Istanbul Technical University

Informatics Institute

Cybersecurity Engineering and Cryptography

RSA Public-Key Encryption and Signature Lab

Detailed Report

Name: Halis Taha Şahin

ID: 707201006

E-Mail: sahinh20@itu.edu.tr

Semester: Fall 2020/2021

Fachsemester: 01

Date: December 8, 2020

Contents

1 Task 1: Deriving the Private Key **1**

2 Task 2: Encrypting a Message **2**

3 Task 3: Decrypting a Message **4**

4 Task 4: Signing a Message **5**

5 Task 5: Verifying a Signature **6**

6 Task 6: Manually Verifying an X.509 Certificate **7**

6.1 Step 1: Download a certificate from a real web server. 7

6.2 Step 2: Extract the public key (e, n) from the issuer's certificate. . 8

6.3 Step 3: Extract the signature from the server's certificate. 9

6.4 Step 4: Extract the body of the server's certificate. 10

6.5 Step 5: Verify the signature. 11

1 Task 1: Deriving the Private Key

Given p, q and e. We must calculate d value from given values. To find the d value, we need to use the following equation.

$$d * e = 1 \; mod \; \phi(n) \tag{1}$$

$$d * e = 1 \; mod \; (p-1)(q-1) \tag{2}$$

p = F7E75FDC469067FFDC4E847C51F452DF

q = E85CED54AF57E53E092113E62F436F4F

e = 0D88C3

Firstly we must initialize values in c file.

```
BN_CTX *ctx = BN_CTX_new();

BIGNUM *p = BN_new();   // p value
BIGNUM *q = BN_new();   // q value
BIGNUM *e = BN_new();   // e value
BIGNUM *d = BN_new();   // d value
BIGNUM *p_minus = BN_new(); // (p-1) value
BIGNUM *q_minus = BN_new(); // (q-1) value
BIGNUM *f_of_fi = BN_new(); // fi value
BIGNUM *one = BN_new(); // define 1 for minus

// initialize p, q, e from PDF file
BN_hex2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF");
BN_hex2bn(&q, "E85CED54AF57E53E092113E62F436F4F");
BN_hex2bn(&e, "0D88C3");
BN_dec2bn(&one, "1");
```

After creating the values, we can continue our operations. First we will get the value of (p-1) and (q-1), and then we will find the inverse of the number e.

```
// Calculate (p-1) value
BN_sub(p_minus, p , one);

// Calculate (q-1) value
BN_sub(q_minus, q , one);

// fi = (p-1)*(q-1)
BN_mul(f_of_fi, p_minus, q_minus, ctx);

// d = e.modInverse(fi)
BN_mod_inverse(d, e, f_of_fi, ctx);

printBN("d = ", d);
```

We calculated d value. If we run the program, output must be like this.

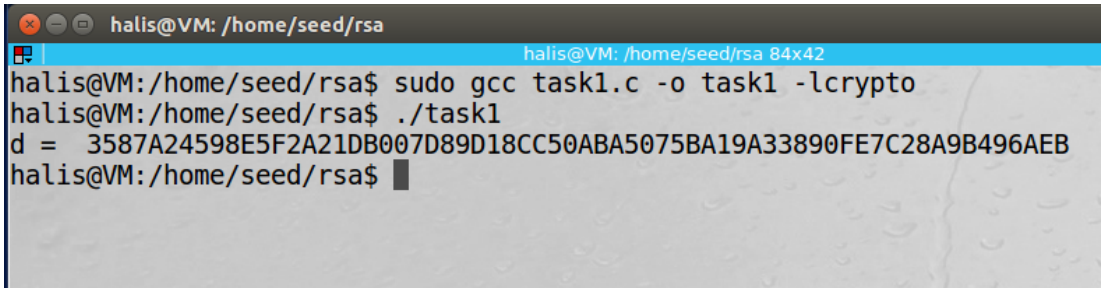

Figure 1: Task1 Output

d = 3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB

2 Task 2: Encrypting a Message

We have a public key and a message that we need to encrypt. First we will encrypt the message using the value of e and n. Then we will check whether we have done the correct encryption by decrypting with the d value. Firstly, we must calculate the hex value of the message. ("A top secret!")

