**ACS 54500 Cryptography and Network Security**

Lab 10: RSA Encryption and Signature Lab

**Name**: Vijayagiridharan Subramanian

**Task 1: Deriving the Private Key– 10 pts**

**Code:**

#include <stdio.h>

#include <openssl/bn.h>

#define NBITS 128

void printBN(char \*msg, BIGNUM \*a)

{ /\* Use BN\_bn2hex(a) for hex string

\* Use BN\_bn2dec(a) for decimal string \*/

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();

BIGNUM \*q = BN\_new();

BIGNUM \*fai\_n = BN\_new();

BIGNUM \*n = BN\_new();

BIGNUM \*e = BN\_new();

BIGNUM \*d = BN\_new();

BIGNUM \*p\_1 = BN\_new();

BIGNUM \*q\_1 = BN\_new();

BN\_hex2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF");

BN\_hex2bn(&q, "E85CED54AF57E53E092113E62F436F4F");

BN\_hex2bn(&e, "0D88C3");

BN\_sub(p\_1, p, BN\_value\_one());

BN\_sub(q\_1, q, BN\_value\_one());

BN\_mul(n, p, q, ctx);

BN\_mul(fai\_n, p\_1, q\_1, ctx);

//printBN("fai\_n=", fai\_n);

BN\_mod\_inverse(d, e, fai\_n, ctx);

printBN("public key e=\t", e);

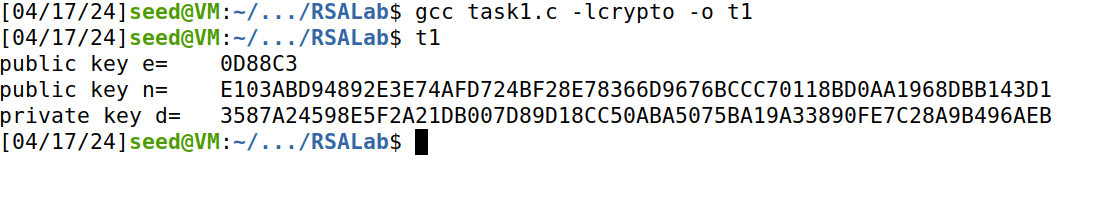
printBN("public key n=\t", n);

printBN("private key d=\t", d);

return 0;

}

This above C program **task1.c** uses OpenSSL's BIGNUM library to generate RSA public and private keys. It initializes variables for key components and context using BIGNUM data types, sets values for primes p and q , and public exponent e . Then, it calculates modulus n , Euler's totient function ( phi(n)), and derives private exponent d using a modular inverse. Finally, it prints the public key (e, n) and private key ( d ).

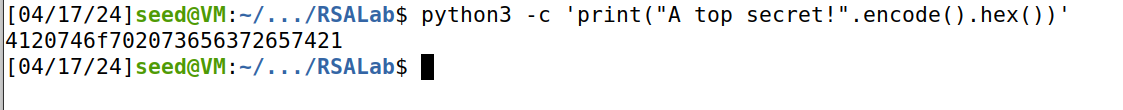


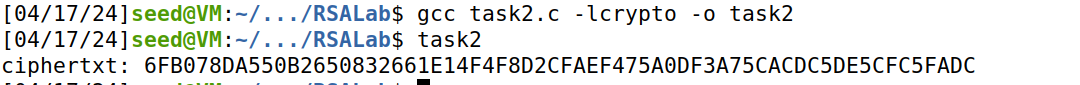
The command compiles the C program "task1.c" with OpenSSL's crypto library and names the executable "t1". While running t1 we get public key e and n and as well as the private key d.

**Task 2: Encrypting a Message – 10 pts**

****

The C program "**task2.c**" implements RSA encryption and decryption using OpenSSL's BIGNUM library. It initializes variables for key components and message/ciphertext using BIGNUM data type, then performs modular exponentiation for encryption. The program prints out the resulting ciphertext after encryption.



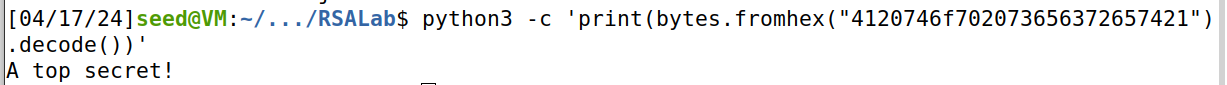


The Python command **python3 -c 'print("A top secret!".encode().hex())'** converts the string **"A top secret!"** into its hexadecimal representation. We are using that value the code.The C compiler command compiles **"task2.c"** with OpenSSL's crypto library and generates the executable **"task2"**. Executing **"task2"** encrypts the hexadecimal representation of "A top secret!" using RSA and prints out the resulting ciphertext.

