CS 458

SHIQI LIU

**RSA Public-Key Encryption and Signature Lab Report**

1. **Introduction**

RSA (RivestShamirAdleman) is one of the first public-key cryptosystems and is widely used for secure communication. The RSA algorithm first generates two large random prime numbers, and then use them to generate public and private key pairs, which can be used to do encryption, decryption, digital signature generation, and digital signature verification. The RSA algorithm is built upon number theories, and it can be quite easily implemented with the support of libraries. Essentially, students will be implementing the RSA algorithm using the C program language.

The lab covers the following security-related topics:

• Public-key cryptography

• The RSA algorithm and key generation

• Big number calculation

• Encryption and Decryption using RSA

• Digital signature

• X.509 certificate

**Lab environment**: This lab requires the openssl library. I am using Ubuntu 16.04, which is already installed on.

1. **Background**

The RSA algorithm involves computations on large numbers. These computations cannot be directly conducted using simple arithmetic operators in programs, because those operators can only operate on primitive SEED Labs data types. In this lab, we will use the Big Number library provided by openssl. To use this library, we will define each big number as a BIGNUM type, and then use the APIs provided by the library for various operations, such as addition, multiplication, exponentiation, modular operations, etc.

First, I create a new folder named cs458lab2.Then I write all of the lab codes in this floder.

1. $ mkdir cs458Lab2
2. **Task**
   1. **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. Please calculate the private key d.

First, I create a Task1.c and use the code showing in the lab insturctions.

1. $ vi task1.c
2. #include <stdio.h>
3. #include <openssl/bn.h>
4. #define NBITS 256
6. **void** printBN(**char** \*msg, BIGNUM \*a){
7. // Convert the BIGNUM to number string
8. **char** \* number\_str = BN\_bn2hex(a);
9. // Print out the number string
10. printf("%s %s\n", msg, number\_str);
11. // Free the dynamically allocated memory
12. OPENSSL\_free(number\_str);
13. }
15. **int** main(){
16. BN\_CTX \*ctx = BN\_CTX\_new();
17. BIGNUM \*p = BN\_new();
18. BIGNUM \*q = BN\_new();
19. BIGNUM \*e = BN\_new();
20. BIGNUM \*d = BN\_new();
21. BIGNUM \*res1 = BN\_new();
22. BIGNUM \*res2 = BN\_new();
23. BIGNUM \*res3 = BN\_new();
24. BIGNUM \*one = BN\_new();
25. // initalize p q e
26. // Assign the first large prime
27. BN\_hex2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF");
29. // Assign the second large prime
30. BN\_hex2bn(&q, "E85CED54AF57E53E092113E62F436F4F");
32. // Assign the Modulus
33. BN\_hex2bn(&e, "0D88C3");
34. BN\_dec2bn(&one,"1");
35. //res1 = p-1
36. BN\_sub(res1, p, one);
37. //res2 = q-1
38. BN\_sub(res2, q, one);
39. //res3=res1\*res2
40. BN\_mul(res3, res1, res2, ctx);
41. //res=a\*b mod n
42. BN\_mod\_inverse(d, e, res3, ctx);
43. //print BN
44. printBN("d= ",d);
45. **return** 0;
46. }

First print out a big number:

Text

Description automatically generated

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)

Text

Description automatically generated

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

Text

Description automatically generated

Finally, we can get d by following commands:

1. seed@VM:~/cs458Lab2$ vi task1.c[10/18/20]
2. seed@VM:~/cs458Lab2$ gcc -o task1 task1.c -lcrypto
3. seed@VM:~/cs458Lab2$ ./task1

Text

Description automatically generated

Therefore, we get the private key

d=3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB

**3.2 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, I create a task2.c using command

1. vi task2.c

Import libary

Text

Description automatically generated

Print method to make the result

Text

Description automatically generated

According to the lab, we know

1. $ python -c ’print("A top secret!".encode("hex"))’
2. 4120746f702073656372657421

Therefore, we can use the hexadecimal of M

Text

Description automatically generatedText

Description automatically generated

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

Text

Description automatically generated

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

Task2 code:

1. #include <stdio.h>
2. #include <openssl/bn.h>
3. #define NBITS 256
5. //print a big number
6. **void** printBN(**char** \*msg, BIGNUM \*a){
7. // Convert the BIGNUM to number string
8. **char** \* number\_str = BN\_bn2hex(a);
9. // Print out the number string
10. printf("%s %s\n", msg, number\_str);
11. // Free the dynamically allocated memory
12. OPENSSL\_free(number\_str);
13. }
15. **int** main(){
16. BN\_CTX \*ctx = BN\_CTX\_new();
17. BIGNUM \*m = BN\_new();
18. BIGNUM \*e = BN\_new();
19. BIGNUM \*n = BN\_new();
20. BIGNUM \*d = BN\_new();
21. BIGNUM \*enc = BN\_new();
22. BIGNUM \*dec = BN\_new();
23. //Initialize
24. BN\_hex2bn(&e,"010001");
25. BN\_hex2bn(&n,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
26. BN\_hex2bn(&m,"4120746f702073656372657421");//A top secret!
27. BN\_hex2bn(&d,"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
28. //encry = m^e mod n
29. BN\_mod\_exp(enc,m,e,n,ctx);
30. printBN("encrypt message = ", enc);
32. //decry = enc^d mod n
33. BN\_mod\_exp(dec,enc,d,n,ctx);
34. printBN("decrypt message = ",dec);
35. **return** 0;
36. }
    1. **Task3 Decrypting a Message**

