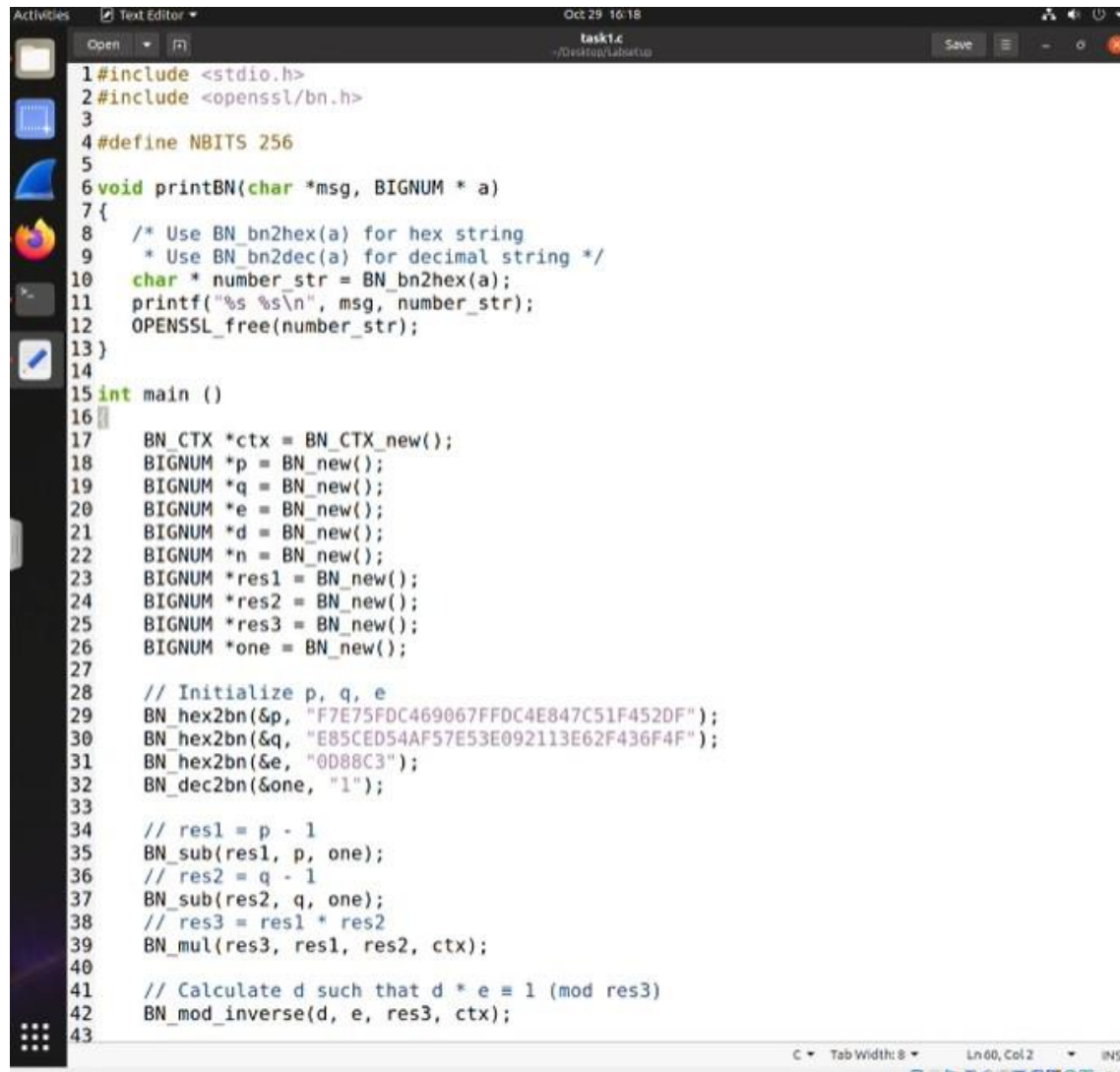


RSA Public Key Encryption and Signature Project

Task 1: Deriving the Private Key

Let p , q , and e be three prime numbers. Let $n = p \cdot q$. 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



```
1#include <stdio.h>
2#include <openssl/bn.h>
3
4#define NBITS 256
5
6void 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}
14
15int 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");
33
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}
```

```

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)

```

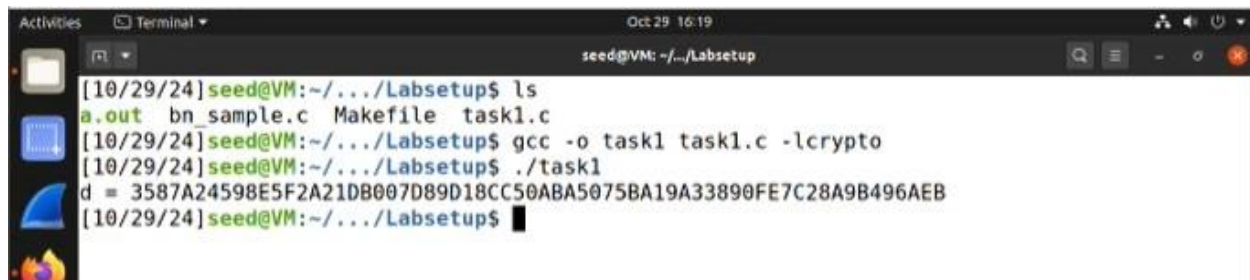
15 int main ()
16 {
17     BN_CTX *ctx = BN_CTX_new();
18     BIGNUM *p = BN_new();
19     BIGNUM *q = BN_new();
20     BIGNUM *e = BN_new();
21     BIGNUM *d = BN_new();
22     BIGNUM *n = BN_new();
23     BIGNUM *res1 = BN_new();
24     BIGNUM *res2 = BN_new();
25     BIGNUM *res3 = BN_new();
26     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 $\text{res1} = p-1$, $\text{res2} = q-1$, $\text{res3} = \text{res1} * \text{res2}$, $d * e \bmod \text{res3} = 1$

```
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:



```
Oct 29 16:19
seed@VM: ~/.../Labsetup
[10/29/24]seed@VM:~/.../Labsetup$ ls
a.out bn_sample.c Makefile task1.c
[10/29/24]seed@VM:~/.../Labsetup$ gcc -o task1 task1.c -lcrypto
[10/29/24]seed@VM:~/.../Labsetup$ ./task1
d = 3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB
[10/29/24]seed@VM:~/.../Labsetup$
```

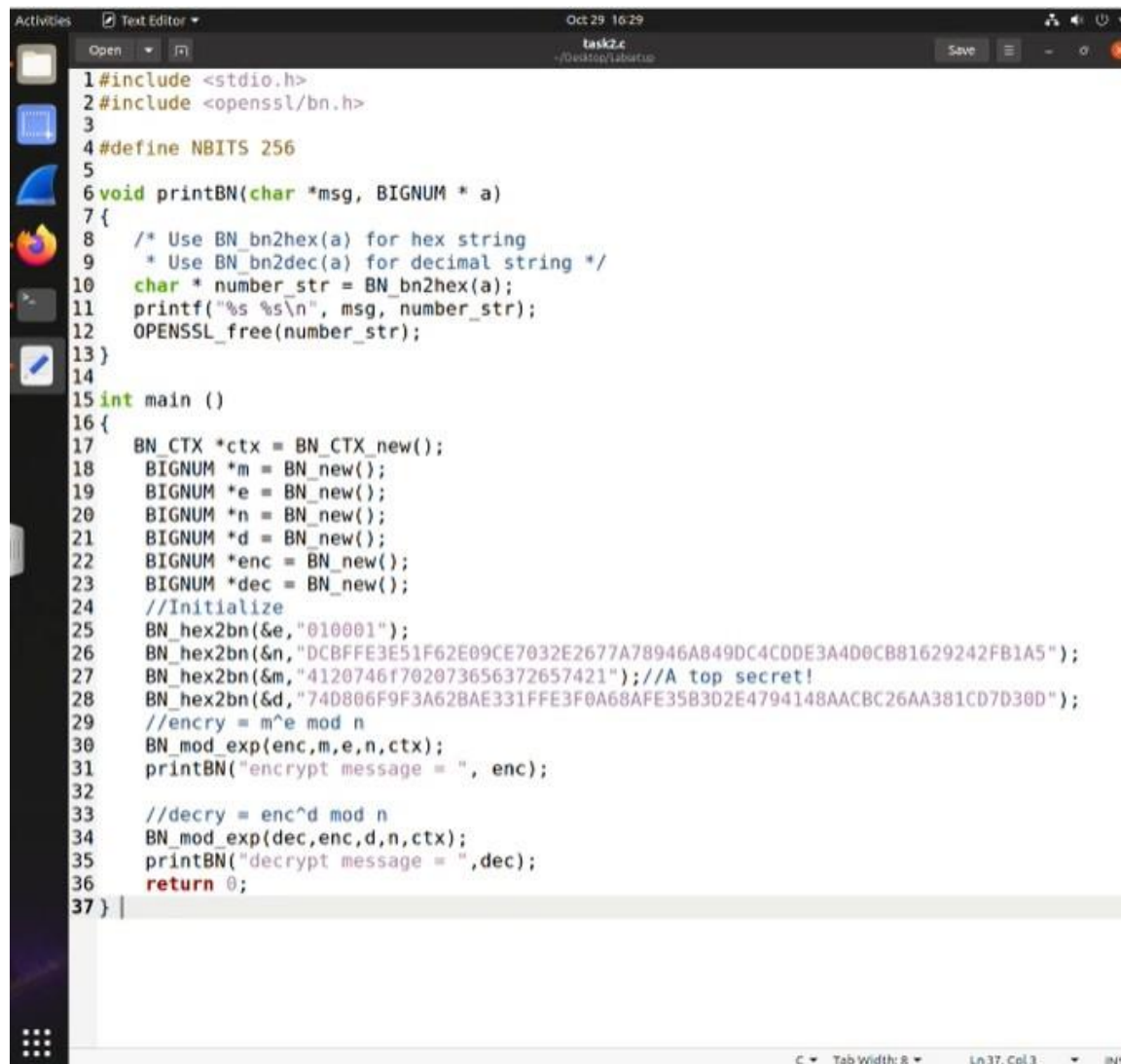
Therefore, we get the private key

$d=3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB$

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



