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    TLS session ticket lifetime hint: 172800 (seconds)
    TLS session ticket:
    0000 - 02 eb 77 a2 ba 3f 6d f3-c1 56 a7 10 f9 ad bd 3c
..w..?m..V.……<
    0010 - be 8b 87 98 bd f1 f4 83-dd 9e 3c be b0 80 f8 fc
..........<.....
    0020 - 48 6a 41 97 42 fd a3 62-fc 5c e0 ac 80 bc 4d 59
HjA.B..b.\....MY
    0030 - e9 3b b6 df 1b ea e1 48-74 76 00 70 13 50 92 9e
.;.....Htv.p.P..
    0040 - 62 31 a1 07 cc 1d b0 6c-40 aa 6c 3e 0f 77 c0 1e
b1.....l@.l>.w..
0050 - 27 5b     17 2b 85 dc f6 82-bb 41 30 aa ca 93 15 ab
'[.+.....A0.....
    0060 - c6 eb db e6 5e 5e da 87-6a 40 3f 23 cf 43 75 05
....^^..j@?#.Cu.
    0070 - 98 43 82 bf 4d ba 5d cc-19 1f 36 07 6d 66 b9 41
.C..M.]..…6.mf.A
    0080 - 20 16 1d 7e ff 0f 41 f7-2c ce 38 5f e6 49 dd b3
..~..A.,.8_.I..
    0090 - c3 e3 81 0e c3 02 99 8b-0e bd 84 3d a8 bc 07 5b
...........=...[
    00a0 - 0b f8 c9 3f 4e 5b 0c d5-6c a1 95 61 04 b3 c5 4c
...?N[..l..a.…L
    00b0 - cd 72 9e 65 9c 00 af 86-e3 42 24 75 11 89 56 0e
.r.e.....B$u..V.
    00c0 - 2b c8 83 99 48 31 48 42-9a 58 72 5a 28 85 97 8e
+...H1HB.XrZ(...
    00d0 - 60 b9 43 4d 85 09 12 70-d6 f5 a5 e3 1f 19 8f 7a
`.CM...p.……z
    00e0 - e9 34 45 22 b4 a8 4d db-c8 56 59 59 5a 0d 41 b3
.4E"..M..VYYZ.A.
    00f0 - 60 ba 1f 67 96                                   `..g.

    Start Time: 1730252867
    Timeout   : 7200 (sec)
    Verify return code: 20 (unable to get local issuer certificate)
    Extended master secret: no
    Max Early Data: 14336
---
read R BLOCK
```

```
---
Post-Handshake New Session Ticket arrived:
SSL-Session:
    Protocol  : TLSv1.3
    Cipher    : TLS_AES_256_GCM_SHA384
    Session-ID:
7F73EE0A207019C88B632B71EAC86E50FAE0AB672E23BD879FC6FD854354E3DE
    Session-ID-ctx:
    Resumption PSK:
A5D97E151AE98167B95F150E236870038BF47D8A5AD104E96A3B3663EB24E74384863F8564
5A41919AF92CF3E10CC7FD
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    TLS session ticket lifetime hint: 172800 (seconds)
    TLS session ticket:
    0000 - 02 eb 77 a2 ba 3f 6d f3-c1 56 a7 10 f9 ad bd 3c
..w..?m..V.……<
    0010 - 32 62 e5 20 34 d0 70 6d-9d 43 f1 76 bf 90 dd 8c   2b.
4.pm.C.v....
    0020 - 29 52 64 f0 e9 2b 30 a2-96 a1 8c aa 05 0c 36 60
)Rd..+0.……6`
    0030 - 89 d2 4f 98 71 ea 2c 0b-b2 51 4f d8 a2 ca 2d 1f
..O.q.,..QO...-.
    0040 - e6 1e 2e 5e 39 c1 42 1c-82 68 1f c3 2a dc 2f 6b
...^9.B..h..*./k
    0050 - 76 14 d8 5a ff 47 53 5b-d3 23 24 4c e4 6c ac d9
v..Z.GS[.#$L.l..
