

Task1: Deriving the Private Key

Code for task1

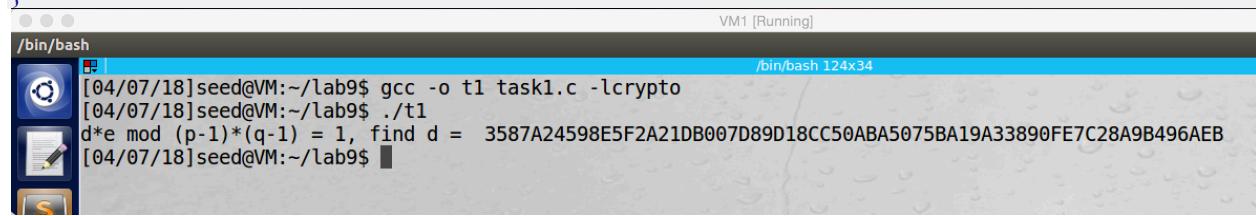
```
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256

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 *one = BN_new();
    BIGNUM *p = BN_new();
    BIGNUM *q = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *x = BN_new();
    BIGNUM *d = BN_new();

    // Initialize p, q, e, n, and one
    BN_hex2bn (&one, "00001"); //hex value of number 1
    BN_hex2bn (&p, "F7E75FDC469067FFDC4E847C51F452DF");
    BN_hex2bn (&q, "E85CED54AF57E53E092113E62F436F4F");
    BN_hex2bn (&e, "0D88C3");

    //calculate d by formula  $e \cdot d \pmod{(p-1) \cdot (q-1)} = 1$ 
    BN_sub (p, p, one); // p=p-1
    BN_sub (q, q, one); // q=q-1
    BN_mul (x, p, q, ctx); //x=(p-1)*(q-1)
    BN_mod_inverse(d, e, x, ctx); // calculate d from  $e \cdot d \pmod{(p-1) \cdot (q-1)} = 1$ 
    //print result
    printBN("d*e mod (p-1)*(q-1) = 1, find d = ", d);
    return 0;
}
```



screenshot1. After run the above code, we got result of d.

Observation and Explanation:

For this task, we are given p, q, and e, and we need to find d. Therefore, we use Extended Euclidian Algorithm: $e \cdot d \bmod (p-1) \cdot (q-1) = 1$. After we compile and run the code, we got result of d which is shown in screenshot1.

Task2: Encrypting a Message

Code for task 2

```
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256

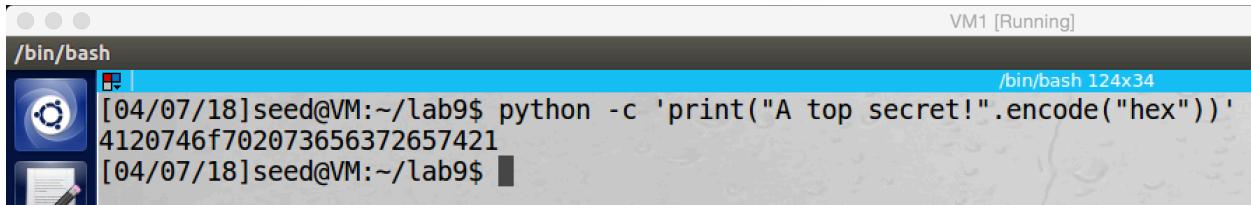
void printBN(char *msg, BIGNUM * a)
{
    /* Use BN_bn2hex(a) for hex string
     * Use BN_bn2dec(a) for decimal string
     */
    char * number_str = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str);
    OPENSSL_free(number_str);
}

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

    // Initialize e, n, d, M
    BN_hex2bn (&M, "4120746f702073656372657421");
    BN_hex2bn (&d,
"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BN_hex2bn (&n,
"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn (&e, "010001");

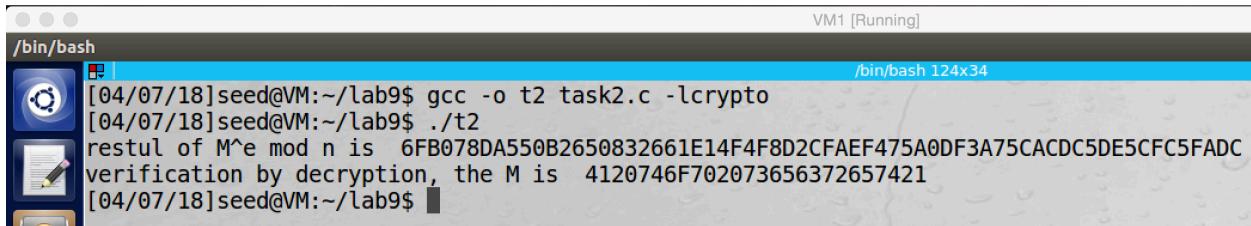
    //calculate ciphertext C by formula M^e mod n
    BN_mod_exp(C, M, e, n, ctx);
    printBN("restul of M^e mod n is ", C);
    // result verification, decrypting the ciphertext by formula C^d mod n
    BN_mod_exp(M, C, d, n, ctx);
    printBN("verification by decryption, the M is ", M);

    return 0;
}
```



```
VM1 [Running]
/bin/bash
[04/07/18]seed@VM:~/lab9$ python -c 'print("A top secret!".encode("hex"))'
4120746f702073656372657421
[04/07/18]seed@VM:~/lab9$
```