```
1#include <stdio.h>
2#include <openssl/bn.h>
3
4#define NBITS 256
5
6void 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}
14
15int main ()
16{
17   BN_CTX *ctx = BN_CTX_new();
18   BIGNUM *m = BN_new();
19   BIGNUM *e = BN_new();
20   BIGNUM *n = BN_new();
21   BIGNUM *d = BN_new();
22   BIGNUM *enc = BN_new();
23   BIGNUM *dec = BN_new();
24   //Initialize
25   BN_hex2bn(&e, "010001");
26   BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4C0DE3A4D0CB81629242FB1A5");
27   BN_hex2bn(&m, "4120746f702073656372657421");//A top secret!
28   BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
29   //encry = m^e mod n
30   BN_mod_exp(enc,m,e,n,ctx);
31   printBN("encrypt message = ", enc);
32
33   //decry = enc^d mod n
34   BN_mod_exp(dec,enc,d,n,ctx);
35   printBN("decrypt message = ",dec);
36   return 0;
37}
```

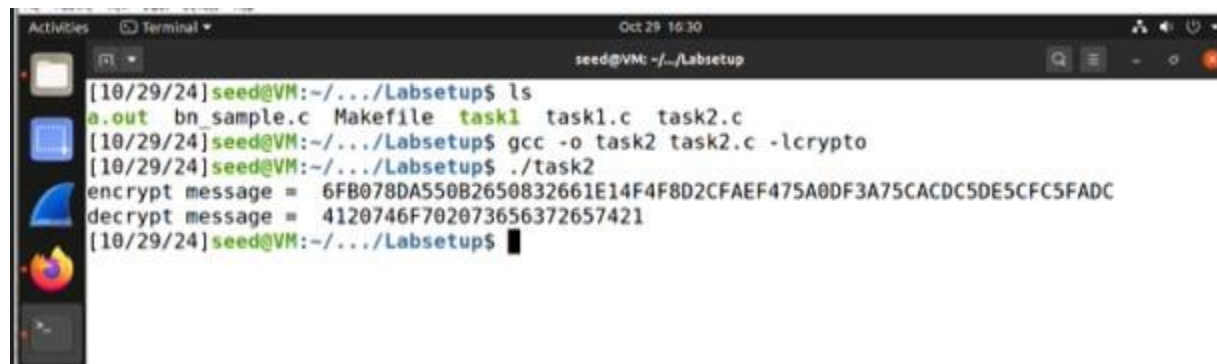
According to the lab, we know

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

Therefore, we can use the hexadecimal of M

```
15 int main ()  
16 {  
17     BN_CTX *ctx = BN_CTX_new();  
18     BIGNUM *m = BN_new();  
19     BIGNUM *e = BN_new();  
20     BIGNUM *n = BN_new();  
21     BIGNUM *d = BN_new();  
22     BIGNUM *enc = BN_new();  
23     BIGNUM *dec = BN_new();  
24     //Initialize  
25     BN_hex2bn(&e, "010001");  
26     BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4C0DE3A4D0CB81629242FB1A5");  
27     BN_hex2bn(&m, "4120746f702073656372657421");//A top secret!  
28     BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");  
29     //encry = m^e mod n  
30     BN_mod_exp(enc, m, e, n, ctx);  
31     printBN("encrypt message = ", enc);  
32  
33     //decry = enc^d mod n  
34     BN_mod_exp(dec, enc, d, n, ctx);  
35     printBN("decrypt message = ", dec);  
36     return 0;  
37 }
```

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



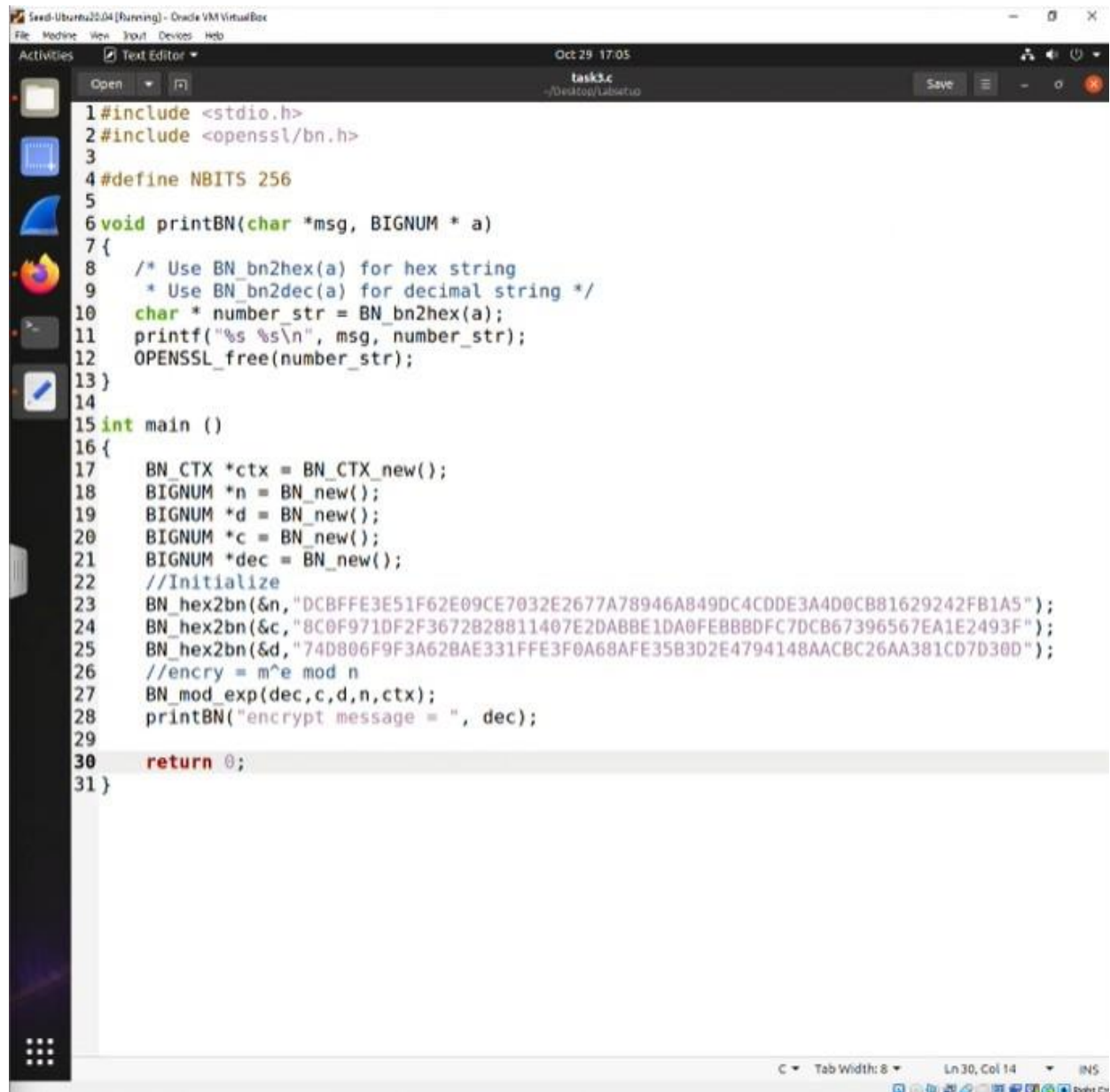
```
Oct 29 16:30  
seed@VM: ~/Labsetup  
[10/29/24] seed@VM:~/../Labsetup$ ls  
a.out bn_sample.c Makefile task1 task1.c task2.c  
[10/29/24] seed@VM:~/../Labsetup$ gcc -o task2 task2.c -lcrypto  
[10/29/24] seed@VM:~/../Labsetup$ ./task2  
encrypt message = 6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC  
decrypt message = 4120746F702073656372657421  
[10/29/24] seed@VM:~/../Labsetup$
```

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

Task 3: Decrypting a Message

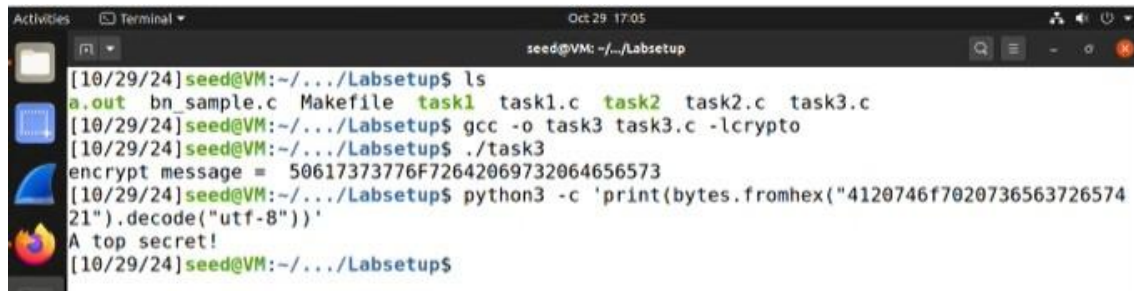
The public/private keys used in this task are the same as the ones used in Task 2. Decrypt the following ciphertext C, and convert it back to a plain ASCII string. We created task3.c.

\$ vim task3.c