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

Code is similar to task2:

1. #include <stdio.h>
2. #include <openssl/bn.h>
3. #define NBITS 256
5. //print a big number
6. **void** printBN(**char** \*msg, BIGNUM \*a){
7. // Convert the BIGNUM to number string
8. **char** \* number\_str = BN\_bn2hex(a);
9. // Print out the number string
10. printf("%s %s\n", msg, number\_str);
11. // Free the dynamically allocated memory
12. OPENSSL\_free(number\_str);
13. }
15. **int** main(){
16. BN\_CTX \*ctx = BN\_CTX\_new();
17. BIGNUM \*n = BN\_new();
18. BIGNUM \*d = BN\_new();
19. BIGNUM \*c = BN\_new();
20. BIGNUM \*dec = BN\_new();
21. //Initialize
22. BN\_hex2bn(&n,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
23. BN\_hex2bn(&c,"8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");
24. BN\_hex2bn(&d,"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
25. //encry = m^e mod n
26. BN\_mod\_exp(dec,c,d,n,ctx);
27. printBN("encrypt message = ", dec);
29. **return** 0;
30. }

We decrypt the given cipher text, *c* using the formula: *c^d mod n.*

By decrypting, we get the hex value of the message.

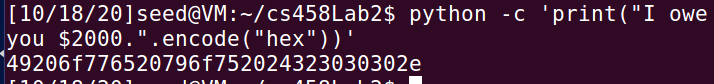
We then use the python to decode the hex value:

Text

Description automatically generated

* 1. **Task4 Signing a Message**

First, we get the hex value of “ I owe you $2000.”

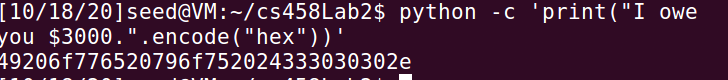


﻿Value is 49206f776520796f752024323030302e

We run our code to produce the signature for the message

Text

Description automatically generated

Then, we get the hex value of “I owe you $3000.”

Value is ﻿49206f776520796f752024333030302e

We run our code to produce the signature for the message:

Text

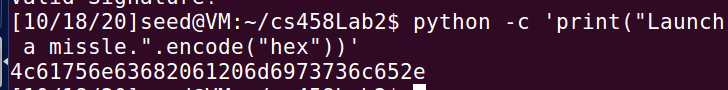
Description automatically generated

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

Code is similar to task3

1. #include <stdio.h>
2. #include <openssl/bn.h>
3. #define NBITS 256
5. //print a big number
6. **void** printBN(**char** \*msg, BIGNUM \*a){
7. // Convert the BIGNUM to number string
8. **char** \* number\_str = BN\_bn2hex(a);
9. // Print out the number string
10. printf("%s %s\n", msg, number\_str);
11. // Free the dynamically allocated memory
12. OPENSSL\_free(number\_str);
13. }
15. **int** main(){
16. BN\_CTX \*ctx = BN\_CTX\_new();
17. BIGNUM \*n = BN\_new();
18. BIGNUM \*d = BN\_new();
19. BIGNUM \*c = BN\_new();
20. BIGNUM \*dec = BN\_new();
21. //Initialize
22. BN\_hex2bn(&n,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
23. BN\_hex2bn(&c,"49206f776520796f752024323030302e");// HEX value of "I owe you $2000."
24. //BN\_hex2bn(&c,"49206f776520796f752024333030302e");// HEX value of "I owe you $3000."
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);
30. **return** 0;
31. }
    1. **Task5: Verifying a Signature**

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



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:

Text

Description automatically generated

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

Code:

1. //  task5.c
2. //
3. //
4. //  Created by SHIQI LIU on 10/18/20.
5. //
6. #include <stdio.h>
7. #include <openssl/bn.h>
8. **void** printBN(**char** \*msg, BIGNUM \*a)
9. {
10. **char** \*number\_str\_a = BN\_bn2hex(a);
11. printf("%s %s\n", msg, number\_str\_a);
12. OPENSSL\_free(number\_str\_a);
13. }
14. **int** main()
15. {
16. // init
17. BN\_CTX \*ctx = BN\_CTX\_new();
18. BIGNUM \*n = BN\_new();
19. BIGNUM \*e = BN\_new();
20. BIGNUM \*M = BN\_new();
21. // BIGNUM \*d = BN\_new();
22. BIGNUM \*C = BN\_new();
23. BIGNUM \*S = BN\_new();
25. // assign values
26. BN\_hex2bn(&n, "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");
27. BN\_dec2bn(&e, "65537");
28. BN\_hex2bn(&M, "4c61756e63682061206d697373696c652e"); //hex encode for " Launch a missile."
29. BN\_hex2bn(&S, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");
30. // Get S^e mod: if S=M^d mod n, C=M
31. BN\_mod\_exp(C, S, e, n, ctx);
32. printBN("Original Message : ", M );
33. printBN("Value of computed : ", C);
34. // verify the signature
35. **if** (BN\_cmp(C, M) == 0)
36. {
37. printf("Valid Signature! \n");
38. }
39. **else**
40. {
41. printf("Verification fails! \n");
42. }
44. **return** 0;
45. }

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:

Text

Description automatically generated

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

* 1. **Task6 Manually Verifying an X.509 Certificate**
* Step 1

Download a certificate from a real web server. We use the [www.google](http://www.google).com