screenshot1. Using python to convert “A top secret!” to hex value



```
VM1 [Running]
/bin/bash
[04/07/18]seed@VM:~/lab9$ gcc -o t2 task2.c -lcrypto
[04/07/18]seed@VM:~/lab9$ ./t2
restul of M^e mod n is 6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC
verification by decryption, the M is 4120746F702073656372657421
[04/07/18]seed@VM:~/lab9$
```

screenshot2. Compile and run task2.c program, we get ciphertext. Moreover, we also decrypt the ciphertext, and we get a M, which is exactly same as the original M.

Observation and Explanation:

In this task, we are given n, e, d, and M. we use formula ($M^e \bmod n$) to get cipher text C. And then we use ($C^d \bmod n$) to get M for verification.

First, we convert the string M to hex value (screenshot1). And then we compile and run task2.c. As the screenshot2 shows, by running the program, we successfully get the cipher text C. And then we also use the private key to decrypt the cipher text, and we get message M, which is exactly same as the original M.

Task3: Decrypting a Message

Code for task3:

```
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256

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 *e = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *C = BN_new();
    BIGNUM *M = BN_new();
```

```

    // Initialize e, n, d, C
    BN_hex2bn (&C,
    "8C0F971DF2F3672B28811407E2DABBE1DA0FEBBDFC7DCB67396567EA1E2493F");
    BN_hex2bn (&d,
    "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BN_hex2bn (&n,
    "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn (&e, "010001");

    //decoding ciphertext C to get message M by formula C^d mod n
    BN_mod_exp(M, C, d, n, ctx);
    printBN("decryption by C^d mod n, the M is ", M);

    return 0;
}

```

```

bash
[04/07/18]seed@VM:~/lab9$ gcc -o t3 task3.c -lcrypto
[04/07/18]seed@VM:~/lab9$ ./t3
decryption by C^d mod n, the M is 50617373776F72642069732064656573
[04/07/18]seed@VM:~/lab9$ python -c 'print("50617373776F72642069732064656573".decode("hex"))'
Password is dees
[04/07/18]seed@VM:~/lab9$ 

```

screenshot1. Decrypting the cipher text by the private key and n, and then we get hex value of the message. Then we use python to decode the hex value, and we get the message “Password is dees”

Observation and Explanation:

In this task, we are given e, n, d, and C. We use the formula $(C^d \bmod n)$ to decrypt the cipher text.

As the screenshot1 shows, we compile and run the program task3.c, and then we get the hex value of the M. Afterwards, we run python to decode the hex value. Finally, we get the message which is “Password is dees”.

