

---

# RSA PUBLIC-KEY ENCRYPTION AND SIGNATURE LAB

---

April 8, 2020

César Pinho - up201604039@fe.up.pt

João Barbosa - up201604156@fe.up.pt

Rui Guedes - up201603854@fe.up.pt

Faculdade de Engenharia da Universidade do Porto

## Introduction

RSA (RivestShamirAdleman) is one of the first public-key cryptosystems and is still widely used for secure data transmission. Its security is derived from the difficulty of factoring large integers that are the product of two large prime numbers. Since, to this date, there hasn't been discovered an efficient non-quantum integer factorization algorithm, the algorithm remains secure as long as the keys are sufficiently large (NIST recommends a length of 2048 bits). In this lab, we go hands-on into the RSA algorithm, using the big number library provided by OpenSSL. We start by experimenting with its various essential steps such as calculating the private key, encrypting and decrypting a message, and generating and verifying a message signature. Finally, we apply these concepts in a more practical context, specifically, to verify the signature of an X.509 certificate.

## Task 1 - Deriving a Private Key

We're tasked with deriving the private key  $d$  from the public key  $(e, n)$ , knowing three prime numbers  $p, q, e$ :  $n=p*q$ . This requires finding a private key where  $e*d \bmod (n) = 1$ . Since we know  $p$  and  $q$ , we can easily calculate the totient of  $n$  ( $\phi(n) = (p-1)*(q-1)$ ). Using the following source code, we performed these steps for deriving the private key.

```
int main()
{
    // Variables declaration
    BN_CTX *ctx = BN_CTX_new();

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

    BIGNUM *one = BN_new();
    BIGNUM *p_minus_one = BN_new();
    BIGNUM *q_minus_one = BN_new();
    BIGNUM *tot_n = BN_new();

    // Initialize <p>, <q>, <e>, <one> values
    BN_dec2bn(&one, "1");
```

```

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

// Calculate <n>
BN_mul(n, p, q, ctx);

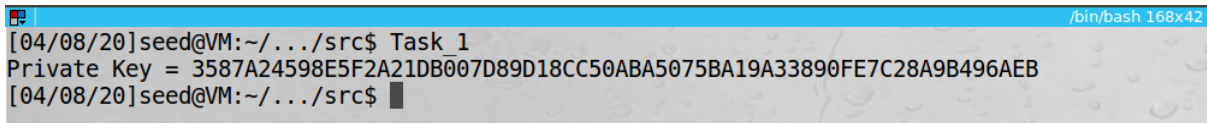
// Calculate the tot_n = (p - 1)*(q - 1)
BN_sub(p_minus_one, p, one);
BN_sub(q_minus_one, q, one);
BN_mul(tot_n, p_minus_one, q_minus_one, ctx);

// Calculate private key: e*d mod totient(n) = 1
BN_mod_inverse(d, e, tot_n, ctx);

// Print final information
printBN("Private Key =", d);
};

```

Executing this program, we get the following result.



```

[04/08/20]seed@VM:~/.../src$ Task_1
Private Key = 3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB
[04/08/20]seed@VM:~/.../src$

```

**Figure 1:** Deriving a Private Key

## Task 2 - Encrypting a Message

Knowing the public key  $(e, n)$ , we have to encrypt the message  $M = \text{"A top secret!"}$ . By the RSA Algorithm, we get an encryption of the message  $C$  by performing the operation  $C = M^e \bmod n$ .

In order to verify our encryption, we're also given access to the private key  $d$ . The encryption is valid if we obtain the original message back by decrypting it, which means  $M = C^d \bmod n$ . Below we showcase the source code used to encrypt the message and validate the encryption.

```

int main()
{
    // Variables declaration
    BN_CTX *ctx = BN_CTX_new();

