Lab Title	Author	Date	Class	Section
RSA Encryption and Signature	Rushabh Prajapati	19/01/2022	CMPT 380 Computer Software Security	

Task 1

Finding private key D

To derive the private key, D,

```
[01/15/22]seed@VM:~/.../RSA$ gcc -o find-private-key find-private-key.c -lcrypto
[01/15/22]seed@VM:~/.../RSA$ ls
find-private-key  find-private-key.c
[01/15/22]seed@VM:~/.../RSA$ ./find-private-key
Private key d: 3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB
[01/15/22]seed@VM:~/.../RSA$ 
```

Figure 1 Private key d output

public key (

e = 0D88C3,

n = E103ABD94892E3E74AFD724BF28E78366D9676BCCC70118BD0AA1968DBB143D1

)

private key (

d = 3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB,

n = E103ABD94892E3E74AFD724BF28E78366D9676BCCC70118BD0AA1968DBB143D1

)

The Value of private key d is = **3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB**

Task 2

Encrypting a message

```
[01/15/22]seed@VM:~/.../Task2$ python -c 'print("A top secret!".encode("hex"))'
4120746f702073656372657421
[01/15/22]seed@VM:~/.../Task2$ 
```

Figure 2 Converting a text string to hex string

```
[01/15/22]seed@VM:~/.../Task2$ gcc -o encrypting_a_message encrypting_a_message.c -lcrypto
[01/15/22]seed@VM:~/.../Task2$ ./encrypting_a_message
Encryption Result:  6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC
Decryption Result:  4120746F702073656372657421
[01/15/22]seed@VM:~/.../Task2$ █
```

Figure 3 Encrypting the message 'A top secret!'

The above C code uses BN.mod_inverse () to calculate the private key exponent d from e and phi(n). This API uses the extended Euclidean algorithm to calculate the modular inverse of e, which is d.

Encryption using – $m^e \ mod \ n$, where m is our secret message, we prepare the plaintext m by assigning a hex string to variable m. This hex string is the hex representation o f the ASCII string "A top secret!".

When the program finishes, we cannot just exit from the program, because the private key d, the secret prime numbers p and q, and other related intermediate results are still stored in the memory. When the same physical memory is assigned to another process, the content may not be cleared by the operating system. That can cause the private key and its related secret information to be leaked out. We need to erase those data. We can use BN_clear (a) to erase the memory used by the variable and set the variable to zero, or use BN_clear_free (a) to erase and free a 's memory.

We can see that after encryption, we get the hash value and once we decrypt it using the decryption, we get the hex string of the original message back, this successfully encrypting a message and decrypting back into the original message.

Task 3

Decrypting a message

Similar to what we did in task 2, here we were given the cipher text C, so the main task to figure out how to get the ASCII text from the given C, for that we were given the function BN_mod_exp(), which takes the parameters a new_hex string, the cipher text c, the private key d and sum of p and q, n. The resulting message is an hex String which when decoded with python gives us the decrypted cypher text.

Decrypted Cipher Text

“password is dees”

```
[01/15/22]seed@VM:~/.../Task3$ gcc -o encrypting_a_message encrypting_a_message.c -lcrypto
[01/15/22]seed@VM:~/.../Task3$ ./encrypting_a_message
Cipher Text:  8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F
ASCII Result:  50617373776F72642069732064656573
[01/15/22]seed@VM:~/.../Task3$ python -c 'print("50617373776F72642069732064656573")'
50617373776F72642069732064656573
[01/15/22]seed@VM:~/.../Task3$ python -c 'print("50617373776F72642069732064656573".decode("hex
"))'
Password is dees
[01/15/22]seed@VM:~/.../Task3$
```

Figure 4 Decy[ted Cipher Text

Task 4

Signing a message

If we apply the public-key operation on **message m** and then
conduct the private-key operation, we will get the original m back; if we reverse the order, applying the private-key firs t, followed by the public-key operation, we will also get m back.