```
1#include <stdio.h>
2#include <openssl/bn.h>
3
4#define NBITS 256
5
6void 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}
14
15int main ()
16{
17   BN_CTX *ctx = BN_CTX_new();
18   BIGNUM *n = BN_new();
19   BIGNUM *d = BN_new();
20   BIGNUM *c = BN_new();
21   BIGNUM *dec = BN_new();
22   //Initialize
23   BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
24   BN_hex2bn(&c, "8C0F971DF2F3672B28811407E2DABBE1DA0FE8BBD7C7DCB67396567EA1E2493F");
25   BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
26   //encry = m^e mod n
27   BN_mod_exp(dec,c,d,n,ctx);
28   printBN("encrypt message = ", dec);
29
30   return 0;
31}
```


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

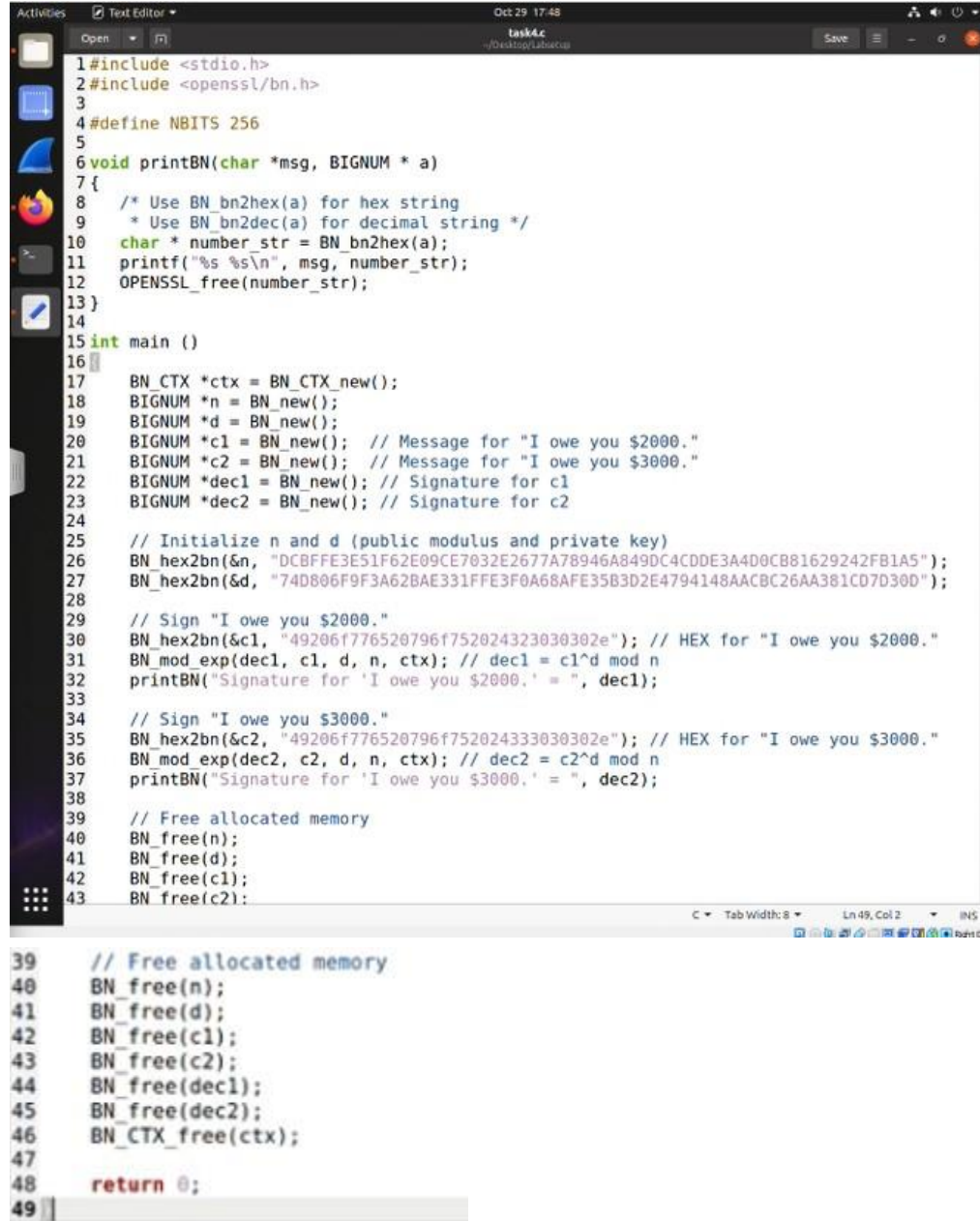
A terminal window titled "Terminal" with a date and time of "Oct 29 17:05". The prompt is "seed@VM: ~/../Labsetup". The user runs "ls" and lists files: "a.out bn_sample.c Makefile task1 task1.c task2 task2.c task3.c". Then they run "gcc -o task3 task3.c -lcrypto". Next, they run "./task3" which outputs "encrypt message = 50617373776F72642069732064656573". Finally, they run "python3 -c 'print(bytes.fromhex(\"4120746f702073656372657421\").decode(\"utf-8\"))'", which outputs "A top secret!".

```
[10/29/24] seed@VM:~/../Labsetup$ ls
a.out bn_sample.c Makefile task1 task1.c task2 task2.c task3.c
[10/29/24] seed@VM:~/../Labsetup$ gcc -o task3 task3.c -lcrypto
[10/29/24] seed@VM:~/../Labsetup$ ./task3
encrypt message = 50617373776F72642069732064656573
[10/29/24] seed@VM:~/../Labsetup$ python3 -c 'print(bytes.fromhex("4120746f702073656372657421").decode("utf-8"))'
A top secret!
[10/29/24] seed@VM:~/../Labsetup$
```

Task 4: Signing a Message

First, we created task4.c.

\$vim task4.c