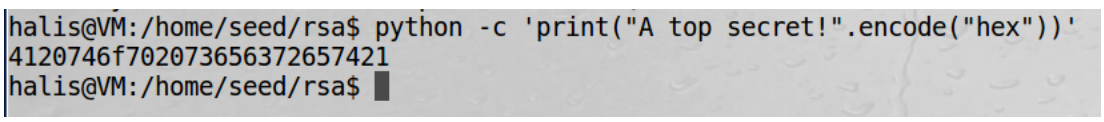

Figure 2: Hex value of the message.

Hex value: 4120746f702073656372657421

After that we must initialize the variables.

```
// initialize m, e, n and d values from PDF file
BN_hex2bn(&m, "4120746f702073656372657421");     // "A top secret!"
BN_hex2bn(&e, "010001");
BN_hex2bn(&n,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
BN_hex2bn(&d,"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
```

We use the public key to encrypt our message. We will then use the private key to decrypt it. Equation 3 will give us the cipher text and equation 4 will give us plain text.

$$c = m^e \ mod \ (n) \tag{3}$$

$$m = c^d \ mod \ (n) \tag{4}$$

```
// Encryption m^e mod n , , Compute c = m^e mod n
BN_mod_exp(c, m, e, n, ctx);
printBN("Encrypted Message = ", c);

// Decryption c^d mod n , Compute m_dec = c^d mod n
BN_mod_exp(m_dec, c, d, n, ctx);
printBN("Decrypted Message = ", m_dec);

printBN("Original  Message = ", m);
```

If we run our program, the output will be as follows.

```
halis@VM:/home/seed/rsa$ sudo gcc task2.c -o task2 -lcrypto
halis@VM:/home/seed/rsa$ ./task2
Encrypted Message =  6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC
Decrypted Message =  4120746F702073656372657421
Original  Message =  4120746F702073656372657421
halis@VM:/home/seed/rsa$
```

Figure 3: Output of task2.c

3 Task 3: Decrypting a Message

We have a cipher text. First, we have to decrypt it with the private key and then decode it.

C = 8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F

```
// Initialize the c, n and d values from PDF
BN_hex2bn(&c,"8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");
BN_hex2bn(&n,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
BN_hex2bn(&d,"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

// Decryption c^d mod n , Compute m_dec = c^d mod n
BN_mod_exp(m_dec, c, d, n, ctx);
printBN("Decrypted Message = ", m_dec);
```

If we run our program, the output will be as follows.

```
halis@VM:/home/seed/rsa$ sudo gcc task3.c -o task3 -lcrypto
halis@VM:/home/seed/rsa$ ./task3
Decrypted Message =  50617373776F72642069732064656573
halis@VM:/home/seed/rsa$
```

Figure 4: Output of task3.c

We must decode the hex value to get string.

```
halis@VM:/home/seed/rsa$ python -c 'print("50617373776F72642069732064656573".decode("hex"))'
Password is dees
halis@VM:/home/seed/rsa$
```

Figure 5: Secret message.

Secret message: "Password is dees"

4 Task 4: Signing a Message

In this task we need to sign our message. First, we will convert our message to hexadecimal. Then we will sign our message with the private key. I will repeat the processes in two different messages and try to observe the results.

Encode the messages.

```
halis@VM:/home/seed/rsa$ python -c 'print("I owe you $2000.".encode("hex"))'
49206f776520796f752024323030302e
halis@VM:/home/seed/rsa$ python -c 'print("I owe you $3000.".encode("hex"))'
49206f776520796f752024333030302e
halis@VM:/home/seed/rsa$
```

Figure 6: Encoded messages.