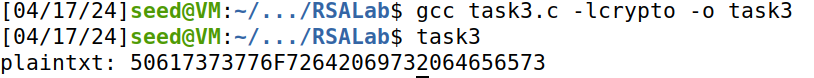
**Task 3: Decrypting a Message – 10 pts**



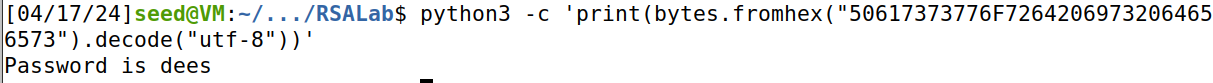
The C program "**task3.c**" implements RSA decryption using OpenSSL's BIGNUM library. It initializes variables for key components, ciphertext, and plaintext using BIGNUM data type. The program decrypts the given ciphertext using the private key and prints out the resulting plaintext.



The Python command converts the hexadecimal representation "**4120746f702073656372657421**" into the string **"A top secret!**".



The C compiler command compiles "**task3.c**" with OpenSSL's crypto library and generates the executable "**task3**". Executing "**task3**" decrypts the ciphertext "**8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F**" using RSA and prints out the resulting plaintext.



The Python command converts the hexadecimal representation "**50617373776F72642069732064656573**" into the string "**Password is dees**".

**Task 4: Signing a Message – 10 pts**

**Code:**

#include <stdio.h>

#include <openssl/bn.h>

#define NBITS 128

void printBN(char \*msg, BIGNUM \*a)

{ /\* Use BN\_bn2hex(a) for hex string

\* Use BN\_bn2dec(a) for decimal string \*/

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 \*n = BN\_new();

BIGNUM \*e = BN\_new();

BIGNUM \*d = BN\_new();

BIGNUM \*m1 = BN\_new();

BIGNUM \*m2 = BN\_new();

BIGNUM \*sig1 = BN\_new();

BIGNUM \*sig2 = BN\_new();

BN\_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");

BN\_hex2bn(&e, "010001");

BN\_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

BN\_hex2bn(&m1, "49206f776520796f75202432303030");

BN\_hex2bn(&m2, "49206f776520796f75202433303030");

BN\_mod\_exp(sig1, m1, d, n, ctx);

BN\_mod\_exp(sig2, m2, d, n, ctx);

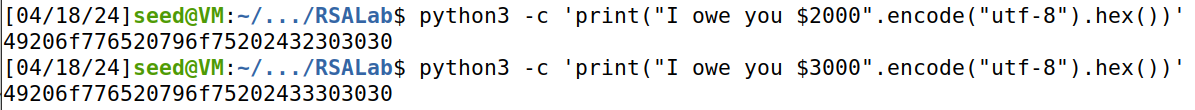
printBN("signature of m1:", sig1);

printBN("signature of m2:", sig2);

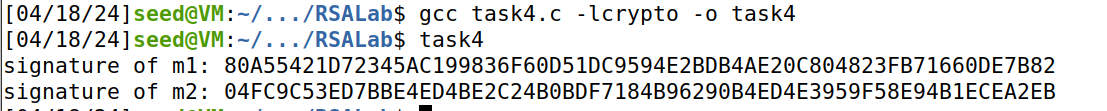
return 0;

}

The C program "task4.c" demonstrates RSA signature generation using OpenSSL's BIGNUM library.It initializes variables for key components, messages, and signatures using BIGNUM data type.The program calculates the signatures of two different messages using the private key and modulus, then prints out the signatures.



The 1st Python command converts the string "**I owe you $2000**" into its hexadecimal representation. The 2nd Python command converts the string **"I owe you $3000**" into its hexadecimal representation.



The C compiler command compiles "**task4.c**" with OpenSSL's crypto library and generates the executable "**task4**". Executing **"task4"** calculates and prints the RSA signatures of the two hexadecimal-encoded messages.

**Task 5: Verifying a Signature – 10 pts**

**Code:**

#include <stdio.h>

#include <openssl/bn.h>

#define NBITS 128

void printBN(char \*msg, BIGNUM \*a)

{ /\* Use BN\_bn2hex(a) for hex string

\* Use BN\_bn2dec(a) for decimal string \*/

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 \*n = BN\_new();

BIGNUM \*e = BN\_new();

BIGNUM \*M = BN\_new();

BIGNUM \*m1 = BN\_new();

BIGNUM \*m2 = BN\_new();

BIGNUM \*sig1 = BN\_new();

BIGNUM \*sig2 = BN\_new();

%TRHrnT#ARGXB

BN\_hex2bn(&M, "4c61756e63682061206d697373696c652e");