Task4: Signing a Message

```

#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256

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 *e = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *C1 = BN_new();
    BIGNUM *C2 = BN_new();
    BIGNUM *M1 = BN_new();
    BIGNUM *M2 = BN_new();

    // initialize M1, M2, d, n, e
    BN_hex2bn (&M1, "49206f776520796f752024323030302e");
    BN_hex2bn (&M2, "49206f776520796f752024333030302e");
    BN_hex2bn (&d,
"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BN_hex2bn (&n,
"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn (&e, "010001");

    //generate signature C1 by M1^d mod n
    BN_mod_exp(C1, M1, d, n, ctx); // generate signature for "I owe you $2000"
    printBN("The signature for I owe you $2000. is ", C1);
    BN_mod_exp(M1, C1, e, n, ctx); // answer verification
    printBN("Verification by decryption ", M1);

    //generate signature C2 by M2^d mod n
    BN_mod_exp(C2, M2, d, n, ctx); // generate signature for "I owe you $3000"
    printBN("The signature for I owe you $3000. is ", C2);
    BN_mod_exp(M2, C2, e, n, ctx); // answer verification
    printBN("Verification by decryption ", M2);

    return 0;
}

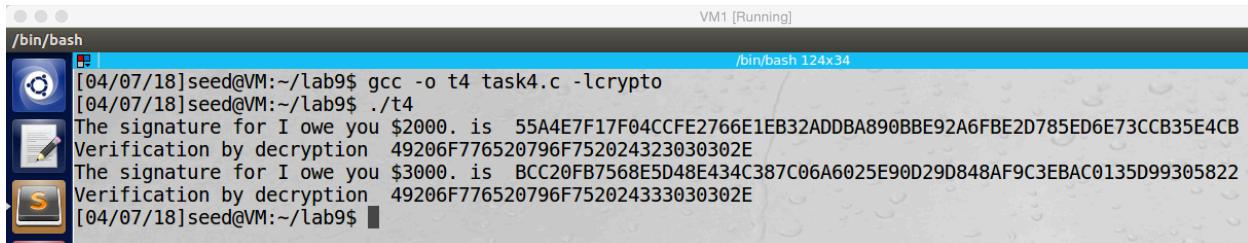
```

```

VM1 [Running]
/bin/bash
[04/07/18]seed@VM:~/lab9$ python -c 'print("I owe you $2000.".encode("hex"))'
49206f776520796f752024323030302e
[04/07/18]seed@VM:~/lab9$ python -c 'print("I owe you $3000.".encode("hex"))'
49206f776520796f752024333030302e
[04/07/18]seed@VM:~/lab9$ 

```

screenshot1. Convert messages into hex value



```
[04/07/18]seed@VM:~/lab9$ gcc -o t4 task4.c -lcrypto
[04/07/18]seed@VM:~/lab9$ ./t4
The signature for I owe you $2000. is 55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73CCB35E4CB
Verification by decryption 49206F776520796F752024323030302E
The signature for I owe you $3000. is BCC20FB7568E5D48E434C387C06A6025E90D29D848AF9C3EBAC0135D99305822
Verification by decryption 49206F776520796F752024333030302E
[04/07/18]seed@VM:~/lab9$
```

screenshot2. Compile and run program. We get the signature for M1 “I owe you \$2000.”, and signature for M2 “I owe you \$3000.”. Otherwise, we also use the public key to generate the hex value for M1 and M2, this is just for answer verification.

Observation and Explanation:

For this task, we are given e, n, d, and two messages M1 (“I owe you \$2000.”), M2 (“I owe you \$3000.”). We use $(M^d \bmod n)$ to generate signature for M1 and M2.

First we convert M1 and M2 into hex value (screenshot1). After we compile and run the program task4.c, we get signature for M1 and M2 (screenshot2). Even though there is just a slight change, the signatures of M1 and M2 are totally different. Moreover, after we get signatures (C1 for M1, and C2 for M2), we also use $(C^e \bmod n)$ to generate M1 and M2 for verification.

Task5: Verifying a Signature

```
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256

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 *e = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *S = BN_new();
    BIGNUM *M = BN_new();

    // Initialize S, n, e
    BN_hex2bn (&S,
    "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");
    BN_hex2bn (&n,
    "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");
```

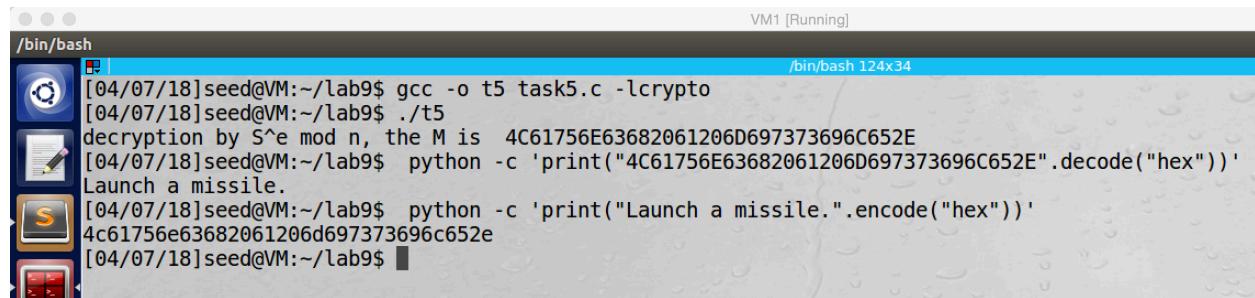
```

BN_hex2bn (&e, "010001");

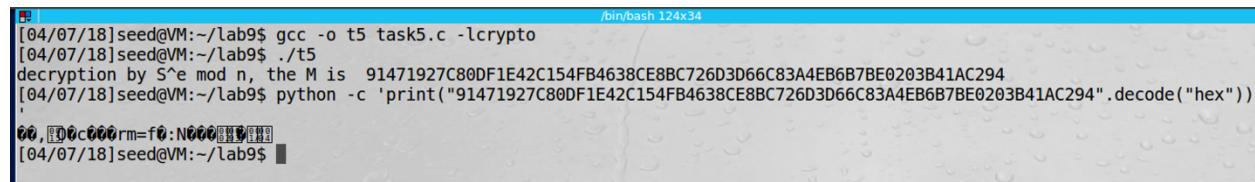
//decoding signature S to get message M by S^e mod n
BN_mod_exp(M, S, e, n, ctx);
printBN("decryption by S^e mod n, the M is ", M);

return 0;
}