As can be seen in Figure 6, there is only one character difference between hex values. We will sign both messages and examine the differences between signatures.

```
// initialize e, m, n from PDF file (Task 2)
BN_hex2bn(&e, "010001");
BN_hex2bn(&m, "49206f776520796f752024323030302e");
BN_hex2bn(&m2, "49206f776520796f752024333030302e");
BN_hex2bn(&n,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");

// Encryption m^e mod n , Compute c = m^e mod n
BN_mod_exp(c, m, e, n, ctx);
printBN("Signed Message (I owe you $2000.) = ", c);

// Encryption m^e mod n , Compute c = m^e mod n
BN_mod_exp(c, m2, e, n, ctx);
printBN("Signed Message (I owe you $3000.) = ", c);
```

If we run our program, the output will be as follows.

```
halis@VM:/home/seed/rsa$ sudo gcc task4.c -o task4 -lcrypto
halis@VM:/home/seed/rsa$ ./task4
Signed Message (I owe you $2000.) =  3A759CBF53901AC41373EEC603955A8E6AF8D3BCD5E9F6DD62C873CBB675051E
Signed Message (I owe you $3000.) =  D06908047527906C724937169FA68CE0AC442FEB99D1880438D331A88F44B074
halis@VM:/home/seed/rsa$
```

Figure 7: Output of task4.c

A single character difference between the messages made the signatures completely different from each other. Therefore, we can say that the confidentiality is very good in the signing process.

5 Task 5: Verifying a Signature

First, let's get the hex value of our message.

```
halis@VM:/home/seed/rsa$ python -c 'print("Launch a missile.".encode("hex"))'
4c61756e63682061206d697373696c652e
halis@VM:/home/seed/rsa$
```

Figure 8: Hex value of a message.

İnitialize the values and decrypt signs with public key.

```
// Initialize m, e and n values from PDF file
BN_hex2bn(&m, "4c61756e63682061206d697373696c652e");
BN_hex2bn(&e, "010001");
BN_hex2bn(&n,"AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");

// Initialize true Sign
BN_hex2bn(&s,"643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");

// Print original message.
printBN("Message = ", m);

// Decryption s^e mod n , Compute m_dec = s^e mod n
BN_mod_exp(m_dec, c, e, n, ctx);
printBN("Decrypted Sign = ", m_dec);

// Corrupted Sign
BN_hex2bn(&s,"643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F");

// Decryption s^e mod n , Compute m_dec2 = s^e mod n
BN_mod_exp(m_dec2, c, e, n, ctx);
printBN("Decrypted Sign (Corrupted)= ", m_dec2);
```

If we run our program, the output will be as follows.

```
halis@VM:/home/seed/rsa$ sudo gcc task5.c -o task5 -lcrypto
halis@VM:/home/seed/rsa$ ./task5
Message =  4C61756E63682061206D697373696C652E
Decrypted Sign =  4C61756E63682061206D697373696C652E
Decrypted Sign (Corrupted)=  91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294
halis@VM:/home/seed/rsa$
```

Figure 9: Output of task5.c

After decrypting our original signature, we got our message correctly. However, we could not get the correct message in decrypting only 1 character corrupted signature.

6 Task 6: Manually Verifying an X.509 Certificate

In this task, we will manually verify an X.509 certificate using our program.