BN\_hex2bn(&n, "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");

BN\_hex2bn(&e, "010001");

BN\_hex2bn(&sig1, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");

BN\_hex2bn(&sig2, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F");

BN\_mod\_exp(m1, sig1, e, n, ctx);

BN\_mod\_exp(m2, sig2, e, n, ctx);

printf("verifying of signature1:");

if (BN\_cmp(m1, M) == 0)

{

printf("valid!\n");

}

else

{

printf("invalid!\n");

}

printf("verifying of signature2:");

if (BN\_cmp(m2, M) == 0)

{

printf("valid!\n");

}

else

{

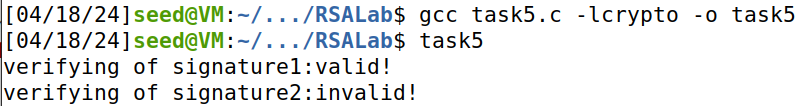
printf("invalid!\n");

}

return 0;

}

The C program "**task5.c**" demonstrates RSA signature verification using OpenSSL's BIGNUM library. It initializes variables for key components, message, signatures, and temporary variables using BIGNUM data type. The program calculates the decryption of signatures using the public key, then compares the decrypted signatures with the original message to verify their validity. It prints out whether each signature is valid or invalid.



The C compiler command compiles "**task5.c**" with OpenSSL's crypto library and generates the executable "**task5**". Executing "**task5**" verifies the validity of two RSA signatures by comparing their decrypted values with the original message. It prints out whether each signature is valid or invalid based on the verification result.

**Task 6: Manually Verifying an X.509 Certificate – 50 pts**

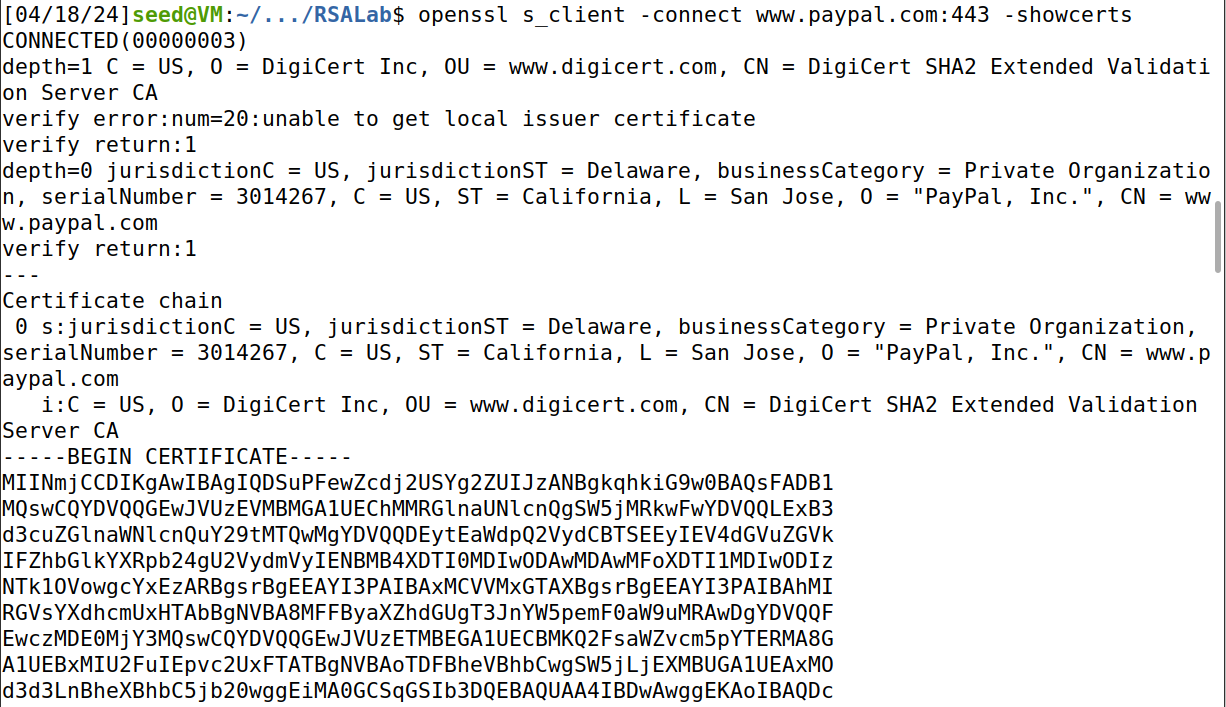
We use “ www.paypal.com” as the targeted web server.

**Step 1: Download a certificate from a real web server. – 10 pts**

The result of the command contains two certificates. The subject field (the entry starting with s:) of the certificate is **www.paypal.com**, i.e. This is **www.paypal.com**’s certificate. The issuer field (the entry starting with i:) provides the issuer’s information.

