Crypto RSA

Seed Lab

By Aneesh Nagdev

October 26, 2025

Task 1:

Code for Task1 file:

seed@VM: ~/.../Labsetup

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

#define NBITS 128

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();  // Prime p
    BIGNUM *q = BN_new();  // Prime q
    BIGNUM *e = BN_new();  // Public exponent e
    BIGNUM *n = BN_new();  // Modulus n
    BIGNUM *phi_n = BN_new(); // Euler's Totient (n)
    BIGNUM *d = BN_new();  // Private exponent d

    // Set values for p, q, e (hexadecimal to BIGNUM conversion
)
    BN_hex2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF");
    BN_hex2bn(&q, "E85CED54AF57E53E092113E62F436F4F");
    BN_hex2bn(&e, "0D88C3");

    // Compute n = p * q
    BN_mul(n, p, q, ctx);
    printBN("Modulus n = ", n);
```

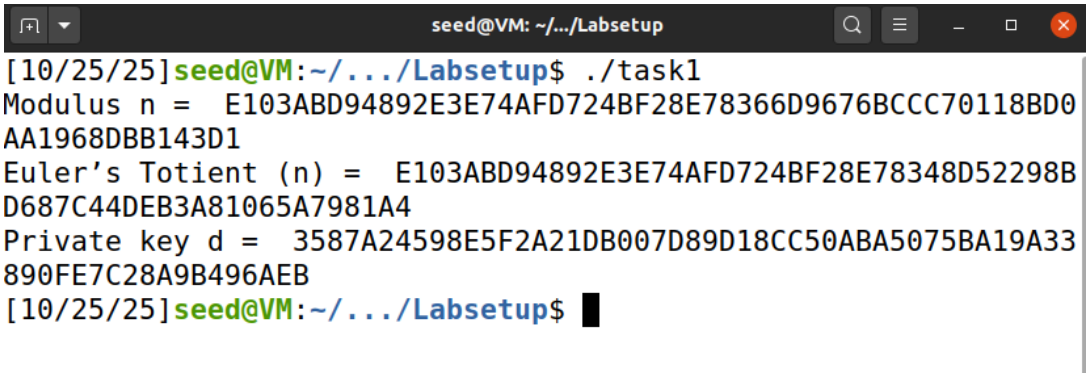
1,5 Top

```
    BN_sub(p, p, BN_value_one());  // p = p - 1
    BN_sub(q, q, BN_value_one());  // q = q - 1
    BN_mul(phi_n, p, q, ctx);
    printBN("Euler's Totient φ(n) = ", phi_n);

    BN_mod_inverse(d, e, phi_n, ctx);  // Calculate d (private
key)
    printBN("Private key d = ", d);

    BN_free(p);
    BN_free(q);
    BN_free(e);
    BN_free(n);
    BN_free(phi_n);
    BN_free(d);
    BN_CTX_free(ctx);