6.1 Step 1: Download a certificate from a real web server.

First I downloaded the certificates from the web server. Then I saved it in "c0.pem" and "c1.pem" files.

```
halis@VM:/home/seed/rsa/task6$ openssl s_client -connect www.example.org:443 -showcerts
CONNECTED(00000003)
depth=2 C = US, O = DigiCert Inc, OU = www.digicert.com, CN = DigiCert Global Root CA
verify return:1
depth=1 C = US, O = DigiCert Inc, CN = DigiCert TLS RSA SHA256 2020 CA1
verify return:1
depth=0 C = US, ST = California, L = Los Angeles, O = Internet Corporation for Assigned Names and Numbers, CN = ww
verify return:1
---
Certificate chain
 0 s:/C=US/ST=California/L=Los Angeles/O=Internet Corporation for Assigned Names and Numbers/CN=www.example.org
   i:/C=US/O=DigiCert Inc/CN=DigiCert TLS RSA SHA256 2020 CA1
-----BEGIN CERTIFICATE-----
MIIG1TCCBb2gAwIBAgIQD74IsIVNBXOKsMzhya/uyTANBgkqhkiG9w0BAQsFADBP
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMSkwJwYDVQQDEyBE
aWdpQ2VydCBUTFMgUlNBIFNIQTI1NiAyMDIwIENBMTAeFw0yMDExMjQwMDAwMDBa
Fw0yMTEyMjUyMzU5NTlaMIGQMQswCQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWZv
cm5pYTEUMBIGA1UEBxMLTG9zIEFuZ2VsZXMxPDA6BgNVBAoTM0ludGVybmV0IENv
cnBvcmF0aW9uIGZvciBBc3NpZ25lZCBOYW1lcyBhbmQgTnVtYmVyczEYMBYGA1UE
AxMPd3d3LmV4YW1wbGUub3JnMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKC
AQEAuvzuzMoKCP8Okx2zvgucA5YinrFPEK5RQP1TX7PEYUAoBO6i5hIAsIKFmFxt
W2sghERilU5rdnxQcF3fEx3sY4OtY6VSBPLPhLrbKozHLrQ8ZN/rYTb+hgNUeT7N
A1mP78IEkxAj4qG5tli4Jq41aCbUlCt7equGXokImhC+UY5IpQEZS0tKD4vu2ksZ
04Qetp0k8jWdAvMA27W3EwgHHNeVGWbJPC0Dn7RqPw13r7hFyS5TpleywjdY1nB7
ad6kcZXZbEcaFZ7ZuerA6RkPGE+PsnZRb1oFJkYoXimsuvkVFhWeHQXCGC1cuDWS
rM3cpQvOzKH2vS7d15+zGls4IwIDAQABo4IDaTCCA2UwHwYDVR0jBBgwFoAUt2ui
6qiqhIx56rTaD5iyxZV2ufQwHQYDVR0OBBYEFCYa+OSxsHKEztqBBtInmPvtOj0X
MIGBBgNVHREEejB4gg93d3cuZXhhbXBsZS5vcmeCC2V4YW1wbGUuY29tggtleGFt
cGxlLmVkdYILZXhhbXBsZS5uZXSCC2V4YW1wbGUub3Jngg93d3cuZXhhbXBsZS5j
b22CD3d3dy5leGFtcGxlLmVkdYIPd3d3LmV4YW1wbGUubmV0MA4GA1UdDwEB/wQE
AwIFoDAdBgNVHSUEFjAUBggrBgEFBQcDAQYIKwYBBQUHAwIwgYsGA1UdHwSBgzCB
gDA+oDygOoY4aHR0cDovL2NybDMuZGlnaWNlcnQuY29tL0RpZ2lDZXJ0VExTUlNB
U0hBMjU2MjAyMENBMS5jcmwwPqA8oDqGOGh0dHA6Ly9jcmw0LmRpZ2ljZXJ0LmNv
bS9EaWdpQ2VydFRMU1JTQVNIQTI1NjIwMjBDQTEuY3JsMEwGA1UdIARFMEMwNwYJ
YIZIAYb9bAEBMCowKAYIKwYBBQUHAgEWHGh0dHBzOi8vd3d3LmRpZ2ljZXJ0LmNv
bS9DUFMwCAYGZ4EMAQICMH0GCCsGAQUFBwEBBHEwbzAkBggrBgEFBQcwAYYYaHR0
cDovL29jc3AuZGlnaWNlcnQuY29tMEcGCCsGAQUFBzAChjtodHRwOi8vY2FjZXJ0
cy5kaWdpY2VydC5jb20vRGlnaUNlcnRUFNSU0FTSEEyNTYyMDIwQ0ExLmNydDAM
BgNVHRMBAf8EAjAAMIIBBQYKKwYBBAHWeQIEAgSB9gSB8wDxAHcA9lyUL9F3MCIU
VBgIMJRWjuNNExkzv98MLyALzE7xZOMAAAF1+73YbgAABAMASDBGAiEApGuo0EOk
8QcyLe2cOX136HPBn+0iSgDFvprJtbYS3LECIQCN6F+Kx1LNDaEj1bW729tiE4gi
```