server in this document

1. [10/18/20]seed@VM:~/cs458Lab2$ openssl s\_client -connect www.google.com:443 -showcerts
2. CONNECTED(00000003)
3. depth=2 OU = GlobalSign Root CA - R2, O = GlobalSign, CN = GlobalSign
4. verify **return**:1
5. depth=1 C = US, O = Google Trust Services, CN = GTS CA 1O1
6. verify **return**:1
7. depth=0 C = US, ST = California, L = Mountain View, O = Google LLC, CN = www.google.com
8. verify **return**:1
9. ---
10. Certificate chain
11. 0 s:/C=US/ST=California/L=Mountain View/O=Google LLC/CN=www.google.com
12. i:/C=US/O=Google Trust Services/CN=GTS CA 1O1
13. -----BEGIN CERTIFICATE-----
14. MIIFkTCCBHmgAwIBAgIQB5q2sT+2NbIIAAAAAFst1zANBgkqhkiG9w0BAQsFADBC
15. MQswCQYDVQQGEwJVUzEeMBwGA1UEChMVR29vZ2xlIFRydXN0IFNlcnZpY2VzMRMw
16. EQYDVQQDEwpHVFMgQ0EgMU8xMB4XDTIwMDkyMjE1MTYxOVoXDTIwMTIxNTE1MTYx
17. OVowaDELMAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWExFjAUBgNVBAcT
18. DU1vdW50YWluIFZpZXcxEzARBgNVBAoTCkdvb2dsZSBMTEMxFzAVBgNVBAMTDnd3
19. dy5nb29nbGUuY29tMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAnG1D
20. Z2Tl8l5ruTFxLz5PmUKZjYFEihvXKsmiu2d0NTKv8RgWvMItgvP/1IfIz+O76jn3
21. meC71lP10LST+wI+vFPE9vEL/Zie5veUijpXE0bUApeud+Rnlsw4UJ1x50PCBdSN
22. r41sNVCAAU98ibSGlc5n2Qr1/YYhs2qnqKKG8i8zNMbjDYyzOWyjfmRVi8W7eU2Z
23. FpJVwc3D6WEACqQrLATPcFNK0Jb3f5j7eohea9eMCslB13e43iIbA2zHb8owE2Mu
24. F7GKhYiTr8OEFy4aq7meEeIFHDB/O5eO+6nSxV0p3dBsdxXu/BlEVRuwnigCtMAn
25. LVS5OEhwBuz1Wl+VWwIDAQABo4ICWzCCAlcwDgYDVR0PAQH/BAQDAgWgMBMGA1Ud
26. JQQMMAoGCCsGAQUFBwMBMAwGA1UdEwEB/wQCMAAwHQYDVR0OBBYEFDDRC4ywBM9P
27. OAk0QWnLJlg3HjmJMB8GA1UdIwQYMBaAFJjR+G4Q68+b7GCfGJAboOt9Cf0rMGgG
28. CCsGAQUFBwEBBFwwWjArBggrBgEFBQcwAYYfaHR0cDovL29jc3AucGtpLmdvb2cv
29. Z3RzMW8xY29yZTArBggrBgEFBQcwAoYfaHR0cDovL3BraS5nb29nL2dzcjIvR1RT
30. MU8xLmNydDAZBgNVHREEEjAQgg53d3cuZ29vZ2xlLmNvbTAhBgNVHSAEGjAYMAgG
31. BmeBDAECAjAMBgorBgEEAdZ5AgUDMDMGA1UdHwQsMCowKKAmoCSGImh0dHA6Ly9j
32. cmwucGtpLmdvb2cvR1RTMU8xY29yZS5jcmwwggEDBgorBgEEAdZ5AgQCBIH0BIHx
33. AO8AdQDwlaRZ8gDRgkAQLS+TiI6tS/4dR+OZ4dA0prCoqo6ycwAAAXS2mgE4AAAE
34. AwBGMEQCIBA000C/IxSaE2sVhS+dJtnXsh7fSYjeybHOnFOtoRFCAiBnTcymqGeb
35. lwBe5U3nJyG3tngeH9YCfBdkeShmHdf6DgB2ALIeBcyLos2KIE6HZvkruYolIGdr
36. 2vpw57JJUy3vi5BeAAABdLaaADoAAAQDAEcwRQIgBFJa178fY3/4Pb95N5hh2JfR
37. 3EJ9AUbb0vff31qx/NsCIQDJTBFLs1QdubTe6S+Xc7EuQC9rh3YVLSYc8+dSRZuV
38. 3zANBgkqhkiG9w0BAQsFAAOCAQEAfznjbmvP1GKbyj7RIT5L/x6dkPBCWp6u6toi
39. 1ak4chqHN7mkJkazcb+DGoSAkz7DWfvrVt6Kruh7Vq93Z90g9Nnp5ZiMvkHd5+JM
40. VqVq3SEK0x+Bd//cW7364zsqnCP97Dg1kvPZz/Rqkq04i9ajSGNxkiMjkkFG4klO
41. tBuMXOmjoIPwa81iXT1tpV8TqV3uQj0FJ+WZXrYP33HSFGgEXO4VJq6cAh1o5V4z
42. +3KOF1Si5pVAzIHjEbTB9RP2WK7XQ2VkmyfqcnEJYCEnLDAReQkkGQMmJVO14jHh
43. SRcKMTc/HXPKihjE7cPmhElEehHzWkJLsYwYZcIGOP+shpz1rw==
44. -----END CERTIFICATE-----
45. 1 s:/C=US/O=Google Trust Services/CN=GTS CA 1O1
46. i:/OU=GlobalSign Root CA - R2/O=GlobalSign/CN=GlobalSign
47. -----BEGIN CERTIFICATE-----
48. MIIESjCCAzKgAwIBAgINAeO0mqGNiqmBJWlQuDANBgkqhkiG9w0BAQsFADBMMSAw
49. HgYDVQQLExdHbG9iYWxTaWduIFJvb3QgQ0EgLSBSMjETMBEGA1UEChMKR2xvYmFs
50. U2lnbjETMBEGA1UEAxMKR2xvYmFsU2lnbjAeFw0xNzA2MTUwMDAwNDJaFw0yMTEy
51. MTUwMDAwNDJaMEIxCzAJBgNVBAYTAlVTMR4wHAYDVQQKExVHb29nbGUgVHJ1c3Qg
52. U2VydmljZXMxEzARBgNVBAMTCkdUUyBDQSAxTzEwggEiMA0GCSqGSIb3DQEBAQUA
53. A4IBDwAwggEKAoIBAQDQGM9F1IvN05zkQO9+tN1pIRvJzzyOTHW5DzEZhD2ePCnv
54. UA0Qk28FgICfKqC9EksC4T2fWBYk/jCfC3R3VZMdS/dN4ZKCEPZRrAzDsiKUDzRr
55. mBBJ5wudgzndIMYcLe/RGGFl5yODIKgjEv/SJH/UL+dEaltN11BmsK+eQmMF++Ac