The command `**openssl s\_client -connect www.paypal.com:443 -showcerts**` establishes a secure connection to www.paypal.com on port 443 and displays its SSL/TLS certificate chain.

**You can see 1st certificate:**



The subject field of the second certificate is the same as the issuer field of the first certificate. Basically, the second certificate belongs to an intermediate CA. In this task, we will use CA’s certificate to verify a server certificate.

**You can see the second certificate:**



Copy and paste each of the certificates (the text between the line containing "Begin CERTIFICATE" and the line containing "END CERTIFICATE", including these two lines) to file. Let us call the first one c0.pem and the second one c1.pem.



**Step 2: Extract the public key (e, n) from the issuer’s certificate. – 10 pts**

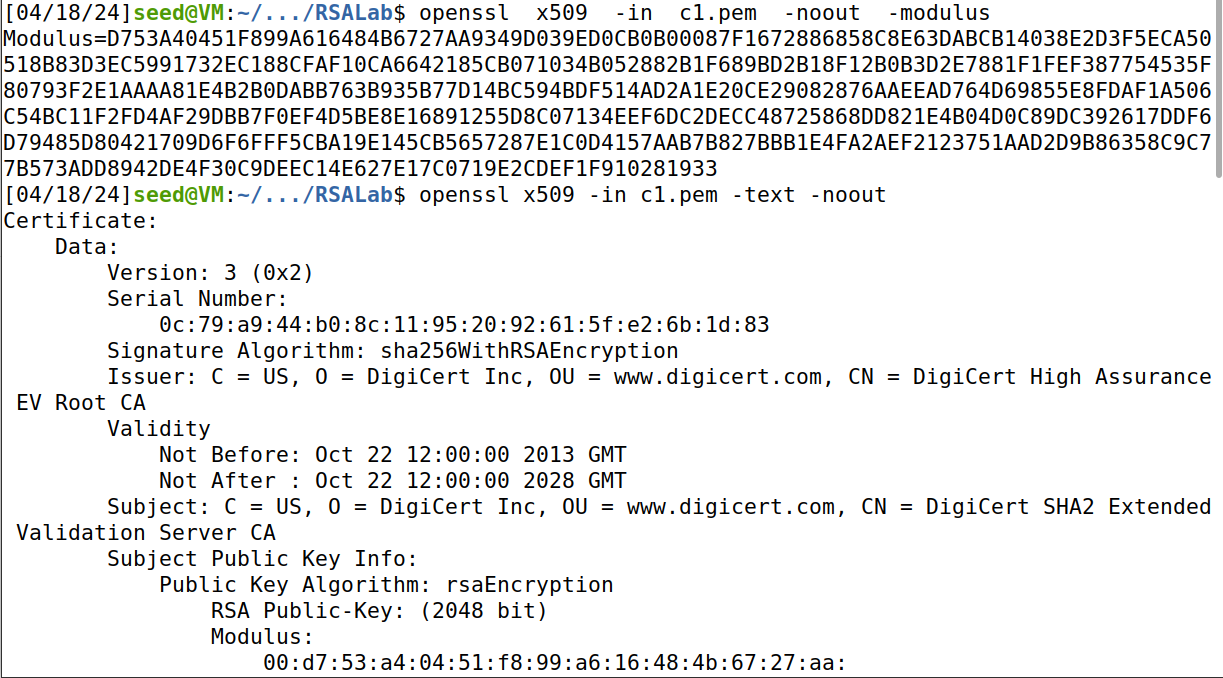
When I type this command **openssl x509 -in c1.pem -noout -modulus**, we can get the modulus value (n).

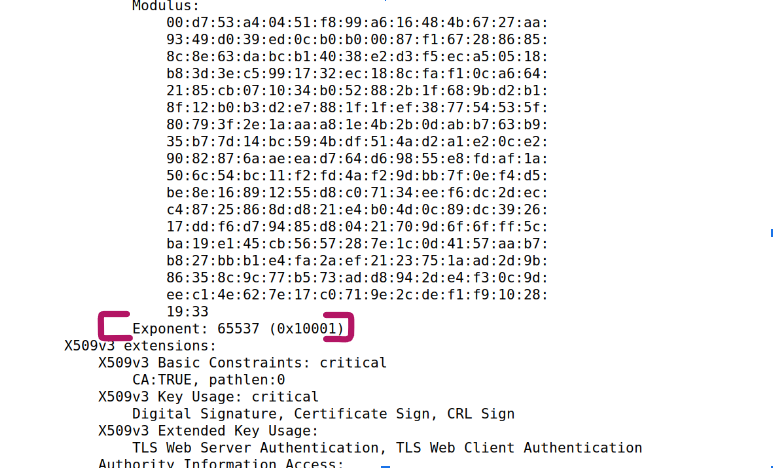
Now we are extracting the public key (e, n) from the issuer certificate.