Figure 10: Download the certificates.

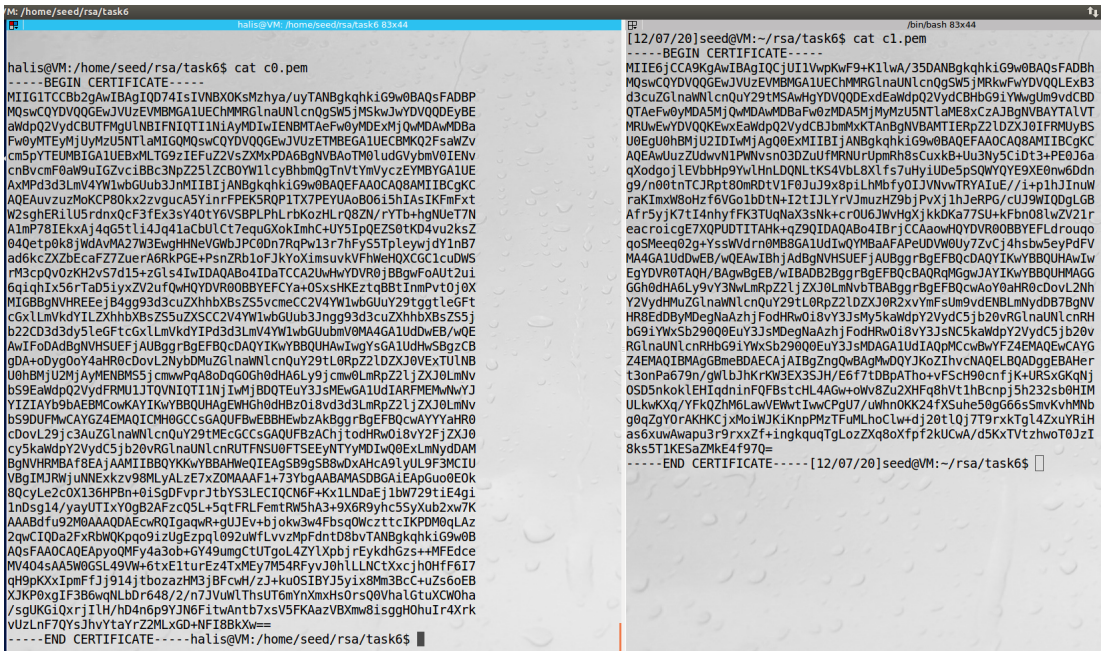

Figure 11: Save the files.

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

I obtained the public key value from the certificate provided by the issuer.

```
halis@VM:/home/seed/rsa/task6$ openssl x509 -in c1.pem -noout -modulus
Modulus=C14BB3654770BCDD4F58DBEC9CEDC366E51F311354AD4A66461F2C0AEC6407E52EDCDCB90A20
52416610604F571349F0E8376783DFE7D34B674C2251A6DF0E9910ED57517426E27DC7CA622E131B7F23
499AECC767D6E33EF5E3D6125E44F1BF71427D58840380B18101FAF9CA32BBB48E278727C52B74D4A8D6
D321300793EA99F5
halis@VM:/home/seed/rsa/task6$
```

Figure 12: n value

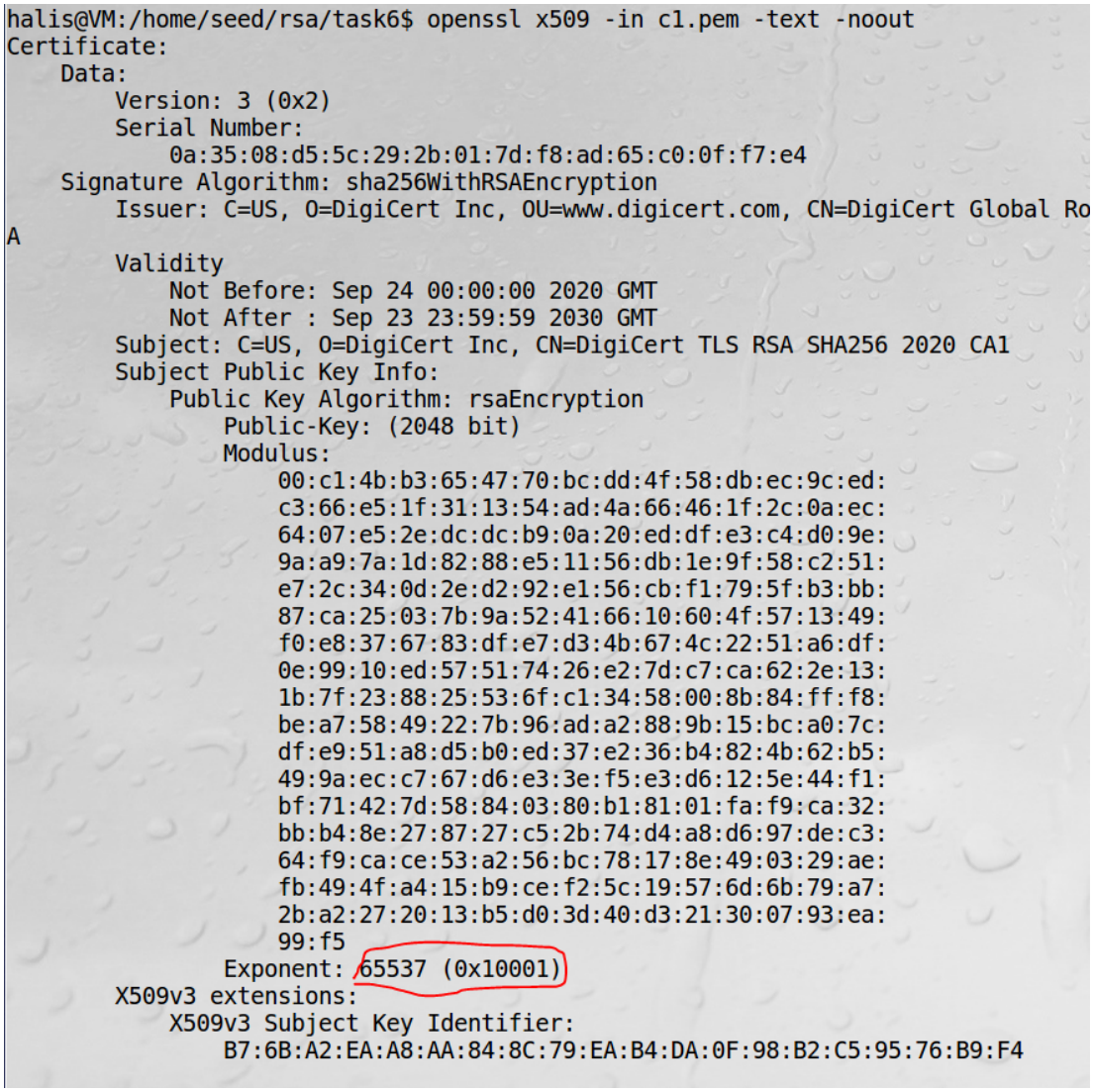

Figure 13: e value

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

I obtained the signature value from the certificate provided by the server.

Then I removed the punctuation marks in the signature.

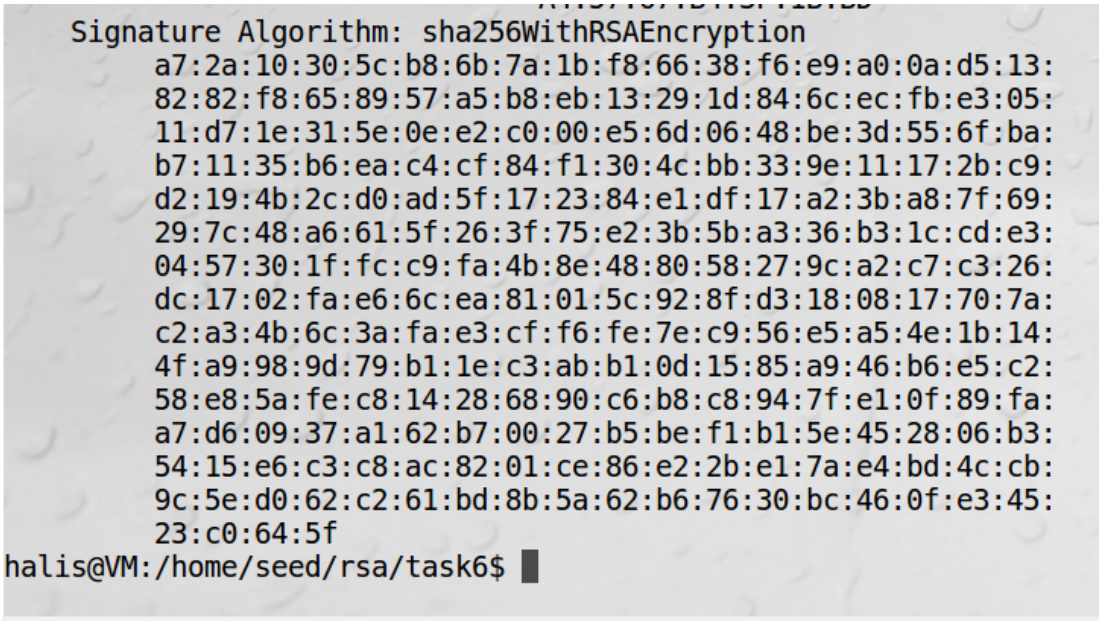

Figure 14: Server's signature

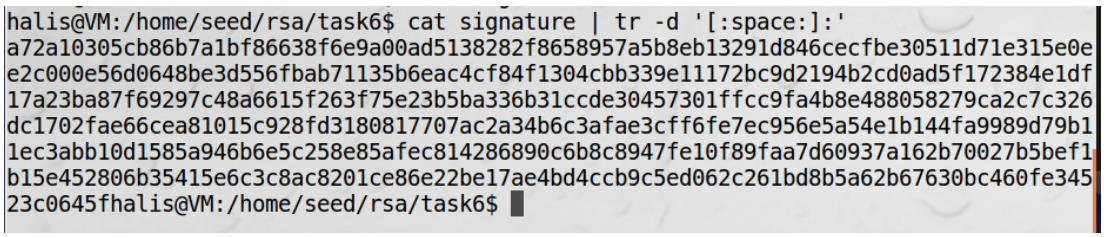

Figure 15: Signature

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

Then I got the body part in the server's certificate with the following command. In the last step I got the hash value for verification