    0060 - 33 7e 4c bd 9c bc df ec-4b 25 47 8a 45 05 87 f6
3~L.....K%G.E...
    0070 - 53 91 af 5e 7b 50 96 d7-99 b1 96 98 1a 24 3d ac
S..^{P.……$=.
    0080 - eb 24 a7 72 2f 32 55 d9-d1 65 00 e3 66 84 c9 1b
.$.r/2U..e..f...
    0090 - e6 9e 80 8a 02 ce 36 da-86 37 d2 df 04 93 63 75
......6..7.……cu
    00a0 - e9 a9 a6 d7 14 93 fd 19-bd ec b3 5d 8f 5d d1 7b
...........].].{
    00b0 - 16 72 a3 79 24 60 17 91-5d ae 1c cf 37 71 ea 2a
.r.y$`..]. … 7q.*
    00c0 - ba 10 ff e7 b1 1e 0c 35-aa 4b 57 8c 8b 3a 80 b3
.......5.KW..:..
    00d0 - 8f 65 65 f7 2f cf 8f 08-4c 07 b5 18 87 b1 c3 bf
.ee./...L.......
    00e0 - 77 a3 75 bf af 57 50 fc-c1 a9 59 59 5a bd 46 54
w.u..WP.….YYZ.FT
    00f0 - 83 c1 a3 60 c5                                  ...`.
```

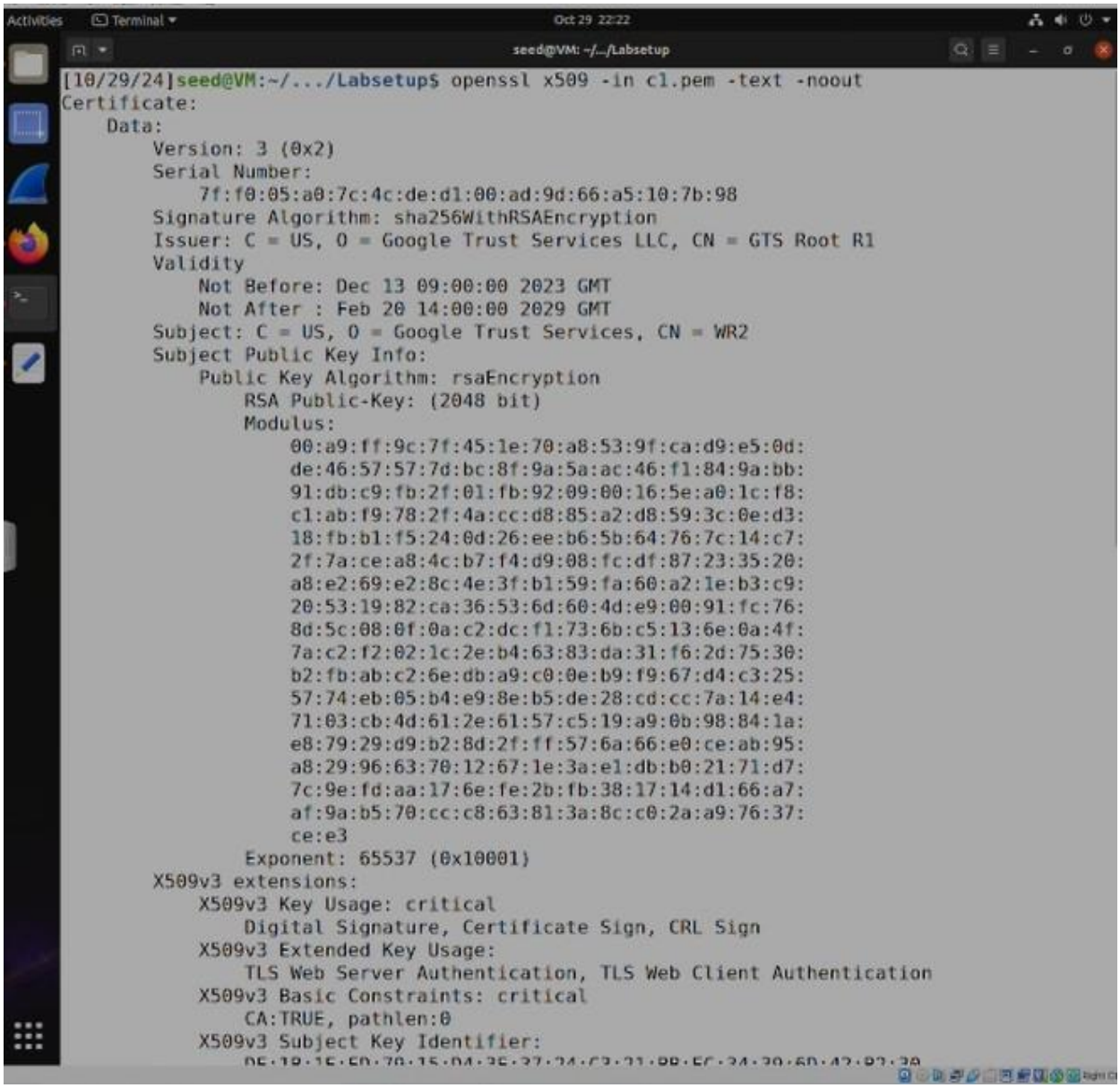
```
    Start Time: 1730252867
    Timeout   : 7200 (sec)
    Verify return code: 20 (unable to get local issuer certificate)
    Extended master secret: no
    Max Early Data: 14336
---
read R BLOCK
```

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

We get the value of n using -modulus:

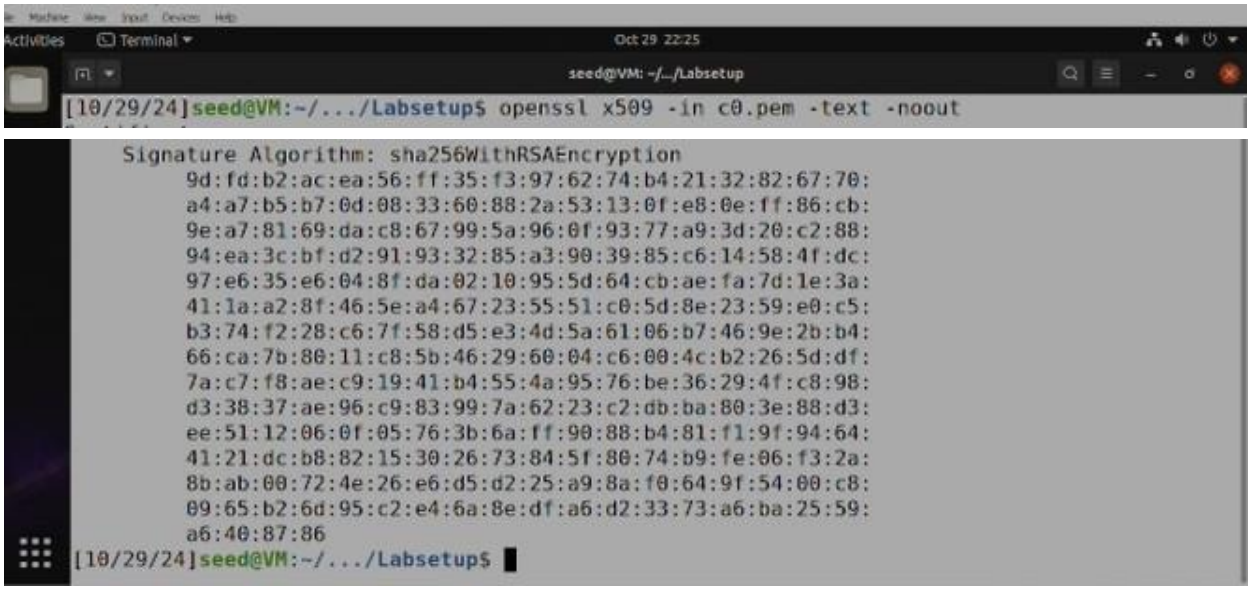


Print all attributes of the certificate, and then find the exponent, which is public key e.