This is the hexadecimal value of n=



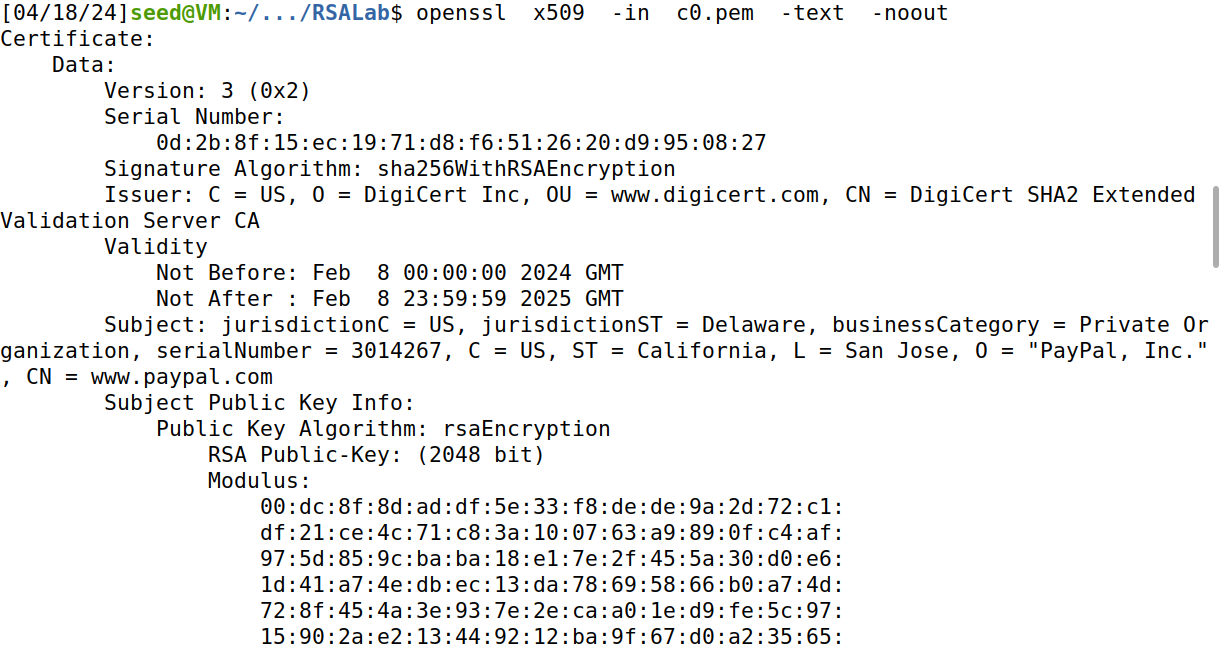
When we enter the command **openssl x509 -in c1.pem -text -noout,** we can get the value of the exponent **(0 X 10001**). e=0X10001