```

```

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

// Variable M for validation purposes
BIGNUM *M_Decrypt = BN_new();

// Initialize <e>, <d>, <n>, <M> values
BN_hex2bn(&e, "010001");

// python -c 'print("A top secret!".encode("hex"))'
BN_hex2bn(&M, "4120746f7020736563726574421");

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

// Calculate <C> = M^e mod n
BN_mod_exp(C, M, e, n, ctx);

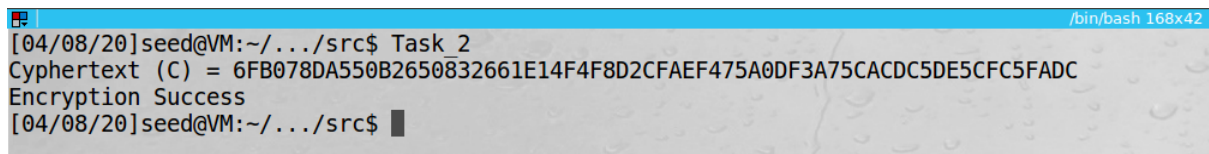
// Calculate <M_Decrypt> = C^d mod n
BN_mod_exp(M_Decrypt, C, d, n, ctx);

// Print final information
printBN("Cyphertext (C) =", C);

if(BN_cmp(M_Decrypt, M) == 0)
    printf("Encryption Success\n");
else
    printf("Encryption Failed\n");
};

```

Executing this program, we get the following result, demonstrating the validity of the encryption method.



```

[04/08/20]seed@VM:~/.../src$ Task_2
Cyphertext (C) = 6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC
Encryption Success
[04/08/20]seed@VM:~/.../src$

```

**Figure 2:** Encrypting a Message

## Task 3 - Decrypting a Message

Using the same keys as the previous task, we wish to decrypt a cyphertext  $C$ , in order to get the encrypted message back. As aforementioned, we can recover the original message  $M$  back using the private key  $M = C^d \bmod n$ .

```
int main()
{
    // Variables declaration
    BN_CTX *ctx = BN_CTX_new();

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

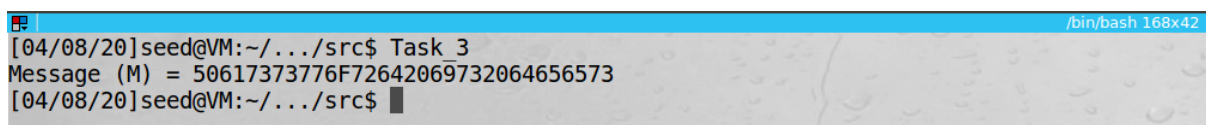
    // Initialize <e>, <d>, <n>, <C> values
    BN_hex2bn(&e, "010001");
    BN_hex2bn(&C, "8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBBDFC7DCB67396567EA1E2493F");
    BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");

    // Calculate <M> = C^d mod n
    BN_mod_exp(M, C, d, n, ctx);

    // Print final information
    printBN("Message (M) =", M);

    // python -c 'print("M".decode("hex"))' —> M = Password is dees
};
```

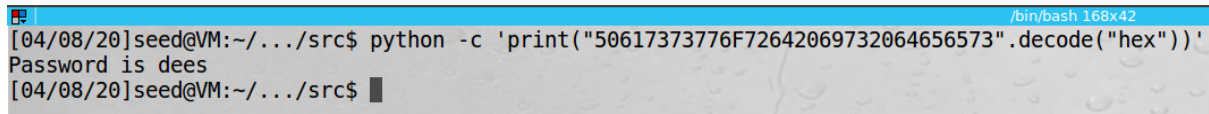
Executing the following program, we obtain the original message back in the hexadecimal format.



```
[04/08/20]seed@VM:~/.../src$ Task 3
Message (M) = 50617373776F72642069732064656573
[04/08/20]seed@VM:~/.../src$
```

**Figure 3:** Decrypting a Message

Converting the hex to a character format, we can read the content of the message.



```

[04/08/20]seed@VM:~/.../src$ python -c 'print("50617373776F72642069732064656573".decode("hex"))'
Password is dees
[04/08/20]seed@VM:~/.../src$

```

**Figure 4:** Decrypting a Message

## Task 4 - Signing a Message

With the same keys as the previous two tasks, we're tasked with signing two different messages, both differing in a single character only. The method for signing a message is similar to its encryption. Its difference lies in using the private key instead of the public key, since our objective is now to provide authenticity, not of confidentiality  $S = M^d \bmod n$ .

```

int main()
{
    // Variables declaration
    BN_CTX *ctx = BN_CTX_new();

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

    // Initialize <e>, <d>, <n>, values
    BN_hex2bn(&e, "010001");
    BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");

    // python -c 'print("I owe you $2000.".encode("hex"))'
    BN_hex2bn(&M, "49206f776520796f752024323030302e");

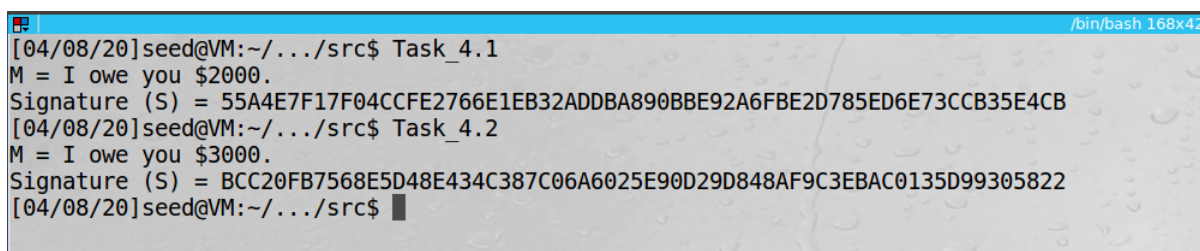
    // python -c 'print("I owe you $3000.".encode("hex"))'
    //BN_hex2bn(&M, "49206f776520796f752024333030302e");

    // Calculate <S> = M^d mod n
    BN_mod_exp(S, M, d, n, ctx);

```

```
// Print final information
printBN("Signature (S) =", S);
};
```

When applying the above source code at both messages, we get two completely different signatures, despite the original messages differing in a single byte only. This confers some collision resistance to the algorithm since there's no immediately apparent correlation between the message and the generated signature.



```
[04/08/20]seed@VM:~/.../src$ Task_4.1
M = I owe you $2000.
Signature (S) = 55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73CCB35E4CB
[04/08/20]seed@VM:~/.../src$ Task_4.2
M = I owe you $3000.
Signature (S) = BCC20FB7568E5D48E434C387C06A6025E90D29D848AF9C3EBAC0135D99305822
[04/08/20]seed@VM:~/.../src$
```

**Figure 5:** Signing a Message

## Task 5 - Verifying a Signature

We now wish to verify the signature  $S$  of a message received from Alice, in order to validate the authenticity of its sender. The rationale is similar, the message  $M$  was sent by Alice if  $M = S^e \bmod n$ , where  $(e, n)$  is Alice's public key.

Below is showcased the source code for verifying this signature of Alice's message, with knowledge of her public key. Observing the result we can conclude that Alice was indeed the sender of the message.

```
int main()
{
    // Variables declaration
    BN_CTX *ctx = BN_CTX_new();

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

    // Initialize <e>, <n>, <M>, <S> values
    BN_hex2bn(&e, "010001");
```

```

BN_hex2bn(&n, "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");

// python -c 'print("Launch a missile.".encode("hex"))'
BN_hex2bn(&M, "4c61756e63682061206d697373696c652e");

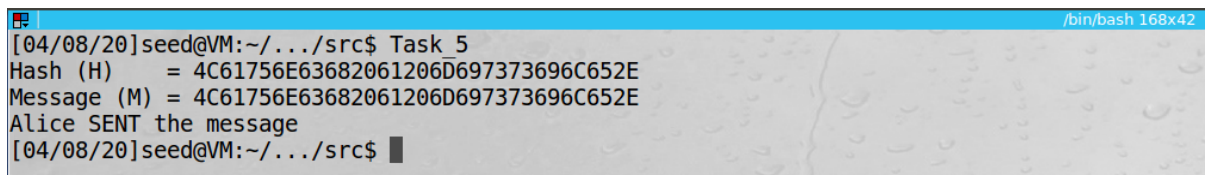
BN_hex2bn(&S, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");
//BN_hex2bn(&S, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F");

// Calculate <H> = S^e mod n
BN_mod_exp(H, S, e, n, ctx);

// Print final information
printBN("Hash (H)      =", H);
printBN("Message (M)   =", M);

if (BN_cmp(H, M) == 0)
    printf("Alice SENT the message\n");
else
    printf("Alice NOT SENT the message\n");
};

```



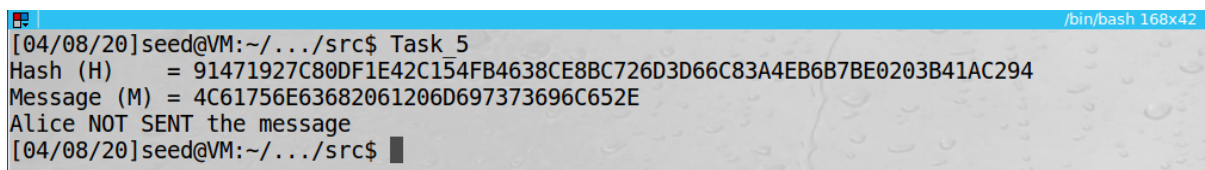
```

[04/08/20]seed@VM:~/.../src$ Task_5
Hash (H)      = 4C61756E63682061206D697373696C652E
Message (M)   = 4C61756E63682061206D697373696C652E
Alice SENT the message
[04/08/20]seed@VM:~/.../src$

```

**Figure 6:** Verifying a Signature

We've experimented with tweaking the original signature from Alice, changing one of the bytes, to simulate a corrupted signature. When applying the same process, there is now a mismatch between the expected message and the original message, therefore we have no guarantee of the message sender's authenticity.



```

[04/08/20]seed@VM:~/.../src$ Task_5
Hash (H)      = 91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294
Message (M)   = 4C61756E63682061206D697373696C652E
Alice NOT SENT the message
[04/08/20]seed@VM:~/.../src$

```

**Figure 7:** Verifying a Corrupted Signature



## Task 6 - Manually Verifying an X.509 Certificate

As a final task, we are given the liberty of choosing a certificate from a web server and verify it. This is similar to the previous task, with the caveat of needing to acquire the certificate's body signature and its issuer public key before comparing the actual and validated/expected content.

We've decided to use Facebook's X.509 Certificate for this task. Following the Lab's instructions, we extracted the body and signature  $S$  from this certificate and extracted the public key  $(e, n)$  of its issue certificate, with which we can now, following the same process as Task 5, verify the signature in Facebook's certificate.

```
int main()
{
    // Variables declaration
    BN_CTX *ctx = BN_CTX_new();

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

    // Initialize <e>, <n>, <M>, <S> values
    BN_hex2bn(&e, "010001");
    BN_hex2bn(&M, "cbd2ae848257a02c4860c2245c5e6b76f6d796674a31cb0c9a0cea8787d1d6de");
    BN_hex2bn(&n, "B6E02FC22406C86D045FD7EF0A6406B27D22266516AE42409BCEDC9F9F76073EC3...");

    BN_hex2bn(&S, "8269386da0a5909fb8a7f59f436ca062659fe0dc50cb1b69ed93031d88a17b74e2...");

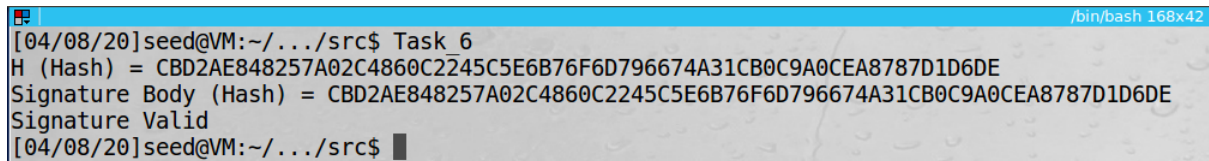
    // Calculate <H> = S^e mod n
    BN_mod_exp(H, S, e, n, ctx);

    // Truncate hash value to 256 bits
    BN_mask_bits(H, 256);

    // Print final information
    printBN("H (Hash) =", H);
    printBN("Signature Body (Hash) =", M);

    if (BN_cmp(H, M) == 0)
        printf("Signature Valid\n");
}
```

```
else  
    printf("Signature Invalid\n");  
};
```

A terminal window with a blue title bar containing a window icon and the text "/bin/bash 168x42". The terminal text shows a user running a program to verify a certificate. The output displays a hash, a signature body hash, and a "Signature Valid" message.

```
[04/08/20]seed@VM:~/.../src$ Task_6  
H (Hash) = CBD2AE848257A02C4860C2245C5E6B76F6D796674A31CB0C9A0CEA8787D1D6DE  
Signature Body (Hash) = CBD2AE848257A02C4860C2245C5E6B76F6D796674A31CB0C9A0CEA8787D1D6DE  
Signature Valid  
[04/08/20]seed@VM:~/.../src$ █
```

**Figure 8:** Manually Verifying an X.509 Certificate

Executing the above program, and observing the obtained result, we conclude that the certificate is valid.