56. xGNhr59qM/9il71I2dN8FGfcddwuaej4bXhp0LcQBbjxMcI7JP0aM3T4I+DsaxmK
57. FsbjzaTNC9uzpFlgOIg7rR25xoynUxv8vNmkq7zdPGHXkxWY7oG9j+JkRyBABk7X
58. rJfoucBZEqFJJSPk7XA0LKW0Y3z5oz2D0c1tJKwHAgMBAAGjggEzMIIBLzAOBgNV
59. HQ8BAf8EBAMCAYYwHQYDVR0lBBYwFAYIKwYBBQUHAwEGCCsGAQUFBwMCMBIGA1Ud
60. EwEB/wQIMAYBAf8CAQAwHQYDVR0OBBYEFJjR+G4Q68+b7GCfGJAboOt9Cf0rMB8G
61. A1UdIwQYMBaAFJviB1dnHB7AagbeWbSaLd/cGYYuMDUGCCsGAQUFBwEBBCkwJzAl
62. BggrBgEFBQcwAYYZaHR0cDovL29jc3AucGtpLmdvb2cvZ3NyMjAyBgNVHR8EKzAp
63. MCegJaAjhiFodHRwOi8vY3JsLnBraS5nb29nL2dzcjIvZ3NyMi5jcmwwPwYDVR0g
64. BDgwNjA0BgZngQwBAgIwKjAoBggrBgEFBQcCARYcaHR0cHM6Ly9wa2kuZ29vZy9y
65. ZXBvc2l0b3J5LzANBgkqhkiG9w0BAQsFAAOCAQEAGoA+Nnn78y6pRjd9XlQWNa7H
66. TgiZ/r3RNGkmUmYHPQq6Scti9PEajvwRT2iWTHQr02fesqOqBY2ETUwgZQ+lltoN
67. FvhsO9tvBCOIazpswWC9aJ9xju4tWDQH8NVU6YZZ/XteDSGU9YzJqPjY8q3MDxrz
68. mqepBCf5o8mw/wJ4a2G6xzUr6Fb6T8McDO22PLRL6u3M4Tzs3A2M1j6bykJYi8wW
69. IRdAvKLWZu/axBVbzYmqmwkm5zLSDW5nIAJbELCQCZwMH56t2Dvqofxs6BBcCFIZ
70. USpxu6x6td0V7SvJCCosirSmIatj/9dSSVDQibet8q/7UK4v4ZUN80atnZz1yg==
71. -----END CERTIFICATE-----
72. ---
73. Server certificate
74. subject=/C=US/ST=California/L=Mountain View/O=Google LLC/CN=www.google.com
75. issuer=/C=US/O=Google Trust Services/CN=GTS CA 1O1
76. ---
77. No client certificate CA names sent
78. Peer signing digest: SHA256
79. Server Temp Key: ECDH, P-256, 256 bits
80. ---
81. SSL handshake has read 3247 bytes and written 431 bytes
82. ---
83. New, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES128-GCM-SHA256
84. Server **public** key is 2048 bit
85. Secure Renegotiation IS supported
86. Compression: NONE
87. Expansion: NONE
88. No ALPN negotiated
89. SSL-Session:
90. Protocol  : TLSv1.2
91. Cipher    : ECDHE-RSA-AES128-GCM-SHA256
92. Session-ID: BC337A8A0CE22588A9845CD955A25131346F558771B2B435CFB7373C3534E9E7
93. Session-ID-ctx:
94. Master-Key: 42EA2EB90D0B1F59618908405370AAB39B2C8E27CD06BB4C99E71435294027B3DAB36CCA97E370AA4A738095F45CDBC0
95. Key-Arg   : None
96. PSK identity: None
97. PSK identity hint: None
98. SRP username: None
99. TLS session ticket lifetime hint: 100800 (seconds)
100. TLS session ticket:
101. 0000 - 01 95 05 1c 6a 74 90 8c-f9 0e 6c 5a cc dc cd 1b   ....jt....lZ....
102. 0010 - e0 a3 61 6d 1a 72 be 3c-e5 8a b1 55 2e d7 5a ab   ..am.r.<...U..Z.
103. 0020 - a9 01 59 f0 e8 eb 0e a2-e5 34 a4 b6 be 05 ec 7c   ..Y......4.....|
104. 0030 - 08 9e e2 70 94 9c e1 8b-47 8e 10 24 c8 e6 e6 5c   ...p....G..$...\
105. 0040 - cc 83 71 a2 a2 1c c8 c3-db 1f df c9 15 23 3e e0   ..q..........#>.
106. 0050 - dc f9 73 33 46 83 27 d9-ab 92 40 0d 92 41 89 6f   ..s3F.'...@..A.o
107. 0060 - 49 bb 2f 30 8c 8e fc fd-bc 3a 44 c5 67 3d c2 15   I./0.....:D.g=..
108. 0070 - 29 2c 86 39 53 66 a0 68-70 48 36 99 50 e7 09 ba   ),.9Sf.hpH6.P...
109. 0080 - 39 ce 7f 5b d1 fb a1 7f-83 b9 6b be 36 b5 c1 a6   9..[......k.6...
110. 0090 - a7 32 f0 85 d7 04 77 1b-e4 c4 03 77 d3 0b 27 92   .2....w....w..'.
111. 00a0 - 1d 4d a5 06 35 67 69 5e-8b 51 cd 64 2e 91 8f 79   .M..5gi^.Q.d...y
112. 00b0 - fa c8 5b 17 8f 63 f4 c5-89 e0 83 2d 16 2f 9e f7   ..[..c.....-./..
113. 00c0 - 33 20 5d 03 00 c6 f4 0d-ec b5 a2 d0 6a 49 a4 49   3 ].........jI.I
114. 00d0 - e5 6a a9 fc f7 05 34 d5-7d 63 af 4d 79            .j....4.}c.My
116. Start Time: 1603076169
117. Timeout   : 300 (sec)
118. Verify **return** code: 0 (ok)
119. ---
120. read:errno=0
121. [10/18/20]seed@VM:~/cs458Lab2$   Text