```
1#include <stdio.h>
2#include <openssl/bn.h>
3
4#define NBITS 256
5
6void 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}
14
15int main ()
16{
17   BN_CTX *ctx = BN_CTX_new();
18   BIGNUM *n = BN_new();
19   BIGNUM *d = BN_new();
20   BIGNUM *c1 = BN_new(); // Message for "I owe you $2000."
21   BIGNUM *c2 = BN_new(); // Message for "I owe you $3000."
22   BIGNUM *dec1 = BN_new(); // Signature for c1
23   BIGNUM *dec2 = BN_new(); // Signature for c2
24
25   // Initialize n and d (public modulus and private key)
26   BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0C881629242FB1A5");
27   BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
28
29   // Sign "I owe you $2000."
30   BN_hex2bn(&c1, "49206f776520796f752024323030302e"); // HEX for "I owe you $2000."
31   BN_mod_exp(dec1, c1, d, n, ctx); // dec1 = c1^d mod n
32   printBN("Signature for 'I owe you $2000.' = ", dec1);
33
34   // Sign "I owe you $3000."
35   BN_hex2bn(&c2, "49206f776520796f752024333030302e"); // HEX for "I owe you $3000."
36   BN_mod_exp(dec2, c2, d, n, ctx); // dec2 = c2^d mod n
37   printBN("Signature for 'I owe you $3000.' = ", dec2);
38
39   // Free allocated memory
40   BN_free(n);
41   BN_free(d);
42   BN_free(c1);
43   BN_free(c2);
44   BN_free(dec1);
45   BN_free(dec2);
46   BN_CTX_free(ctx);
47
48   return 0;
49}
```


Second, we get the hex value of "I owe you \$2000."

Value is 49206f776520796f75203030302e

```
[10/29/24]seed@VM:~/.../Labsetup$ ls
a.out bn_sample.c Makefile task1 task1.c task2 task2.c task3 task3.c task4.c
[10/29/24]seed@VM:~/.../Labsetup$ python3 -c "print('I owe you $2000.'.encode('utf-8').hex())"
49206f776520796f75203030302e
```

Third, we get the hex value of "I owe you \$3000."

Value is 49206f776520796f75203030302e

```
[10/29/24]seed@VM:~/.../Labsetup$ python3 -c "print('I owe you $3000.'.encode('utf-8').hex())"
49206f776520796f75203030302e
```

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

```
Seed-Ubuntu20.04 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal Oct 29 17:48
seed@VM: ~/.../Labsetup

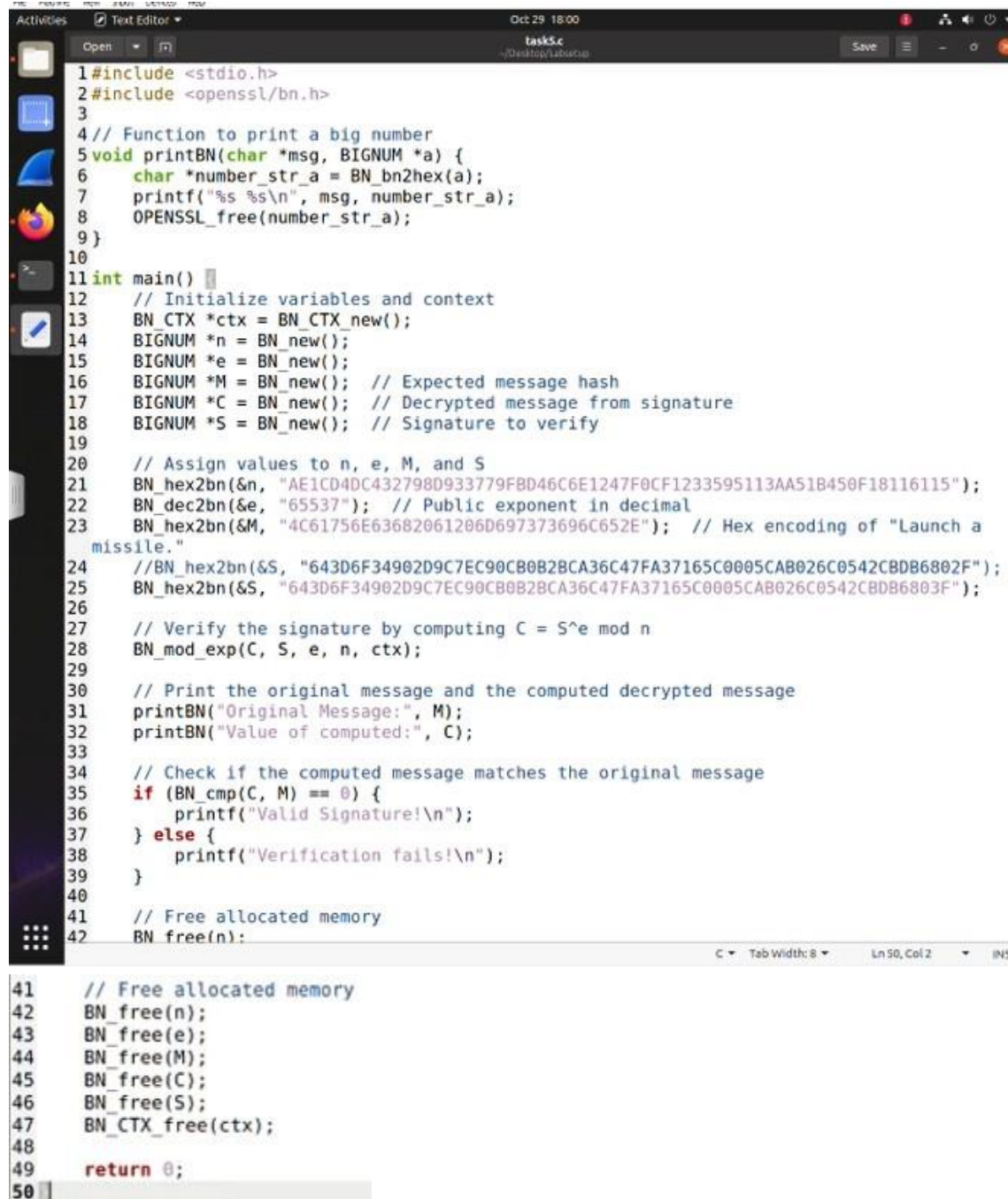
[10/29/24]seed@VM:~/.../Labsetup$ ls
a.out bn_sample.c Makefile task1 task1.c task2 task2.c task3 task3.c task4.c
[10/29/24]seed@VM:~/.../Labsetup$ python3 -c "print('I owe you $2000.'.encode('utf-8').hex())"
49206f776520796f75203030302e
[10/29/24]seed@VM:~/.../Labsetup$ python3 -c "print('I owe you $3000.'.encode('utf-8').hex())"
49206f776520796f75203030302e
[10/29/24]seed@VM:~/.../Labsetup$ gcc -o task4 task4.c -lcrypto
[10/29/24]seed@VM:~/.../Labsetup$ ./task4
Signature for 'I owe you $2000.' = 55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73CCB35E4CB
Signature for 'I owe you $3000.' = BCC20FB7568E5D48E434C387C06A6025E90D29D848AF9C3EBAC0135D99305822
[10/29/24]seed@VM:~/.../Labsetup$
```

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



```
1#include <stdio.h>
2#include <openssl/bn.h>
3
4// Function to print a big number
5void printBN(char *msg, BIGNUM *a) {
6    char *number_str_a = BN_bn2hex(a);
7    printf("%s %s\n", msg, number_str_a);
8    OPENSSL_free(number_str_a);
9}
10
11int main() {
12    // Initialize variables and context
13    BN_CTX *ctx = BN_CTX_new();
14    BIGNUM *n = BN_new();
15    BIGNUM *e = BN_new();
16    BIGNUM *M = BN_new(); // Expected message hash
17    BIGNUM *C = BN_new(); // Decrypted message from signature
18    BIGNUM *S = BN_new(); // Signature to verify
19
20    // Assign values to n, e, M, and S
21    BN_hex2bn(&n, "AE1CD4DC432798D933779F8D46C6E1247F0CF1233595113AA51B450F18116115");
22    BN_dec2bn(&e, "65537"); // Public exponent in decimal
23    BN_hex2bn(&M, "4C61756E63682061206D697373696C652E"); // Hex encoding of "Launch a
    missile."
24    //BN_hex2bn(&S, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542C8DB6802F");
25    BN_hex2bn(&S, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542C8DB6803F");
26
27    // Verify the signature by computing C = S^e mod n
28    BN_mod_exp(C, S, e, n, ctx);
29
30    // Print the original message and the computed decrypted message
31    printBN("Original Message:", M);
32    printBN("Value of computed:", C);
33
34    // Check if the computed message matches the original message
35    if (BN_cmp(C, M) == 0) {
36        printf("Valid Signature!\n");
37    } else {
38        printf("Verification fails!\n");
39    }
40
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 missile.'.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