If we apply the private-key operation on m using our own private key and get a number s = $m^\wedge d \ mod \ n$, everybody can get the ın back from s using our public key, so obviously, this cannot be used for encryption. However, since we are the only one who know the private key d, we are the only one who can produce the numbers from m; nobody else can, but they can easily verify the relationship between s and m. This is reminiscent of the hand-written signature.

For a message m that needs to be signed, we calculate s = m^d mod n using our private key, and s will serve as our signature on the message. If neither m nor s is modified, everybody can verify that se mod n equal s to m, but if any of them is modified by an attacker, such a relationship will not hold any more, unless the attacker knows our private key.

In the below, figure we can observe that a tiny change can change the value of our signature completely, thus making it quite impossible to forge.

```
[01/15/22]seed@VM:~/.../Task4$ python -c 'print("I owe you $2000".encode("hex"))'
49206f776520796f75202432303030
[01/15/22]seed@VM:~/.../Task4$ python -c 'print("I owe you $3000.".encode("hex"))'
49206f776520796f752024333030302e
[01/15/22]seed@VM:~/.../Task4$ gcc -o signing_a_message signing_a_message.c -lcrypto
[01/15/22]seed@VM:~/.../Task4$ ./signing_a_message
Cipher Signature (I owe you $2000):  55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73CCB35E4CB
Cipher Signature (I owe you $3000):  BCC20FB7568E5D48E434C387C06A6025E90D29D848AF9C3EBAC0135D99305822
[01/15/22]seed@VM:~/.../Task4$
```

Figure 5 Cipher Signature of 2 Different Text just with a 2 changed to 3 for one of the messages.

Task 5

Verify the Signature

So given the public key (e,n), we fiest need to generate the HEX code the message "Launch a missile", which can be done through python, to generate a hex string M and lastly we have the cipher signature S to match.

Firstly, we try to generate the cipher C, using openssl's BN_mod_exp() function, taking S^e mod n, and once we generate the cipher text C, comparing this with the generated hex string M, using openssl's BN_cmp() function, and if C == M, the signature is verified. Therefore, the message belong's to Alice.

```
[01/15/22]seed@VM:~/.../Task5$ python -c 'print("Launch a missile.".encode("hex"))'
4c61756e63682061206d697373696c652e
[01/15/22]seed@VM:~/.../Task5$ gcc -o signing_a_message signing_a_message.c -lcrypto
[01/15/22]seed@VM:~/.../Task5$ ./signing_a_message
Cipher Sinature: 643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F
Verified Signature[01/15/22]seed@VM:~/.../Task5$ 
```

Figure 6Verifying the Signature

Task 5.1

Changing the Signature value from 2F to 3F, as we can see in the below image that, changing the value of the signature, also made the verification fail, thus our program will also reject the message and it can be conclude that the message does not belong to Alice or maybe is forged during transfer.

```
[01/15/22]seed@VM:~/.../Task5$ gcc -o tampered_signature tampered_signature.c -lcrypto
[01/15/22]seed@VM:~/.../Task5$ ./tampered_signature
Tampered Cipher Signature: 643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F
Verification Failed[01/15/22]seed@VM:~/.../Task5$ 
```