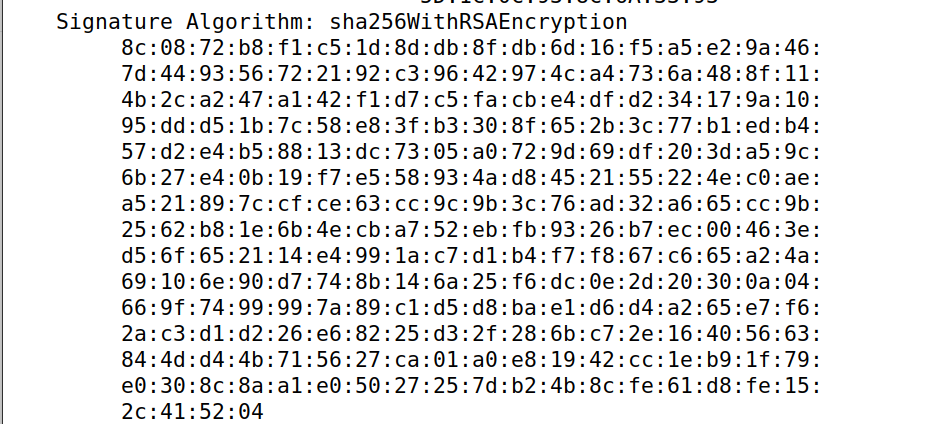




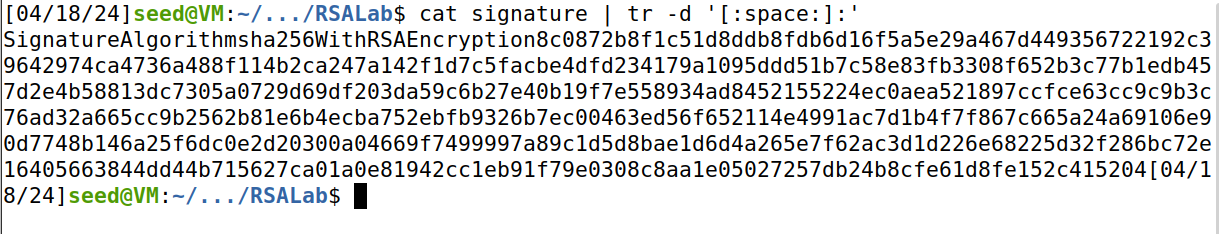
**Step 3: Extract the signature from the server’s certificate. – 10 pts**

When we type the command **openssl x509 -in c0.pem -text -nout ,** it prints out all fields and we can find the signature block at the end.



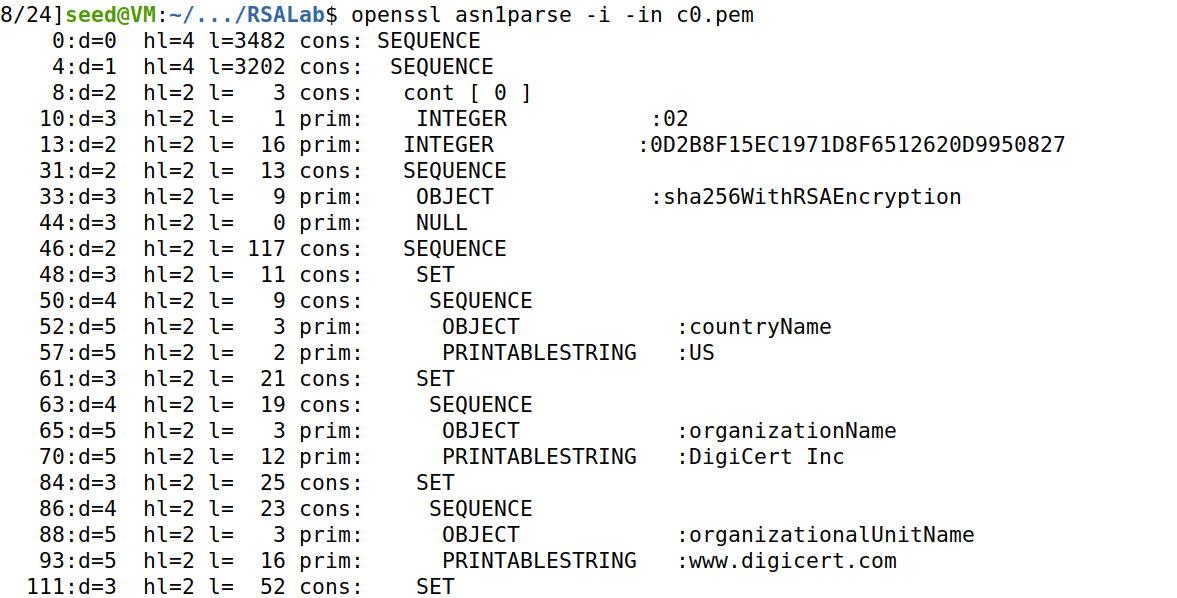


Then we copy that signature block into signature file and when we open it through **cat signature | tr -d ‘[:space:]:’** it helps us to remove the spaces, colons and fake numbers from the data so that we can get a 16-into string.