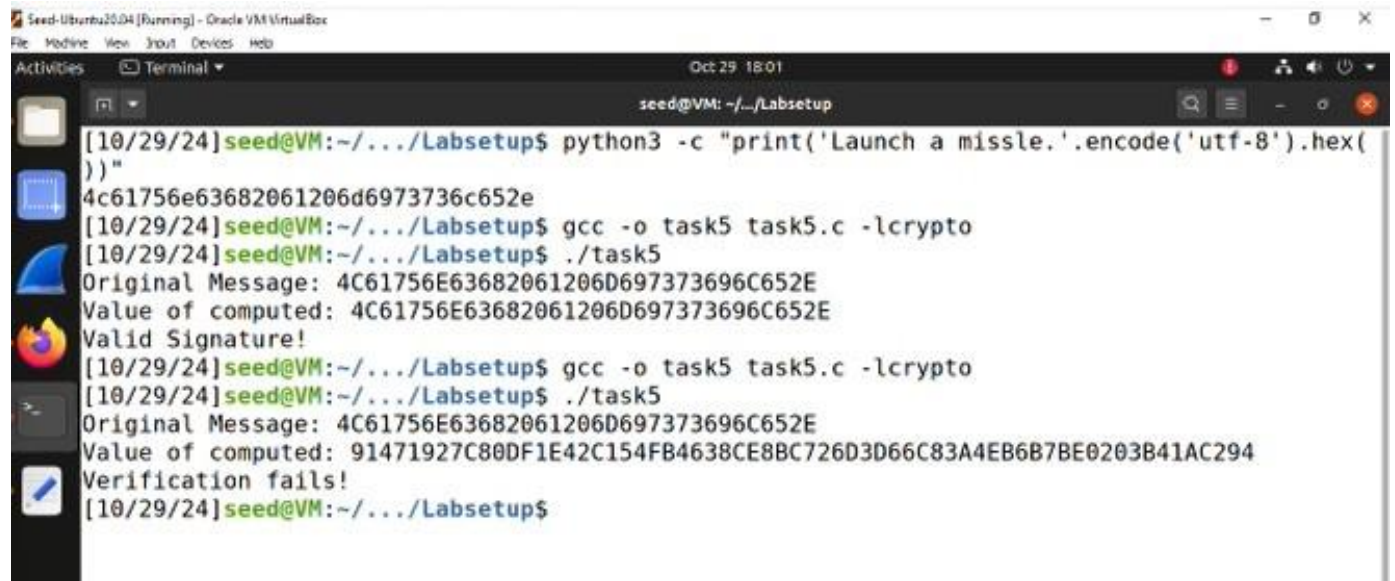
We use the signature to compute the value of the message C.

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:



```
Seed-Ubuntu20.04 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal
Oct 29 18:01
seed@VM: ~/.../Labsetup

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

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
CgYDVQQDEwNXUjIwHhcNMjQxMjM0MDgyNjM2MjM0MDgyNjM1WjAZMRcw
FQYDVQQDEw53d3cuZ29vZ2xlLmNvbTBZMBMGByqGSM49AgEGCCqGSM49AwEHA0IA
BO4/mHOK7c0x6J5PAWAQYK5a9FoKEEKo6yIebzVGmj13TdY5CwIpoXNER+tsQH/B
ijis3XTXpooyvmvgmKu0ohbqjggJCMIICPjA0BgNVHQ8BAf8EBAMCB4AwEwYDVR01
BAwwCgYIKwYBBQUHAwEwDAYDVR0TAQH/BAIwADAdBgNVHQ4EFgQUk9u51kdRRete
dUnhEnK9kbOpfC8wHwYDVR0jBBgwFoAU3hse7XkV1D43JMMhu+w0OW1CsJAwWAYI
KwYBBQUHAQEETDBKMCEGCCsGAQUFBzABhhVodHRwOi8vby5wa2kuZ29vZy93cjIw
JQYIKwYBBQUHMAKGGWh0dHA6Ly9pLnBraS5nb29nL3dyMi5jcnQwGQYDVR0RBBIw
EIIOD3d3Lmdvb2dsZS5jb20wEwYDVR0gBAwwCjAIBGZngQwBAgEwNgYDVR0fBC8w
LTArCmGjY4YlaHR0cDovL2MucGtpLmdvb2cvd3IyLz1VVMJOMHclRTZZLmNybDCC
AQUGCisGAQQB1nkCBAIEgfYEgfMA8QB3AO7N0GTV2xrOxVy3nbTNE6Iyh0Z8vOze
w1FIWUZXh7WbAAABkmZNV8MAAAQDAEgwRgIhAPP9CVkB0GXIS2b7kTXWPPHfK2Ty
Nd87dMf9OB4rNBAXAiEAufW+murZkpZ7q/RQEV2zTK/9o67r/y8GB68lxEZ03dAA
dgBISOnr2qZHNA/lagL6nTDrHFIBylbdLIHzu7+rOdiEcwAAAZJmTVfnAAAEAwBH
MEUCIGTyK07CmC5fWrNvdbOyqWUJy5pHLY+5cUfnPQbtv1KxAiEAwrIZK079rAlR
IU+heWW8IE8KeQGxntqAxDjmkPfm9aAwDQYJKoZIhvcNAQELBQADggEBAJ39sqzq
Vv8185didLQhMoJncKSntbcNCDNgiCpTEw/oDv+Gy56ngWnayGeZWpYPk3epPSDC
iJTqPL/SkZMyhaOQOYXGFfP3JfmNeYEj9oCEJvdZMuu+n0eOkEao09GXqRnI1VR
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/AFoHxM3tEARZlmpRB7mDANBgkqhkiG9w0BAQsFADBH
MQswCQYDVQQGEwJVUzEiMCAGA1UEChMZR29vZ2xlIFRydXN0IFNlcnZpY2VzIEExM
QZEUmBIGA1UEAxMLR1RTIFJvb3QgUjEwHhcNMjQxMjM0MDgyNjM2MjM0MDgyNjM1Wj
MTQwMDAwWjA7MQswCQYDVQQGEwJVUzEeMBwGA1UEChMVR29vZ2xlIFRydXN0IFNl
cnZpY2VzMQwwCgYDVQQDEwNXUjIwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEK
```