     Description automatically generatedText

     Description automatically generatedText

     Description automatically generated

We save these two certificates in the files c0.pem and c1.pem respectively

* Step2 Extract the public key (e, n) from the issuer’s certificate.

We get the value of n using -modulus:

* [10/18/20]seed@VM:~/cs458Lab2$ openssl x509 -in c1.pem -noout -modulus
* Modulus

Text

Description automatically generated

Print all attributes of the certificate, and then find the exponent, which is public key e(line 35)

1. [10/18/20]seed@VM:~/cs458Lab2$ openssl x509 -in c1.pem -text -noout
2. Certificate:
3. Data:
4. Version: 3 (0x2)
5. Serial Number:
6. 01:e3:b4:9a:a1:8d:8a:a9:81:25:69:50:b8
7. Signature Algorithm: sha256WithRSAEncryption
8. Issuer: OU=GlobalSign Root CA - R2, O=GlobalSign, CN=GlobalSign
9. Validity
10. Not Before: Jun 15 00:00:42 2017 GMT
11. Not After : Dec 15 00:00:42 2021 GMT
12. Subject: C=US, O=Google Trust Services, CN=GTS CA 1O1
13. Subject Public Key Info:
14. Public Key Algorithm: rsaEncryption
15. Public-Key: (2048 bit)
16. Modulus:
17. 00:d0:18:cf:45:d4:8b:cd:d3:9c:e4:40:ef:7e:b4:
18. dd:69:21:1b:c9:cf:3c:8e:4c:75:b9:0f:31:19:84:
19. 3d:9e:3c:29:ef:50:0d:10:93:6f:05:80:80:9f:2a:
20. a0:bd:12:4b:02:e1:3d:9f:58:16:24:fe:30:9f:0b:
21. 74:77:55:93:1d:4b:f7:4d:e1:92:82:10:f6:51:ac:
22. 0c:c3:b2:22:94:0f:34:6b:98:10:49:e7:0b:9d:83:
23. 39:dd:20:c6:1c:2d:ef:d1:18:61:65:e7:23:83:20:
24. a8:23:12:ff:d2:24:7f:d4:2f:e7:44:6a:5b:4d:d7:
25. 50:66:b0:af:9e:42:63:05:fb:e0:1c:c4:63:61:af:
26. 9f:6a:33:ff:62:97:bd:48:d9:d3:7c:14:67:dc:75:
27. dc:2e:69:e8:f8:6d:78:69:d0:b7:10:05:b8:f1:31:
28. c2:3b:24:fd:1a:33:74:f8:23:e0:ec:6b:19:8a:16:
29. c6:e3:cd:a4:cd:0b:db:b3:a4:59:60:38:88:3b:ad:
30. 1d:b9:c6:8c:a7:53:1b:fc:bc:d9:a4:ab:bc:dd:3c:
31. 61:d7:93:15:98:ee:81:bd:8f:e2:64:47:20:40:06:
32. 4e:d7:ac:97:e8:b9:c0:59:12:a1:49:25:23:e4:ed:
33. 70:34:2c:a5:b4:63:7c:f9:a3:3d:83:d1:cd:6d:24:
34. ac:07
35. Exponent: 65537 (0x10001)

Text

Description automatically generatedText

Description automatically generated

* Step3 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.

1. [10/18/20]seed@VM:~/cs458Lab2$  openssl x509 -in c0.pem -text -noout
2. Signature Algorithm: sha256WithRSAEncryption
3. 7f:39:e3:6e:6b:cf:d4:62:9b:ca:3e:d1:21:3e:4b:ff:1e:9d:
4. 90:f0:42:5a:9e:ae:ea:da:22:d5:a9:38:72:1a:87:37:b9:a4:
5. 26:46:b3:71:bf:83:1a:84:80:93:3e:c3:59:fb:eb:56:de:8a:
6. ae:e8:7b:56:af:77:67:dd:20:f4:d9:e9:e5:98:8c:be:41:dd:
7. e7:e2:4c:56:a5:6a:dd:21:0a:d3:1f:81:77:ff:dc:5b:bd:fa:
8. e3:3b:2a:9c:23:fd:ec:38:35:92:f3:d9:cf:f4:6a:92:ad:38:
9. 8b:d6:a3:48:63:71:92:23:23:92:41:46:e2:49:4e:b4:1b:8c:
10. 5c:e9:a3:a0:83:f0:6b:cd:62:5d:3d:6d:a5:5f:13:a9:5d:ee:
11. 42:3d:05:27:e5:99:5e:b6:0f:df:71:d2:14:68:04:5c:ee:15:
12. 26:ae:9c:02:1d:68:e5:5e:33:fb:72:8e:17:54:a2:e6:95:40:
13. cc:81:e3:11:b4:c1:f5:13:f6:58:ae:d7:43:65:64:9b:27:ea:
14. 72:71:09:60:21:27:2c:30:11:79:09:24:19:03:26:25:53:b5:
15. e2:31:e1:49:17:0a:31:37:3f:1d:73:ca:8a:18:c4:ed:c3:e6:
16. 84:49:44:7a:11:f3:5a:42:4b:b1:8c:18:65:c2:06:38:ff:ac:
17. 86:9c:f5:af
18. [10/18/20]seed@VM:~/cs458Lab2$