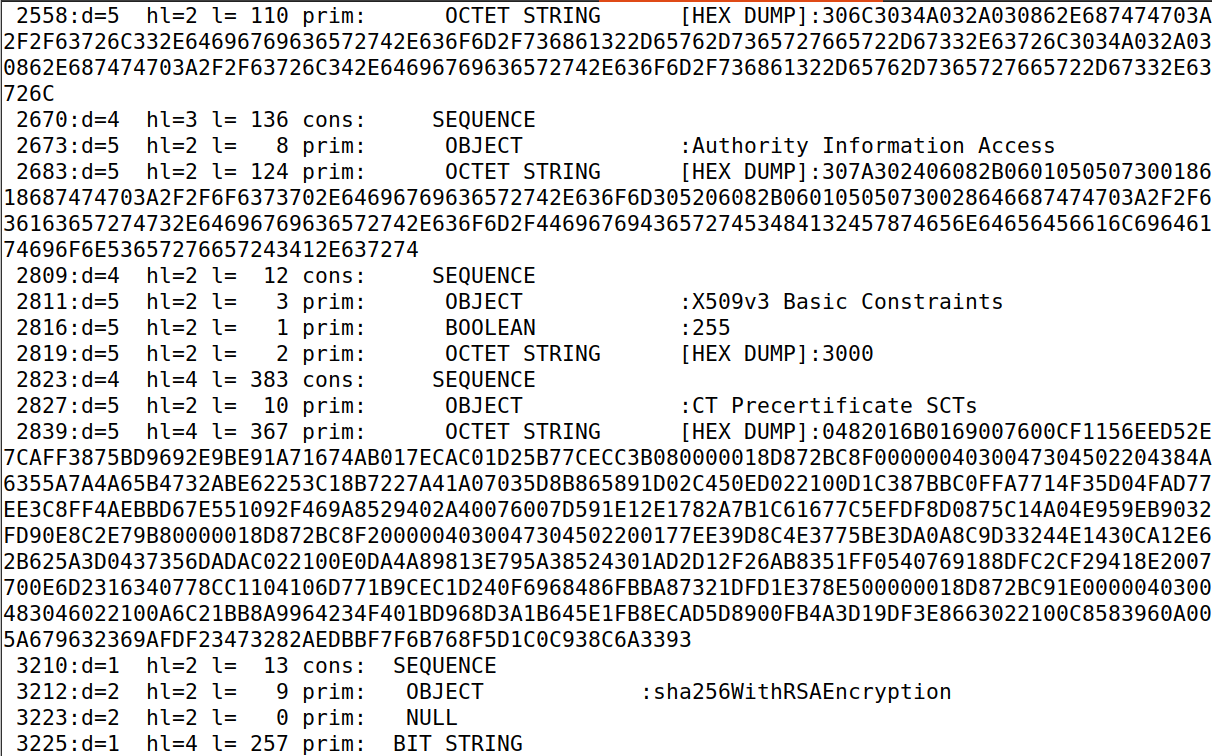
****

**Step 4: Extract the body of the server’s certificate. – 10 pts**

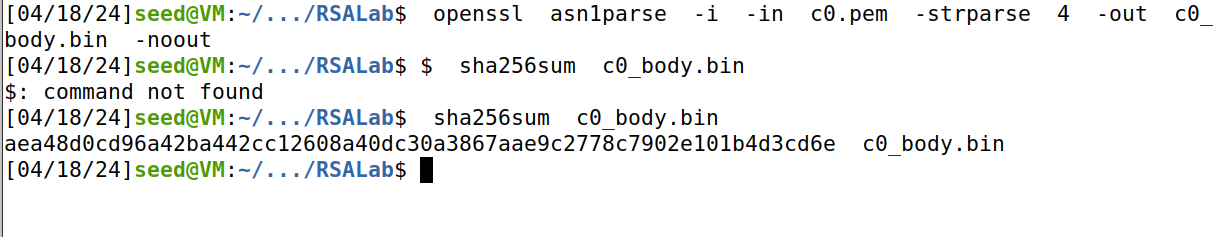
The command openssl **asn1parse -i -in c0.pem** parses the ASN.1 structure of the file "c0.pem", displaying its components. It reveals detailed information such as sequence, integers, objects, and octet strings. This output provides insights into the structure and content of the PEM file.







