

SYSTEM AND NETWORK SECURITY

FALL 2024 – RSA Public-Key Encryption and Signature Lab

Due: 9th September, 2024

Task 1: Deriving the Private Key

- We create a C file “task1.c” which will derive the private key using the values of p , q and e which are provided to us.

```
p = F7E75FDC469067FFDC4E847C51F452DF
q = E85CED54AF57E53E092113E62F436F4F
e = 0D88C3
```

- BIGNUM are created such as p , q , n , $euler_func$, e , d and all that is necessary. We then assign p , q and e as given by converting them from HEX to Binary using `BN_hex2bn()`.
- Steps we follow for RSA:
 - Assign Values: Initialize the big numbers (p , q , and e) with specific hexadecimal values.
 - Calculate $p * q$: Compute the modulus n as the product of p and q .
 - Calculate $p - 1$ and $q - 1$: Subtract 1 from both p and q to prepare for the next step.
 - Compute $\phi(n)$ i.e, $euler_func(n) = (p - 1 * (q - 1))$: Calculate $\phi(n)$, which is crucial for determining the private key.
 - Check if e and $\phi(n)$ are Relatively Prime: Ensure that e and $\phi(n)$ have no common divisors other than 1.
 - Calculate the Private Key (d): Derive d as the modular inverse of e modulo $\phi(n)$.
 - Cleanup: Free the memory used for the big numbers.

```
#include <stdio.h>
#include <openssl/bn.h>

#define NBITS 256

int main()
{
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *p = BN_new();
    BIGNUM *q = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *euler_func = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *res = BN_new();
    BIGNUM *p_1 = BN_new();
    BIGNUM *q_1 = BN_new();

    char p_str[NBITS], q_str[NBITS], e_str[NBITS];

    printf("Enter value of p (hex): ");
    scanf("%s", p_str);
    printf("Enter value of q (hex): ");
    scanf("%s", q_str);
    printf("Enter value of e (hex): ");
```

```

scanf("%s", e_str);

BN_hex2bn(&p, p_str);
BN_hex2bn(&q, q_str);
BN_hex2bn(&e, e_str);

// n = p * q
BN_mul(n, p, q, ctx);

// Euler's totient function:  $\phi(n)$  or  $\text{fai}(n) = (p - 1) \times (q - 1)$ 
BN_sub(p_1, p, BN_value_one());
BN_sub(q_1, q, BN_value_one());
BN_mul(euler_func, p_1, q_1, ctx);

// Ensure e and  $\phi(n)$  or  $\text{fai}(n)$  are relatively prime
BN_gcd(res, euler_func, e, ctx);
if (!BN_is_one(res))
{
    char *euler_func_str = BN_bn2hex(euler_func);
    char *e_str_converted = BN_bn2hex(e);
    printf("Error: %s and %s are not relatively prime\n", e_str_converted,
euler_func_str);
    OPENSSL_free(euler_func_str);
    OPENSSL_free(e_str_converted);
    return 1;
}

// Calculate the private key d (modular inverse of e mod  $\phi(n)$ )
BN_mod_inverse(d, e, euler_func, ctx);

// Print the private key
char *private_key_str = BN_bn2hex(d);
printf("Private key 'd': %s\n", private_key_str);
OPENSSL_free(private_key_str);

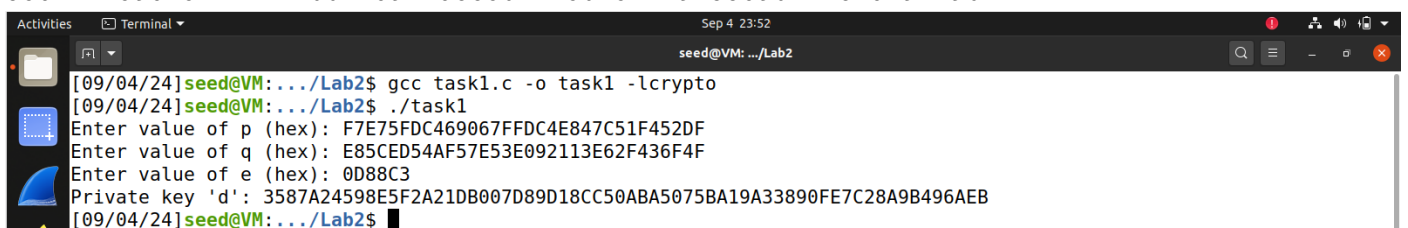
// Clear and free allocated memory
BN_clear_free(p);
BN_clear_free(q);
BN_clear_free(n);
BN_clear_free(res);
BN_clear_free(euler_func);
BN_clear_free(e);
BN_clear_free(d);
BN_clear_free(p_1);
BN_clear_free(q_1);

return 0;
}

```

The private key 'd' is

3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB



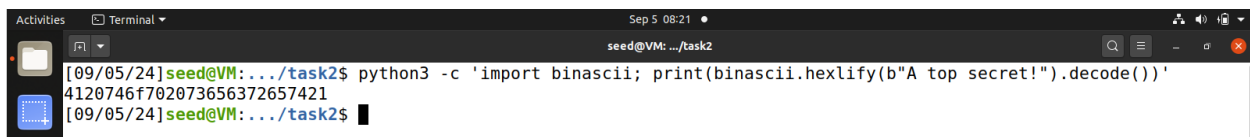
```

Activities Terminal
Sep 4 23:52
seed@VM: .../Lab2
[09/04/24]seed@VM:.../Lab2$ gcc task1.c -o task1 -lcrypto
[09/04/24]seed@VM:.../Lab2$ ./task1
Enter value of p (hex): F7E75FDC469067FFDC4E847C51F452DF
Enter value of q (hex): E85CED54AF57E53E092113E62F436F4F
Enter value of e (hex): 0D88C3
Private key 'd': 3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB
[09/04/24]seed@VM:.../Lab2$

```

Task 2: Encrypting a Message

- I attempted to run the following commands from the lab instructions:
 - `python3 -c 'print("A top secret!".encode("utf-8").hex())'`
 - `python -c 'print("A top secret!".encode("hex"))'`
- However, it appears that these commands are no longer supported in Python 3. To resolve this and successfully convert the string "A top secret!" into its hexadecimal format, I used the following command:
 - `python3 -c 'import binascii; print(binascii.hexlify(b"A top secret!").decode())'`
- This command provides the `binascii` module to perform the hexadecimal conversion, which works properly in Python 3.
- Message in HEX: 4120746f702073656372657421



```
Activities Terminal
Sep 5 08:21
seed@VM: .../task2
[09/05/24]seed@VM: .../task2$ python3 -c 'import binascii; print(binascii.hexlify(b"A top secret!").decode())'
4120746f702073656372657421
[09/05/24]seed@VM: .../task2$
```

- We also have the below parameters for encryption.
`n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5`
`e = 010001 (this hex value equals to decimal 65537)`
`M = A top secret!`
`d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D`
- I've created 2 C scripts. `task2.c` for encryption and `task2_verify.c` for decryption and verification.
- The first script: `task2.c`
 - We first create `BIGNUM` structures for `n`, `e`, `M`, and `C`. Next we take all the inputs that are `n`, `e`, and `M` (`M` is taken as input in decimal format but then while encryption we convert it to HEX format)
 - We perform the RSA Encryption using the formula $C = M^e \bmod n$ where `M` is the message, (`e`, `n`) is the public key pair.
 - The last step is to print this encrypted text in HEX format using the `printBN` function.
 - Before terminating the code, we free all the allocated memory for `BIGNUM` variables.
- The verification script: `task2_verify.c`
 - Similar to `task2.c`, we first create `BIGNUM` structures for `n`, `d`, `C`, and `M`.
 - Next, we take inputs for `n`, `d`, and `C` (`C` is in HEX). These are converted into `BIGNUM` format using the `BN_hex2bn()` function.
 - We perform the RSA Decryption using the formula $M = C^d \bmod n$, where `C` is the cipher text, and (`d`, `n`) is the private key pair.
 - The decrypted message is printed in both hexadecimal and ASCII format using the `printBN` and `printBNasASCII` functions. Before terminating the code, we free all the allocated memory for the `BIGNUM` variables
- Below are both the codes separated by a bolded line (---). First is **`task2.c`** and the second is **`task2_verify.c`**

```

#include <stdio.h>
#include <openssl/bn.h>

#define NBITS 256

void printBN(char *msg, BIGNUM *a)
{
    char *number_str_a = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str_a);
    OPENSSL_free(number_str_a);
}

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

    char n_str[NBITS], e_str[NBITS], M_str[NBITS];

    // Take inputs
    printf("Enter value of n (hex): ");
    scanf("%s", n_str);
    printf("Enter value of e (decimal): ");
    scanf("%s", e_str);
    printf("Enter message M to encrypt (hex): ");
    scanf("%s", M_str);

    // Convert all inputs to BIGNUM
    BN_hex2bn(&n, n_str);
    BN_dec2bn(&e, e_str);
    BN_hex2bn(&M, M_str);

    // RSA Encryption:  $C = M^e \bmod n$ 
    BN_mod_exp(C, M, e, n, ctx);
    printBN("Cipher Text:", C);

    // Clear all data and free allocated memory
    BN_clear_free(n);
    BN_clear_free(e);
    BN_clear_free(M);
    BN_clear_free(C);
    BN_CTX_free(ctx);

    return 0;
}