Exponent: 65537 (0x10001)



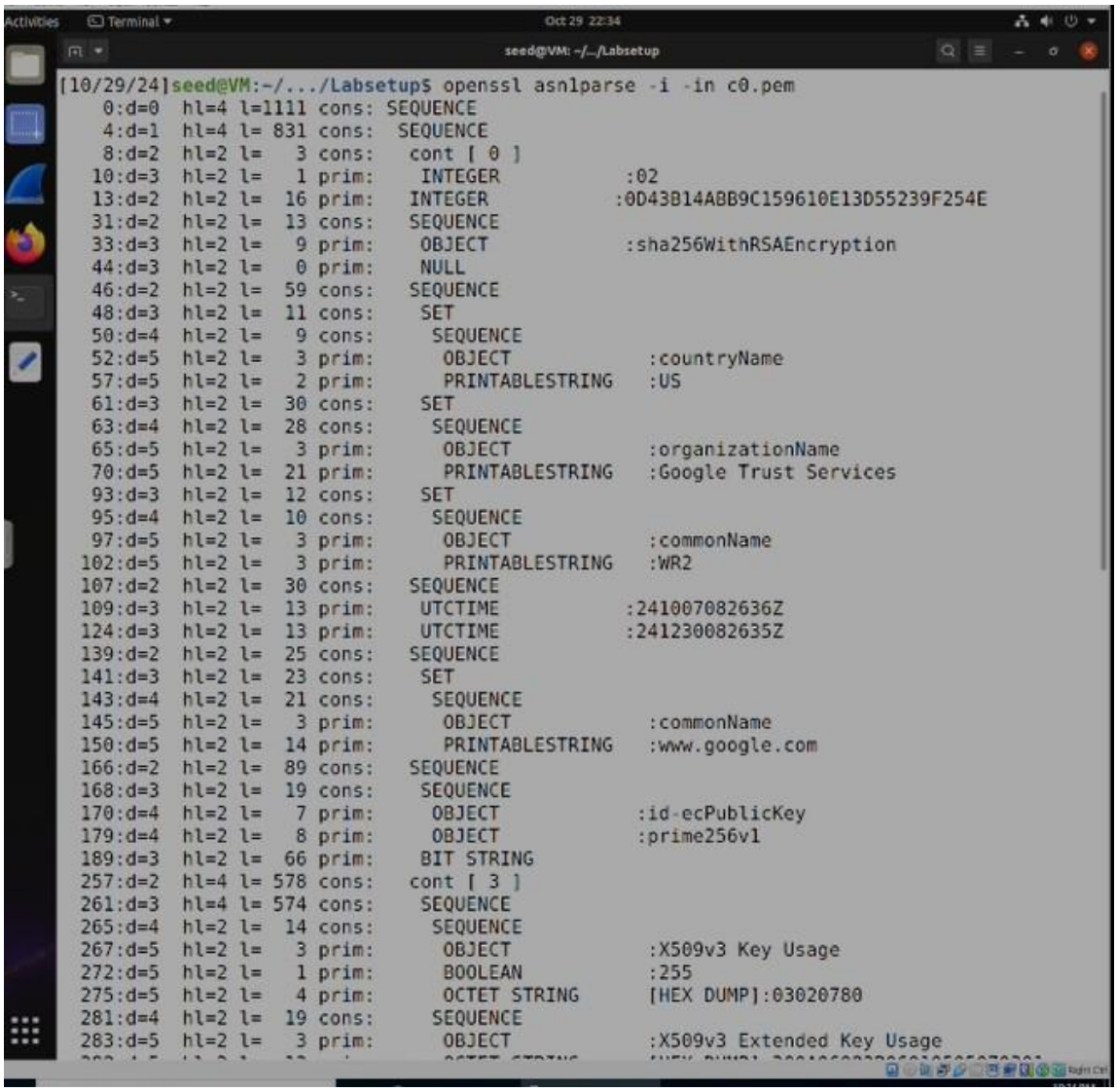
Step 3: Extract the signature from the server's certificate.