    return 0;
}
```

:

Compiling task1.c screenshot:

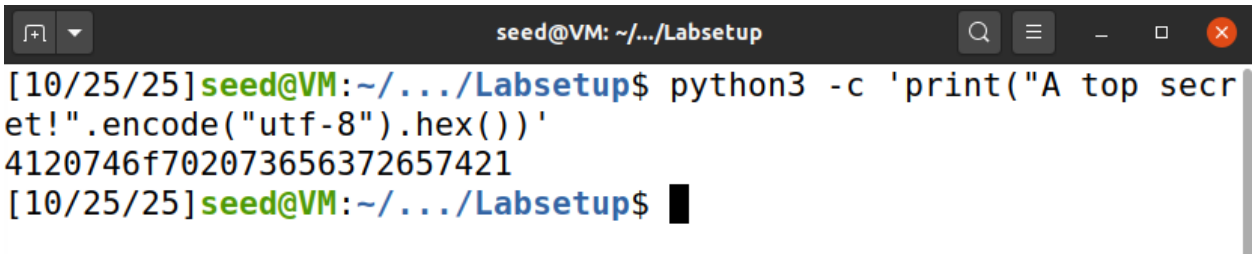
Running task1 screenshot:



Description: In this file, we created task1 file and derived the private key d from the public key components e and n. The output for the file can be seen in the screenshot above.

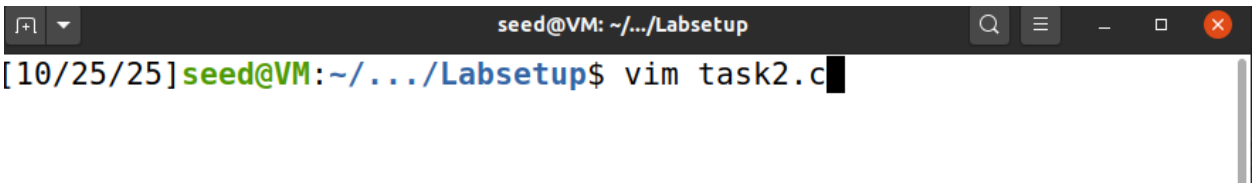
Task 2:

Running the python3 code snippet screenshot:

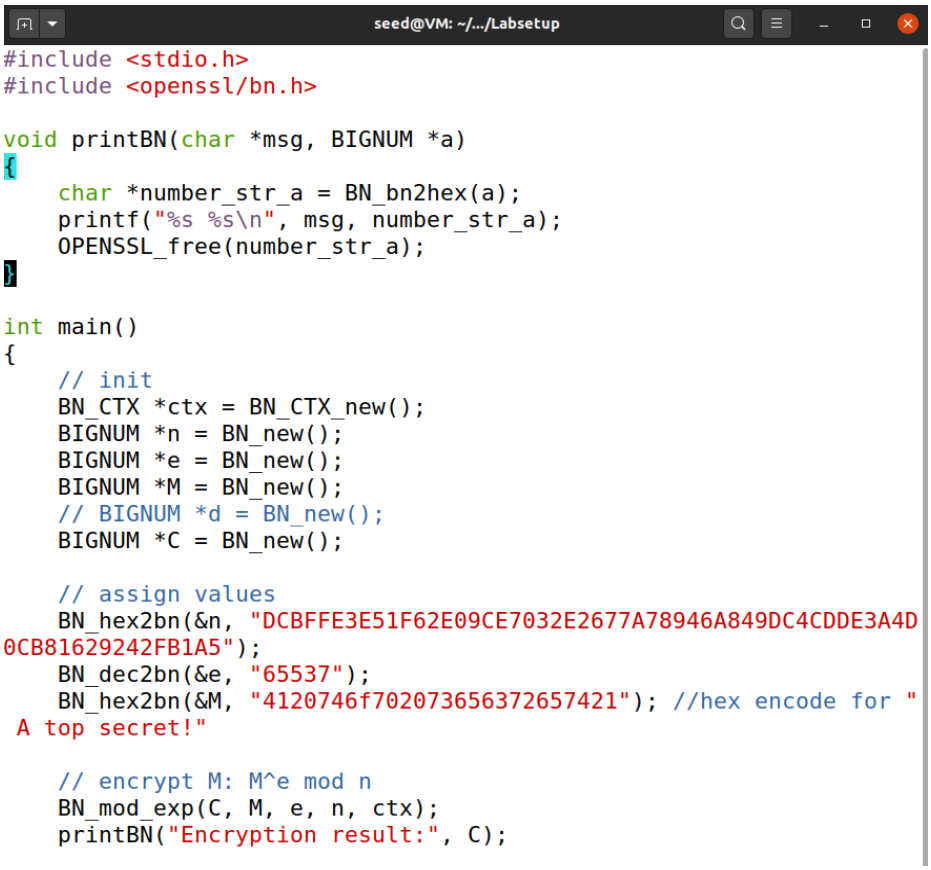


Description: The hex string returned represents the ASCII values of the message “A top secret!”.

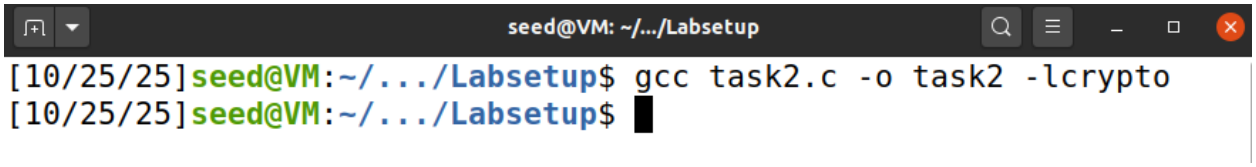
Creating task2.c file screenshot:



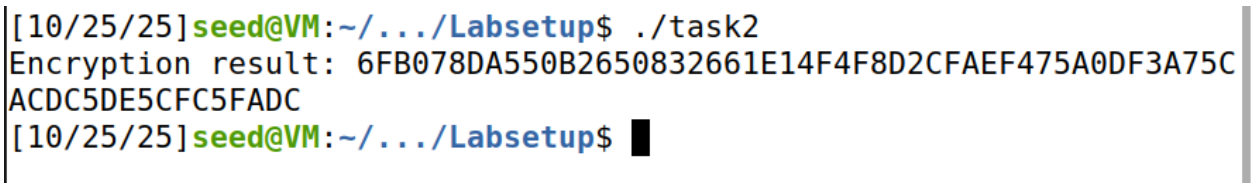
Partial screenshot of task2 code snippet:



Compiling task2.c file screenshot:



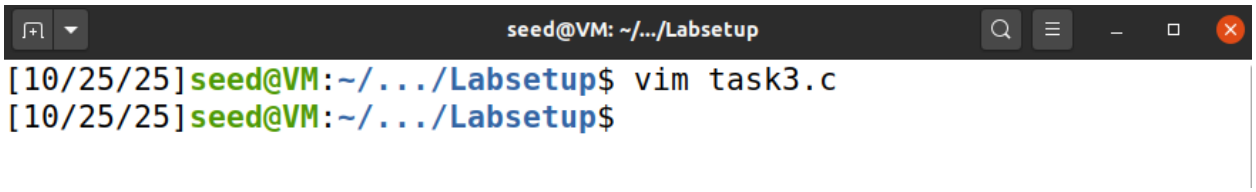
Running task2 file screenshot:



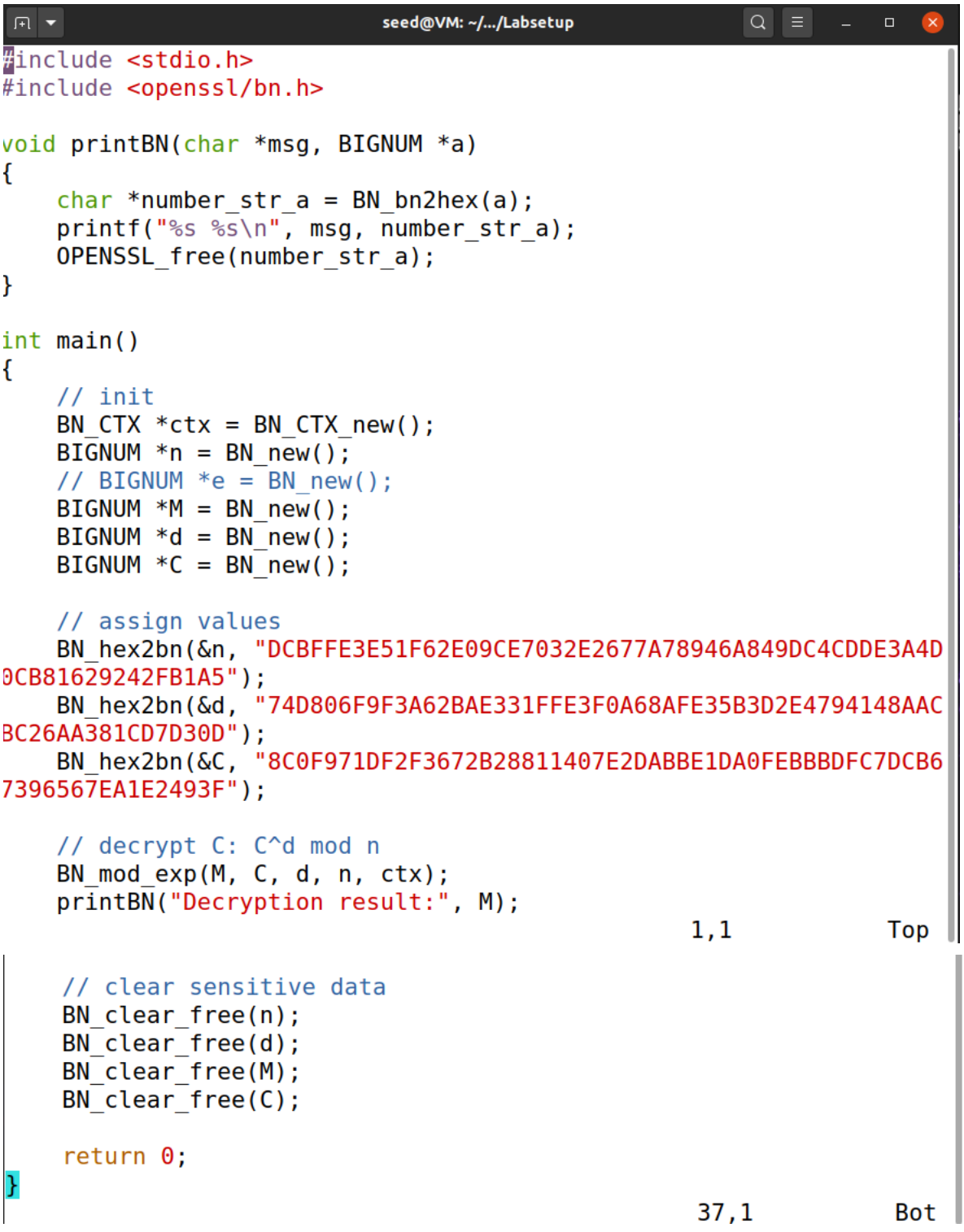
Description: The message “A top secret!” was encrypted using the public key (e,n). The message was first converted to a hexadecimal string then to a BIGNUM, and encrypted using the modular exponentiation formula. The encrypted message was printed as a hexadecimal string as seen in the screenshot above.

Task 3:

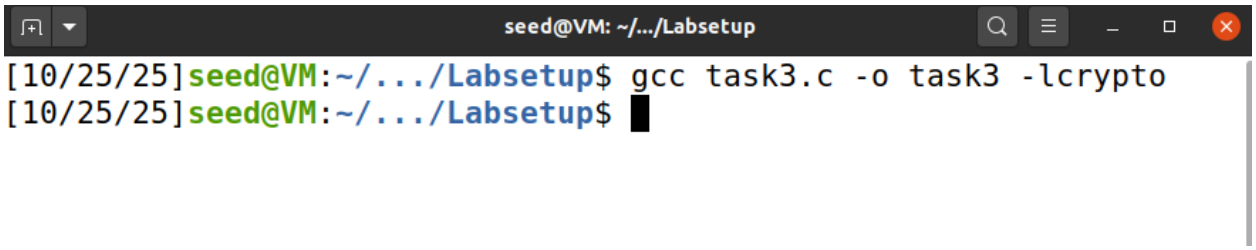
Creating task3.c decrypting file screenshot:



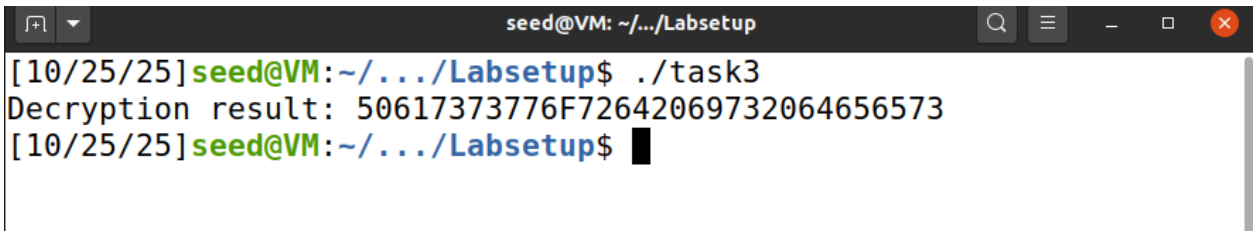
Screenshot of task3.c decrypting file:



Compiling task3.c file screenshot:

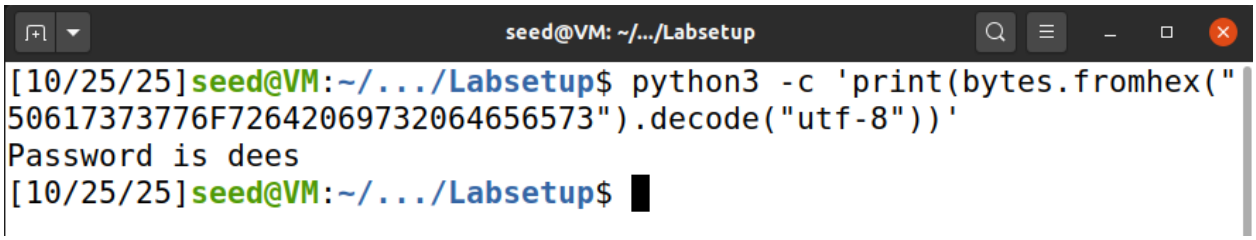


Running task3 file screenshot:



Description and explanation: First we decrypt the cipherText using the private key and modulus. The decryption process was done using modular exponentiation. The decrypted result can be seen in the screenshot above. Next we will convert this hexadecimal to ASCII using python to get the original message.

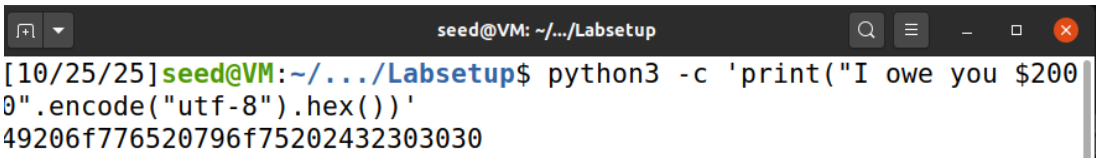
Converting Hex string to Plain ASCII:



Explanation and description: Initially, we encrypted the message “A top secret!” using RSA encryption resulting in a ciphertext. Then, the encryption was done using the public key components e and n. Afterwards, we decrypted the ciphertext using the private key and recovering the original message. The decrypted result, which was in hex was converted to the plain ASCII string, revealing the message “Password is dees”, showing that the RSA encryption and decryption process.

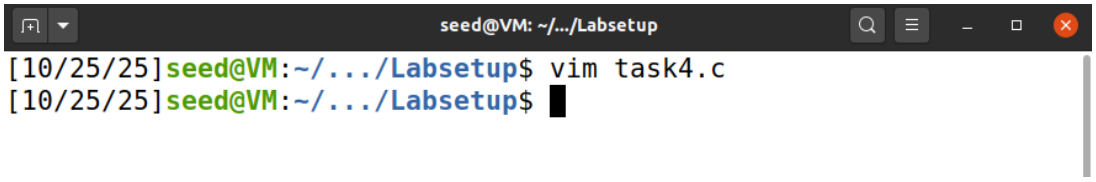
Task 4:

Getting the Hex for two string screenshot:

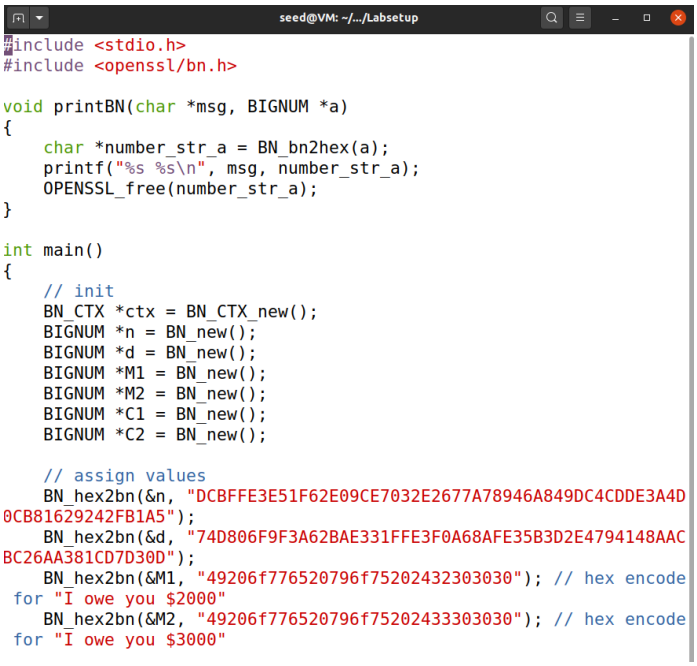


Explanation: This command gives us the hex for the strings.

Creating task4.c file screenshot:

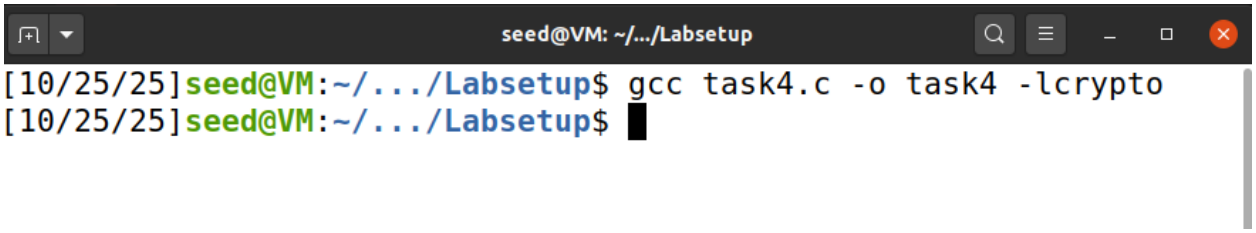


Screenshot of task4.c file:

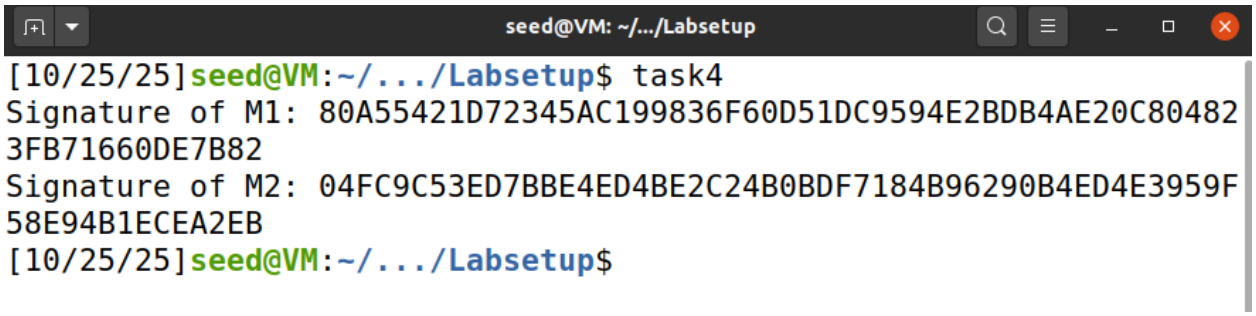