Figure 7 Verification fails as the value of signature changes

Task 6

Manually Verify an X.509 Certificate

Here, we are tying to manually verify an X.509 certification of www.google.com

Step 1 Downloading the certificate from the real web server using openssl.

```
[01/19/22]seed@VM:~/.../TG6$ openssl s_client -connect www.google.com:443 -showcerts
CONNECTED(00000003)
depth=2 C = US, O = Google Trust Services LLC, CN = GTS Root R1
verify error:num=20:unable to get local issuer certificate
verify return:1
depth=1 C = US, O = Google Trust Services LLC, CN = GTS CA 1C3
verify return:1
depth=0 CN = www.google.com
verify return:1
---
Certificate chain
 0 s:CN = www.google.com
   i:C = US, O = Google Trust Services LLC, CN = GTS CA 1C3
-----BEGIN CERTIFICATE-----
MIIEhzCCA2+gAwIBAgIRAPS7Wjp6OCxoCgAAAAEn3j0wDQYJKoZIhvcNAQELBQAw
RjELMAkGA1UEBhMCVVMxIjAgBgNVBAoTGUdvb2dsZSBUcnVzdCBTZXJ2aWNlcyBM
TEMxEzARBgNVBAMTCkdUUyBDQSAxQzMwHhcNMjExMjA4MjI1MDM0WhcNMjIwMzAy
MjI1MDMzWjAZMRcwFQYDVQQDEw53d3cuZ29vZ2xlLmNvbTBZMBMGByqGSM49AgEG
CCqGSM49AwEHA0IABMx8+MQr43Hcgpj1ncFhdctKpwqL52UkKr/QTfJsQ0PEQSAM
```

Figure 8 openssl to get the certificate of google.com

We get 2 certificates by using the above openssl command. C0.pem being the Server's CA and c1.pem being the Issuer's CA.

```
-----BEGIN CERTIFICATE-----
MIIEhzCCA2+gAwIBAgIRAPS7Wjp6OCxoCgAAAAEn3j0wDQYJKoZIhvcNAQELBQAw
RjELMAkGA1UEBhMCVVMxIjAgBgNVBAoTGUdvb2dsZSBUcnVzdCBTZXJ2aWNlcyBM
TEMxEzARBgNVBAMTCkdUUyBDQSAxQzMwHhcNMjExMjA4MjI1MDM0WhcNMjIwMzAy
MjI1MDMzWjAZMRcwFQYDVQQDEw53d3cuZ29vZ2xlLmNvbTBZMBMGByqGSM49AgEG
CCqGSM49AwEHA0IABMx8+MQr43Hcgpj1ncFhdctKpwqL52UkKr/QTfJsQ0PEQSAM
G1gHCbgchLgLxciHfqzc6oleRE0DL87CnESz5QijggJmMIICYjAOBgNVHQ8BAf8E
BAMCB4AwEwYDVR0lBAwwCgYIKwYBBQUHAwEwDAYDVR0TAQH/BAIwADAdBgNVHQ4E
FgQUxdRktCODmFCYI1rrKQepkM8ukqgwHwYDVR0jBBgwFoAUinR/r4XN7pXNPZzQ
4kYU83E1HScwagYIKwYBBQUHAQEEXjBcMCcGCCsGAQUFBzABhhtodHRwOi8vb2Nz
cC5wa2kuZ29vZy9ndHMxYzMwMQYIKwYBBQUHMAKGJWh0dHA6Ly9wa2kuZ29vZy9y
ZXBvL2NlcnRzL2d0czFjMy5kZXIwGQYDVR0RBBIwEIIOd3d3Lmdvb2dsZS5jb20w
IQYDVR0gBBowGDAIBgZngQwBAgEwDAYKKwYBBAHWeQIFAzA8BgNVHR8ENTAzMDGg
L6AthitodHRwOi8vY3Jscy5wa2kuZ29vZy9ndHMxYzMvbW9WRGZJU2lhMmsuY3Js
MIIBAwYKKwYBBAHWeQIEAgSB9ASB8QDvAHUARqVV63X6kSAwtaKJafTzfREsQXS+
/Um4havy/HD+bUcAAAF9nHS84wAABAMARjBEAiBlnKeyJxN31CThxKdYlgL/rz9j
7gJEPJLgtE7/92Q6uAIgfA2qhDtkP3PplBrsiPLzDPuRxbmQWVFm6AG/J/A3zOMA
dgDfpV6raIJPH2yt7rhfTj5a6s2iEqRqXo47EsAgRFwqcwAAAX2cdLy/AAAEAwBH
MEUCIQCRahdv0d2654opa0R8jkkti8ToWnbQwu6qQP+G/yBjWAIgeM/PmAChWclb
c0.pem
```