AoIBAQCp/5x/RR5wqFOfytnlDd5GVld9vI+aWqxG8YSau5HbyfsvAfuSCQAWXqAc
+MGr+XgvSszYhaLYWTW00xj7sfUkDSbutltkdnwUxy96zqhMt/TZCPzfhyM1IKji
aeKMTj+xWfpgoh6zySBTGYLKNlNtYE3pAJH8do1cCA8Kwtzxc2vFE24KT3rC8gIc
LrRjg9ox9i1lMLL7q8Ju26nADrn5Z9TDJVd06wW06Y613ijNzHoU5HEDy01hLmFX
xRmpC5iEGuh5KdmyjS//V2pm4M6rlagplmNwEmceOuHbsCFx13ye/aoXbv4r+zgX
FNFmp6+atXDMYGOBOozAKql2N87jAgMBAAGjgf4wgfsWdgYDVR0PAQH/BAQDAgGG
MB0GA1UdJQQWMBQGCCsGAQUFBwMBBggrBgEFBQcDAjASBgNVHRMBAf8ECDAGAQH/
AgEAMBOGA1UdDgQWBBTeGx7teRXUPjckwyG77DQ5bUKyMDAfBgNVHSMEGDAWgBTk
rysmcRorSCeFL1JmLO/wiRNxPjA0BggrBgEFBQcBAQQoMCYwJAYIKwYBBQUHMAKG
GGh0dHA6Ly9pLnBraS5nb29nL3IxLmNyddArBgNVHR8EJDAiMCCgHqAchhpodHRw
Oi8vYy5wa2kuZ29vZy9yL3IxLmNybdATBgNVHSAEDDAKMAgGBmeBDAECATANBgkq
hkiG9w0BAQsFAAOCAgEAXWL5R87RBOWGqtY8TXJbz3S0DNKhjO6V1FP7sQ02hYS
TL8Tnw3UV0lIecAwPJQl8hr0ujKUtjNyC4XuCRElNJThb0Lbgpt7fyqaqf9/qdLe
SiDLs/sDA7j4BwXaWZiVGEaYzq9yviQmsR4ATb0IrZNBRAq7x9UBhb+TV+PfdBJT
DhEl05vc3ssnbrPCuTNiOcLgNeFbpwkuGcuRKnZc8d/KI4RApW//mkHgte8y0YWu
ryUJ8GLFbsLIbjL9uNrzkqRSvOFVU6xddZIMy9vhNkSXJ/UcZhjJY1pXAprffJB
vei7j+Qi151lRehMCOfa6WBmiA4fx+FOVsV2/7R6V2nyAiIJJkEd2nSi5SnzxJr1
Xdagev3htytMOPvoKwa676ATL/hzfvDaQBECxd2Ppvy+275W+DKcH0FBbX62xevG
iza3F4ydzx16NJ8hk8R+dDXSqv1MbRTlybB5W0k8878XSOjvmiYTDIfyc9acxVJr
Y/cyKHipa+telPohv7wYPytZ9orGBV5SGOJm4NrB3K1aJar0RfzxC3ikr7Dyc6Qw
qDTBU39CluVIQeuQRgwG3MuSxl7zRERDRilGoKb8uY45JzmxWuKxrfwT/478JuHU
/oTxUFqOl2stKnn7QGTq8z29W+GgBLCSBxC9epaHM0myFH/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/E8FjTDTANBgkqhkiG9w0BAQsFAADBx
MQswCQYDVQQGEWJCRCRTEZMBcGA1UEChMQR2xvYmFsU2lnbiBudilzYTEQMA4GA1UE
CxMHU9vdCBDQTEBMBkGA1UEAxMSR2xvYmFsU2lnbiBSb290IENBMB4XDTIwMDYx
OTAwMDA0Ml0XDTI0MDEyODAwMDA0MlowRzELMAkGA1UEBhMCVVMxIjAgBgNVBAoT
GUdub2dsZSBUcnVzdCBTZXJ2aWNlcyBMTEMxZDASBgNVBAMTC0dUUyBSb290IFIx
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAEthECix7joXebO9y/ld63
ladAPKH9gv19MgaCcfb2jH/76Nu8ai6Xl6OMS/kr9rH5zoQdsfnFl97vufKj6bwS
iV6nqlKr+CMny6SxnGPb15l+8Ape62im9MZArlNEDPjTrET08gYbEvs/AmQ351k
KSUjB6G00j0uYODP0gmHu81I8E3CwnqIiru6z1kZ1q+PsAewnJHxgsHA3y6mbWwZ
DrXYfiYaRQM9sHmklCitD38m5agI/pboPGiUU+6DOogrFZYJsuB6jC511pzrp1Zk
j5ZPaK49l8KEj8C8QMALXL32h7M1bKwYUH+E4EzNktMg6TO8UpmvMrUpsyUqtEj5
cuHKZPfmghCN6J3Cioj60GaK/GP5Afl4/Xtcd/p2h/rs37EOeZVXtL0m79YB0esW
CruOC7XFxYpVq9Os6pFLKcwZpDilTirxZUTQAs6qzkm06p98g7BAe+dDq6dso499
iYH6TKX/1Y7DzkgvtdizjKXPdsDtQCv9Uw+wp9U7DbGKogPeMa3Md+pvez7W35Ei
Eua++tgY/BBjFFfy3l3WFpO9KWgz7zpm7AeKJt8T11dleCfeXkkUAKIAf5qoIbap
sZWwpbkNfHhax2xIPEDgfg1azVY80ZcFuctL7TlLnMQ/0lUTbiSwlnH69MG6z00b
9f6BQdgAmD06yK56mDcYBZUCAwEAAaOCATgwggE0MA4GA1UdDwEB/wQEAWIBhjAP
BgNVHRMBAf8EBTADAQH/MB0GA1UdDgQWBBTkrysmcRorSCeFL1JmLO/wiRNxPjAf
BgNVHSMEGDAWgBRge2YarQ2XyolQL30EzTSO//z9SzBgBggrBgEFBQcBAQRUMFIw
JQYIKwYBBQUHMAGGGWh0dHA6Ly9vY3NwLnBraS5nb29nL2dzcjEwKQYIKwYBBQUH
MAKGHWh0dHA6Ly9wa2kuZ29vZy9nc3IxL2dzcjEuY3J0MDIGA1UdHwQrMCkwJ6A1

oCOGIWh0dHA6Ly9jcmwucGtpLmdvb2cvZ3NyMS9nc3IwLmNybDA7BgNVHSAENDAY
MAgGBmeBDAECAITAIBgZngQwBAGIwDQYLKwYBBAHWeQIFAwIwDQYLKwYBBAHWeQIF
AwMwDQYJKoZIhvcNAQELBQADggEBADSkHrEoo9C0dhemMXoh6dFSPsjbdBZBiLg9
NR3t5P+T4Vxfq7vqfM/b5A3RilfyJm9bvhdGaJQ3b2t6yMAYN/olUazsaL+yyEn9
WprKASOshIArAoyZl+tJaox118fessmXnlhIVw41oeQalv1vg4Fv74zPl6/AhSrw
9U5pCZEt4Wi4wStz6dTZ/CLANx8LZh1J7QJVj2fhMtfTJr9w4z30Z209fOU0iOMy
+qduBmpvvYuR7hZL6Dupszfnw0Skfths18dG9ZKb59UhvmaSGZRVbnQpsg3BZlvi
d01IKO2dlxozclOzgJXPYovJJIultzkMu34qQb9Sz/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

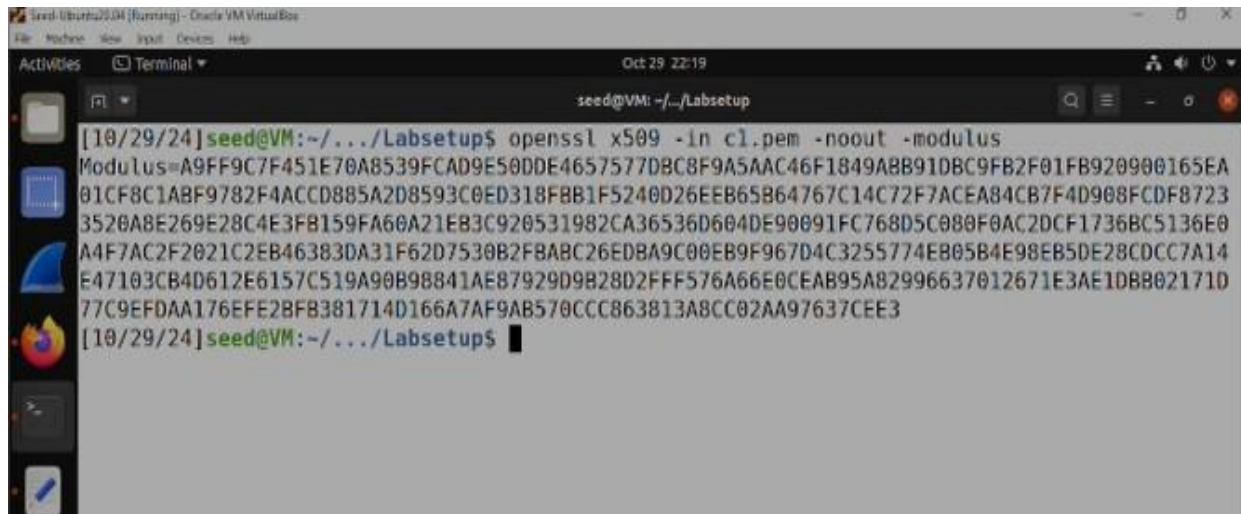
...`.

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:



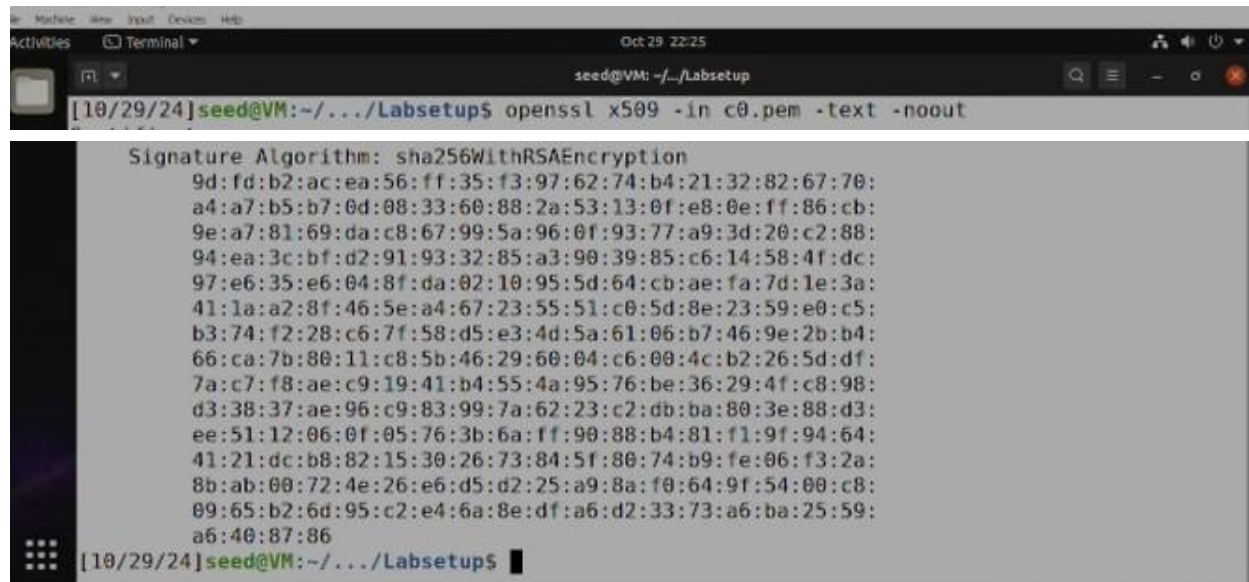
```
[10/29/24]seed@VM:~/../Labsetup$ openssl x509 -in cl.pem -noout -modulus
Modulus=A9FF9C7F451E70A8539FCAD9E500DE4657577DBC8F9A5AAC46F1849A8B91DBC9FB2F01FB920900165EA
01CF8C1ABF9782F4ACCD885A2D8593C0ED318F8B1F5240D26EEB65864767C14C72F7ACEA84CB7F4D908FCDF8723
3520A8E269E28C4E3FB159FA60A21E83C920531982CA36536D604DE90091FC768D5C080F0AC2DCF1736BC5136E0
A4F7AC2F2021C2EB46383DA31F62D7530B2F8ABC26EDBA9C00EB9F967D4C3255774E805B4E98EB5DE28DCC7A14
E47103CB4D612E6157C519A90B98841AE87929D9828D2FFF576A66E0CEAB95A82996637012671E3AE1DBB02171D
77C9EFDAA176EFE28FB381714D166A7AF9AB570CCC863813A8CC02AA97637CEE3
[10/29/24]seed@VM:~/../Labsetup$
```