```
    // encrypt M: M^d mod n
    BN_mod_exp(C1, M1, d, n, ctx);
    BN_mod_exp(C2, M2, d, n, ctx);
    printBN("Signature of M1:", C1);
    printBN("Signature of M2:", C2);

    // clear sensitive data
    BN_clear_free(n);
    BN_clear_free(d);
    BN_clear_free(M1);
    BN_clear_free(M2);
    BN_clear_free(C1);
    BN_clear_free(C2);

    return 0;
}
                                                43,1          Bot
```

Compiling task4.c file screenshot:



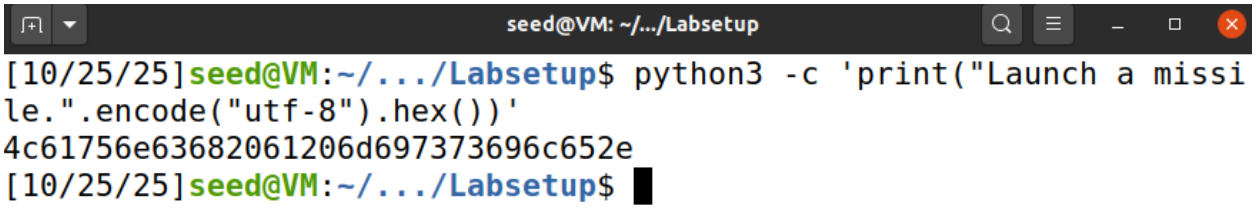
Running task4 file screenshot:



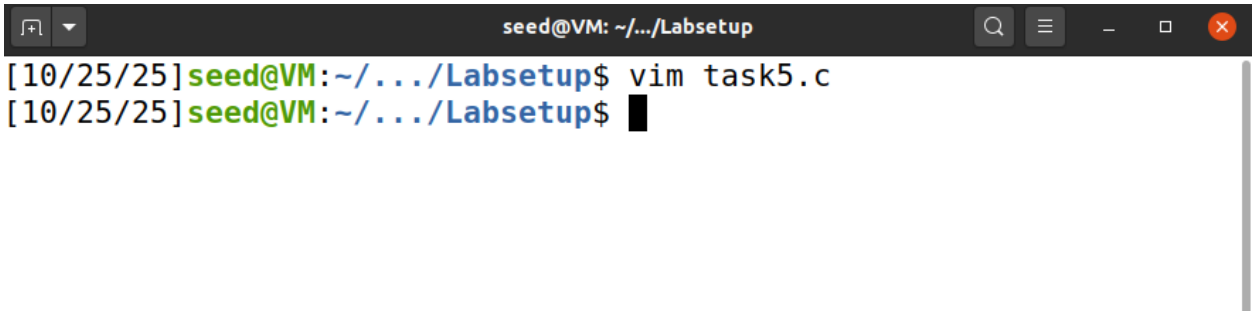
Description and Explanation: The signatures for both messages calculated using the private key can be seen in the screenshot above. As expected, they both are completely different. This shows how RSA ensures message integrity. Any small alteration in the message results in a completely different signature. By comparing the signature, we can confirm whether the message has been changed.

Task 5:

Getting the Hex for the Bob's string screenshot:



Creating task5.c file screenshot:



Screenshot of task4.c file:


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

void printBN(char *msg, BIGNUM *a)
{
    char *number_str_a = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str_a);
    OPENSSL_free(number_str_a);
}

int main()
{
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *n = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *M = BN_new();
    BIGNUM *C = BN_new();
    BIGNUM *S = BN_new();

    BN_hex2bn(&n, "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113
AA51B450F18116115");
    BN_dec2bn(&e, "65537");
    BN_hex2bn(&M, "4c61756e63682061206d697373696c652e");
    BN_hex2bn(&S, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB
026C0542CBDB6802F");

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

    if (BN_cmp(C, M) == 0)
    {
        printf("Valid Signature! \n");
```

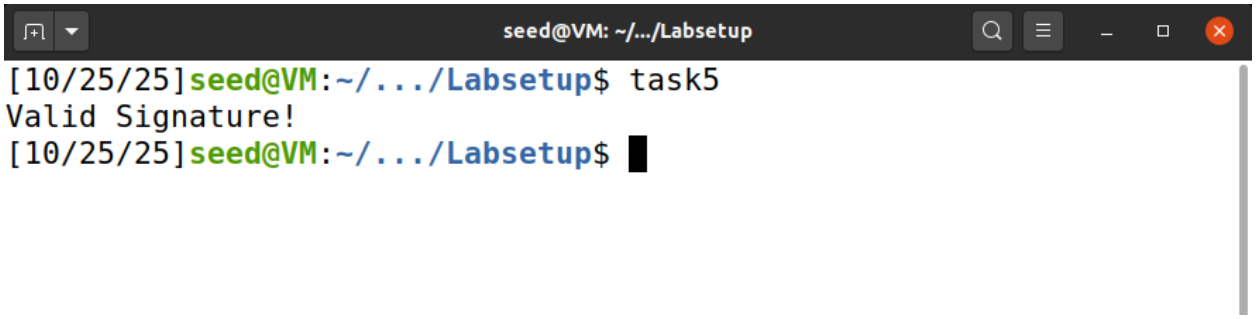
```
    {
        printf("Verification fails! \n");
    }

    BN_clear_free(n);
    BN_clear_free(e);
    BN_clear_free(M);
    BN_clear_free(C);
    BN_clear_free(S);

    return 0;
}
```

43,1 Bot

Description: This code basically verifies an RSA signature by decrypting the given signature S using Alice;s public key. The public exponent is provided as 65537 and used in the code as seen in the screenshot. The modules and the message M are also written and message M is represented as a hexadecimal string which we got from the python snippet. Next, we will run the file where the signature is decrypted using the modular exponentiation, and if the result is “Valid Signature!” it means the signature is valid else it will print “Verification fails!”.

Compiling task5.c file screenshot:

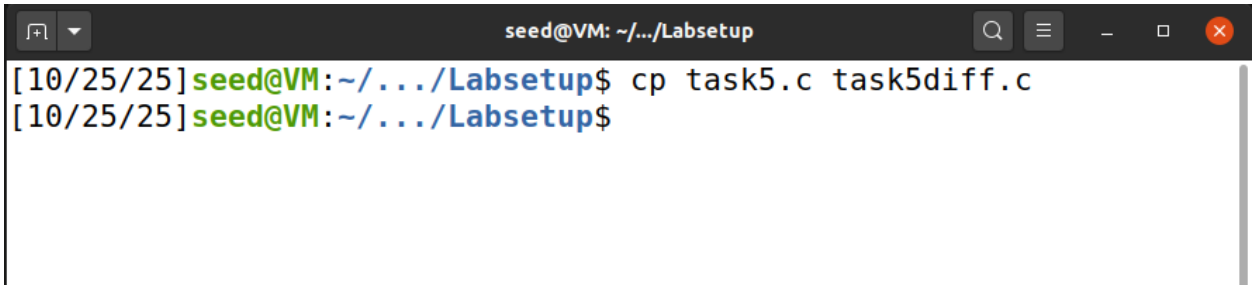
Running task5 file screenshot:



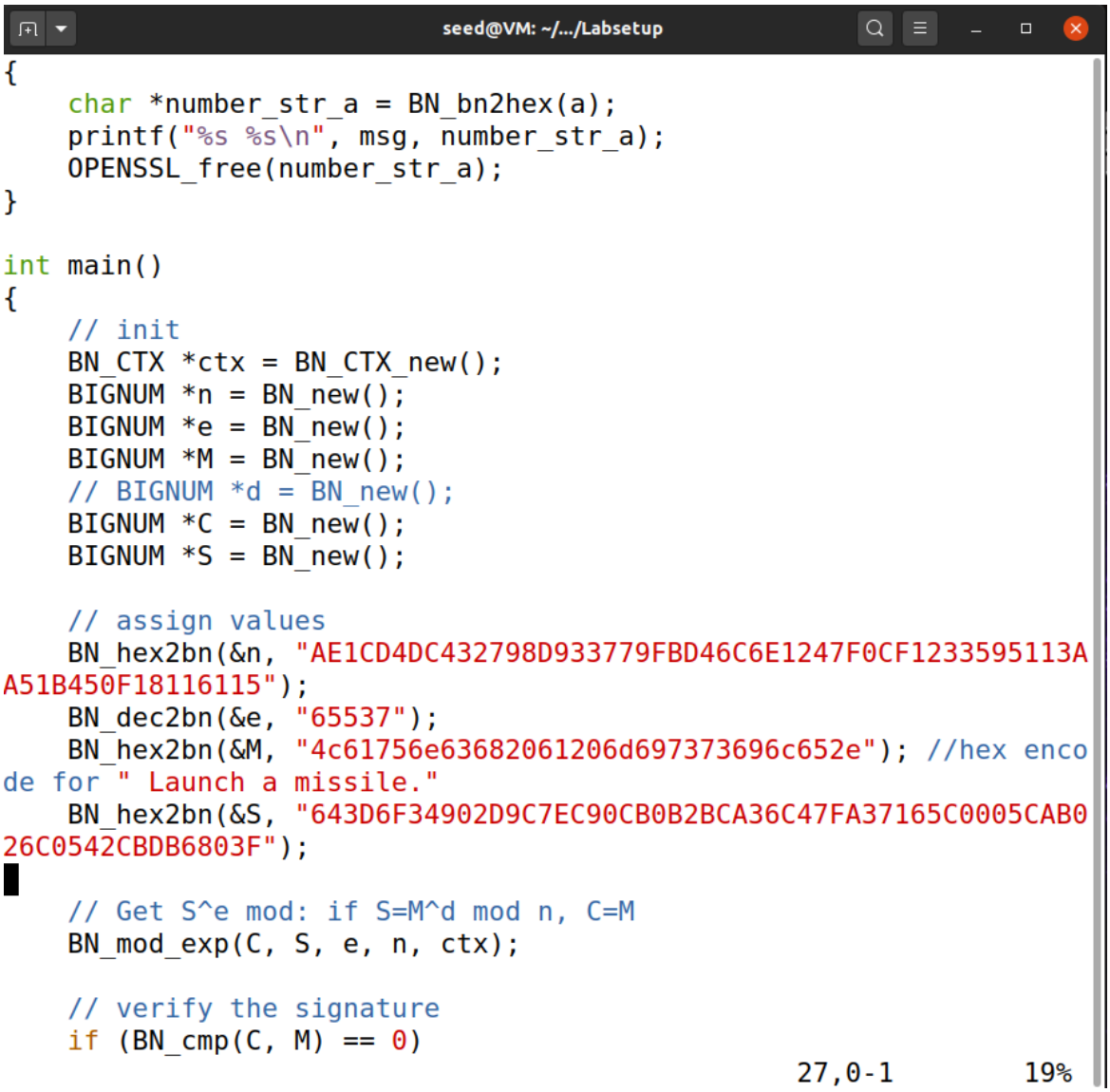
Description and Explanation: As it can be seen in the screenshot, “Valid Signature!” was outputted. This means the signature correctly matches the message, confirming that the message “Launch a missile.” is authentic and has not been changed. The decryption of the signature using Alice’s public key was successful, verifying the integrity of the message is valid.

Redoing the whole task with 3F in signature.

Coping tak5.c to task5diff.c file screenshot:



Changing the Signature in the code screenshot:



Compiling task5diff.c file screenshot:

Running the task5diff file screenshot:

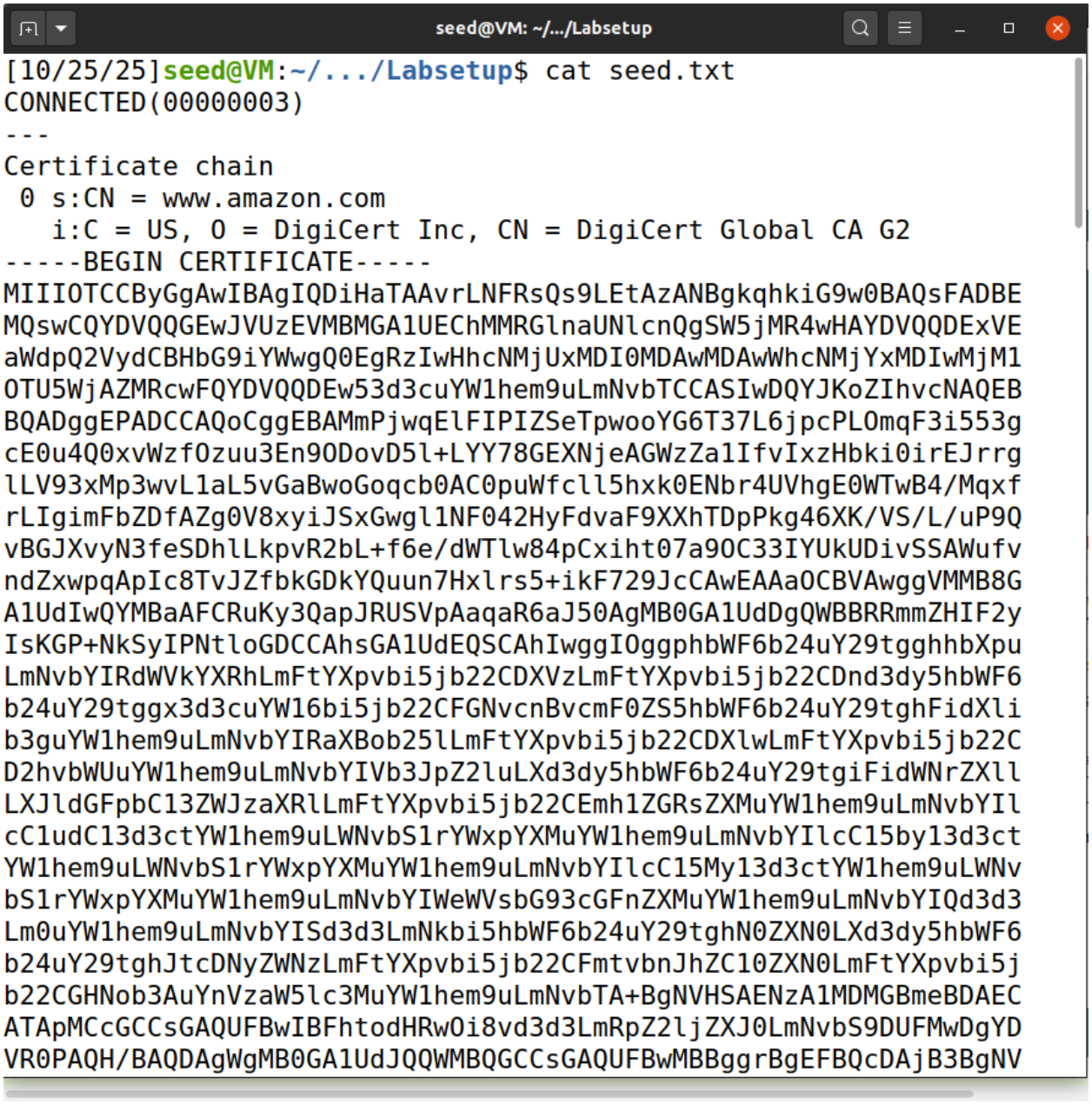
Explanation: As seen in the screenshot above, after corrupting the signature by changing the last byte from **2F** to **3F**, the verification results in "Verification fails!". This shows that even a small change to the signature causes the verification process to fail, demonstrating how RSA ensures the integrity of the message by detecting any alterations.

Task 5:

Using www.google.com for certificate.

Generating certificate screenshot:

Viewing the certificate using Cat screenshot:



```
HR8EcDBuMDWgM6Axhi9odHRwOi8vY3JsMy5kaWdpY2VydC5jb20vRGlnaUNlcnRH
bG9iYWxDQUcyLmNybDA1oDOgMYYvaHR0cDovL2NybDQuZGlnaWNlcnQuY29tL0Rp
Z2lDZXJ0R2xvYmFsQ0FHMi5jcmwwdAYIKwYBBQUHAQEEaDBmMCQGCCsGAQUFBzAB
hhhodHRwOi8vb2NzcC5kaWdpY2VydC5jb20wPgYIKwYBBQUHMAKGMmh0dHA6Ly9j
YWNlcnRzLmRpZ2ljZXJ0LmNvbS9EaWdpQ2VydEdsb2JhbENBRzIuY3J0MAwGA1Ud
EwEB/wQCMAAwggF9BgorBgEEAdZ5AgQCBIIBbQSCAWkBZwB2ANdtfRDRp/V3wsfp
X9cAv/mCyTNaZeHQswFzF8DIxWl3AAABmhRnX5gAAAQDAEcwRQIhALy+XEal28nG
YnA/PEMwvCB+/Pg2OUq8UNA6ehC3RnZiAiBmndMjaXZmJEbSSFa9Zy4Ls13PukA9
y8yIFKkgwwfK8QB1AMIxfldFGaNF7n843rKQQevHwiFaIr9/1bWtdprZDlLNAAAB
mhRnX1YAAAQDAEYwRAIgTYr5VQEFK7RpQIcI0tLWWF7GxNDVHaQHurklOfPKTvkC
IDY1Mkj6KDsaAsWfwoyPQfP8iK37ZV1QvVO/hXsJqkyVAHYAlE5Dh/rswe+B8xkk
JqgYZQHH0184AgE/cmd9VTcuGdgAAAGaFGdfZAAABAMARzBFAiEA6Y3TR6gpvN1b
qftSw2Y4VO5LWWoe/vfXqGwqLo/e8rQCIHZpk4qWcpk6O6X9VumJ7fVdLuD8I0yE
mgBYbzAN1MtzMA0GCSqGSIb3DQEBCwUAA4IBAQAmpWhQ3sGV4wbQshoVOPZiP9Zi
W3YxHU7F3iO4Qbi9JEI/DvlY5mK+RU2RYr1ZlofQ2jvkQF86nCXQjGHPdXF/eSd6
Y5M46kNDSomv9pN8aC9mMD+wDafwbAUa9zFdotJAPio3o8x4QejnbkK4nn9A4iXR
OFK5sMixgMDTBmqcA53JrUS7V+Ox7uTQ+NSIQY7D9fprm/xs1fnQHDiTe8PQ+bWT
m2A6HTYXDBxzTkHE8Ov2jjiMMLOp8bsWau3Up6ruchpPIrfvcsjux8JE6KA8cC//
Ioy/6CZ5wgM0v2NCN0p2eze+wPPSvMLnA511bKBhWIPLsh5OzmOG/1OeqjNU
-----END CERTIFICATE-----
 1 s:C = US, O = DigiCert Inc, CN = DigiCert Global CA G2
   i:C = US, O = DigiCert Inc, OU = www.digicert.com, CN = DigiCert
 Global Root G2
-----BEGIN CERTIFICATE-----
MIIEizCCA3OgAwIBAgIQDI7gyQ1qiRWIBAYe4kH5rzANBgkqhkiG9w0BAQsFADBh
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
d3cuZGlnaWNlcnQuY29tMSAwHgYDVQQDExdEaWdpQ2VydCBHbG9iYWwgUm9vdCBH
MjAeFw0xMzA4MDExMjAwMDBaFw0yODA4MDExMjAwMDBaMEQxCzAJBgNVBAYTAlVT
MRUwEwYDVQQKEwxEaWdpQ2VydCBJbmMxHjAcBgNVBAMTFURpZ2lDZXJ0IEdsb2Jh
bCBDQSBHMjCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBANNIfL7zBYZd
W9UvhU5L4IatFaxhz1uvPmoKR/uadpFgC4przc/cV35gmAvkVNlW7SHMArZagV+X
au4CLyMnuG3UsOcGAngLH1ypmTb+u6wbBfpXzYEQQGfWMItYNdSWYb7QjHqXnxr5
```

