16-10-2024

Session 6: ECDSA for real implementations

Nombre: Álvarez García Brandon Azarael

Nombre de la materia: Selected topics in cryptography

Grupo: 7CM1

Nombre del profesor: Dra. Sandra Díaz Santiago

Ejercicios de programación

1. Implement a function to generate a key pair for ECDSA. The private key and the public key must go to different textfiles. You must store each key using base64.

Para esta implementación utilizaremos la librería Crypto con sus diferentes módulos, la cual nos ayudara a implementar lo necesario para esta practica

```python
# Function to generate key pair
1 usage
def generate_key_pair():

    # Generate private key
    private_key = ECC.generate(curve='P-256') # P-192, P-224

    # Save public key
    public_key = private_key.public_key()

    with open("private_key.txt", "w") as priv_file:
        priv_file.write(b64encode(private_key.export_key(format='DER')).decode('utf-8'))

    with open("public_key.txt", "w") as pub_file:
        pub_file.write(b64encode(public_key.export_key(format='DER')).decode('utf-8'))

```

Esta función nos ayudara a crear el par de llaves tanto publica como privada, todo se realiza mientras la función ECC.generate , la cual lleva como parámetro una curva estandarizada, yo escogí la P-256, sin embargo soporta otras, como P-192, P-224 etc, de ahí extraemos la llave publica y privada, y las guardamos en un archivo de texto por separado, así como la codificamos en base 64.

Adicionalmente, para los ejercicios posteriores utilizamos la cardinalidad, sin embargo, dentro de la librería no hay algún método que te retorne esta, así que cree una función que devuelve la cardinalidad de la curva P-256

```python
3 usages
def get_curve_order():
    return int("FFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551", 16)
```

2. Implement a function to do the signature generation for ECDSA. Your function must receive a private key d and a valid message, i.e. $0 < m < q$ Your function must return the pair (r, s). When you test your function, consider the public parameters must be already fixed .

```python
def sign_message(m):

    with open("private_key.txt", "r") as priv_file:
        private_key = ECC.import_key(b64decode(priv_file.read()))

    q = get_curve_order()

    if not (0 < m < q):
        raise ValueError("El mensaje debe ser un entero entre 0 y q")

    # Create hash(x)
    h = SHA256.new(str(m).encode('utf-8'))

    # Create signature
    signer = DSS.new(private_key, 'fips-186-3')

    # Get pair r and s
    signature = signer.sign(h)

    # deconstructing r, s
    r, s = int.from_bytes(signature[:len(signature) // 2], 'big'), int.from_bytes(signature[len(signature) // 2:],
                                                                                  'big')
    return r, s

```

En este función leemos el archivo de texto donde contenemos nuestra llave privada y hacemos uso de la función creada previamente para obtener el orden, esta función lleva como parámetro un mensaje m representado en enteros y utilizamos una función hash , la más conocida es la SHA256 y creamos la firma, posteriormente obtenemos los parámetros r y s , sin embargo la función que devuelve la firma, los devuelve pero de forma concatenada, así que lo separamos en variables independientes y las retornamos.

3. Design and implement a function to do the signature verification for ECDSA. Your function must read the textfile where the public key is stored and must receive the message m, and the pair (r, s) and must return a boolean value: true if the signature is valid or false if it is not valid.

```python
def verify_signature(m, r, s):

    with open("public_key.txt", "r") as pub_file:
        public_key = ECC.import_key(b64decode(pub_file.read()))

    q = get_curve_order()

    if not (0 < m < q):
        raise ValueError("El mensaje debe ser un entero entre 0 y q")

    # Create hash(x)
    h = SHA256.new(str(m).encode('utf-8'))

    # Constructing signature
    signature = r.to_bytes(32, 'big') + s.to_bytes(32, 'big')

    # Verify signature
    verifier = DSS.new(public_key, 'fips-186-3')
    try:
        verifier.verify(h, signature)
        return True
    except ValueError:
        return False

```

Para esta función vamos a usar loa parámetros previamente calculados, r,s y el mensaje, posteriormente obtenemos la llave publica de nuestro archivo y utilizamos la función hash256, definida a base del mensaje y vovlemos a generar la firma como la habíamos obtenido inicialmente (sin desestructurarla en dos parámetros) y procedemos a verificar con la función, pasando la firma y el valor de nuestro hash.

4. Design and implement a computer program to test the previous functions. **Please avoid to edit your source code to change any parameter of ECDSA**
5. Run your program for at least 5 different files.

```
82  def test_program():
83
84      generate_key_pair()
85
86      for i in range(5):
87          # Generate m
88          q = get_curve_order()
89          m = randint(1, q - 1)
90
91          # Signature generation
92          r, s = sign_message(m)
93          print(f"Firma para el mensaje: {m}:")
94          print(f"r: {r}")
95          print(f"s: {s}")
96
97          # Verify the signature
98          is_valid = verify_signature(m, r, s)
99          print(f"¿Firma válida?: {is_valid}")
100         print("-" * 40)
101
```

Para el punto 4,5 cree una función que junta las funciones previamente creadas y el proceso se repite las 5 veces que se mencionan.

Resultados.

Para los resultados, cree otro archivo Python, en donde simplemente mande a llamar la última función creada test_program(), para no juntar todo en un solo archivo.

```
1
2  from ECDSA import test_program
3
4
5  test_program()
6
7
```

```
D:\Programs\Anaconda\python.exe "D:\ESCOM\10°Semestre\Cripto 2\Practicas\Practice_7\main.py"
Firma para el mensaje: 950852881309313538496607114726221397196435018727730214702678834156735783307 37:
r: 24557598431471532601339871813227285702077021660032891144409419466840363581432
s: 10397362111048103691093917439248766086634718022949597499121798044409814591813 2
¿Firma válida?: True
----------------------------------------
Firma para el mensaje: 587421248417192587509678066154999855485265998762265766029109335254302231668 63:
r: 96740904839356759372657391270131817320678160422133697103878210804306739121551
s: 21597047152439674985624917214392159489126215054807112026413995419571901106246
¿Firma válida?: True
----------------------------------------
Firma para el mensaje: 107086489152058785355931406590882574078191936364910247109893572788495632659843:
r: 73839794726862946425605666448990365299018483940216035345977990722810689754661
s: 154115357813858131442421740278741712540780468317414140956126605259074705641 00
¿Firma válida?: True
----------------------------------------
Firma para el mensaje: 466533273251653090645809214588120644840949852963971754628311364366585961050 05:
r: 85034077339628501111951785792099772587723752175863448881108413208756328467025
s: 974276321155869120453442361596910047636387740705154198747982280306316081211 20
¿Firma válida?: True
----------------------------------------
Firma para el mensaje: 741821000084122545947941775567447415021428663768101027263283413665774497287 06:
r: 11729198881147292416483001479971379469332353889345989177228151366821057806853
s: 11078834286980157104360791742968820836614318186453841396132634097614581812124 2
¿Firma válida?: True
----------------------------------------

Process finished with exit code 0
```