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

```
Oct 29 22:22
seed@VM: ~/Labsetup
[10/29/24]seed@VM:~/Labsetup$ openssl x509 -in c1.pem -text -noout
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number:
      7f:f0:05:a0:7c:4c:de:d1:00:ad:9d:66:a5:10:7b:98
    Signature Algorithm: sha256WithRSAEncryption
    Issuer: C = US, O = Google Trust Services LLC, CN = GTS Root R1
    Validity
      Not Before: Dec 13 09:00:00 2023 GMT
      Not After : Feb 20 14:00:00 2029 GMT
    Subject: C = US, O = Google Trust Services, CN = WR2
    Subject Public Key Info:
      Public Key Algorithm: rsaEncryption
      RSA Public-Key: (2048 bit)
      Modulus:
        00:a9:ff:9c:7f:45:1e:70:a8:53:9f:ca:d9:e5:0d:
        de:46:57:57:7d:bc:8f:9a:5a:ac:46:f1:84:9a:bb:
        91:db:c9:fb:2f:01:fb:92:09:00:16:5e:a0:1c:f8:
        c1:ab:f9:78:2f:4a:cc:d8:85:a2:d8:59:3c:0e:d3:
        18:fb:b1:f5:24:0d:26:ee:b6:5b:64:76:7c:14:c7:
        2f:7a:ce:a8:4c:b7:f4:d9:08:fc:df:87:23:35:20:
        a8:e2:69:e2:8c:4e:3f:b1:59:fa:60:a2:1e:b3:c9:
        20:53:19:82:ca:36:53:6d:60:4d:e9:00:91:fc:76:
        8d:5c:08:0f:0a:c2:dc:f1:73:6b:c5:13:6e:0a:4f:
        7a:c2:f2:02:1c:2e:b4:63:83:da:31:f6:2d:75:30:
        b2:fb:ab:c2:6e:db:a9:c0:0e:b9:f9:67:d4:c3:25:
        57:74:eb:05:b4:e9:8e:b5:de:28:cd:cc:7a:14:e4:
        71:03:cb:4d:61:2e:61:57:c5:19:a9:0b:98:84:1a:
        e8:79:29:d9:b2:8d:2f:ff:57:6a:66:e0:ce:ab:95:
        a8:29:96:63:70:12:67:1e:3a:e1:db:b0:21:71:d7:
        7c:9e:fd:aa:17:6e:fe:2b:fb:38:17:14:d1:66:a7:
        af:9a:b5:70:cc:c8:63:81:3a:8c:c0:2a:a9:76:37:
        ce:e3
      Exponent: 65537 (0x10001)
    X509v3 extensions:
      X509v3 Key Usage: critical
        Digital Signature, Certificate Sign, CRL Sign
      X509v3 Extended Key Usage:
        TLS Web Server Authentication, TLS Web Client Authentication
      X509v3 Basic Constraints: critical
        CA:TRUE, pathlen:0
      X509v3 Subject Key Identifier:
        0C:10:1E:CB:70:15:04:2E:27:74:03:71:00:EE:24:70:60:A7:07:30
```


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.



```
[10/29/24]seed@VM:~/.../Labsetup$ openssl x509 -in c0.pem -text -noout
Signature Algorithm: sha256WithRSAEncryption
9d:fd:b2:ac:ea:56:ff:35:f3:97:62:74:b4:21:32:82:67:70:
a4:a7:b5:b7:0d:08:33:60:88:2a:53:13:0f:e8:0e:ff:86:cb:
9e:a7:81:69:da:c8:67:99:5a:96:0f:93:77:a9:3d:20:c2:88:
94:ea:3c:bf:d2:91:93:32:85:a3:90:39:85:c6:14:58:4f:dc:
97:e6:35:e6:04:8f:da:02:10:95:5d:64:cb:ae:fa:7d:1e:3a:
41:1a:a2:8f:46:5e:a4:67:23:55:51:c0:5d:8e:23:59:e0:c5:
b3:74:f2:28:c6:7f:58:d5:e3:4d:5a:61:06:b7:46:9e:2b:b4:
66:ca:7b:80:11:c8:5b:46:29:60:04:c6:00:4c:b2:26:5d:df:
7a:c7:f8:ae:c9:19:41:b4:55:4a:95:76:be:36:29:4f:c8:98:
d3:38:37:ae:96:c9:83:99:7a:62:23:c2:db:ba:80:3e:88:d3:
ee:51:12:06:0f:05:76:3b:6a:ff:90:88:b4:81:f1:9f:94:64:
41:21:dc:b8:82:15:30:26:73:84:5f:80:74:b9:fe:06:f3:2a:
8b:ab:00:72:4e:26:e6:d5:d2:25:a9:8a:f0:64:9f:54:00:c8:
09:65:b2:6d:95:c2:e4:6a:8e:df:a6:d2:33:73:a6:ba:25:59:
a6:40:87:86
[10/29/24]seed@VM:~/.../Labsetup$
```

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



```
[10/29/24]seed@VM:~/.../Labsetup$ vim signature
[10/29/24]seed@VM:~/.../Labsetup$ cat signature | tr -d '[:space:]'
9dfdb2acea56ff35f3976274b42132826770a4a7b5b70d083360882a53130fe80eff86cb9ea78169dac867995a9
60f9377a93d20c28894ea3cbfd291933285a3903985c614584fdc97e635e6048fda0210955d64cbaefa7d1e3a41
1aa28f465ea467235551c05d8e2359e0c5b374f228c67f58d5e34d5a6106b7469e2bb466ca7b8011c85b4629600
4c6004cb2265ddf7ac7f8aec91941b4554a9576be36294fc898d33837ae96c983997a6223c2dbba803e88d3ee51
12060f05763b6aff9088b481f19f94644121dcb88215302673845f8074b9fe06f32a8bab00724e26e6d5d225a98
af0649f5400c80965b26d95c2e46a8edfa6d23373a6ba2559a6408786[10/29/24]seed@VM:~/.../Labsetup$
```