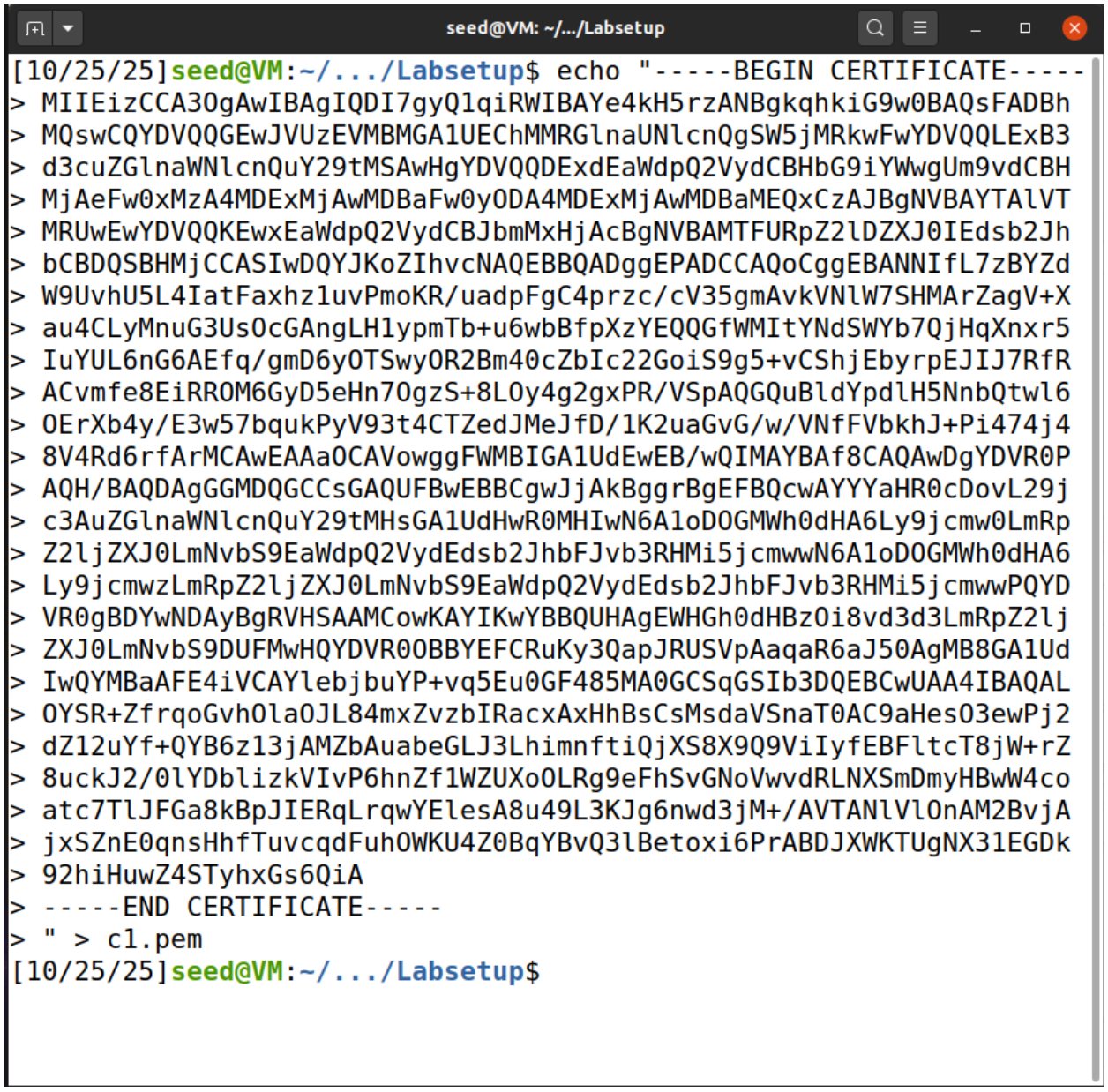
```
IuYUL6nG6AEfq/gmD6yOTSwyOR2Bm40cZbIc22GoiS9g5+vCShjEbyrpEJIJ7RfR
ACvmfe8EiRROM6GyD5eHn7OgzS+8LOy4g2gxPR/VSpAQGQuBldYpdlH5NnbQtwl6
OErXb4y/E3w57bqukPyV93t4CTZedJMeJfD/1K2uaGvG/w/VNfFVbkhJ+Pi474j4
8V4Rd6rfArMCAwEAAaOCAVowggFWMBIGA1UdEwEB/wQIMAYBAf8CAQAwDgYDVR0P
AQH/BAQDAgGGMDQGCCsGAQUFBwEBBCgwJjAkBggrBgEFBQcwAYYYaHR0cDovL29j
c3AuZGlnaWNlcnQuY29tMHsGA1UdHwR0MHIwN6A1oDOGMWh0dHA6Ly9jcmw0LmRp
Z2ljZXJ0LmNvbS9EaWdpQ2VydEdsb2JhbFJvb3RHMi5jcmwwN6A1oDOGMWh0dHA6
Ly9jcmwzLmRpZ2ljZXJ0LmNvbS9EaWdpQ2VydEdsb2JhbFJvb3RHMi5jcmwwPQYD
VR0gBDYwNDAyBgRVHSAAMCowKAYIKwYBBQUHAgEWHGh0dHBzOi8vd3d3LmRpZ2lj
ZXJ0LmNvbS9DUFMwHQYDVR0OBBYEFCRuKy3QapJRUSVpAaqaR6aJ50AgMB8GA1Ud
IwQYMBaAFE4iVCAYlebjbuYP+vq5Eu0GF485MA0GCSqGSIb3DQEBCwUAA4IBAQAL
OYSR+ZfrqoGvhOlaOJL84mxZvzbIRacxAxHhBsCsMsdaVSnaT0AC9aHesO3ewPj2
dZ12uYf+QYB6z13jAMZbAuabeGLJ3LhimnftiQjXS8X9Q9ViIyfEBFltcT8jW+rZ
8uckJ2/0lYDblizkVIvP6hnZf1WZUXoOLRg9eFhSvGNoVwvdRLNXSmDmyHBwW4co
atc7TlJFGa8kBpJIERqLrqwYElesA8u49L3KJg6nwd3jM+/AVTANlVlOnAM2BvjA
jxSZnE0qnsHhfTuvcqdFuhOWKU4Z0BqYBvQ3lBetoxi6PrABDJXWKTUgNX31EGDk
92hiHuwZ4STyhxGs6QiA
-----END CERTIFICATE-----
 2 s:C = US, O = DigiCert Inc, OU = www.digicert.com, CN = DigiCert
 Global Root G2
   i:C = US, O = "VeriSign, Inc.", OU = VeriSign Trust Network, OU
= "(c) 2006 VeriSign, Inc. - For authorized use only", CN = VeriSig
n Class 3 Public Primary Certification Authority - G5
-----BEGIN CERTIFICATE-----
MIIE3zCCA8egAwIBAgIQSysBFc3lx0gbPN3+3hEWnjANBgkqhkiG9w0BAQsFADCB
yjELMAkGA1UEBhMCVVMxFzAVBgNVBAoTDlZlcmlTaWduLCBJbmMuMR8wHQYDVQQL
ExZWZXJpU2lnbiBUcnVzdCBOZXR3b3JrMTowOAYDVQQLEzEoYykgMjAwNiBWZXJp
U2lnbiwgSW5jLiAtIEZvciBhdXRob3JpemVkIHVzZSBvbmx5MUUwQwYDVQQDEzxW
ZXJpU2lnbiBDbGFzcyAzIFB1YmxpYyBQcmltYXJ5IENlcnRpZmljYXRpb24gQXV0
aG9yaXR5IC0gRzUwHhcNMTgwNDAzMDAwMDAwWhcNMjgwNDAyMjM1OTU5WjBhMQsw
CQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3d3cu
ZGlnaWNlcnQuY29tMSAwHgYDVQQDExdEaWdpQ2VydCBHbG9iYWwgUm9vdCBHMjCC
```

```
ASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALs3zTTce2vJsmiQrUp1/0a6
IQoIjfUZVMn7iNvzrvI6iZE8euarBhprz6wt6F4JJES6Ypp+1qOofuBUdSAFrFC3
nGMabDDc2h8Zsdce3v3X4MuUgzeu7B9DTt17LNK9LqUv5Km4rTrUmaS2JembawBg
kmD/TyFJGPdnkKthBpyP8rrptOmSMmu181foXRvNjB2rlQSVSfM1LZbjSW3dd+P7
SUu0rFUHqY+Vs7Qju0xtRfD2qbKVMLT9TFWMJ0pXFHyCnc1zktMWSgYMjFDRjx4J
vheh5iHK/YPlELyDpQrEZyj2cxQUPUZ2w4cUiSE0Ta8PRQymSaG6u5zFsTODKYUC
AwEAAaOCAScwggEjMB0GA1UdDgQWBBROIlQgGJXm427mD/r6uRLtBhePOTAPBgNV
HRMBAf8EBTADAQH/MF8GA1UdIARYMFYwVAYEVR0gADBMMCMGCCsGAQUFBwIBFhdo
dHRwczovL2Quc3ltY2IuY29tL2NwczAlBggrBgEFBQcCAjAZDBdodHRwczovL2Qu
c3ltY2IuY29tL3JwYTAvBgNVHR8EKDAmMCSgIqAghh5odHRwOi8vcy5zeW1jYi5j
b20vcGNhMy1nNS5jcmwwDgYDVR0PAQH/BAQDAgGGMC4GCCsGAQUFBwEBBCIwIDAe
BggrBgEFBQcwAYYSaHR0cDovL3Muc3ltY2QuY29tMB8GA1UdIwQYMBaAFH/TZafC
3ey78DAJ80M5+gKvMzEzMA0GCSqGSIb3DQEBCwUAA4IBAQCOStidCZ2ebdzF1Jkd
ijPBIAyeed+Chn4dWDm0ptwghHtZ7Q1DRFGLlOQQgXmcFBZtRCcGRgC2ZZDTjFYE
fs1mnDD3J+fKxm+cM6rR5JTlLxP4j3v6k91tk1l3TtZhucfF5qs+SNmpp/QQtneN
JoyWHawD8Knb1BZOUA9sxrFeFdJz4uAz+ld6Fdm1ZtU5gMQXUOKHL6ZGry8+Qnax
9A35U6m4eroyFSlmrbteDOW5jkZtDuu3E33vx7Di7yLZV/Pfto/28wAcHxnhukPk
CmMNKS5wFI5j7dStFFn6QWYpQX5SGuu7ywTgOE/nbsDXwqcUsO+elydJvwFEwtwX
b0ES
-----END CERTIFICATE-----
---
Server certificate
subject=CN = www.amazon.com

issuer=C = US, O = DigiCert Inc, CN = DigiCert Global CA G2

---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: RSA-PSS
Server Temp Key: X25519, 253 bits
---
```