Text

Description automatically generated

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

1. [10/18/20]seed@VM:~/cs458Lab2$ vi signature
2. [10/18/20]seed@VM:~/cs458Lab2$ cat signature | tr -d '[:space:]:'
3. ee87b56af7767dd20f4d9e9e5988cbe41dde7e24c56a56add210ad31f8177ffdc5bbdfae33b2a9c23fdec383592f3d9cff46a92ad388bd6a3486371922323924146e2494eb41b8c5ce9a3a083f06bcd625d3d6da55f13a95dee423d0527e5995eb60fdf71d21468045cee1526ae9c021d68e55e33fb728e1754a2e69540[10/18/20]seed@VM:~/cs458Lab2$

Text

Description automatically generated

* Step4 : Extract the body of the server’s certificate
* [10/18/20]seed@VM:~/cs458Lab2$ openssl asn1parse -i -in c0.pem
* 0:d=0  hl=4 l=1425 cons: SEQUENCE
* 4:d=1  hl=4 l=1145 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           :079AB6B13FB635B208000000005B2DD7
* 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=  66 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=  19 cons:    SET
* 95:d=4  hl=2 l=  17 cons:     SEQUENCE
* 97:d=5  hl=2 l=   3 prim:      OBJECT            :commonName
* 102:d=5  hl=2 l=  10 prim:      PRINTABLESTRING   :GTS CA 1O1
* 114:d=2  hl=2 l=  30 cons:   SEQUENCE
* 116:d=3  hl=2 l=  13 prim:    UTCTIME           :200922151619Z
* 131:d=3  hl=2 l=  13 prim:    UTCTIME           :201215151619Z
* 146:d=2  hl=2 l= 104 cons:   SEQUENCE
* 148:d=3  hl=2 l=  11 cons:    SET
* 150:d=4  hl=2 l=   9 cons:     SEQUENCE
* 152:d=5  hl=2 l=   3 prim:      OBJECT            :countryName
* 157:d=5  hl=2 l=   2 prim:      PRINTABLESTRING   :US
* 161:d=3  hl=2 l=  19 cons:    SET
* 163:d=4  hl=2 l=  17 cons:     SEQUENCE
* 165:d=5  hl=2 l=   3 prim:      OBJECT            :stateOrProvinceName
* 170:d=5  hl=2 l=  10 prim:      PRINTABLESTRING   :California
* 182:d=3  hl=2 l=  22 cons:    SET
* 184:d=4  hl=2 l=  20 cons:     SEQUENCE
* 186:d=5  hl=2 l=   3 prim:      OBJECT            :localityName
* 191:d=5  hl=2 l=  13 prim:      PRINTABLESTRING   :Mountain View
* 206:d=3  hl=2 l=  19 cons:    SET
* 208:d=4  hl=2 l=  17 cons:     SEQUENCE
* 210:d=5  hl=2 l=   3 prim:      OBJECT            :organizationName
* 215:d=5  hl=2 l=  10 prim:      PRINTABLESTRING   :Google LLC
* 227:d=3  hl=2 l=  23 cons:    SET
* 229:d=4  hl=2 l=  21 cons:     SEQUENCE
* 231:d=5  hl=2 l=   3 prim:      OBJECT            :commonName
* 236:d=5  hl=2 l=  14 prim:      PRINTABLESTRING   :www.google.com
* 252:d=2  hl=4 l= 290 cons:   SEQUENCE
* 256:d=3  hl=2 l=  13 cons:    SEQUENCE
* 258:d=4  hl=2 l=   9 prim:     OBJECT            :rsaEncryption
* 269:d=4  hl=2 l=   0 prim:     NULL
* 271:d=3  hl=4 l= 271 prim:    BIT STRING
* 546:d=2  hl=4 l= 603 cons:   cont [ 3 ]
* 550:d=3  hl=4 l= 599 cons:    SEQUENCE
* 554:d=4  hl=2 l=  14 cons:     SEQUENCE
* 556:d=5  hl=2 l=   3 prim:      OBJECT            :X509v3 Key Usage
* 561:d=5  hl=2 l=   1 prim:      **BOOLEAN**           :255
* 564:d=5  hl=2 l=   4 prim:      OCTET STRING      [HEX DUMP]:030205A0
* 570:d=4  hl=2 l=  19 cons:     SEQUENCE
* 572:d=5  hl=2 l=   3 prim:      OBJECT            :X509v3 Extended Key Usage
* 577:d=5  hl=2 l=  12 prim:      OCTET STRING      [HEX DUMP]:300A06082B06010505070301
* 591:d=4  hl=2 l=  12 cons:     SEQUENCE
* 593:d=5  hl=2 l=   3 prim:      OBJECT            :X509v3 Basic Constraints