```



screenshot1. After compile and run program task5.c, we generate hex value for M. and then we decode the hex value, and we get message “Launch a missile.” which is exactly same as the original message. So this signature is indeed Alice’s.



screenshot2. After we change the last byte of the signature from 2F to 3F and run the program again, we get a very different hex value for the message M. After we decode the hex value, we get some corrupted characters. We have no idea what this is.

Observation and Explanation:

In this task, we are given e , n , S , and original message M . Our task is to verify the signature S is generated from M . For this purpose, we can use $(S^e \bmod n)$ to get message M' . If M and M' are exactly same, then the signature S is indeed Alice's.

First, we compile and run the program task4.c, and it prints the message M which is generated by formula $(S^e \bmod n)$ (screenshot1). Because the M is hex value, so we decode the M, and we get string “Launch a missile.”, this message is exactly same as the Alice’s message (screenshot1). So the signature is indeed Alice’s.

As the screenshot2 shows, if we changed the last byte of the signature S from 2F to 3F, then after we run the program, we got a totally different hex value for M. After we decode the hex value, we got some corrupted characters. And these corrupted characters are not the message “Launch a missile.” Obviously. So the signature is not Alice’s.

Task6: Manually Verifying an X.509 Certificate

Code for task 6

```
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256

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 *e = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *S = BN_new();
    BIGNUM *M = BN_new();

    // Initialize S, n, e
    BN_hex2bn (&S,
    "22ff6e0b9110cab7692d43c03cb4bebe7f1534b1ba11be458f740357a73d4e4220255eff298e9d1d0f9fd20f050278bf3
a4c1362cf18466447ae84da6083fd876f3a285dcc7f7fcce5a5976cc4d1ae13c404629fdd6778d7e986a149f1ce36e805
c1a2897196ffc8dd31341750aabf6f9c709d7e50aab8442b2f51c0309565fb6eba7e3e64526eaec715429643aa7c3298b
5ad6b2f742d87da5383b14452f2873cbb72cd0be7596b1c2a8b2746b3930ad0df8e8c5120c095e65f0a1552cf9b933bb
b214f3c9d64a5fd4fc746f4a69f6bc4c6674e6a6bfff272fc65888cde6b8c0610599db05a20b6853e3817fd3ea6ff653bf2c
8c93a2c55f57263e43359563d");
    BN_hex2bn (&n,
    "CA524BEA1EFFCE246BA8DA721868D5565D0E485A2D3509765ACFA4C81CB1A9FE5389FBAD34FF885B9
FBBE7E80001DC35737503ADB3B1B9A47D2B2679CE15400AEF51B89F328C7C7086524B16FE6A276BE6367
A6250D8DF9A89CC0929EB4F291488800B8F381E806A187C1DBD973B787D4549364F41CDA2E076573C6831
7964C96ED7511E66C3A2642C79C0E765C35684535A436DCB9A0220D2EF1A69D1B09D73A2E02A60655031
CFFBB32FBF1188402EB549100F0A6EDC97FABF2C9F05390B5854AF0696E8C58E0116BCA81A4D41C59391
A21EA18BF2FEC1882449A3474BC51301DDA7571269622BEBFE20EF69FB3AA5F07E29EEED9616F7B11FA
0E49025E033");
    BN_hex2bn (&e, "010001");

    //decoding signature S to get certificate body M by S^e mod n
    BN_mod_exp(M, S, e, n, ctx);
    printBN("decryption by S^e mod n, the M is ", M); //print message M

    return 0;
}
```