$ openssl asn1parse -i -in c0.pem

$ openssl asn1parse -i -in c0.pem -strparse 4 -out c0_body.bin -noout

$ sha256sum c0_body.bin

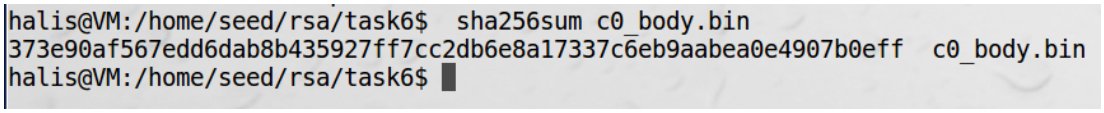

Figure 16: Hash of server's certificate's body

Hash: 373e90af567edd6dab8b435927ff7cc2db6e8a17337c6eb9aabea0e4907b0eff

6.5 Step 5: Verify the signature.

At the last task, we will decrypt the signature value using the values we obtained and check it with the hash value we obtained. If we can get the same hash value, we will have successfully completed our task.

```
// n value
BN_hex2bn(&n, "C14BB3654770BCDD4F58DBEC9CEDC366E51F311354AD4A6646\
1F2C0AEC6407E52EDCDCB90A20EDDFE3C4D09E9AA97A1D8288E51156DB1E9F58C2\
51E72C340D2ED292E156CBF1795FB3BB87CA25037B9A52416610604F571349F0E8\
376783DFE7D34B674C2251A6DF0E9910ED57517426E27DC7CA622E131B7F238825\
536FC13458008B84FFF8BEA75849227B96ADA2889B15BCA07CDFE951A8D5B0ED37\
E236B4824B62B5499AECC767D6E33EF5E3D6125E44F1BF71427D58840380B18101\
FAF9CA32BBB48E278727C52B74D4A8D697DEC364F9CACE53A256BC78178E490329\
AEFB494FA415B9CEF25C19576D6B79A72BA2272013B5D03D40D321300793EA99F5");

// e value - 65537
BN_hex2bn(&e, "10001");

// Signature
BN_hex2bn(&s, "a72a10305cb86b7a1bf86638f6e9a00ad5138282f8658957a5\
b8eb13291d846cecfbe30511d71e315e0ee2c000e56d0648be3d556fbab71135b6\
eac4cf84f1304cbb339e11172bc9d2194b2cd0ad5f172384e1df17a23ba87f69297\
c48a6615f263f75e23b5ba336b31ccde30457301ffcc9fa4b8e488058279ca2c7c32\
6dc1702fae66cea81015c928fd3180817707ac2a34b6c3afae3cff6fe7ec956e5a54\
e1b144fa9989d79b11ec3abb10d1585a946b6e5c258e85afec814286890c6b8c8947\
fe10f89faa7d60937a162b70027b5bef1b15e452806b35415e6c3c8ac8201ce86e22\
be17ae4bd4ccb9c5ed062c261bd8b5a62b67630bc460fe34523c0645f");

// Decryption s^e mod n , Compute m_dec = c^e mod n
BN_mod_exp(s_dec, s, e, n, ctx)
printBN("Decrypted Sign = ", s_dec);

// Last 32 bit must be same with this hash
printf("the pre-computed hash was: ");
printf("373e90af567edd6dab8b435927ff7cc2db6e8a17337c6eb9aabea0e4907b0eff\n");
```

If we run our program, the output will be as follows.

```
halis@VM:/home/seed/rsa/task6$ sudo gcc task6.c -o task6 -lcrypto
halis@VM:/home/seed/rsa/task6$ ./task6
Decrypted Sign =  01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFF003031300D060960864801650304020105000420373E90AF567EDD6DAB8B435927FF7CC2DB6E8A17
337C6EB9AABEA0E4907B0EFF
the pre-computed hash was: 373e90af567edd6dab8b435927ff7cc2db6e8a17337c6eb9aabea0e49
07b0eff
halis@VM:/home/seed/rsa/task6$
```

Figure 17: Output of task6.c : Result

As can be seen in Figure 17, when we deciphered our signature, the last 32 bits were the same as the hash value we obtained. Therefore, we manually verified the certificate.