Figure 9 c0.pem -Server's Certificate Authority


```
-----BEGIN CERTIFICATE-----
MIIFljCCA36gAwIBAgINAgO8U1lrNMcY9QFQZjANBgkqhkiG9w0BAQsFADBHMQsw
CQYDVQQGEwJVUzEiMCAGA1UEChMZR29vZ2xlIFRydXN0IFNlcnZpY2VzIExMQzEU
MBIGA1UEAxMLR1RTIFJvb3QgUjEwHhcNMjAwODEzMDAwMDQyWhcNMjcwOTMwMDAw
MDQyWjBGMQswCQYDVQQGEwJVUzEiMCAGA1UEChMZR29vZ2xlIFRydXN0IFNlcnZp
Y2VzIExMQzETMBEGA1UEAxMKR1RTIENBIDFDMzCCASIwDQYJKoZIhvcNAQEBBQAD
ggEPADCCAQoCggEBAPWI3+dijB43+DdCkH9sh9D7ZYIl/ejLa6T/belaI+KZ9hzp
kgOZE3wJCor6QtZeViSqejOEH9Hpabu5dOxXTGZok3c3VVP+ORBNtzS7XyV3NzsX
lOo85Z3VvMO0Q+sup0fvsEQRY9i0QYXdQTBIkxu/t/bgRQIh4JZCF8/ZK2VWNAcm
BA2o/X3KLu/qSHw3TT8An4Pf73WELnlXXPxXbhqW//yMmqaZviXZf5YsBvcRKgKA
gOtjGDxQSYflispfGStZloEAoPtR28p3CwvJlk/vcEnHXG0g/Zm0tOLKLnf9LdwL
tmsTDIwZKxeWmLnwi/agJ7u2441Rj72ux5uxiZ0CAwEAAaOCAYAwggF8MA4GA1Ud
DwEB/wQEAwIBhjAdBgNVHSUEFjAUBggrBgEFBQcDAQYIKwYBBQUHAwIwEgYDVR0T
AQH/BAgwBgEB/wIBADAdBgNVHQ4EFgQUinR/r4XN7pXNPZzQ4kYU83E1HScwHwYD
VR0jBBgwFoAU5K8rJnEaK0gnhS9SZizv8IkTcT4waAYIKwYBBQUHAQEEXDBaMCYG
CCsGAQUFBzABhhpodHRwOi8vb2NzcC5wa2kuZ29vZy9ndHNyMTAwBggrBgEFBQcw
AoYkaHR0cDovL3BraS5nb29nL3JlcG8vY2VydHMvZ3RzcjEuZGVyMDQGA1UdHwQt
MCswKaAnoCWGI2h0dHA6Ly9jcmwucGtpLmdvb2cvZ3RzcjEvZ3RzcjEuY3JsMFcG
A1UdIARQME4wOAYKKwYBBAHWeQIFAzAqMCgGCCsGAQUFBwIBFhxodHRwczovL3Br
c1.pem
```


Figure 10 c1.pem – Immediate CA, Issuer's Certificate Authority

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

```
[01/19/22]seed@VM:~/.../TG6$ openssl x509 -in c1.pem -noout -modulus
Modulus=F588DFE7628C1E37F83742907F6C87D0FB658225FDE8CB6BA4FF6DE95A23E299F61CE9920399137C090A8AFA42
D65E5624AA7A33841FD1E969BBB974EC574C66689377375553FE39104DB734BB5F2577373B1794EA3CE59DD5BCC3B443EB
2EA747EFB0441163D8B44185DD413048931BBFB7F6E0450221E0964217CFD92B6556340726040DA8FD7DCA2EEFEA487C37
4D3F009F83DFEF75842E79575CFC576E1A96FFFC8C9AA699BE25D97F962C06F7112A028080EB63183C504987E58ACA5F19
2B59968100A0FB51DBCA770B0BC9964FEF7049C75C6D20FD99B4B4E2CA2E77FD2DDC0BB66B130C8C192B179698B9F08BF6
A027BBB6E38D518FBDAEC79BB1899D
[01/19/22]seed@VM:~/.../TG6$ openssl x509 -in c1.pem -text -noout | grep exponent
[01/19/22]seed@VM:~/.../TG6$ openssl x509 -in c1.pem -text -noout | grep Exponent
                Exponent: 65537 (0x10001)
```