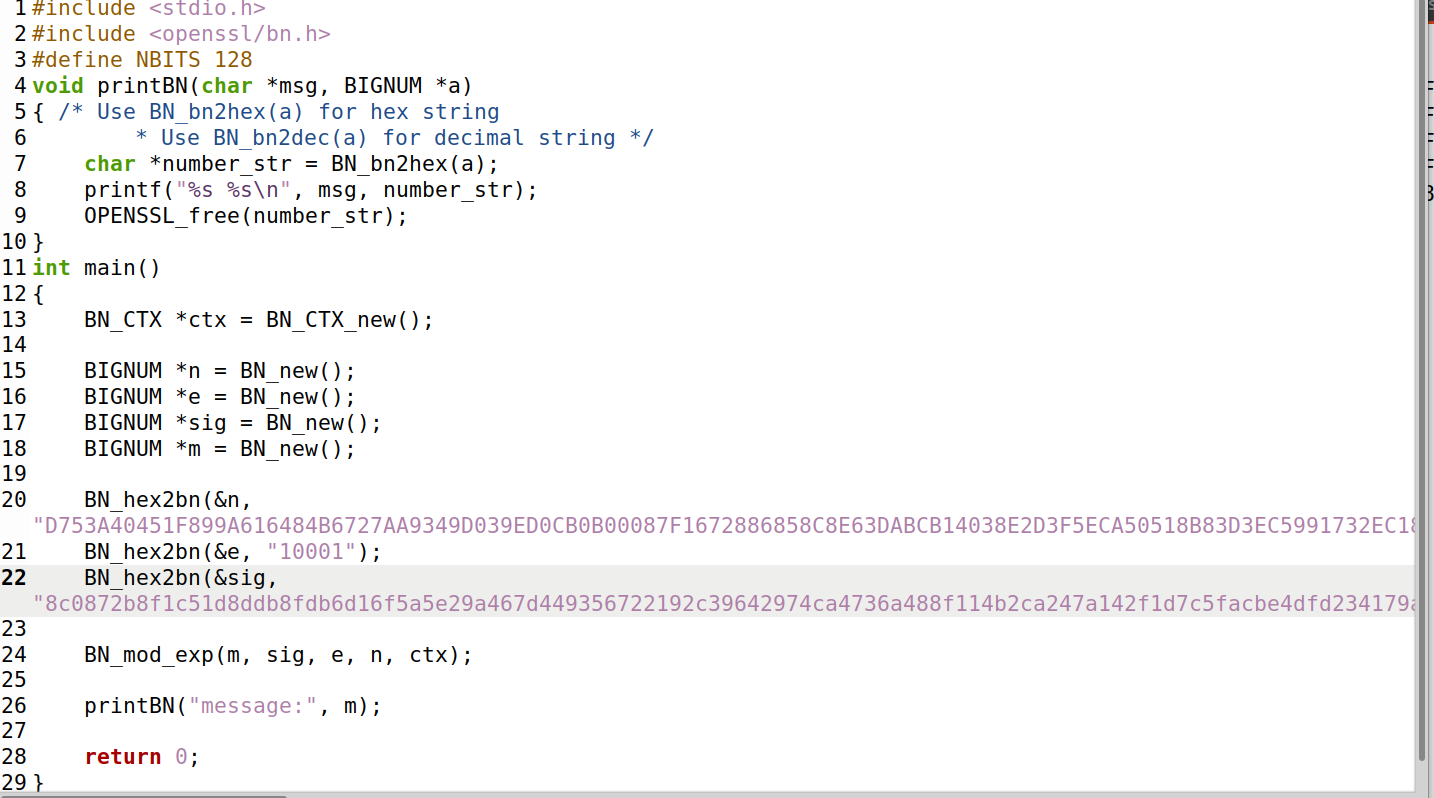
The command **openssl asn1parse -i -in c0.pem -strparse 4 -out c0\_body.bin -noout** extracts the body of the ASN.1 structure from "c0.pem" and saves it as "c0\_body.bin". Then it calculates the SHA256 hash of the binary file "c0\_body.bin", providing a unique identifier for its content. The hash value is "**aea48d0cd96a42ba442cc12608a40dc30a3867aae9c2778c7902e101b4d3cd6e**".



**Step 5: Verify the signature. – 10 pts**

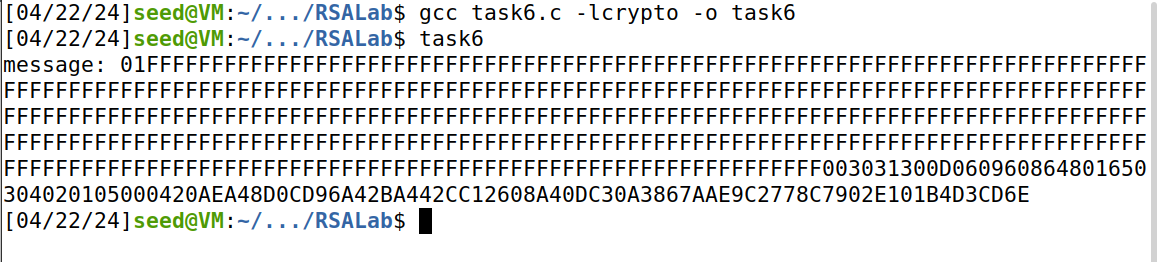
The below is the command for verifying the signature. I replaced the values of n, e and signature values that we got before.

**Code:**



The code task6.c sets up variables for RSA decryption, like the modulus, public exponent, signature, and the recovered message. It decrypts the signature using modular exponentiation with the public key and modulus, saving the outcome as the recovered message. Lastly, it displays the recovered message in hexadecimal form using a custom printing function.

The command prompt compiles "**task6.c**" with GCC and OpenSSL's crypto library, generating the executable "**task6**". Upon execution of **"task6**", it demonstrates RSA decryption and displays the recovered message in hexadecimal format.



The script checks if the last 64 characters of string `m` (a 256-bit hash in hex format) match a given hash value in variable `hash`. It outputs `True` for a match, indicating validity, and `False` for no match, indicating an error or alteration.

The verification is successful and the signature is valid for the correct hash value and prints ”True”.