```

```

#include <stdio.h>
#include <openssl/bn.h>
#include <string.h>

#define NBITS 256

void printBN(char *msg, BIGNUM *a)
{
    char *number_str_a = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str_a);
    OPENSSL_free(number_str_a);
}

void printBNasASCII(char *msg, BIGNUM *a)
{
    char *number_str_a = BN_bn2hex(a);

```

```

printf("%s ", msg);
for (size_t i = 0; i < strlen(number_str_a); i += 2)
{
    char hex_byte[3] = {number_str_a[i], number_str_a[i+1], '\0'};
    printf("%c", (char)strtol(hex_byte, NULL, 16));
}
printf("\n");
OPENSSL_free(number_str_a);
}

int main()
{
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *n = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *C = BN_new();
    BIGNUM *M = BN_new(); // For decrypted message (plaintext)

    char n_str[NBITS], d_str[NBITS], C_str[NBITS];

    // Take inputs
    printf("Enter value of n (hex): ");
    scanf("%s", n_str);
    printf("Enter value of d (hex): ");
    scanf("%s", d_str);
    printf("Enter ciphertext C (hex): ");
    scanf("%s", C_str);

    // Convert all inputs to BIGNUM
    BN_hex2bn(&n, n_str);
    BN_hex2bn(&d, d_str);
    BN_hex2bn(&C, C_str);

    // RSA Decryption  $M = C^d \bmod n$ 
    BN_mod_exp(M, C, d, n, ctx);

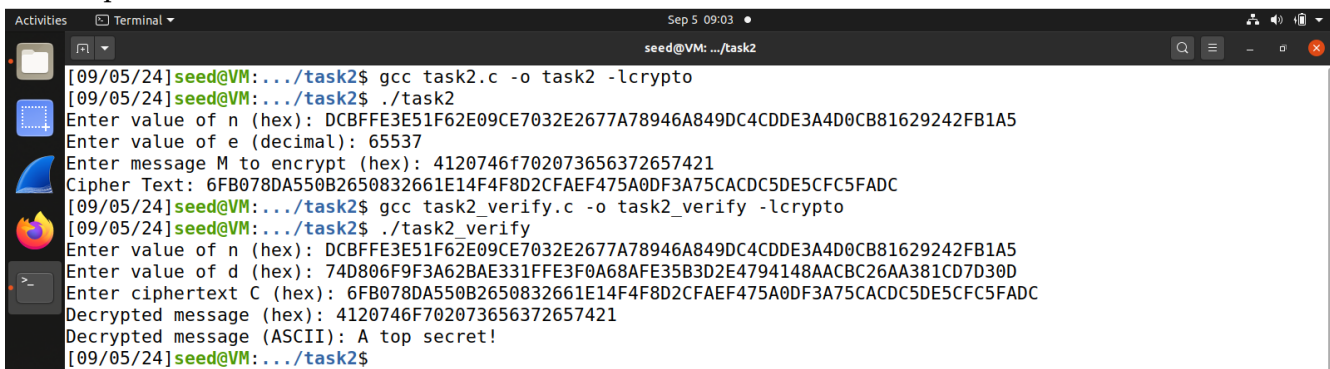
    // Print the decrypted text in hex and ASCII
    printBN("Decrypted message (hex):", M);
    printBNasASCII("Decrypted message (ASCII):", M);

    // Clear all data and free allocated memory
    BN_clear_free(n);
    BN_clear_free(d);
    BN_clear_free(C);
    BN_clear_free(M);
    BN_CTX_free(ctx);

    return 0;
}

```

• Output:



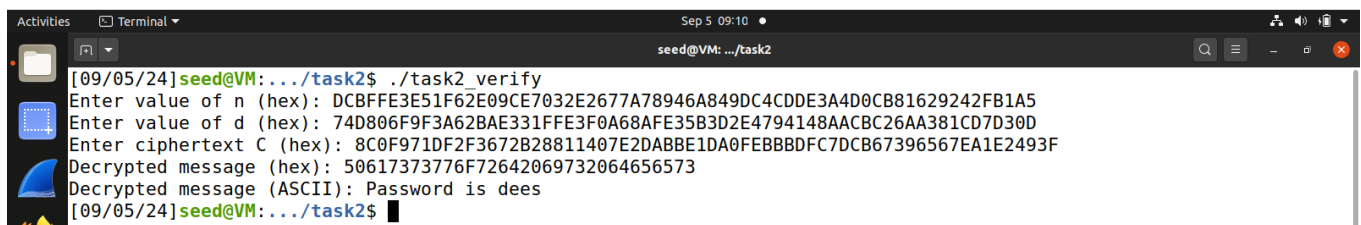
```

[09/05/24]seed@VM: .../task2$ gcc task2.c -o task2 -lcrypto
[09/05/24]seed@VM: .../task2$ ./task2
Enter value of n (hex): DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
Enter value of e (decimal): 65537
Enter message M to encrypt (hex): 4120746f702073656372657421
Cipher Text: 6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC
[09/05/24]seed@VM: .../task2$ gcc task2_verify.c -o task2_verify -lcrypto
[09/05/24]seed@VM: .../task2$ ./task2_verify
Enter value of n (hex): DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
Enter value of d (hex): 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D
Enter ciphertext C (hex): 6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC
Decrypted message (hex): 4120746f702073656372657421
Decrypted message (ASCII): A top secret!
[09/05/24]seed@VM: .../task2$

```

Task 3: Decrypting a Message

- I have already performed a similar decryption operation in Task 2 using the same public/private key pair. The process of decrypting the cipher text C and converting it back to an ASCII string is identical to what I completed in Task 2.
- In Task 2, I decrypted the cipher text using the private key d with the formula $M = C^d \bmod n$, and printed the plaintext message in both hexadecimal and ASCII format within the script without using the python command.
- I can apply the same decryption process to the new cipher text C that is given for this task (8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBD7C7DCB67396567EA1E2493F) using the same approach for verification.
- Here's the output using the same private key pair of (d,n):

A terminal window titled 'Terminal' with a search bar and window controls. The prompt is 'seed@VM: .../task2'. The user enters './task2_verify'. The script prompts for 'Enter value of n (hex):', 'Enter value of d (hex):', and 'Enter ciphertext C (hex):'. The user provides the values: 'DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5', '74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D', and '8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBD7C7DCB67396567EA1E2493F' respectively. The script outputs 'Decrypted message (hex): 50617373776F72642069732064656573' and 'Decrypted message (ASCII): Password is dees'.

```
[09/05/24]seed@VM: .../task2$ ./task2_verify
Enter value of n (hex): DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
Enter value of d (hex): 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D
Enter ciphertext C (hex): 8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBD7C7DCB67396567EA1E2493F
Decrypted message (hex): 50617373776F72642069732064656573
Decrypted message (ASCII): Password is dees
[09/05/24]seed@VM: .../task2$
```

Task 4: Signing a Message

- I've created a C file `task4.c` that will perform RSA signature operation on 2 messages.
- We begin by the same thing here as well. `BIGNUM` structures for `n`, `d`, `M1`, `M2`, `C1` and `C2`.
- We assign the values of `n` and the private key `d` using the `BN_hex2bn()` function, converting the HEX strings into `BIGNUM` format.
- Next, we give the code our inputs of two messages in the ASCII format. These are converted to HEX using the `ascii_to_bn()` function and then stored in `M1` and `M2` as `BIGNUM`s.
- The RSA signature is computed for both messages using the formula $C = M^d \bmod n$.
- Once we have the signatures of both the messages, we clear all the allocated memory.
- Below is the `task4.c` code:

```
#include <stdio.h>
#include <openssl/bn.h>
#include <string.h>

#define NBITS 256

void printBN(char *msg, BIGNUM *a)
{
    char *number_str_a = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str_a);
    OPENSSL_free(number_str_a);
}

void ascii_to_bn(BIGNUM *bn, const char *ascii)
{
    char hex_str[NBITS * 2 + 1];
    int i;

    for (i = 0; i < strlen(ascii); i++)
    {
        sprintf(hex_str + i * 2, "%02x", ascii[i]);
    }
}
```

```

    hex_str[i * 2] = '\\0';
    BN_hex2bn(&bn, hex_str);
}

int main()
{
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *n = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *M1 = BN_new();
    BIGNUM *M2 = BN_new();
    BIGNUM *C1 = BN_new();
    BIGNUM *C2 = BN_new();

    // RSA parameters (n and d) are set
    BN_hex2bn(&n,
"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&d,
"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

    // Input for the messages
    char ascii_M1[NBITS], ascii_M2[NBITS];
    printf("Enter M1 (ASCII): ");
    fgets(ascii_M1, NBITS, stdin);
    ascii_M1[strcspn(ascii_M1, "\\n")] = 0;

    printf("Enter M2 (ASCII): ");
    fgets(ascii_M2, NBITS, stdin);
    ascii_M2[strcspn(ascii_M2, "\\n")] = 0;
    ascii_to_bn(M1, ascii_M1);
    ascii_to_bn(M2, ascii_M2);
    // RSA signature: C = M^d mod n
    BN_mod_exp(C1, M1, d, n, ctx);
    BN_mod_exp(C2, M2, d, n, ctx);

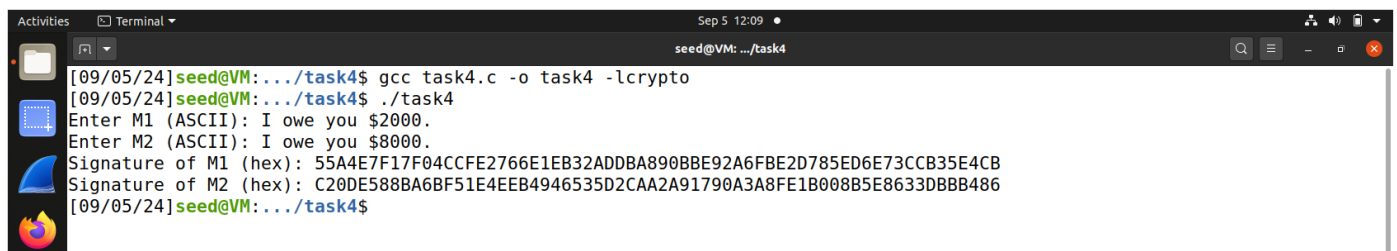
    // Print the signatures of both the messages in hex format
    printBN("Signature of M1 (hex):", C1);
    printBN("Signature of M2 (hex):", C2);

    // Clear all data and free allocated memory
    BN_clear_free(n);
    BN_clear_free(d);
    BN_clear_free(M1);
    BN_clear_free(M2);
    BN_clear_free(C1);
    BN_clear_free(C2);
    BN_CTX_free(ctx);

    return 0;
}

```

- Output:



```

[09/05/24]seed@VM:.../task4$ gcc task4.c -o task4 -lcrypto
[09/05/24]seed@VM:.../task4$ ./task4
Enter M1 (ASCII): I owe you $2000.
Enter M2 (ASCII): I owe you $8000.
Signature of M1 (hex): 55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73CCB35E4CB
Signature of M2 (hex): C20DE588BA6BF51E4EEB4946535D2CAA2A91790A3A8FE1B008B5E8633DBBB486
[09/05/24]seed@VM:.../task4$

```

Task 5: Verifying a Signature

- I've created `task5.c` file that will verify the RSA signature provided by Alice.
- The initial steps are same as before; initialize `BIGNUM` structures for all the parameters. Here, we have `n`, `e`, `M`, `S`, and `C`. Inputs are given by the user based on the format mentioned.
- Next, to verify the signature, we perform the operation $C = S^e \bmod n$ to decrypt the signature using Alice's public key i.e., (e, n) .
- We then compare the output with the original message (in HEX format as the output obtained after decryption will be in HEX).
- If they match, the signature is declared valid; else it is invalid. Once the verification is complete, all the allocated memory is cleared as usual.
- Below is the `task5.c` code:

```
#include <stdio.h>
#include <openssl/bn.h>
#include <string.h>

#define NBITS 256

void printBN(char *msg, BIGNUM *a)
{
    char *number_str_a = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str_a);
    OPENSSL_free(number_str_a);
}

void ascii_to_bn(BIGNUM *bn, const char *ascii)
{
    char hex_str[NBITS * 2 + 1];
    int i;

    for (i = 0; i < strlen(ascii); i++)
    {
        sprintf(hex_str + i * 2, "%02x", ascii[i]);
    }
    hex_str[i * 2] = '\0';
    BN_hex2bn(&bn, hex_str);
}

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

    // Input for Alice's Public Key and Signature
    char n_str[NBITS], e_str[NBITS], ascii_msg[NBITS], S_str[NBITS];

    printf("Enter Alice's Public Key: \n");
    printf("\te (DECIMAL): ");
    scanf("%s", e_str);
    printf("\tn (HEX): ");
    scanf("%s", n_str);

    printf("Enter Message (ASCII): ");
    getchar();
    fgets(ascii_msg, NBITS, stdin);
    ascii_msg[strcspn(ascii_msg, "\n")] = 0;
```



```

printf("Enter Signature provided by Alice (HEX): ");
scanf("%s", S_str);

// Convert inputs to BIGNUM format
BN_hex2bn(&n, n_str);
BN_dec2bn(&e, e_str);
ascii_to_bn(M, ascii_msg);
BN_hex2bn(&S, S_str);

// Signature verification: C = S^e mod n
BN_mod_exp(C, S, e, n, ctx);

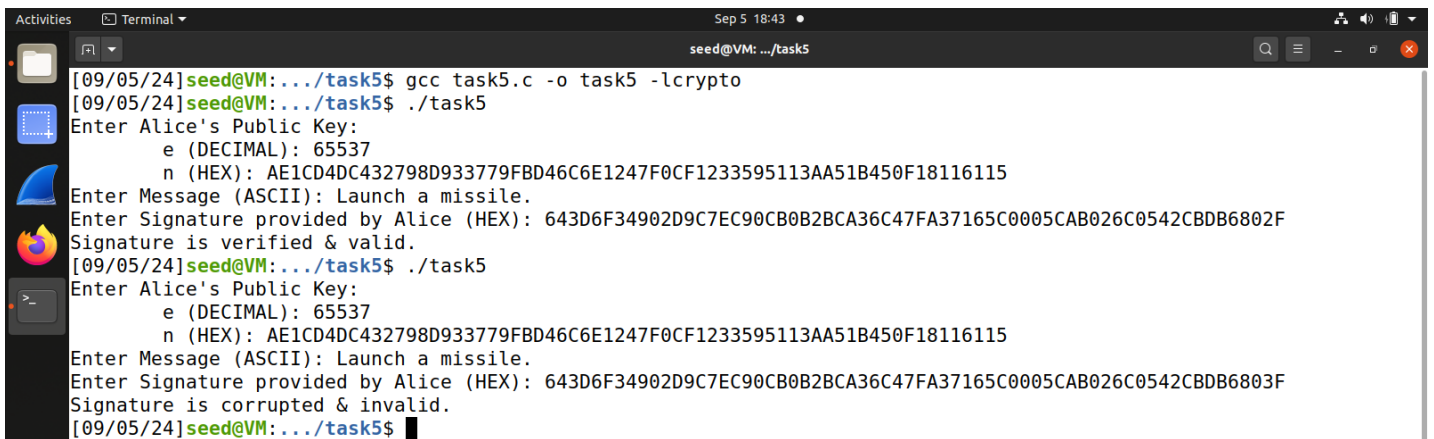
// Compare the calculated ciphertext C with the original message M
if (BN_cmp(C, M) == 0)
{
    printf("Signature is verified & valid.\n");
}
else
{
    printf("Signature is corrupted & invalid.\n");
}

// Clear all data and free allocated memory
BN_clear_free(n);
BN_clear_free(e);
BN_clear_free(M);
BN_clear_free(S);
BN_clear_free(C);
BN_CTX_free(ctx);

return 0;
}

```

- Output:



```

[09/05/24]seed@VM: .../task5$ gcc task5.c -o task5 -lcrypto
[09/05/24]seed@VM: .../task5$ ./task5
Enter Alice's Public Key:
  e (DECIMAL): 65537
  n (HEX): AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115
Enter Message (ASCII): Launch a missile.
Enter Signature provided by Alice (HEX): 643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F
Signature is verified & valid.
[09/05/24]seed@VM: .../task5$ ./task5
Enter Alice's Public Key:
  e (DECIMAL): 65537
  n (HEX): AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115
Enter Message (ASCII): Launch a missile.
Enter Signature provided by Alice (HEX): 643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F
Signature is corrupted & invalid.
[09/05/24]seed@VM: .../task5$

```

- Changing the last byte of the signature causes the verification to fail because the decrypted signature no longer matches the original message, making it clear that the signature is invalid.

Task 6: Manually Verifying an X.509 Certificate

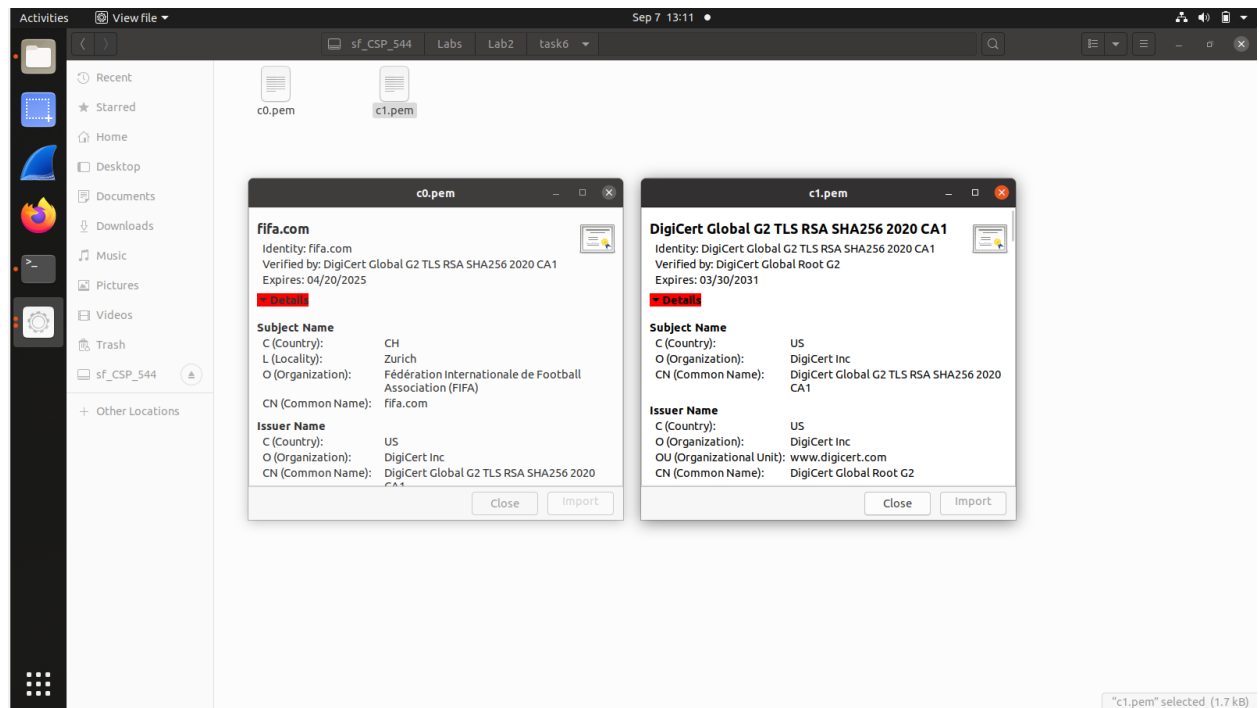
- In this task, I'll be using the FIFA WORLD CUP 2026 website (www.fifa.com)

STEP 1: Download a certificate from a real web server

- Run the below command to download the certificate from this website.
`openssl s_client -connect www.fifa.com:443 -showcerts`

```
Activities Terminal Sep 7 13:07 seed@VM: .../task6
[09/07/24]seed@VM: .../task6$ openssl s_client -connect www.fifa.com:443 -showcerts
CONNECTED(00000003)
depth=1 C = US, 0 = DigiCert Inc, CN = DigiCert Global G2 TLS RSA SHA256 2020 CA1
verify error:num=20:unable to get local issuer certificate
verify return:1
depth=0 C = CH, L = Zurich, 0 = F\C3\A9d\C3\A9ration Internationale de Football Association (FIFA), CN = fifa.com
verify return:1
---
Certificate chain
 0 s:C = CH, L = Zurich, 0 = F\C3\A9d\C3\A9ration Internationale de Football Association (FIFA), CN = fifa.com
  i:C = US, 0 = DigiCert Inc, CN = DigiCert Global G2 TLS RSA SHA256 2020 CA1
-----BEGIN CERTIFICATE-----
MIIlIjCCB36gAwIBAgIQDnvHV7z58CnvqIAL39Q/qDANBgkqhkiG9w0BAQsFADBZ
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMTMwMQYDVQQDEypE
aWdpQ2VydCBhbG9iYWwgRzIgVExTIFJTQSBTSEEyNTYgMjAyMCMBDQTEwHhcNMjQw
NzExMDAwMDAwMWhcNMjUwNDE5MjM1OTU5WjB2MQswCQYDVQQGEwJDSEDEPMA0GA1UE
BxMGMWVyaWNoMUMwMQYDVQQKDDpGw6lkW6lyYXRpb24gSW50ZXJ1eXRpb25hbGUg
ZGUGRm9vdGJhbGwGwGQXNzb2NpYXRpb24gKEZJRkEPMREwDyYDVQDEwHmaWZlLnVb
bTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAlbL33QQCJ38xX4Hu2wk
kwqnCJo1X5c9/UmkQ1araEz+Zbg971pEtMqyFoWb3wIaoEcPeWbNR5y3jzQueCkW
yyFgjNkb3UjYh/dUD74U5cFHV19+39yEaF7cAfB099g50GeLUh7TKEh2ldWb4t9
yTQB6TzVrNmMD89F7z2u0JVbj8Gj1zYHtAT2bs1RzBFRPqNCoNhrGHI3reAw2+jgX
eqtVvJKHA/FX2bzIk0DHRG69VkiyNSzCKLT1GE6KGczrTa+YgokIe9LRUuHm934A
HPLx1ZX5iTyOPT310FEgF50EjN8WNQrNn65xEPniowtULrxXRQ5jn0L0Jge0Es87
Dp0CAwEAAoCBTswgU3MB8GA1UdIwQYMBAfHFSFGMBmx9833s+9KTeqAx2+7c0X
MB0GA1UdDgQWBBS4MczHLazkLpILC5Y478SJB8hntzCCAccGA1UdEQSCAb4wggG6
ggghmaWZlLnVbYIMZmLmY5SkaWdpdGFsggoLmZpZmEuY29tgg4QLmZpZmEuZGln
aXRhbIIRK15xYS5maWZlLnRpZ2l0YWYyCDyouGvdZC5maWZlLnVbYISK15wcmQu
ZmLmY5SkaWdpdGFsgghIqLnByZS5maWZlLnRpZ2l0YWYyCDSouCWUzmlmY5Sjb22C
EioudG1ja2V0cy5maWZlLnVbYI0Ki5wcmUuZmLmY5Sjb22CDioucHJkLmZpZmEu
Y29tghUqLnZvbHVudGVlcnMuZmLmY5Sjb22CFcoudm9sdW50ZWVYLnZpZmEuY29t
giIqLmVbXBldGl0aW9uc2FmZWd1YXJkaW5nLnZpZmEuY29tghQmLmdyZWVvY2Fy
ZC5maWZlLnVbYIYK15wcC10aWNrZXRzLmZpZmEuY29tghsQLm5vZG1zY3JpbWlu
YXRpb24uZmLmY5Sjb22CFyouc3RnZXh0cmFuZXRzLmZpZmEuY29tghQmLmV4dHJh
bmV0cy5maWZlLnVbYIRK15pbnNpZGUuZmLmY5Sjb22CGCoudm9sdW50ZWVYLVXVh
dC5maWZlLnVbYTA+BgNVHSAENZAlMDMGBmeBDAECAjApMCcGCCsGAQUFBwIBFhto
dHRw0i8vd3d3LmRpZ2ljZXJ0LmNvbS9DUFMwDgYDVROPAQH/BAQDAgWgMB0GA1Ud
JQQMBOGQCcGAQUFBwMBBggrBgEFBQcDAjCBnwYDVROfBIGXMIgUMEIgRBEhkiJo
dHRw0i8vY3J3Sj5MaWdpY2VydC5jb20vRGlnaUNlcnRhbG9iYXwHLMRMU1JTQVNI
QT1lNjIwMjB0TEtMS5jcmwSKBGoESG0mh0dHA6Ly9jcmw0LmRpZ2ljZXJ0LmNv
bS9EaWdpQ2VydEdsb2JhbEcyVExTUlNBU0hBMjU2MjAyMENBMS0xLmNybDcBhwYI
KwYBBQUHAQEezB5MCQGCCsGAQUFBzABhhodHRw0i8vb2NzcC5kaWdpY2VydC5j
b20wUUYIKwYBBQUHMAKGRWh0dHA6Ly9jYWNlcnRzLmRpZ2ljZXJ0LmNvbS9EaWdp
Q2VydEdsb2JhbEcyVExTUlNBU0hBMjU2MjAyMENBMS0xLmNydDAMBGNVHRMBAF8E
AjAAMIIBfwYKKwYBBAHWeQIEAgSCAW8EggFrAwkAdgB0daMnXJoQwzhbNTfP1Lr
HfDgjhuNacCx+mSxYpo53wAAAZCh2sHAAAEAwBHMEUCIQDeAc/Tea3DgJtkzsnu
gkhcpvFw5+0vWFG0nobyiRA0AIGcpzKktv/xU/We0Jk3vII0JSGnK5tgnS1Pe0x
8IwYN9sAdwB9WR4S4XgqexxhZ3xe/fjQh1wUoE6VnrkDL9k0jC55uAAAAZCh2ssi
AAAEAwBIMEYCIQDd9H80PaAbH4I6PS+/RqCotmkCMwaZsAq7rddA/CDtMwIhAMhL
lmeKrQraffl1LF3Zhzvdmf+K8qsbJ2MwzntPhwUAHYAzPsPaoVxCWx+LZtTzumi
fCLphVwN1422qX5UwP5MDbAAAAGQodrK8wAABAMARzBFAiBeCqpJRf6R8v3iu6QN
8M5TfttiElLLa3EN1Zp/TjtyFAiH436NBmg10Uvv+yZ9G8yotfqZnxXeBxx2LB
ZNLquh6+MA0GCCSGSIB3DQEBcWUAA4IBAQAQnXq3m6z0ik/IKhJ9590oFF5bNIU
Ogg7yh42LNVb03JcsLiKkeJGvAcPpFoohZ02ox0IyCwhFa2kKu73wJCLfHVRm9j3
dXyPUBxsJJ0Iyi1SbXWYUjm2d+/2LAKBvftdfwC2mc4drkEL67P+prF7gE8rVEY
sDetodSgluonD+pVjTddRZK5Z6z42Vmsw6Fu0rbjlvnPKGRMM3Y9Xo6MyA9weNm
kPM7IQPtYdNds3z+eQRSVwRlgy6aT/36fwfJ3f4+zLmUP0BBAE30MEB50Gv3SVb
HUK8DYDiX4Piuj1/GJmrgP0UdvF+BXG5HDSLoiUHuXol4fZVzWgVPimM
-----END CERTIFICATE-----
 1 s:C = US, 0 = DigiCert Inc, CN = DigiCert Global G2 TLS RSA SHA256 2020 CA1
  i:C = US, 0 = DigiCert Inc, OU = www.digicert.com, CN = DigiCert Global Root G2
-----BEGIN CERTIFICATE-----
MIIEyDCCA7CgAwIBAgIQDPW9BItwAvR6uFasI8zwZjANBgkqhkiG9w0BAQsFADBh
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
```

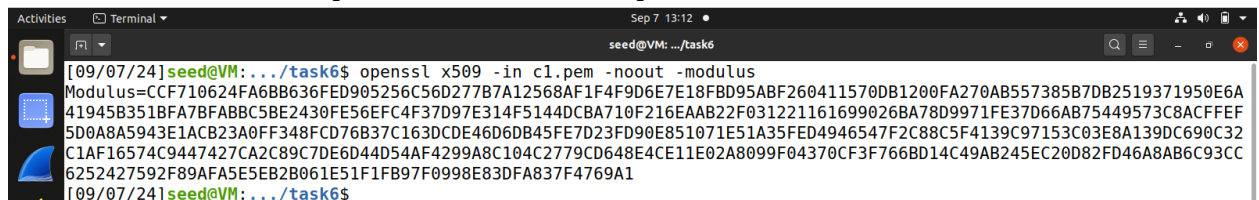
- Save the two certificates in `c0.pem` and `c1.pem`



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

- As mentioned in the document, for finding the modulus (n), we run the below command

```
openssl x509 -in c1.pem -noout -modulus
```



Modulus =

```
CCF710624FA6BB636FED905256C56D277B7A12568AF1F4F9D6E7E18FBD95ABF260411570DB120
0FA270AB557385B7DB2519371950E6A41945B351BFA7BFABBC5BE2430FE56EFC4F37D97E314F5
144DCBA710F216EAAB22F031221161699026BA78D9971FE37D66AB75449573C8ACFFFEF5D0A8A5
943E1ACB23A0FF348FCD76B37C163DCDE46D6DB45FE7D23FD90E851071E51A35FED4946547F2C
88C5F4139C97153C03E8A139DC690C32C1AF16574C9447427CA2C89C7DE6D44D54AF4299A8C10
4C2779CD648E4CE11E02A8099F04370CF3F766BD14C49AB245EC20D82FD46A8AB6C93CC625242
7592F89AFA5E5EB2B061E51F1FB97F0998E83DFA837F4769A1
```

- And for finding the exponent (e), we run this command

```
openssl x509 -in c1.pem -text -noout
```

```

0c:f5:bd:06:2b:56:02:f4:7a:b8:50:2c:23:cc:f0:66
Signature Algorithm: sha256WithRSAEncryption
Issuer: C = US, O = DigiCert Inc, OU = www.digicert.com, CN = DigiCert Global Root G2
Validity
  Not Before: Mar 30 00:00:00 2021 GMT
  Not After : Mar 29 23:59:59 2031 GMT
Subject: C = US, O = DigiCert Inc, CN = DigiCert Global G2 TLS RSA SHA256 2020 CA1
Subject Public Key Info:
  Public Key Algorithm: rsaEncryption
  RSA Public-Key: (2048 bit)
  Modulus:
    00:cc:f7:10:62:4f:a6:bb:63:6f:ed:90:52:56:c5:
    6d:27:7b:7a:12:56:8a:f1:f4:f9:d6:e7:e1:8f:bd:
    95:ab:f2:60:41:15:70:db:12:00:fa:27:0a:b5:57:
    38:5b:7d:b2:51:93:71:95:0e:6a:41:94:5b:35:1b:
    fa:7b:fa:bb:c5:be:24:30:fe:56:ef:c4:f3:7d:97:
    e3:14:f5:14:4d:cb:a7:10:f2:16:ea:ab:22:f0:31:
    22:11:61:69:90:26:ba:78:d9:97:1f:e3:7d:66:ab:
    75:44:95:73:c8:ac:ff:ef:5d:0a:8a:59:43:e1:ac:
    b2:3a:0f:f3:48:fc:d7:6b:37:c1:63:dc:de:46:d6:
    db:45:fe:7d:23:fd:90:e8:51:07:1e:51:a3:5f:ed:
    49:46:54:7f:2c:88:c5:f4:13:9c:97:15:3c:03:e8:
    a1:39:dc:69:0c:32:c1:af:16:57:4c:94:47:42:7c:
    a2:c8:9c:7d:e6:d4:4d:54:af:42:99:a8:c1:04:c2:
    77:9c:d6:48:e4:ce:11:e0:2a:80:99:f0:43:70:cf:
    3f:76:6b:d1:4c:49:ab:24:5e:c2:0d:82:fd:46:a8:
    ab:6c:93:cc:62:52:42:75:92:f8:9a:fa:5e:5e:b2:
    b0:61:e5:1f:1f:b9:7f:09:98:e8:3d:fa:83:7f:47:
    69:a1
  Exponent: 65537 (0x10001)

```

STEP 3: Extract the signature from the server's certificate

- We run the command `$ openssl x509 -in c0.pem -text -noout` for the signature.

```

7D:F9:75:2C:5D:D9:87:3B:DD:0C:C7:FE:2B:CA:AC:6C:
9D:8C:5B:39:ED:3E:1C:14
Signed Certificate Timestamp:
  Version : v1 (0x0)
  Log ID  : CC:FB:0F:6A:85:71:09:65:FE:95:9B:53:CE:E9:B2:7C:
  22:E9:85:5C:0D:97:8D:B6:A9:7E:54:C0:FE:4C:0D:B0
  Timestamp : Jul 11 12:53:08.979 2024 GMT
  Extensions: none
  Signature : ecdsa-with-SHA256
    30:45:02:20:5E:0A:AA:49:45:FE:91:F2:FD:E2:BB:A4:
    0D:F0:CE:53:7E:D8:A1:12:59:4B:6B:71:0D:D5:9A:7F:
    4E:3B:72:14:02:21:00:DE:37:E8:D0:66:83:5D:14:BE:
    FF:B2:67:D1:BC:CA:8B:5F:A9:99:F1:C5:E0:71:C7:62:
    C1:64:D9:6A:BA:1E:BE
Signature Algorithm: sha256WithRSAEncryption
  20:36:ac:50:de:6e:b3:3a:29:3f:20:a8:49:f7:9f:4e:a0:51:
  79:6c:d2:14:3a:08:3b:ca:1e:36:2c:db:db:3b:72:5c:b0:b8:
  8a:91:e2:46:bc:07:29:3c:5a:28:85:9d:36:a3:1d:08:c8:2c:
  21:15:ad:a4:2a:ee:f7:c0:90:8b:7c:75:51:9b:d8:f7:75:7c:
  8f:50:1c:6c:24:9a:08:ca:29:52:70:15:d6:c9:48:e6:d9:df:
  bf:d8:b0:0a:06:f7:ed:75:fc:02:da:67:38:76:b9:04:2f:ae:
  cf:fa:9a:c5:ee:01:3c:ad:51:18:b0:37:ad:a1:d4:a0:d6:ea:
  27:0f:ea:55:8d:37:5d:75:16:4a:e5:9e:b3:e3:65:66:b3:0e:
  85:bb:4a:db:8e:5b:e7:3c:a8:11:30:cd:d8:f5:7a:3a:33:20:
  3d:c1:e3:66:90:f3:3b:21:03:ed:c9:d3:5d:b3:7c:fe:79:04:
  52:57:04:65:83:2e:9a:4f:fd:fa:7f:07:c9:dd:fe:3e:cc:b9:
  94:3c:e0:41:00:4d:ce:30:40:52:d0:66:2f:dd:25:5b:1d:49:
  3c:0d:80:e2:5f:83:e2:ba:3d:7f:18:99:ab:80:fd:14:76:f1:
  7e:05:71:b9:1c:34:8b:a2:25:07:b9:7a:25:e1:f6:55:cd:68:
  15:3e:29:8c
[09/07/24]seed@VM:.../task6$

```

- We take this signature and save it in a file and name it signature. Now we remove all the spaces and colons from the signature in order to get a hexadecimal format of the signature. We can do this by running the following command.

```
cat signature | tr -d '[:space:]:'
```

```

[09/07/24]seed@VM:.../task6$ cat signature | tr -d '[:space:]:'
2036ac50de6eb33a293f20a849f79f4ea051796cd2143a083bca1e362cddb3b725cb0b88a91e246bc07293c5a28859d36a31d08c82c2115ada4
2aeef7c0908b7c75519bd8f7757c8f501c6c249a08ca29527015d6c948e6d9dfbfd8b00a06f7ed75fc02da673876b9042faecffa9ac5ee013cad
5118b037ada1d4a0d6ea270fea558d375d75164ae59eb3e36566b30e85bb4adb8e5be73ca81130cdd8f57a3a33203dc1e36690f33b2103edc9d3
5db37cfe790452570465832e9a4ffdfaf707c9ddfe3eccb9943ce041004dce304052d0662fdd255b1d493c0d80e25f83e2ba3d7f1899ab80fd14
76f17e0571b91c348ba22507b97a25e1f655cd68153e298c[09/07/24]seed@VM:.../task6$

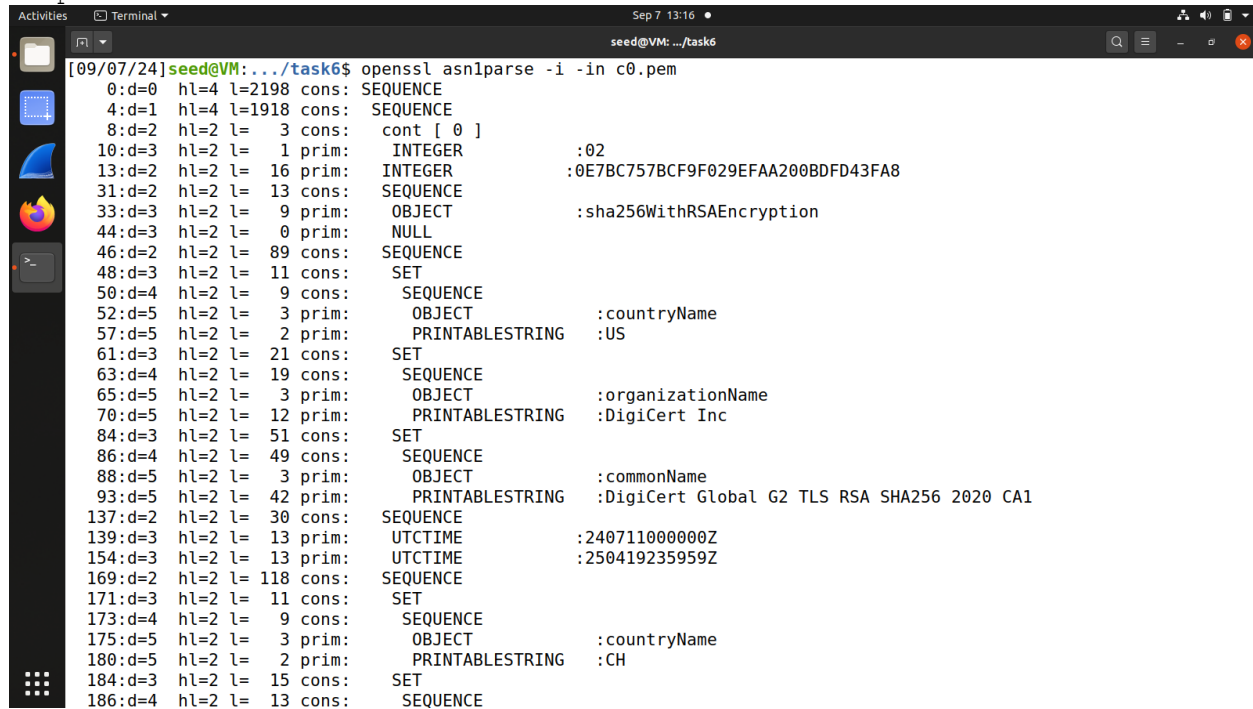
```

Certificate Signature (hex) =

```
2036ac50de6eb33a293f20a849f79f4ea051796cd2143a083bca1e362cdbdb3b725cb0b88a91e
246bc07293c5a28859d36a31d08c82c2115ada42aee7c0908b7c75519bd8f7757c8f501c6c24
9a08ca29527015d6c948e6d9dfbfd8b00a06f7ed75fc02da673876b9042faecffa9ac5ee013ca
d5118b037ada1d4a0d6ea270fea558d375d75164ae59eb3e36566b30e85bb4adb8e5be73ca811
30cdd8f57a3a33203dc1e36690f33b2103edc9d35db37cfe790452570465832e9a4ffdfa7f07c
9ddfe3eccb9943ce041004dce304052d0662fdd255b1d493c0d80e25f83e2ba3d7f1899ab80fd
1476f17e0571b91c348ba22507b97a25e1f655cd68153e298c
```

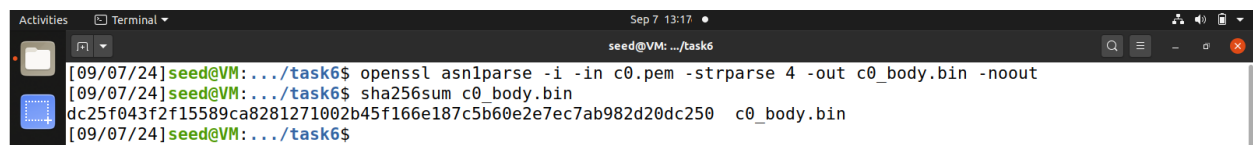
STEP 4: Extract the body of the server's certificate.

- Openssl has a command called `asn1parse` used to extract data from ASN.1 formatted data, and is able to parse our X.509 certificate. We run the command `$ openssl asn1parse -i -in c0.pem`



```
[09/07/24]seed@VM: .../task6$ openssl asn1parse -i -in c0.pem
0:d=0 hl=4 l=2198 cons: SEQUENCE
4:d=1 hl=4 l=1918 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           :0E7BC757BCF9F029EFAA200BDFD43FA8
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= 89 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= 21 cons: SET
63:d=4 hl=2 l= 19 cons: SEQUENCE
65:d=5 hl=2 l= 3 prim: OBJECT              :organizationName
70:d=5 hl=2 l= 12 prim: PRINTABLESTRING    :DigiCert Inc
84:d=3 hl=2 l= 51 cons: SET
86:d=4 hl=2 l= 49 cons: SEQUENCE
88:d=5 hl=2 l= 3 prim: OBJECT              :commonName
93:d=5 hl=2 l= 42 prim: PRINTABLESTRING    :DigiCert Global G2 TLS RSA SHA256 2020 CA1
137:d=2 hl=2 l= 30 cons: SEQUENCE
139:d=3 hl=2 l= 13 prim: UTCTIME           :240711000000Z
154:d=3 hl=2 l= 13 prim: UTCTIME           :250419235959Z
169:d=2 hl=2 l= 118 cons: SEQUENCE
171:d=3 hl=2 l= 11 cons: SET
173:d=4 hl=2 l= 9 cons: SEQUENCE
175:d=5 hl=2 l= 3 prim: OBJECT              :countryName
180:d=5 hl=2 l= 2 prim: PRINTABLESTRING    :CH
184:d=3 hl=2 l= 15 cons: SET
186:d=4 hl=2 l= 13 cons: SEQUENCE
```

- Next we run `$ openssl asn1parse -i -in c0.pem -strparse 4 -out c0_body.bin -noout`.
- We have our body of the certificate stored in `c0_body.bin` file. We now find the hash of this file using the command `$ sha256sum c0_body.bin`



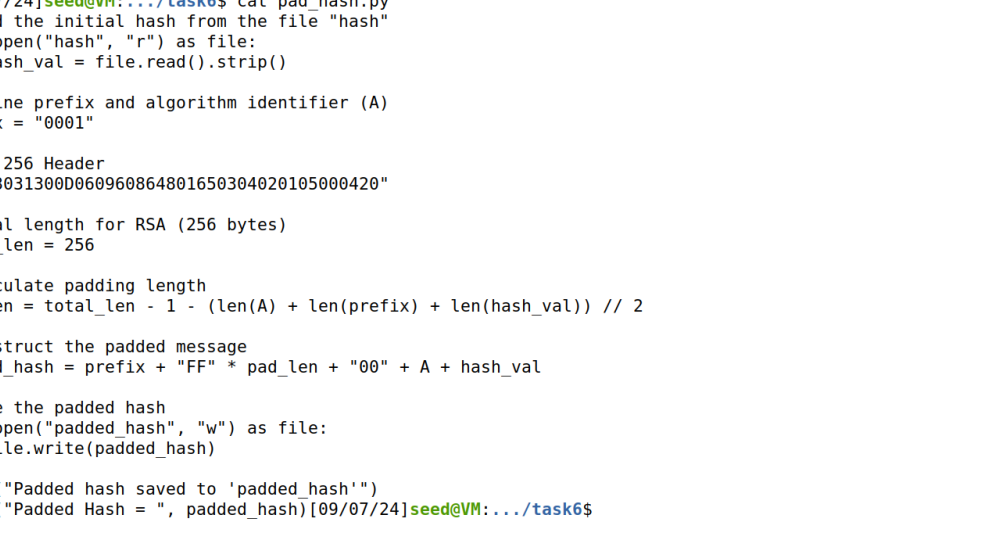
```
[09/07/24]seed@VM: .../task6$ openssl asn1parse -i -in c0.pem -strparse 4 -out c0_body.bin -noout
[09/07/24]seed@VM: .../task6$ sha256sum c0_body.bin
dc25f043f2f15589ca8281271002b45f166e187c5b60e2e7ec7ab982d20dc250  c0_body.bin
[09/07/24]seed@VM: .../task6$
```

- The hash value is `dc25f043f2f15589ca8281271002b45f166e187c5b60e2e7ec7ab982d20dc250`
- Note here that the sizes of modulus (`n`) and the signature are equal to 256 bytes (unlike `task5` where the size was just 32 bytes) and the size of the hash value is way smaller (32 bytes). So, for that reason we will need to pad the hash value to 256 bytes.
- The hash value (32 bytes) is wrapped into two successive layers:
 - In practice, if the hash value is `H`, then the first wrapping results in the sequence of bytes `A || H` where `"||"` is concatenation, and `"A"` is a header which is specific to

`"3031300D060960864801650304020105000420"` (20 bytes).

- The "A || H" is now expanded with some extra bytes:
0x00 0x01 0xFF 0xFF ... 0xFF 0x00 || A || H

- The number of bytes of value `0xFF` is adjusted to that the total size equals the size of the RSA modulus (i.e. 256 bytes for a 2048-bit RSA key).
- Below is the python code (`pad_hash.py`) which performs padding to the hash.



```
[09/07/24]seed@VM:.../task6$ cat pad_hash.py
# Read the initial hash from the file "hash"
with open("hash", "r") as file:
    hash_val = file.read().strip()

# Define prefix and algorithm identifier (A)
prefix = "0001"

# SHA 256 Header
A = "3031300D060960864801650304020105000420"

# Total length for RSA (256 bytes)
total_len = 256

# Calculate padding length
pad_len = total_len - 1 - (len(A) + len(prefix) + len(hash_val)) // 2

# Construct the padded message
padded_hash = prefix + "FF" * pad_len + "00" + A + hash_val

# Save the padded hash
with open("padded_hash", "w") as file:
    file.write(padded_hash)

print("Padded hash saved to 'padded_hash'")
print("Padded Hash = ", padded_hash)[09/07/24]seed@VM:.../task6$
```

```
# Read the initial hash from the file "hash"
with open("hash", "r") as file:
    hash_val = file.read().strip()

# Define prefix and algorithm identifier (A)
prefix = "0001"
A = "30 31 30 0D 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20".replace(' ', '')

# Total length for RSA (256 bytes)
total_len = 256

# Calculate padding length
pad_len = total_len - 1 - (len(A) + len(prefix) + len(hash_val)) // 2

# Construct the padded message
padded_message = prefix + "FF" * pad_len + "00" + A + hash_val

# Save the padded message to a file named "padded_hash"
with open("padded_hash", "w") as file:
    file.write(padded_message)

print("Padded hash saved to 'padded_hash'")
```

- The output of the above code:

[illegible]

- I now go back to `task5` folder and copy the C file, `task5.c`, where I can verify whether the signature is valid or not. Once I copy, I'll need to make a change in the code as the code I've written

takes the message in ASCII format. I'll need to change that to HEX format and then we compile and run the file `task5_modified.c` within the `task6` folder.

- We have
 - Modulus (n) (hex) =
 CCF710624FA6BB636FED905256C56D277B7A12568AF1F4F9D6E7E18FBD95ABF260411570DB12
 00FA270AB557385B7DB2519371950E6A41945B351BFA7BFABBC5BE2430FE56EFC4F37D97E314
 F5144DCBA710F216EAAB22F031221161699026BA78D9971FE37D66AB75449573C8ACFFEF5D0A
 8A5943E1ACB23A0FF348FCD76B37C163DCDE46D6DB45FE7D23FD90E851071E51A35FED494654
 7F2C88C5F4139C97153C03E8A139DC690C32C1AF16574C9447427CA2C89C7DE6D44D54AF4299
 A8C104C2779CD648E4CE11E02A8099F04370CF3F766BD14C49AB245EC20D82FD46A8AB6C93CC
 6252427592F89AFA5E5EB2B061E51F1FB97F0998E83DFA837F4769A1
 - Exponent (e) (hex) = 65537 (0x10001)
 - Message (hash value) (hex) =
 0001FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 003031300D060960864801650304020105000420dc25f043f2f15589ca8281271002
 b45f166e187c5b60e2e7ec7ab982d20dc250
 - Certificate Signature (hex) =
 2036ac50de6eb33a293f20a849f79f4ea051796cd2143a083bca1e362cddb3b725c
 b0b88a91e246bc07293c5a28859d36a31d08c82c2115ada42aef7c0908b7c75519b
 d8f7757c8f501c6c249a08ca29527015d6c948e6d9dfbfd8b00a06f7ed75fc02da67
 3876b9042faecffa9ac5ee013cad5118b037ada1d4a0d6ea270fea558d375d75164a
 e59eb3e36566b30e85bb4adb8e5be73ca81130cdd8f57a3a33203dc1e36690f33b21
 03edc9d35db37cfe790452570465832e9a4ffdfa7f07c9ddfe3eccb9943ce041004d
 ce304052d0662fdd255b1d493c0d80e25f83e2ba3d7f1899ab80fd1476f17e0571b9
 1c348ba22507b97a25e1f655cd68153e298c
- We now compile and run the `task5_modified.c` code.

```

[09/07/24]seed@VM: .../task6$ gcc task5_modified.c -o task5_modified -lcrypto
[09/07/24]seed@VM: .../task6$ ./task5_modified
*** Modified Code for Task 6 ***
Enter Public Key:
    e (DECIMAL): 65537
    n (HEX): CCF710624FA6BB636FED905256C56D277B7A12568AF1F4F9D06E7E18FBD95ABF260411570DB1200FA270AB557385B7DB2519
371950E6A41945B3518FA7BFA8BC5BE2430FE56EFC4F37D97E314F5144DCBA710F216EAB822F031221161699026BA78D9971FE37D66AB7544957
3C8ACFFEF5D0A8A5943E1ACB23A0FF348FCD76B37C163DCDE46D6DB45FE7D23FD90E851071E51A35FED4946547F2C88C5F4139C97153C03E8A13
9DC690C32C1AF16574C9447427CA2C89C7DE6D44D54AF4299A8C104C2779CD648E4CE11E02A8099F04370CF3F766BD14C49AB245EC20D82FD46A
8AB6C93CC6252427592F89AFA5E5EB2B061E51F1FB97F0F0998E83DFA837F4769A1
Enter Hash Value (HEX): 0001FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
0003031300D0609068648016503040201
05000420dc25f043f2f15589ca8281271002b45f166e187c5b60e2e7ec7ab982d20dc250
Enter Signature to be Verified (HEX): 2036ac50de6eb33a293f20a849f79f4ea051796cd2143a083bca1e362cddb3b725cb0b88a91e2
46bc07293c5a28859d36a3d10d8c82c2115ada42aeef7c0908b7c75519bd8f7757c8f501c6c249a08ca29527015d6c948e6d9dfbdf8b00a06f7ed
75fc02da673876b9042faecffa9ac5ee013cad5118b037ada1d4a0d6ea270fea558d375d75164ae59eb3e36566b30e85bb4adb8e5be73ca81130
cdd8f57a3a33203dc1e36690f33b2103edc9d35db37cfe790452570465832e9a4ffdafa7f07c9ddf3eccb9943ce041004dce304052d0662fdd25
5b1d493c0d80e25f83e2ba3d7f1899ab80fd1476f17e0571b91c348ba22507b97a25e1f655cd68153e298c
Signature is verified & valid.
[09/07/24]seed@VM: .../task6$

```

```
[09/07/24]seed@VM:.../task6$ gcc task5_modified.c -o task5_modified -lcrypto
[09/07/24]seed@VM:.../task6$ ./task5_modified
*** Modified Code for Task 6 ***
Enter Public Key:
    e (DECIMAL): 65537
    n (HEX):
CCF710624FA6BB636FED905256C56D277B7A12568AF1F4F9D6E7E18FBD95ABF260411570DB1200FA270AB557385B7BD2519371950E6A41945B3
51BFA7BFBABC5BE2430FE56EFC4F37D97E314F5144DCBA710F216EAB22F031221161699026BA78D9971FE37D66AB75449573C8ACFFEF5D0A8A
5943E1ACB23A0FF348FC7D6B37C163DCDE46D6DB45FE7D23FD90E851071E51A35FED4946547F2C88C5F4139C97153C03E8A139DC690C32C1AF1
6574C9447427CA2C89C7DE6D4454F4299A8AC104C2779CD648E4CB11E02A809FF4370CF3F766BD14C49AB245EC20D82FD46A8AB6C93CC6252
472592F89AFA5E5EB2B061E51F1F907098AE83DFA837F4769A1
```

```
0001FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
003031300D60960864801650304020105000420dc25f043f2f1
5589ca8281271002b45f166e187c5b60e2e7ec7ab982d20dc250
```

```
2036ac50de6eb33a293f20a849f7914ea051796cd2143a083bca1e362cddbdb3b725cb0b88a91e246bc07293c5a28859d36a31d08c82c2115ada
42aeef7c0908b7c75519bd8f7757c8f501c6c249a08ca29527015d6c948e6d9dfbdf8b00a06f7ed75fc02da673876b9042faecffa9ac5ee013c
ad5118b037adald40a06dea270fcae558d37f5164ae59eb3e3656b30e85bb4adb8e5be73ca81130cdd8f57a3a33203dc1e36690f33b2103edc
9d355b3bc7990452570465832e9a4f7dfaf7079cddffe3ecbc9943ce041004dce304052d0662fdd255b1d7493cdd08e025f83e2ba3d7f1f899ab80
fd1476f17e0571b91c348ba22507b97a25e1f655cd68153e298c
```