* 598:d=5  hl=2 l=   1 prim:      **BOOLEAN**           :255
* 601:d=5  hl=2 l=   2 prim:      OCTET STRING      [HEX DUMP]:3000
* 605:d=4  hl=2 l=  29 cons:     SEQUENCE
* 607:d=5  hl=2 l=   3 prim:      OBJECT            :X509v3 Subject Key Identifier
* 612:d=5  hl=2 l=  22 prim:      OCTET STRING      [HEX DUMP]:041430D10B8CB004CF4F3809344169CB2658371E3989
* 636:d=4  hl=2 l=  31 cons:     SEQUENCE
* 638:d=5  hl=2 l=   3 prim:      OBJECT            :X509v3 Authority Key Identifier
* 643:d=5  hl=2 l=  24 prim:      OCTET STRING      [HEX DUMP]:3016801498D1F86E10EBCF9BEC609F18901BA0EB7D09FD2B
* 669:d=4  hl=2 l= 104 cons:     SEQUENCE
* 671:d=5  hl=2 l=   8 prim:      OBJECT            :Authority Information Access
* 681:d=5  hl=2 l=  92 prim:      OCTET STRING      [HEX DUMP]:305A302B06082B06010505073001861F687474703A2F2F6F6373702E706B692E676F6F672F677473316F31636F7265302B06082B06010505073002861F687474703A2F2F706B692E676F6F672F677372322F475453314F312E637274
* 775:d=4  hl=2 l=  25 cons:     SEQUENCE
* 777:d=5  hl=2 l=   3 prim:      OBJECT            :X509v3 Subject Alternative Name
* 782:d=5  hl=2 l=  18 prim:      OCTET STRING      [HEX DUMP]:3010820E7777772E676F6F676C652E636F6D
* 802:d=4  hl=2 l=  33 cons:     SEQUENCE
* 804:d=5  hl=2 l=   3 prim:      OBJECT            :X509v3 Certificate Policies
* 809:d=5  hl=2 l=  26 prim:      OCTET STRING      [HEX DUMP]:30183008060667810C010202300C060A2B06010401D679020503
* 837:d=4  hl=2 l=  51 cons:     SEQUENCE
* 839:d=5  hl=2 l=   3 prim:      OBJECT            :X509v3 CRL Distribution Points
* 844:d=5  hl=2 l=  44 prim:      OCTET STRING      [HEX DUMP]:302A3028A026A0248622687474703A2F2F63726C2E706B692E676F6F672F475453314F31636F72652E63726C
* 890:d=4  hl=4 l= 259 cons:     SEQUENCE
* 894:d=5  hl=2 l=  10 prim:      OBJECT            :CT Precertificate SCTs
* 906:d=5  hl=3 l= 244 prim:      OCTET STRING      [HEX DUMP]:0481F100EF007500F095A459F200D18240102D2F93888EAD4BFE1D47E399E1D034A6B0A8AA8EB27300000174B69A0138000004030046304402201034D340BF23149A136B15852F9D26D9D7B21EDF4988DEC9B1CE9C53ADA111420220674DCCA6A8679B97005EE54DE72721B7B6781E1FD6027C17647928661DD7FA0E007600B21E05CC8BA2CD8A204E8766F92BB98A2520676BDAFA70E7B249532DEF8B905E00000174B69A003A0000040300473045022004525AD7BF1F637FF83DBF79379861D897D1DC427D0146DBD2F7DFDF5AB1FCDB022100C94C114BB3541DB9B4DEE92F9773B12E402F6B8776152D261CF3E752459B95DF
* 1153:d=1  hl=2 l=  13 cons:  SEQUENCE
* 1155:d=2  hl=2 l=   9 prim:   OBJECT            :sha256WithRSAEncryption
* 1166:d=2  hl=2 l=   0 prim:   NULL
* 1168:d=1  hl=4 l= 257 prim:  BIT STRING

Text

Description automatically generated

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.

1. [10/18/20]seed@VM:~/cs458Lab2$ openssl asn1parse -i -in c0.pem -strparse 4 -out c0\_body.bin -noout
2. [10/18/20]seed@VM:~/cs458Lab2$ sha256sum c0\_body.bin
3. fe5623d9b8e79ffa12b3b6471ae96c3ebc8ee5b6144305320c8c7de06443269a  c0\_body.bin

Text

Description automatically generated

* Step5: Verify the signature.

Code is similar to 3.5, We use the values obtained from the previous steps, get the signature and verify the signature obtained with the original signatureText

Description automatically generated

Text

Description automatically generated

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.