screenshot1. Run command “openssl s_client -connect www.google.com:443 -showcerts” to get Google’s certificates. And we save the above two certificates into c0.pem and c1.pem

```
[04/07/18]seed@VM:~/lab9$ openssl x509 -in c1.pem -noout -modulus
Modulus=CA524BEA1E7FC246BA8DA721868D5565D0E485A2D3509765A0CFA4C81CB1A9FE5389FBAD34FF885B9FBBE7E80001DC35737503ADB3B1B9A47D2B
2679CE15400AEF51B889F328C7C086524B16FE6A276BE6367A6250D8DF9A89CC0929EB4F291488800B8F381E806A187C1DBD973B787D4549364F41CDA2E0
76573C68317964C96ED7511E66C3A2642C79C0E765C35684535A436DCB9A0220D2F1A69D1B09D73AE02A60655031CFFBB32FBF1188402EB549100F0A6E
DC97FABF2C9F05390B5854AF0696E8C58E0116BCA81A4D41C59391A21EA18BF2FEC1882449A3474BC51301DDA7571269622BEBFE20EF69FB3AA5F07E29EE
ED9616F7B11FA0E49025E033
```

screenshot2. By running -modulus command, we can get “n”

```
Data:
Version: 3 (0x2)
Serial Number:
 01:e3:a9:30:1c:fc:72:06:38:3f:9a:53:1d
Signature Algorithm: sha256WithRSAEncryption
Issuer: OU=GlobalSign Root CA - R2, O=GlobalSign, CN=GlobalSign
Validity
  Not Before: Jun 15 00:00:42 2017 GMT
  Not After : Dec 15 00:00:42 2021 GMT
Subject: C=US, O=Google Trust Services, CN=Google Internet Authority G3
Subject Public Key Info:
  Public Key Algorithm: rsaEncryption
  Public-Key: (2048 bit)
  Modulus:
  00:ca:52:4b:ea:1e:ff:ce:24:6b:a8:da:72:18:68:
  d5:56:5d:0e:48:5a:2d:35:09:76:5a:cf:a4:c8:1c:
  b1:a9:fe:53:89:fb:ad:34:ff:88:5b:9f:bb:7e:88:
  00:01:dc:35:73:75:03:ad:b3:b1:b9:a4:7d:2b:26:
  79:ce:15:40:0a:ef:51:b8:9f:32:8c:7c:70:86:52:
  4b:16:fe:6a:27:6b:e6:36:7a:62:50:d8:df:9a:89:
  cc:09:29:eb:4f:29:14:88:80:0b:8f:38:1e:80:6a:
  18:7c:1d:bd:97:3b:78:7d:45:49:36:4f:41:cd:a2:
  e0:76:57:3c:68:31:79:64:c9:6e:d7:51:1e:66:c3:
  a2:64:2c:79:c0:e7:65:c3:56:84:53:5a:43:6d:cb:
  9a:02:20:d2:ef:1a:69:d1:b0:9d:73:a2:e0:2a:60:
  65:50:31:cf:fb:b3:2f:bf:11:88:40:2e:b5:49:10:
  0f:0a:6e:dc:97:fa:bf:2c:9f:05:39:0b:58:54:af:
  06:96:e8:c5:8e:01:16:bc:a8:1a:d4:1c:5:93:91:
  a2:1e:a1:8b:f2:fe:c1:88:24:49:a3:47:4b:c5:13:
  01:dd:a7:57:12:69:62:2b:eb:fe:20:ef:69:fb:3a:
  a5:f0:7e:29:ee:ed:96:16:f7:b1:1f:a0:e4:90:25:
  e0:33
  Exponent: 65537 (0x10001)
X509v3 extensions:
```

screenshot3. We get “e” by running command -noout

```
Signature Algorithm: sha256WithRSAEncryption
22:ff:6e:0b:91:10:ca:b7:69:2d:43:c0:3c:b4:be:be:7f:15:
34:b1:ba:11:be:45:8f:74:03:57:a7:3d:4e:42:20:25:5e:ff:
29:8e:9d:1d:0f:9f:d2:0f:05:02:78:bf:3a:4c:13:62:cf:18:
46:64:47:ae:84:da:60:83:fd:d8:76:f3:a2:85:dc:c7:f7:fc:
ce:5a:59:76:cc:4d:1a:e1:3c:40:46:29:fd:d6:77:8d:7e:98:
6a:14:9f:1c:e3:6e:80:5c:1a:28:97:19:6f:fc:8d:d3:13:41:
75:0a:ab:f6:f9:c7:09:d7:e5:0a:ab:84:42:b2:f5:1c:03:09:
56:5f:b6:eb:a7:e3:e6:45:26:ea:ec:71:54:29:64:3a:a7:c3:
29:8b:5a:d6:b2:f7:42:d8:7d:a5:38:3b:14:45:2f:28:73:cb:
b7:2c:d0:be:75:96:b1:c2:a8:b2:74:6b:39:30:ad:0d:f8:e8:
c5:12:0c:09:5e:65:f0:a1:55:2c:f9:b9:33:bb:b2:14:f3:c9:
d6:4a:5f:d4:fc:74:6f:4a:69:f6:bc:4c:66:74:e6:a6:bf:f2:
72:fc:65:88:8c:de:6b:8c:06:10:59:9d:b0:5a:20:b6:85:3e:
38:17:fd:3e:a6:ff:65:3b:f2:c8:c9:3a:2c:55:f5:72:63:e4:
33:59:56:3d
[04/07/18]seed@VM:~/lab9$
```

screenshot4. Get signature by using command “openssl x509 -in c0.pem -text -noout”

```
VM1 [Running]
/bin/bash
[04/07/18]seed@VM:~/lab9$ cat signature | tr -d '[:space:]'
2ff6fe0b9110cab7692d43c03cb4bebef7f1534b1ba11be458f740357a73d4e4220255eff298e9d1d0f9fd20f050278bf3a4c1362cf18466447ae84da6083
fdd876f3a285dcc7f7fcce5a5976cc4d1ae13c404629ffd6778d7e986a149f1ce36e8051ca2897196ffc8dd31341750aabf69c709d7e50aab8442b2f51c
030965fbbebae3e64526eaec715429643aa7c298bb5ad6b2f742d87da5383b14452f2873cbb72cd0be7596b1c288b2746b3930ad0df8e8c5120c095e65
f0a1552c9fb933bb214f3c9d64a5df4fc746f4a69fb6c4c6674ea6b6ff272fc65888cde6b8c0610599db05a20b6853e3817fd3ea6ff653bf2c8c93a2c55
f57263e43359563d[04/07/18]seed@VM:~/lab9$
```

screenshot5. Removing ":" and " " in the signature

```
VM1 [Running]
/bin/bash
[04/07/18]seed@VM:~/lab9$ openssl asn1parse -i -in c0.pem -strparse 4 -out c0_body.bin -noout
[04/07/18]seed@VM:~/lab9$
```

screenshots. We can use the `-strpase` option to get the field from the offset 4, which will give us the body of the certificate, excluding the signature block

```
VM1 [Running]  
/bin/bash  
[04/07/18] seed@VM:~/lab9$ sha256sum c0_body.bin  
8bf154d598cc2355a74981dbce7170016bf674d39ee588d47734704051d34175 c0_body.bin  
[04/07/18] seed@VM:~/lab9$
```

screenshot7. Calculate the hash value for the body of the certificate

```
VM1 [Running]
/bin/bash
[04/07/18]seed@VM:~/lab9$ sha256sum c0_body.bin
8bf154d598cc2355a74981dbc7170016bf674d39ee588d47734704051d34175 c0_body.bin
[04/07/18]seed@VM:~/lab9$ gcc -o t6 task6.c -lcrypto
[04/07/18]seed@VM:~/Lab9$ ./t6
decryption by S' mod n, the M is 01FFFFFF...003031300D060960864801650304020105000420BF154D598CC2355A74981DBC7170016BF674D39EE588D47734704051D34175
[04/07/18]seed@VM:~/Lab9$
```

screenshot8. After running the program, we get the message M, it is same as the hash value for the body of the certificate. so we verified that the signature is valid.

Explanation and Observation:

In this task, we want to verify the X.509 certificate of www.google.com. As the screenshot 1 to screenshot7 shows, I follow the steps on the lab description to get all necessary information, which includes the public key (n , e), the signature, the certificate body, and the hash value of the certificate body.

To verify the signature is valid or not, we can use formula $(S^e \bmod n)$ to get the hash value of the certificate body, now we already have these values (S , e , n), and we also have the hash value of the certificate body. If the M generated by my program is same as the hash value of the certificate body, then we verified that the signature is valid.

For this purpose, we feed n , e , and signature S into the program `task6.c`, and then we compile and run it. As screenshot8 shows, the message M is printed. We see that the last 64bytes are exactly same as the hash value of the certificate body, so we verified that the signature is valid.