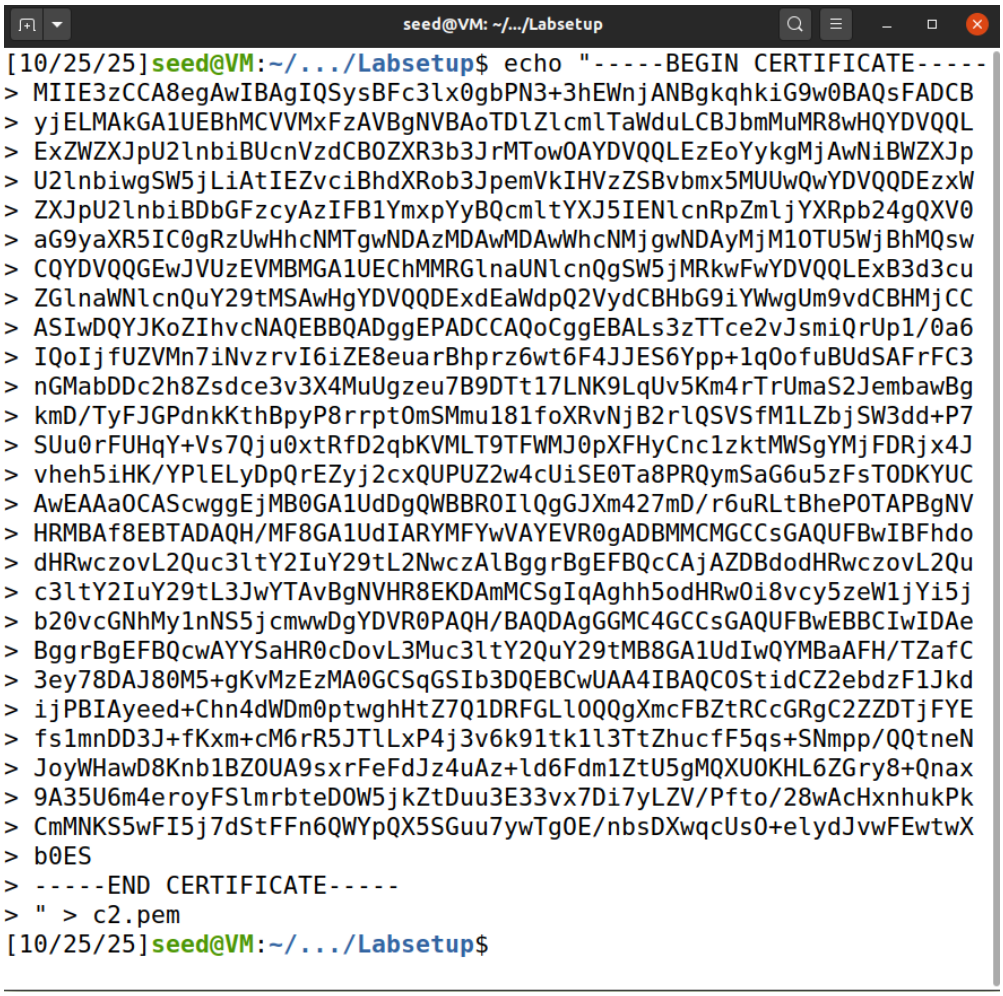
```
SSL handshake has read 5081 bytes and written 380 bytes
Verification error: unable to get local issuer certificate
---
New, TLSv1.3, Cipher is TLS_AES_128_GCM_SHA256
Server public key is 2048 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 20 (unable to get local issuer certificate)
---
---
Post-Handshake New Session Ticket arrived:
SSL-Session:
    Protocol  : TLSv1.3
    Cipher    : TLS_AES_128_GCM_SHA256
    Session-ID: F0453C9C9572C64327D8C63D52FCF93DC43A2815A6690AB4533
3655FC33A3C42
    Session-ID-ctx:
    Resumption PSK: 868FF979D95ABA878A2E7864B7A796F5EEA3B49E9473E57
6D3C25514ECEC71E3
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    TLS session ticket lifetime hint: 64864 (seconds)
    TLS session ticket:
    0000 - 01 31 37 36 31 34 32 32-32 33 34 30 30 30 00 00   .17614
22234000..
    0010 - 00 a5 26 ff 29 78 81 1a-be bf 5e 8f 32 21 30 02   ..&.)x
....^.2!0.
    0020 - 51 bd a9 2c fd be a1 9f-63 37 e0 cd 8f 68 80 62   Q..,..
..c7...h.b
    0030 - 15 77 69 2d 0d 21 dc 8d-8c 40 29 59 55 07 4d 9e   .wi-.!
...@)YU.M.
    0040 - db 9e 7a 03 3a ea 2d f8-00 43 59 75 d0 f0 47 f8   ..z.:.
-..CYu..G.
    0050 - aa a2 f1 d3 d4 e4 8f 95-12 66 bf 12 31 50 e3 b4   ......
...f..1P..
    0060 - bd 30 42 3b 10 14 e8 7b-c4 53 e6 6a 39 7f eb cb   .0B;..
.{.S.j9...
    0070 - 4b 1a 8f c8 d8 74 30 fb-65 19 34 c7 92 6f 7c 18   K....t
0.e.4..o|.
    0080 - 77 76 ce 74 a2 bb 03 0c-ac 09                     wv.t..
....

    Start Time: 1761443768
    Timeout   : 7200 (sec)
    Verify return code: 20 (unable to get local issuer certificate)
    Extended master secret: no
    Max Early Data: 0
---
read R BLOCK
closed
[10/25/25]seed@VM:~/.../Labsetup$
```

Saving the first certificate as c0.pem:

Saving the second certificate as c1.pem:

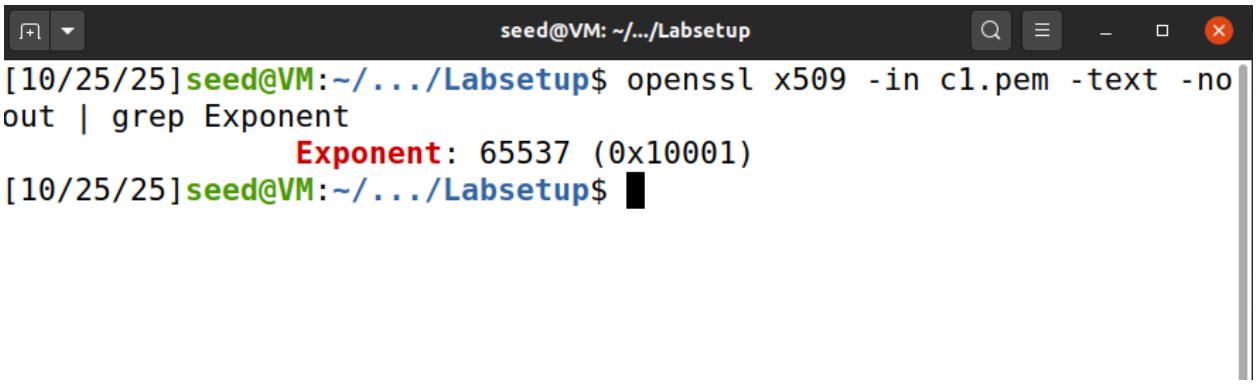


Saving the third certificate as c2.pem screenshot:

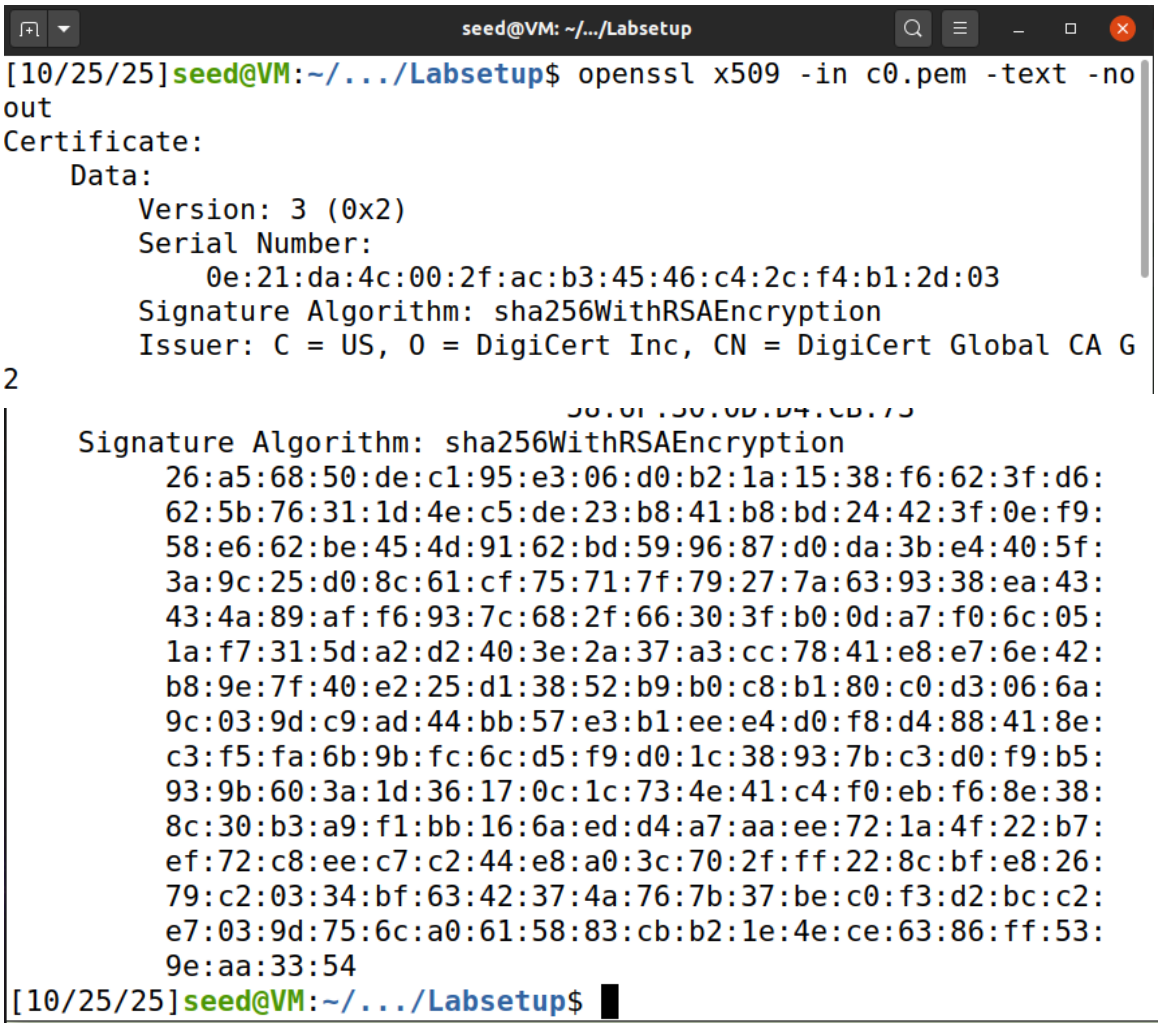


Finding Modulus from the c1.pem file screenshot:

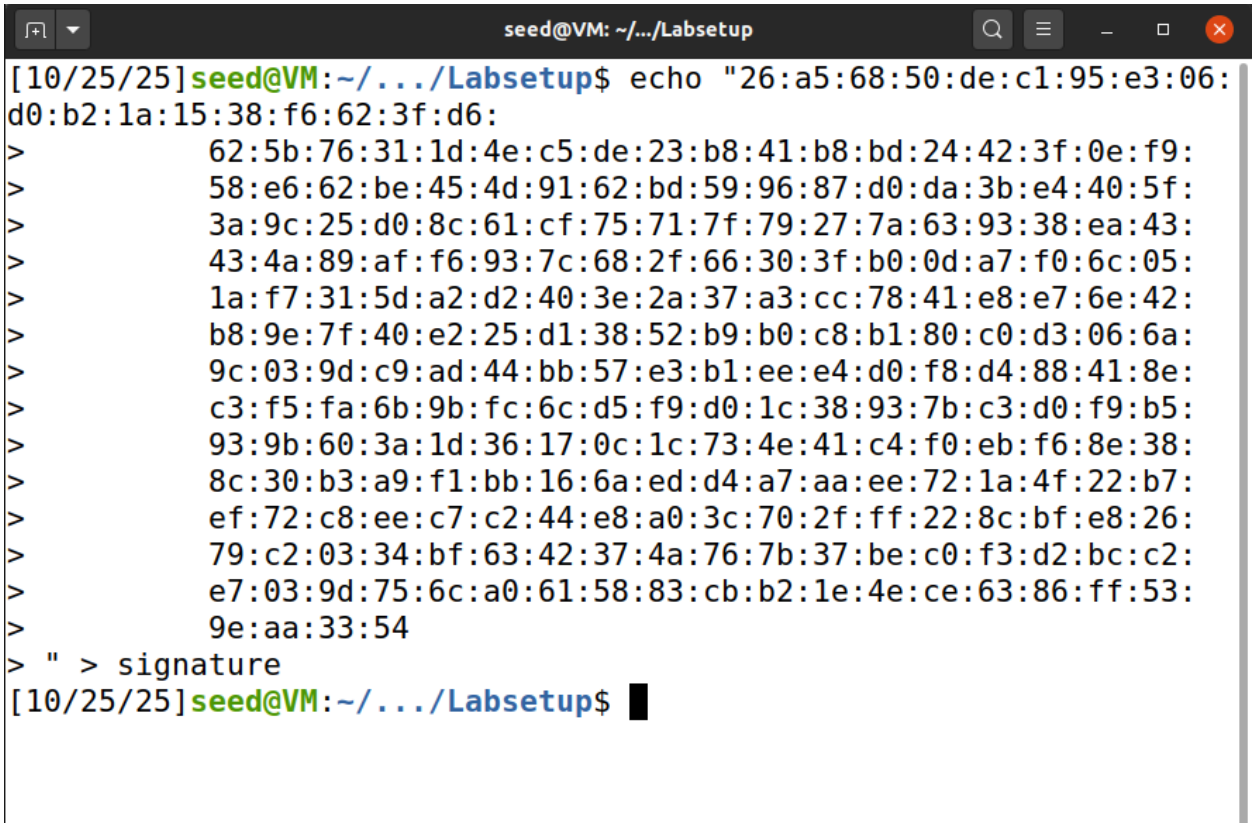
Getting Expononet from the c1.pem file screenshot:



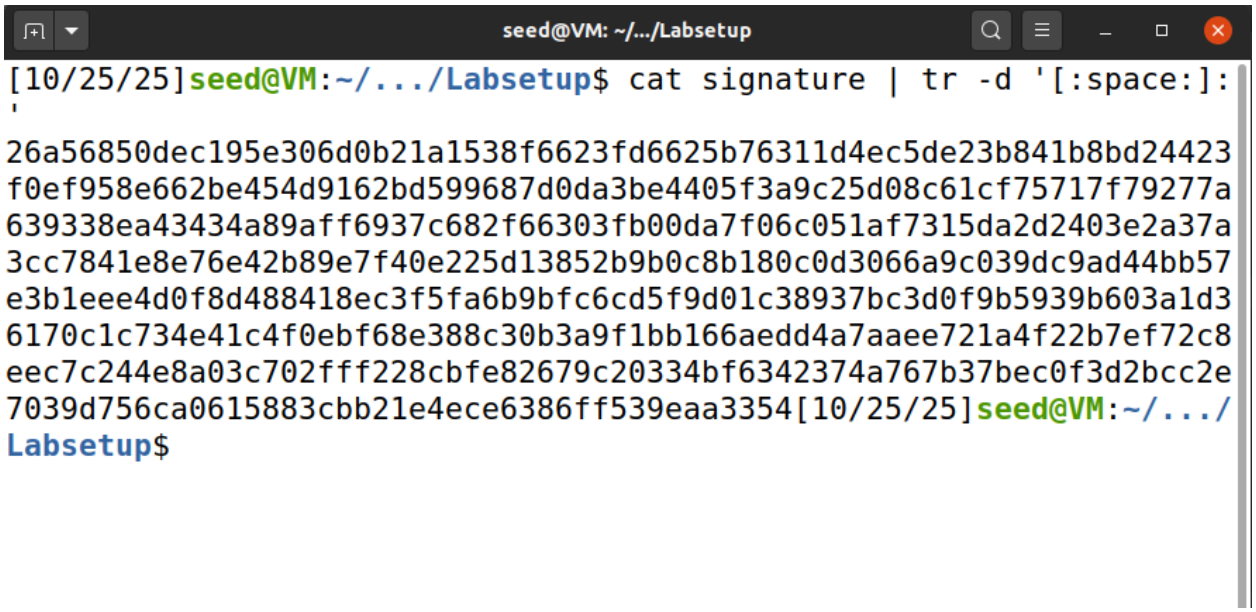
Extracting the signature from the server's certificate:



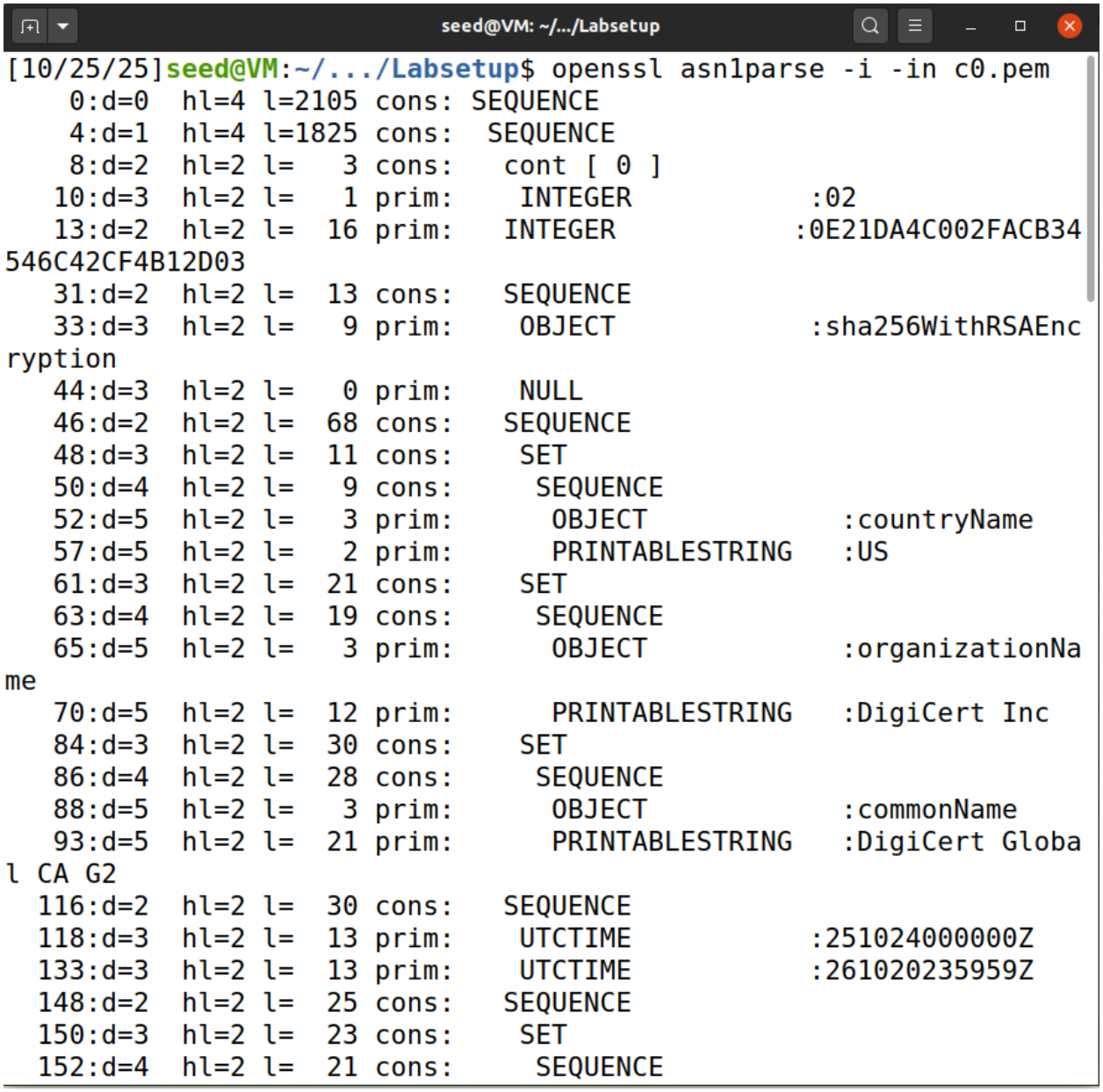
Saving it in a signature file screenshot:



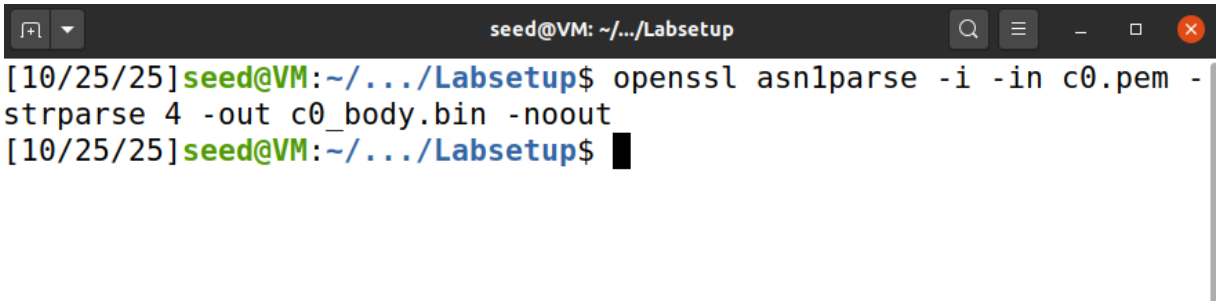
Removing colons and spaces from the signature:



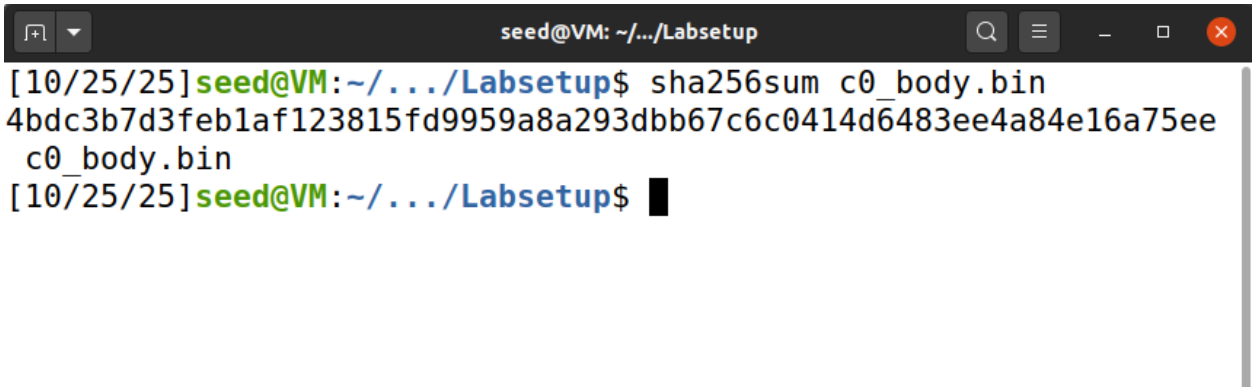
Extracting the body of the server's certificate:



Extracting the body of the certificate screenshot:



Calculating the hash for the body screenshot:



Padding the hash so it becomes 256 byte:

```
prefix = "0001"
hash = "4bdc3b7d3feb1af123815fd9959a8a293dbb67c6c0414d6483ee4a84e
16a75ee"
A = "3031300D060960864801650304020105000420"
total_len = 256

pad_len = total_len - 1 - (len(A) + len(prefix) + len(hash)) // 2

padded_message = prefix + 'FF' * pad_len + '00' + A + hash

print(padded_message)

~
```

Description and explanation: The python code is used to pad the hash of the certificate's body so it can be correctly verified using RSA encryption. RSA requires the data to be a specific size, so we add extra padding to make it the right length(256 bytes). Without padding, the verification would fail because the data wouldn't match the format expected by RSA.

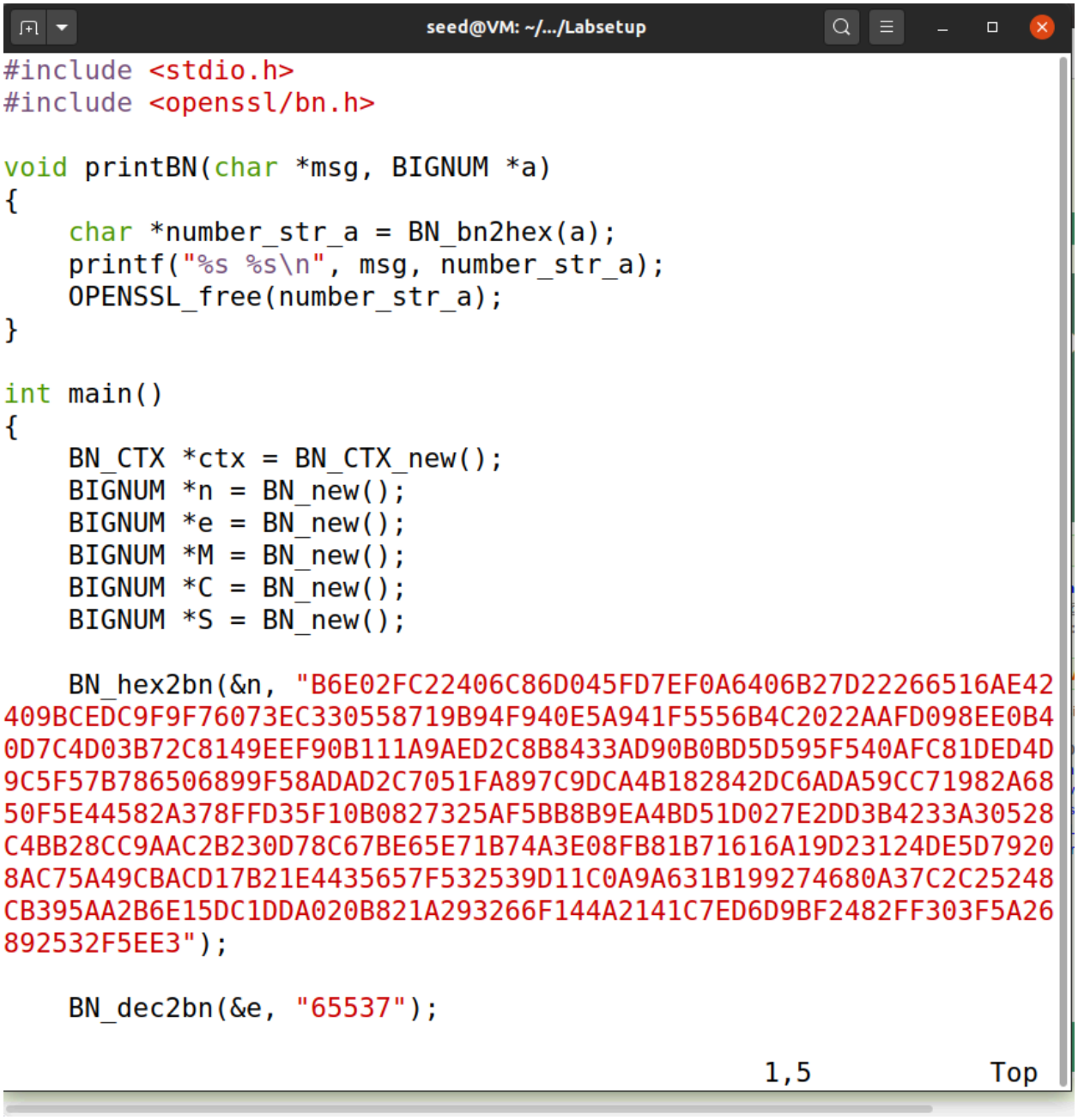
Padded hash output:

```
seed@VM: ~/.../Labsetup
[10/25/25]seed@VM:~/.../Labsetup$ python3 pad_hash.py
0001FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFF003031300D0609608648016503040201050004204bdc3b7
d3feb1af123815fd9959a8a293dbb67c6c0414d6483ee4a84e16a75ee
[10/25/25]seed@VM:~/.../Labsetup$
```

Creating file to verify:

```
seed@VM: ~/.../Labsetup
[10/25/25]seed@VM:~/.../Labsetup$ vim verify.c
```

Code snippet for the verify.c file:

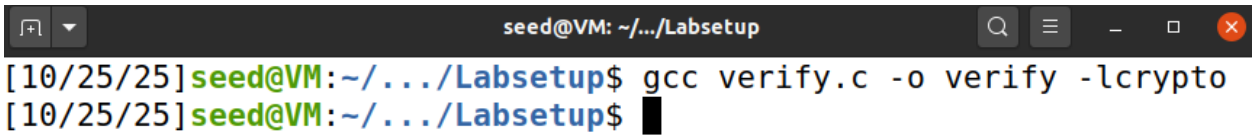


```
    BN_hex2bn(&M, "0001FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF003031300D0609608648016503040201050
004200640f8d13c0789ff0ed5437cf4bc9f2827d52146dddff38aefc2c17747d45
f28");

    // Signature (S) - The signature obtained from the certificate
    BN_hex2bn(&S, "00f3bbf23fe1d30fc06e10ccc1476668101659dcff1a97b
5a34ba8e348cd73f39c14261d08b8f35c4a8004788d93934e49e5c0e2c15e70d7b
d5eab250657badde9c474af54993692fbb20cedd10b4bae75df35017214b1de8f9
e3b760fa5ddff2a54028324c84fbc7ae604484164e07967ae95ed37b3924c65586
50934689ac320db255dd9942fd13a01088861a448a51311763e2cb46e8290f2697
d26ae59ad7d911799ea14d04797fcf4beb1e74bacec6b969661fa12654521b85ff
443b4d9003709c53b6c4d622d630798a714eb2b619a0b2f3515394e2931bc5efb2
45bfb9f5ff2f062eba6b98aa41e900dfe0f03c4bd44e5fd4738307b729320ceaa7
8a5");

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

    if (BN_cmp(C, M) == 0)
    {
        printf("Valid Signature! \n");
    }
    else
    {
        printf("Verification fails! \n");
                                                 31,5           66%
```

```
    }

    BN_clear_free(n);
    BN_clear_free(e);
    BN_clear_free(M);
    BN_clear_free(C);
    BN_clear_free(S);

    return 0;
}

                                                 47,1           Bot
```

Description and explanation: The C code basically verifies the signature of the server certificate using RSA encryption. It begins by using the modulus n and exponent e, which are provided from the certificate's public key. The hash of the certificate's body is stored in M, and the signature from the certificate is stored in S. The code calculator C, which is the expected result if the signature is valid. If the computed value of C matches the original certificate hash M, it confirms the signature is valid meaning the certificate was not changed. In the screenshot below we can see that "valid signature" is printed hence the signature matches.

Compiling the verify.c file:



Running the verify file: