RSA PUBLIC-KEY ENCRYPTION AND SIGNATURE LAB

SEED Lab #7

Identification

- **Group nº3**
  - **José Rodrigues** : 201708806
  - **Paulo Ribeiro** : 201806505
  - **Pedro Ferreira** : 201806506

Task 1:

To calculate the private key `d`, we created the following script:

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

void printBN(char *msg, BIGNUM * a) {
    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 *minusone = BN_new();

    BIGNUM *p = BN_new();
    BIGNUM *q = BN_new();
    BIGNUM *e = BN_new();

    BIGNUM *p1 = BN_new();
    BIGNUM *q1 = BN_new();
    BIGNUM *r = BN_new();

    BIGNUM *pkey = BN_new();

    BN_dec2bn(&minusone, "-1");

    BN_hex2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF");
    BN_hex2bn(&q, "E85CED54AF57E53E092113E62F436F4F");
    BN_hex2bn(&e, "0D88C3");

    BN_add(p1, p, minusone);
    BN_add(q1, q, minusone);

    BN_mul(r, p1, q1, ctx);
    BN_mod_inverse(pkey, e, r, ctx);

    printBN("Private key:", pkey);

    return 0;
}
```

Basically, it starts by calculating the Euler's totient function using `Φ(n) = (p-1) * (q-1)`. Then, we need to pick an `e` value. This `e` value can be any number `e` where 1 < `e` < `Φ(n)` and `e` are coprime to `Φ(n)`, common values being the *Fermat Primes* (3, 17, and 65537). In this instance, it was already defined for us as 886979. Now we can compute the private key `d`, which is the modular multiplicative inverse of `e` (`mod Φ(n)`).

By executing that script, we obtained the value of the private key: `d = 3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB` We verified that this value is correct by entering the values into an online RSA calculator.

Task 2:

The first step is to get the hex encoding of the secret message, which is `4120746f702073656372657421`, corresponding to our plaintext `P`. We did this by running the following python command:

```
python -c 'print("A top secret!".encode("hex"))'
```

Now, to encrypt the message, we generate the ciphertext using $C = P^e \pmod{n}$. To verify the encryption result, we decrypted the resulting ciphertext using $P = C^d \pmod{n}$ and noticed that the plaintext remains the same. This procedure is done in the following script:

```
(...)

int main () {
    BN_CTX *ctx = BN_CTX_new();

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

    BIGNUM *C = BN_new();

    BIGNUM *secret = BN_new();

    BN_hex2bn(&n,
"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&e, "010001");
    BN_hex2bn(&M, "4120746f702073656372657421"); // Hex equivalent of "A top
secret!"
    BN_hex2bn(&d,
"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
```

```
    BN_mod_exp(C, M, e, n, ctx);

    printBN("Ciphertext:", C);

    BN_mod_exp(secret, C, d, n, ctx);

    printBN("Decrypted message:", secret);

    return 0;
}
```

Task 3:

In this task, the goal is to decrypt a ciphertext. For that, we repeat the procedure of the previous task, but now we just need to get the corresponding plaintext and convert it to an ASCII string. The following script does that:

```
(...)

int main () {
    BN_CTX *ctx = BN_CTX_new();

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

    BIGNUM *C = BN_new();

    BN_hex2bn(&n,
"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&C,
"8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");
    BN_hex2bn(&d,
"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

    BN_mod_exp(M, C, d, n, ctx);

    printBN("Decrypted message:", M);

    return 0;
}
```

By running it, we discovered that the secret message is `50617373776F72642069732064656573`. This is the hexadecimal value, and converting it to ASCII, the message is revealed as being `Password is dees`.

Task 4:

The first step is to get the hex encoding of both messages, the first one is `49206F776520796F752024323030302E`, and the second one is `49206F776520796F752024333030302E`. We did this by running the following python commands:

```
python -c 'print(" I owe you $2000.".encode("hex"))'
python -c 'print(" I owe you $3000.".encode("hex"))'
```

Now, we must generate the signatures for both messages, which is done by encrypting them with the private key, using $S = P^d \pmod n$:

```
(...)

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 *S1 = BN_new();
    BIGNUM *S2 = BN_new();

    BN_hex2bn(&n,
"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&d,
"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

    BN_hex2bn(&M1, "49206F776520796F752024323030302E");
    BN_hex2bn(&M2, "49206F776520796F752024333030302E");

    BN_mod_exp(S1, M1, d, n, ctx);
    BN_mod_exp(S2, M2, d, n, ctx);

    printBN("Hex message 1:", M1);
    printBN("Hex message 2:", M2);

    printBN("Signed message 1:", S1);
    printBN("Signed message 2:", S2);

    return 0;
}
```

We obtained the following signatures:

- S1 = `55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73CCB35E4CB`
- S2 = `BCC20FB7568E5D48E434C387C06A6025E90D29D848AF9C3EBAC0135D99305822`

As we can see, in the digital signatures above, even though the messages only differ on a single byte, their encrypted counterparts are extremely different due to the protection given by the modulus operation.

Task 5:

To verify the signature, we must decrypt it using the public key, using `P = Sᵉ (mod n)`. Then, we converted it to an ASCII string and noticed that the message is the same, meaning that the signature is indeed Alice's.

Then, we experimented corrupting the signature by changing its last byte from 2F to 3F:

- Decrypted message (hex): `4C61756E63682061206D697373696C652E`
- Corrupted decrypted message (hex): `91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294`
- Decrypted message (ASCII): `Launch a missile.`
- Corrupted decrypted message (ASCII): `□G□'Èñä,□O´c□è¼rm=fÈ:N¶·¾□□´□Â□`

As we can see, we got two entirely different results, where the plaintext of the corrupted signature is totally imperceptible. This happens because even though we're raising it to the same exponent, the base (the signature) is different, which leads to a completely different result.

The entire procedure is in the following script:

```c
(...)

int main () {
    BN_CTX *ctx = BN_CTX_new();

    BIGNUM *M = BN_new();
    BIGNUM *bad_M = BN_new();
    BIGNUM *S = BN_new();
    BIGNUM *bad_S = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *n = BN_new();

    BN_hex2bn(&S,
"643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");
    BN_hex2bn(&bad_S,
"643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F");
    BN_hex2bn(&e, "010001");
    BN_hex2bn(&n,
"AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");

    BN_mod_exp(M, S, e, n, ctx);

    printBN("Decrypted message:", M);

    BN_mod_exp(bad_M, bad_S, e, n, ctx);

    printBN("Corrupted decrypted message:", bad_M);
    return 0;
}
```

Task 6:

Step 1:

For this first step, we must save both certificates (the server CA and the intermediate CA) of a given web server. We chose to use the Twitter website (www.twitter.com).

Server certificate:

```
-----BEGIN CERTIFICATE-----
MIIGwzCCBaugAwIBAgIQAjf+8b3vOp6vWPt1+RUzwzANBgkqhkiG9w0BAQsFADBP
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMSkwJwYDVQQDEyBE
aWdpQ2VydCBUTFMgUlNBIFNIQTI1NiAyMDIwIENBMTAeFw0yMjAxMjQwMDAwMDBa
(...)
```

Intermediate certificate:

```
-----BEGIN CERTIFICATE-----
MIIEvjCCA6agAwIBAgIQBtjZBNVYQ0b2ii+nVCJ+xDANBgkqhkiG9w0BAQsFADBh
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
d3cuZGlnaWNlcnQuY29tMSAwHgYDVQQDExdEaWdpQ2VydCBHbG9iYWwgUm9vdCBD
(...)
```

Step 2:

To extract the public key (e, n) from the issuer's certificate, we extract the value of n using the command:

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

And we print out all the fields to find the exponent e using the command:

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

- Modulus (n): 00:c1:4b:b3:65:47:70:bc:dd:4f:58:db:ec:9c:ed: c3:66:e5:1f:31:13:54:ad:4a:66:46:1f:2c:0a:ec: 64:07:e5:2e:dc:dc:b9:0a:20:ed:df:e3:c4:d0:9e:

  (...)

- Exponent (e): 65537 (0x10001)

Step 3:

To extract the signature, we must print all the fields using the command:

```
openssl x509 -in c0.pem -text -noout
```

- Signature Algorithm: sha384WithRSAEncryption 90:e8:32:79:96:50:59:39:06:18:bc:07:40:7e:2f:54:d7:9d: 2d:2a:94:c0:08:7d:97:7f:d9:c0:86:d6:71:c0:2c:ef:b7:65: 13:d4:2c:98:68:bf:9f:5e:32:be:2a:1b:58:b1:bc:cf:d8:2d: (...)

We then need to remove the spaces and colons from the data, to obtain a hex string that can be used in our program. We do that by running:

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

- Signature without the spaces and colons: 90e8327996505939061 8bc07407e2f54d79d2d2a94c0087d977fd9c086d671c02cefb76513d42c9868bf9f5e32be2a1b58b1bccfd82d68be78a794f664b561dc1d9445a1bc470cee41a6cbd3e6 (...)

Step 4:

Now, we extract the body of the certificate by running the command:

```
openssl asn1parse -i -in c0.pem -strparse 4 -out c0_body.bin -noout
```

After this, we generate its hash by running:

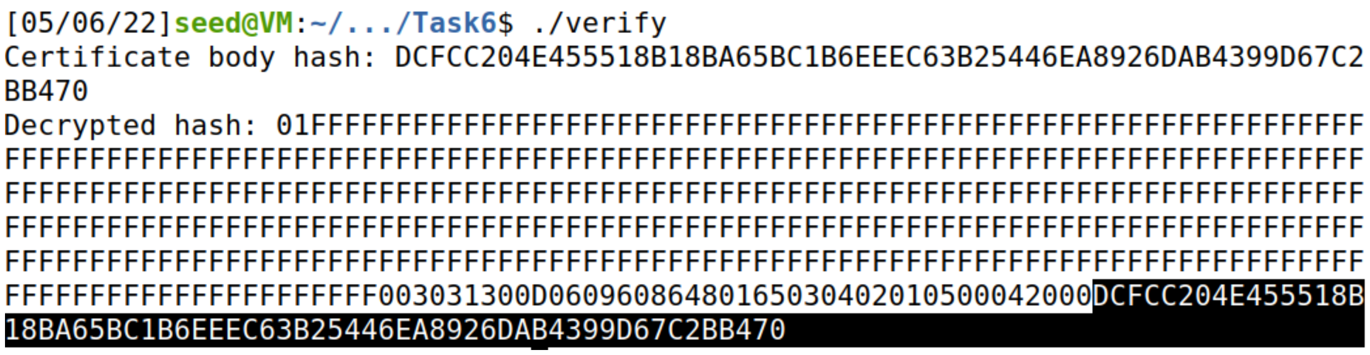
```
sha256sum c0_body.bin
```

Step 5:

Now that we have all the information we need, we can verify whether the signature is valid or not by comparing the hash of the certificate body with the plaintext obtained through `P = Sᵉ (mod n)`. The following program does that verification:

```
(...)

int main () {
    BN_CTX *ctx = BN_CTX_new();

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

    BN_hex2bn(&e, "10001");
    BN_hex2bn(&n,
"C14BB3654770BCDD4F58DBEC9CEDC366E51F311354AD4A66461F2C0AEC6407E52EDCDCB90A20EDDFE
3C4D09E9AA97A1D8288E51156DB1E9F58C251E72C340D2ED292E156CBF1795FB3BB87CA25037B9A524
16610604F571349F0E8376783DFE7D34B674C2251A6DF0E9910ED57517426E27DC7CA622E131B7F238
825536FC13458008B84FFF8BEA75849227B96ADA2889B15BCA07CDFE951A8D5B0ED37E236B4824B62B
5499AECC767D6E33EF5E3D6125E44F1BF71427D58840380B18101FAF9CA32BBB48E278727C52B74D4A
8D697DEC364F9CACE53A256BC78178E490329AEFB494FA415B9CEF25C19576D6B79A72BA2272013B5D
03D40D321300793EA99F5");
    BN_hex2bn(&S,
"90e8327996505939 0618bc07407e2f54d79d2d2a94c0087d977fd9c086d671c02cefb76513d42c986
8bf9f5e32be2a1b58b1bccfd82d68be78a794f664b561dc1d9445a1bc470cee41a6cbd3e63b3f4e17d
67c8f1bbba617061bb9d2b7b135c518432745c20726f307a20f84b4daca730354d4f543a1e3da9001e
c9c7c5d098b435f7ecafce7a49ebaa8a9b22ca4245913344232853 04dccf02da71f7fa3fd457720cea
0628af29c42fa2d6d785a20886c18e91cd7901d8ca7901c0e417430421918ec8c1be076aa7eb40bc77
ee21fe310832b4c90f0e3441b36f8831866c3ae4370b02aae9eeb613b6961411929a7b7d4e36f73666
63c8c38b377eb8e4f20f6");
    BN_hex2bn(&cmp_M,
"00dcfcc204e455518b18ba65bc1b6eeec63b25446ea8926dab4399d67c2bb470");

    BN_mod_exp(M, S, e, n, ctx);

    printBN("Certificate body hash:", cmp_M);
    printBN("Decrypted hash:", M);

    return 0;
}
```

After running it, we noticed that the ending of the resulting plaintext (highlighted in black) is equivalent to the certificate body hash, so we conclude that the signature is valid.



```
[05/06/22]seed@VM:~/.../Task6$ ./verify
Certificate body hash: DCFCC204E455518B18BA65BC1B6EEEC63B25446EA8926DAB4399D67C2
BB470
Decrypted hash: 01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFF003031300D060960864801650304020105000420 00DCFCC204E455518B
18BA65BC1B6EEEC63B25446EA8926DAB4399D67C2BB470
```