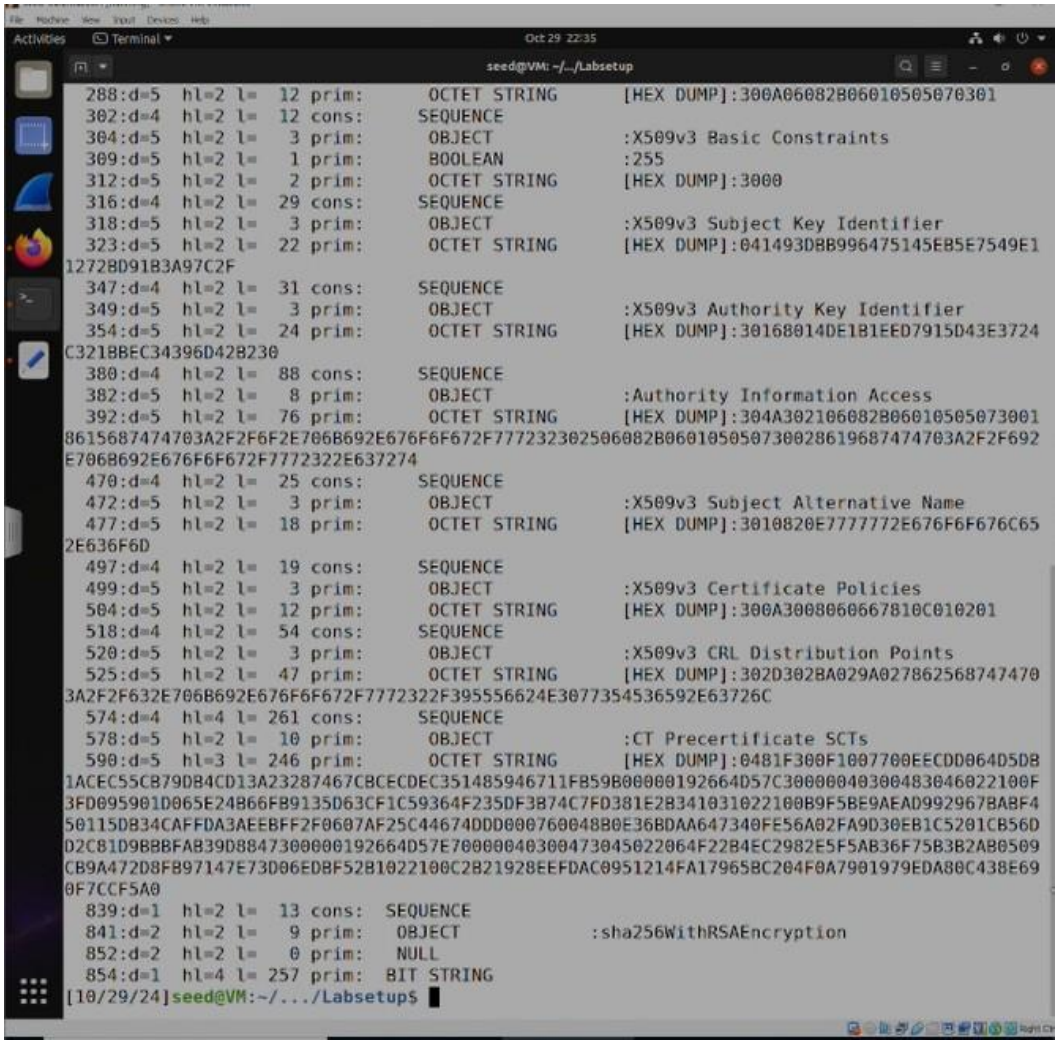
We run the command: openssl x509 -in c0.pem -text –noout to extract the signature from the server's certificate, c0.pem. We put this signature into a file.