Figure 11 Modulus(n) and E =65537 (0x10001)

Step 3 - Extract the signature from the server's certificate

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

```
    Signature Algorithm: sha256WithRSAEncryption
         72:ea:fb:c8:a1:ee:36:ad:22:1f:9d:62:56:13:ef:ad:85:3e:
         11:2e:11:a5:35:c6:58:9d:b9:26:62:98:55:b0:3d:4b:78:a4:
         eb:c0:58:cf:46:e2:ff:76:33:39:ac:de:69:56:e7:10:5e:99:
         09:b5:a2:e7:24:34:22:8e:ec:12:9e:44:e5:f1:12:7d:e8:d1:
         55:90:9e:03:4d:84:b6:fe:af:a2:05:82:73:f1:aa:f7:d6:61:
         ba:15:03:c4:dc:af:5a:b7:af:92:df:57:39:fa:ca:06:48:7a:
         67:78:4d:d2:15:68:e0:77:15:3f:de:67:ee:6e:f6:be:ed:99:
         d2:c7:79:35:bd:c9:fc:cc:da:c7:7e:6f:48:5d:96:f7:cd:47:
         6b:1e:e3:82:1c:de:e1:5c:44:87:c7:b2:86:f1:16:fa:da:0e:
         73:6d:9a:e3:5b:08:ed:63:53:6f:1f:99:3f:3a:98:6a:c0:ea:
         22:e9:c3:55:ba:fb:c1:6a:c2:70:70:0d:30:36:f5:a3:68:ff:
         f8:b8:49:d2:27:97:78:10:9f:06:51:0d:bc:75:44:7f:66:ab:
         6c:13:cd:47:a6:3b:75:bc:66:1e:b3:48:08:8d:72:a4:85:be:
         9b:44:db:08:4b:00:f0:df:f1:11:39:71:dc:0b:05:79:05:23:
         db:55:15:84
[01/19/22]seed@VM:~/.../TG6$ vim signature_algorithm
```

Figure 12 Signature Block

```
[01/19/22]seed@VM:~/.../TG6$ cat signature_algorithm | tr -d '[:space:]:'
72eafbc8a1ee36ad221f9d625613efad853e112e11a535c6589db926629855b03d4b78a4ebc058cf46e2ff763339acde69
56e7105e9909b5a2e72434228eec129e44e5f1127de8d155909e034d84b6feafa2058273f1aaf7d661ba1503c4dcaf5ab7
af92df5739faca06487a67784dd21568e077153fde67ee6ef6beed99d2c77935bdc9fcccdac77e6f485d96f7cd476b1ee3
821cdee15c4487c7b286f116fada0e736d9ae35b08ed63536f1f993f3a986ac0ea22e9c355bafbc16ac270700d3036f5a3
68fff8b849d2279778109f06510dbc75447f66ab6c13cd47a63b75bc661eb348088d72a485be9b44db084b00f0dff11139
71dc0b05790523db551584[01/19/22]seed@VM:~/.../TG6$ cat signature_algorithm | tr -d '[:space:]:' >
sign_withoutcolons
```