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

```
Oct 29 22:34
seed@VM: ~/Labsetup
[10/29/24] seed@VM:~/.../Labsetup$ openssl asn1parse -i -in c0.pem
0:d=0 hl=4 l=1111 cons: SEQUENCE
4:d=1 hl=4 l= 831 cons: SEQUENCE
8:d=2 hl=2 l= 3 cons: cont [ 0 ]
10:d=3 hl=2 l= 1 prim: INTEGER :02
13:d=2 hl=2 l= 16 prim: INTEGER :0D43B14ABB9C159610E13D55239F254E
31:d=2 hl=2 l= 13 cons: SEQUENCE
33:d=3 hl=2 l= 9 prim: OBJECT :sha256WithRSAEncryption
44:d=3 hl=2 l= 0 prim: NULL
46:d=2 hl=2 l= 59 cons: SEQUENCE
48:d=3 hl=2 l= 11 cons: SET
50:d=4 hl=2 l= 9 cons: SEQUENCE
52:d=5 hl=2 l= 3 prim: OBJECT :countryName
57:d=5 hl=2 l= 2 prim: PRINTABLESTRING :US
61:d=3 hl=2 l= 30 cons: SET
63:d=4 hl=2 l= 28 cons: SEQUENCE
65:d=5 hl=2 l= 3 prim: OBJECT :organizationName
70:d=5 hl=2 l= 21 prim: PRINTABLESTRING :Google Trust Services
93:d=3 hl=2 l= 12 cons: SET
95:d=4 hl=2 l= 10 cons: SEQUENCE
97:d=5 hl=2 l= 3 prim: OBJECT :commonName
102:d=5 hl=2 l= 3 prim: PRINTABLESTRING :WR2
107:d=2 hl=2 l= 30 cons: SEQUENCE
109:d=3 hl=2 l= 13 prim: UTCTIME :241007082636Z
124:d=3 hl=2 l= 13 prim: UTCTIME :241230082635Z
139:d=2 hl=2 l= 25 cons: SEQUENCE
141:d=3 hl=2 l= 23 cons: SET
143:d=4 hl=2 l= 21 cons: SEQUENCE
145:d=5 hl=2 l= 3 prim: OBJECT :commonName
150:d=5 hl=2 l= 14 prim: PRINTABLESTRING :www.google.com
166:d=2 hl=2 l= 89 cons: SEQUENCE
168:d=3 hl=2 l= 19 cons: SEQUENCE
170:d=4 hl=2 l= 7 prim: OBJECT :id-ecPublicKey
179:d=4 hl=2 l= 8 prim: OBJECT :prime256v1
189:d=3 hl=2 l= 66 prim: BIT STRING
257:d=2 hl=4 l= 578 cons: cont [ 3 ]
261:d=3 hl=4 l= 574 cons: SEQUENCE
265:d=4 hl=2 l= 14 cons: SEQUENCE
267:d=5 hl=2 l= 3 prim: OBJECT :X509v3 Key Usage
272:d=5 hl=2 l= 1 prim: BOOLEAN :255
275:d=5 hl=2 l= 4 prim: OCTET STRING [HEX DUMP]:03020780
281:d=4 hl=2 l= 19 cons: SEQUENCE
283:d=5 hl=2 l= 3 prim: OBJECT :X509v3 Extended Key Usage
285:d=5 hl=2 l= 1 prim: OBJECT :sha256WithRSAEncryption
```

```

Oct 29 22:35
seed@VM: ~/.../Labsetup

288:d=5 hl=2 l= 12 prim: OCTET STRING [HEX DUMP]:300A06082B06010505070301
302:d=4 hl=2 l= 12 cons: SEQUENCE
304:d=5 hl=2 l= 3 prim: OBJECT :X509v3 Basic Constraints
309:d=5 hl=2 l= 1 prim: BOOLEAN :255
312:d=5 hl=2 l= 2 prim: OCTET STRING [HEX DUMP]:3000
316:d=4 hl=2 l= 29 cons: SEQUENCE
318:d=5 hl=2 l= 3 prim: OBJECT :X509v3 Subject Key Identifier
323:d=5 hl=2 l= 22 prim: OCTET STRING [HEX DUMP]:04149308B996475145E85E7549E1
1272B09183A97C2F
347:d=4 hl=2 l= 31 cons: SEQUENCE
349:d=5 hl=2 l= 3 prim: OBJECT :X509v3 Authority Key Identifier
354:d=5 hl=2 l= 24 prim: OCTET STRING [HEX DUMP]:30168014DE1B1EED7915D43E3724
C32188EC34396D428230
380:d=4 hl=2 l= 88 cons: SEQUENCE
382:d=5 hl=2 l= 8 prim: OBJECT :Authority Information Access
392:d=5 hl=2 l= 76 prim: OCTET STRING [HEX DUMP]:304A302106082B06010505073001
8615687474703A2F2F6F6E706B692E676F6F672F777232302506082B060105050730028619687474703A2F2F692
E706B692E676F6F672F7772322E637274
470:d=4 hl=2 l= 25 cons: SEQUENCE
472:d=5 hl=2 l= 3 prim: OBJECT :X509v3 Subject Alternative Name
477:d=5 hl=2 l= 18 prim: OCTET STRING [HEX DUMP]:3010820E777772E676F6F676C65
2E636F6D
497:d=4 hl=2 l= 19 cons: SEQUENCE
499:d=5 hl=2 l= 3 prim: OBJECT :X509v3 Certificate Policies
504:d=5 hl=2 l= 12 prim: OCTET STRING [HEX DUMP]:300A3008060667810C010201
518:d=4 hl=2 l= 54 cons: SEQUENCE
520:d=5 hl=2 l= 3 prim: OBJECT :X509v3 CRL Distribution Points
525:d=5 hl=2 l= 47 prim: OCTET STRING [HEX DUMP]:302D302BA029A027862568747470
3A2F2F632E706B692E676F6F672F7772322F395556624E3077354536592E63726C
574:d=4 hl=4 l= 261 cons: SEQUENCE
578:d=5 hl=2 l= 10 prim: OBJECT :CT Precertificate SCTs
590:d=5 hl=3 l= 246 prim: OCTET STRING [HEX DUMP]:0481F300F1007700EECD0064D5D8
1ACEC55CB79D84CD13A23287467CBCECDEC351485946711FB59B00000192664D57C30000040300483046022100F
3FD095901D065E24866F89135D63CF1C59364F235DF3B74C7F0381E28341031022100B9F5BE9AED992967BABF4
50115D834CAFFDA3AE8BF2F0607AF25C44674DD000760048B0E36BDA647340FE56A02FA9D30EB1C5201CB56D
D2C81D98BBFAB39D8847300000192664D57E70000040300473045022064F22B4EC2982E5F5AB36F75B3B2AB0509
CB9A472D8F897147E73D06E0BF5281022100C2B21928EEFDAC0951214FA17965BC204F0A7901979EDA80C438E69
0F7CCF5A0
839:d=1 hl=2 l= 13 cons: SEQUENCE
841:d=2 hl=2 l= 9 prim: OBJECT :sha256WithRSAEncryption
852:d=2 hl=2 l= 0 prim: NULL
854:d=1 hl=4 l= 257 prim: BIT STRING
[10/29/24]seed@VM:~/.../Labsetup$

```

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.

```

Oct 29 22:33
seed@VM: ~/.../Labsetup

[10/29/24]seed@VM:~/.../Labsetup$ openssl asn1parse -i -in c0.pem -strparse 4 -out c0_body.
bin -noout
[10/29/24]seed@VM:~/.../Labsetup$ sha256sum c0_body.bin
534c219ac3dabca895eb904a49b148f301c550c304dde4b37fa8a2204758b2dd c0_body.bin
[10/29/24]seed@VM:~/.../Labsetup$

```

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

```
#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");

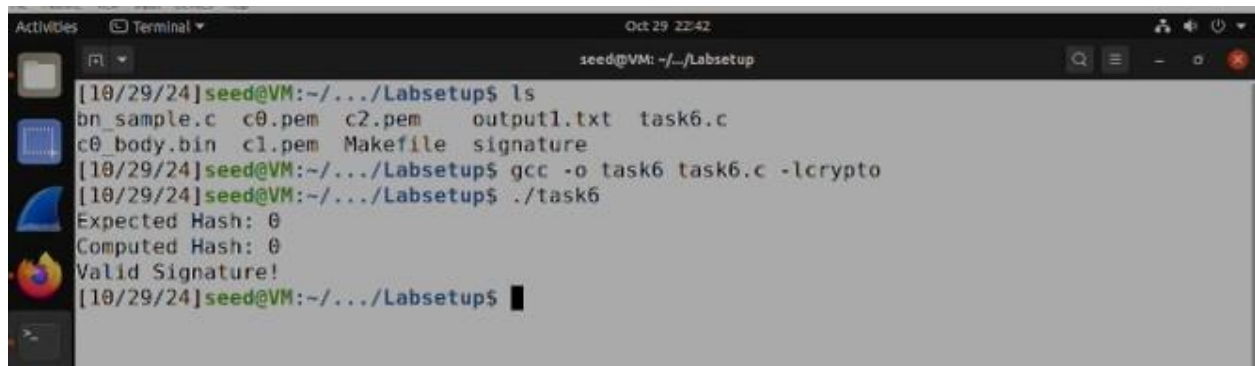
    BN_mod_exp(M1, S, e, n, ctx);

    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;
}
```

A terminal window titled 'Terminal' with a date and time of 'Oct 29 22:42'. The user is 'seed@VM' in the directory '~/../Labsetup'. The terminal shows the following commands and output:

```
[10/29/24]seed@VM:~/../Labsetup$ ls
bn_sample.c  c0.pem  c2.pem  output1.txt  task6.c
c0_body.bin  c1.pem  Makefile  signature
[10/29/24]seed@VM:~/../Labsetup$ gcc -o task6 task6.c -lcrypto
[10/29/24]seed@VM:~/../Labsetup$ ./task6
Expected Hash: 0
Computed Hash: 0
Valid Signature!
[10/29/24]seed@VM:~/../Labsetup$
```

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.