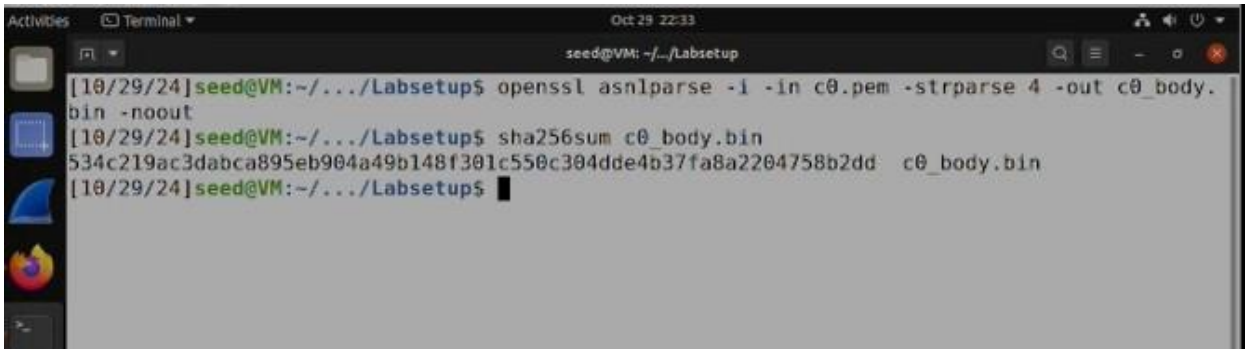
We put this signature into a file, then remove all the colons and spaces from the signature:

Step 4: Extract the body of the server's certificate.





In this we cannot determine the end of the body. So we use -strparse to get the field from the offset 4, which will give us the body of the certificate, excluding the signature block.

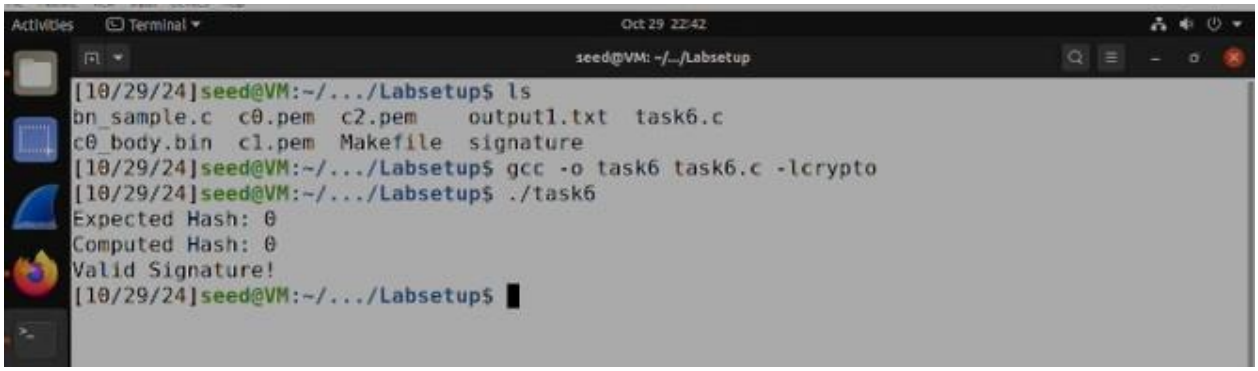


Step 5: Verify the signature.

We use the values obtained from the previous steps, get the signature and verify the signature obtained with the original signature

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

#define NBITS 256

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 *S = BN_new();
   BIGNUM *M = BN_new();
   BIGNUM *n = BN_new();
   BIGNUM *e = BN_new();
   BIGNUM *M1 = BN_new();

   BN_hex2bn(&S, "<signature-hex>");
   BN_hex2bn(&M, "<expected-hash-hex>");
   BN_hex2bn(&n, "<modulus-hex>");
   BN_hex2bn(&e, "010001");

   BN_mod_exp(M1, S, e, n, ctx);

   printBN("Expected Hash:", M);
   printBN("Computed Hash:", M1);

   if (BN_cmp(M, M1) == 0) {
       printf("Valid Signature!\n");
   } else {
       printf("Verification failed!\n");
   }
```

```
    // Free memory
    BN_free(S);
    BN_free(M);
    BN_free(n);
    BN_free(e);
    BN_free(M1);
    BN_CTX_free(ctx);

    return 0;
}
```



We can notice that the original message and the hash value of the computed message is the same. Hence we can conclude that the www.google.com certificate is verified to be right.