Figure 13 Pasting it into a file to use it later to verify the certificate

Step 4 - Extract the Body of the server's certificate

We use the following command to extract the body of the certificate

```
[01/19/22]seed@VM:~/.../TG6$ openssl asn1parse -i -in c0.pem
    0:d=0  hl=4 l=1159 cons: SEQUENCE
    4:d=1  hl=4 l= 879 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=  17 prim:   INTEGER           :F4BB5A3A7A382C680A0000000127DE3D
   32:d=2  hl=2 l=  13 cons:   SEQUENCE
   34:d=3  hl=2 l=   9 prim:    OBJECT            :sha256WithRSAEncryption
   45:d=3  hl=2 l=   0 prim:    NULL
   47:d=2  hl=2 l=  70 cons:   SEQUENCE
   49:d=3  hl=2 l=  11 cons:    SET
   51:d=4  hl=2 l=   9 cons:     SEQUENCE
   53:d=5  hl=2 l=   3 prim:      OBJECT            :countryName
   58:d=5  hl=2 l=   2 prim:      PRINTABLESTRING   :US
   62:d=3  hl=2 l=  34 cons:    SET
   64:d=4  hl=2 l=  32 cons:     SEQUENCE
```

Figure 14 Start of the certificate

```
  624:d=4  hl=4 l= 259 cons:     SEQUENCE
  628:d=5  hl=2 l=  10 prim:      OBJECT            :CT Precertificate SCTs
  640:d=5  hl=3 l= 244 prim:      OCTET STRING      [HEX DUMP]:0481F100EF00750046A555EB75FA912030B
5A28969F4F37D112C4174BEFD49B885ABF2FC70FE6D470000017D9C74BCE30000040300463044022065 9CA7B2271377D42
4E1C4A7589602FFAF3F63EE02443C92E0B44EFFF7643AB802207C0DAA843B643F73E9941AEC88F2F30CFB91C5B99059516
6E801BF27F037CCE3007600DFA55EAB68824F1F6CADEEB85F4E3E5AEACDA212A46A5E8E3B12C020445C2A730000017D9C7
4BCBF00000403004730450221009 16A176FD1DDBAE78A296B447C8E492D8BC4E85A76D0C2EEAA40FF86FF206358022078C
FCF9800A159C95B2E81A6CAC8C78B001BFCD0F68DFE86025E83CFA2DA5955
  887:d=1  hl=2 l=  13 cons:  SEQUENCE
  889:d=2  hl=2 l=   9 prim:   OBJECT            :sha256WithRSAEncryption
  900:d=2  hl=2 l=   0 prim:   NULL
  902:d=1  hl=4 l= 257 prim:  BIT STRING
```

Figure 15 "sha256WithRSAEncryption" – Line 889

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.

So the body of the certificate is from 4 to 887

Calculating the hash of the body of the certificate using sha256sum.

```
[01/19/22]seed@VM:~/.../TG6$ openssl asn1parse -i -in c0.pem -strparse 4 -out c0_body.bin -noout
[01/19/22]seed@VM:~/.../TG6$ sha256sum c0
c0_body.bin  c0.pem
[01/19/22]seed@VM:~/.../TG6$ sha256sum c0_body.bin
9b71d33565fa095b8555c2ee50fd2b6c59de5eca60abaddae6b2265684986927  c0_body.bin
```

Figure 16 body of the certificate - hash

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.

```
[01/19/22]seed@VM:~/.../TG6$ vim verfied_message.c
[01/19/22]seed@VM:~/.../TG6$ gcc -o verfied_message verfied_message.c -lcrypto
[01/19/22]seed@VM:~/.../TG6$ ./verfied_message
Original Message  9B71D33565FA095B8555C2EE50FD2B6C59DE5ECA60ABADDAE6B2265684986927
Value of C Computed:  01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF003031300D0609608648016503040201050004209B71D33565FA095B8555C2
EE50FD2B6C59DE5ECA60ABADDAE6B2265684986927
[01/19/22]seed@VM:~/.../TG6$
```

Figure 17 As you can see the signatures match

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.