TP1 - Estruturas Criptofráficas

Elementos do grupo 4:

- André Morais, A83899
- Tiago Magalhães, A84485

Parte 2

KEM - RSA

Para gerar chave pública e chave privada, primeiramente tivemos que gerar os parâmetros:

- **q** e **p**, que são dois números primos de modo a que o módulo **n** tenha como tamanho de parâmetro segurança
- **phi** servirá para calcular o expoente da chave pública

In [1...

```
import os
import math
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.hashes import Hash,SHA256

# Secure hash FIPS 180
def hash_sha256(seed):
    digest = hashes.Hash(hashes.SHA256())
    digest.update(seed)
    digest.update(seed)
    return digest.finalize()

class KEM_RSA:
    def __init__(self, param):
        p = random_prime(2^(param/2) - 1, 2^(param/2-1))
        q = random_prime(2^(param/2) - 1, 2^(param/2-1))
        self.n = p * q

        # Função totiente de Euler
        phi = (p - 1)*(q - 1)

        e = self.gen_e(phi)
        d = self.gen_d(phi, e)

        self.param = param

        # Public Key: (n,e)
        self.public_key = (self.n, e)
        # Private Key: (n,d)
        self.private_key = (self.n, d)
```

Após definidos estes parâmetros, podemos começar a definir as chaves públicas e privadas. Para isso foram definidas duas funções:

- **gen_e** : Escolhe uma chave pública (**n**, **e**) tal que, 0 < **e** < **phi(n)** & co-primo com **n** e

phi(n)

- **gen_d** : Escolhe uma chave privada(**n**, **d**) tal que, **d*e** *mod* **phi(n)** = 1

In [1...

```
class KEM_RSA(KEM_RSA):
    def gen_e(self, phi):
        # 1 < e < phi
        e = ZZ.random_element(phi)
        # coprime com phi e N
        while gcd(e, phi) != 1:
            e = ZZ.random_element(phi)
        return e

    def gen_d(self, phi, e):
        # Inversa modular
        d = inverse_mod(e, phi)
        return d
```

Para aplicarmos a técninca KEM construímos dois algoritmos, o **encaps**, vocacionado para ofuscar pequenas quantidades de informação,"chaves", que ele próprio gera. O **decaps** revela a chave partir do encapsulamento de desta.
Para o encaps usamos o seguinte algoritmo:

- Gerar inteiro aleatório z entre 0 e n - 1;
- Cifrar z com a chave pública rsa, obtendo-se o encapsulamento;
- Derivar chave simétrica k através de um kdf, em que k = kdf(z).

O decaps segue a seguintes operações:

- Decifra o encapsulamento com a sua chave privada, obtendo z;
- Com z, deriva a chave simétrica k através de um kdf, em que k = kdf(z).

In [1...

```
class KEM_RSA(KEM_RSA):
    def encrypt_asym(self, pub_key, msg):
        n, e = pub_key
        return pow(msg, e, n)

    def decrypt_asym(self, priv_key, ct):
        n, d = priv_key
        return pow(ct, d, n)

    def encaps(self, pub_key):
        n, ex = pub_key
        # 1 < z < n
        z = ZZ.random_element(n)
        z_as_bytes = int(z).to_bytes(int(z).bit_length() + 7 // 8, 'big')

        salt = os.urandom(16)
        key = self.kdf(z_as_bytes, salt)

        e = self.encrypt_asym(pub_key, z)
        e_as_bytes = int(e).to_bytes(int(e).bit_length() + 7 // 8, 'big')

        return key, e_as_bytes, salt

    def decaps(self, e, salt):
        e_int = int.from_bytes(e, 'big')

        z = self.decrypt_asym(self.private_key, e_int)
        z_as_bytes = int(z).to_bytes(int(z).bit_length() + 7 // 8, 'big')

        key = self.kdf(z_as_bytes, salt)
        return key

    def kdf(self, password, salt):
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=32,
            salt=salt,
            iterations=100000,)
        key = kdf.derive(password)
        return key
```

Verificação/Teste dos algoritmos

In [1...

```
Bob_rsa = KEM_RSA(1024)
Alice_rsa = KEM_RSA(1024)

alice_pub_key = Alice_rsa.public_key

k, e, salt = Bob_rsa.encaps(alice_pub_key)
print("Chave partilhada: ", k)

key  = Alice_rsa.decaps(e, salt)
print("Chave partilhada: ",key)
```

```
Chave partilhada:  b'E\xc32n\xff]i\xb2\xc2`\x89k[c\x99\xf79B\x80\x04\xef-\x
ec7\xaaJp%\xc1\x8f\xf8\xa9'
Chave partilhada:  b'E\xc32n\xff]i\xb2\xc2`\x89k[c\x99\xf79B\x80\x04\xef-\x
ec7\xaaJp%\xc1\x8f\xf8\xa9'
```

Transformar um KEM em um PKE-IND-CCA usando FOT

In [1...

```
class PKE_RSA:
    def __init__(self, param, salt):
        p = random_prime(2^(param/2) - 1, 2^(param/2-1))
        q = random_prime(2^(param/2) - 1, 2^(param/2-1))
        self.n = p * q
        # Função totiente de Euler
        phi = (p - 1)*(q - 1)
        e = self.gen_e(phi)
        d = self.gen_d(phi, e)

        self.param = param

        # Public Key: (n,e)
        self.public_key = (self.n, e)
        # Private Key: (n,d)
        self.private_key = (self.n, d)

        self.salt = salt

    def gen_e(self, phi):
        # 1 < e < phi
        e = ZZ.random_element(phi)
        # coprime com phi e N
        while gcd(e, phi) != 1:
            e = ZZ.random_element(phi)
        return e

    def gen_d(self, phi, e):
        # Inversa modular
        d = inverse_mod(e, phi)
        return d

    def otp_enc(self, key, msg):
        return bytes(a ^^ b for a, b in zip(msg, key))

    def otp_dec(self, key, ct):
        return bytes(a ^^ b for a, b in zip(ct, key))

    def encrypt_sym(self, pub_key, msg):
        k, e = self.encaps(pub_key)
        ct = self.otp_enc(k, msg)
        return e, ct

    def decrypt_sym(self, e, ct):
        k = self.decaps(e)
        pt = self.otp_dec(k, ct)
        return pt

    def hash_g(self, msg):
        h = hashes.Hash(hashes.SHA3_256())
        h.update(msg)
        digest = h.finalize()
        return digest

    def encrypt_asym(self, pub_key, msg):
        n, e = pub_key
        return pow(msg, e, n)

    def decrypt_asym(self, priv_key, ct):
        n, d = priv_key
        return pow(ct, d, n)
```

```
    def encrypt(self, pub_key, msg):
        n, e = pub_key

        x = ZZ.random_element(n)

        # Gerar r
        r = os.urandom(32)
        r_as_bytes =  r
        r_as_int = int.from_bytes(r_as_bytes,"big")

        msg_as_bytes = msg.encode('utf-8')

        # Obter y ofuscando o plaintext x
        y = bytes(a ^^ b for a, b in zip( msg_as_bytes, self.kdf(r_as_bytes

        # Obter yr
        yr = y + r_as_bytes
        yr_as_int = int.from_bytes(yr, "big")

        # (e,k) <- KEM
        k = self.kdf(yr,32)
        e = self.encrypt_asym(pub_key, yr_as_int)
        e_as_bytes = int(e).to_bytes((int(e).bit_length() + 7) // 8, 'big'

        # Obter tag c ofuscando r com chave k
        c = self.otp_enc(k, r_as_bytes)

        return y, e_as_bytes , c

    def decrypt(self, y, e, c):
        k = self.decaps(e)

        r = self.otp_dec(k, c)
        r_as_int = int.from_bytes(r,"big")
        yr = y + r
        yr_as_int = int.from_bytes(yr, "big")

        ee = self.encrypt_asym(self.public_key, yr_as_int)
        ee = int(ee).to_bytes((int(ee).bit_length() + 7) // 8, 'big')

        kk = self.kdf(yr,32)

        if ee != e or kk != kk:
            return "Error"
        else:
            pt = bytes(a ^^ b for a, b in zip(y, self.kdf(r,len(y))))
            return pt


    def kdf(self, password, len):
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=len,
            salt=self.salt,
            iterations=100000,)
        key = kdf.derive(password)
```

In [1...

```
salt = os.urandom(16)

msg = "Secret Message"
msg_as_bytes = str.encode(msg)

Bob_rsa = PKE_RSA(1024,salt)
Alice_rsa = PKE_RSA(1024, salt)

alice_pub_key = Alice_rsa.public_key

y, e, c = Bob_rsa.encrypt(alice_pub_key, msg)


pt  = Alice_rsa.decrypt(y, e, c)
print(pt.decode())
```

```
Secret Message
```

DSA

Neste exercício temos de implementar um algoritmo de DSA

Foi dividida em 4 processos:

- Gerar parâmetros
- Gerar chaves
- Gerar assinatura da mensagem
- Verificação da assinatura de uma mensagem

Gerar parâmetros

Precisamos de gerar 3 parâmetros de domíno iniciais que são a base deste algoritmo:

- **p** : Um número primo de N-bits
- **q** : Um número primo de L-bits e p-1 é múltiplo de q
- **g** : Escolher um gerador g com expressão: h^ ((p-1)/q) *mod* p

Para gerar estes parâmetros baseamo-nos no FIPS PUB 186-4.

In [1...

```
class DSA:
    def __init__(self, n, l):
        q, p ,g = self.gen_params(n, l)
        self.q = q
        self.p = p
        self.g = g

        priv, pub = self.gen_key()
        self.priv_key = priv
        self.pub_key = pub

    def gen_params(self, n, l):
        p, q = self.gen_pq(n,l)
        g = self.gen_g(p, q)
        return q, p, g

    def gen_g(self, p, q):
        h = ZZ.random_element(1, p-1)
        e = (p-1)/q
        g = pow(h, int(e), p)
        while g == 1:
            h = ZZ.random_element(1, p-1)
            g = pow(h, int(e), p)
        return g

    def gen_pq(self, N, L):
        outlen = 256 # hash length output
        seedlen = N

        n = (L//outlen) - 1
        b = L - 1 - (n * outlen)
        i = 0
        q = 0
        p = 0
        is_valid_p = False
        while is_valid_p == False:
            while is_prime(q) == False:
                domain_parameter_seed = os.urandom(N)
                hash_res = hash_sha256(domain_parameter_seed)
                hash_int = int.from_bytes(hash_res,'big')
                U = hash_int % pow(2,(N-1))
                q = pow(2,N-1) + U + 1 - (U % 2)
            offset = 1

            for counter in range(0,4*L):
                V = []
                for j in range(0,n+1):
                    x = int((int.from_bytes(domain_parameter_seed,'big') +
                    y = x % (pow(2,seedlen))
                    y_as_bytes = int(y).to_bytes(int(y).bit_length() + 7 //
                    hash_res = hash_sha256(y_as_bytes)
                    hash_as_int = int.from_bytes(hash_res, 'big')
                    V.append(hash_as_int)
                W = 0
                for i in range(0,len(V)-1):
                    W += V[i] * pow(2,i*outlen)
                W += (V[-1] % (pow(2,b))) * pow(2,n*outlen)
                X = W + pow(2,L-1)
                c = X % (q)
                p = X - (c-1)
                if p >= pow(2, L - 1):
                    if is_prime(p) and (p - 1) % q == 0:
```

Criação da chaves

- **x**
  - Inteiro aleatório compreendido no intervalo : 0<x<q
  - Corresponde à **Chave Privada**
- **y**
  - Gerar g^x *mod* p
  - Corresponde à **Chave Pública**

In [1...

```
class DSA(DSA):
    def gen_key(self):
        x = ZZ.random_element(1, self.q-1)
        y = pow(self.g, x, self.p)
        return x, y
```

Assinatura de uma mensagem

A assinatura consiste num par de números (r,s) que é computado da seguinte maneira:

- **Calcular r**: (g^k *mod* p) *mod* q. Se r=0, gerar outro k e repetir o processo
- **Calcular s**: [k^(-1) *(hash(m) + xr)*] *mod* q

In [1...

```
class DSA(DSA):
    def calculate_r(self):
        k = ZZ.random_element(1, self.q-1)
        r = mod(pow(self.g, k, self.p), self.q)
        while r == 0:
            k = ZZ.random_element(1, self.q-1)
            r = mod(pow(self.g, k, self.p), self.q)
        return r, k

    def calculate_s(self, k, msg, r):
        hash_m = hash_sha256(msg)
        hash_as_int = int.from_bytes(hash_m, 'big')
        s1 = pow(k, -1, self.q)
        xr = self.priv_key * r
        s2 = mod(hash_as_int + xr, self.q)
        return s1 * s2

    def sign(self, msg):
        r, k = self.calculate_r()
        s = self.calculate_s(k, msg, r)
        while s == 0:
            (r, k) = self.calculate_r()
            s = calculate_s(k, msg, r)
        return r, s
```

Verificar a assinatura de uma mensagem

Este processo também pode ser divido em algumas etapas:

- Tamanho de **r** e de **s**
  - 0 < r < q
  - 0 < s < q
- Calcular **w** : s^(-1) *mod* q
- Calcular **u1** : (hash(m) * w) *mod* q
- Calcular **u2** : (r * w) *mod* q
- Calcular **v** : (g^u1 * y^u2 _mod_p) *mod* q
- Verificar : v==r

In [1...

```
class DSA(DSA):
    def verify(self, r, s, pub_key, msg):
        if self.validate_sign_vars(r, s) == False:
            print("Erro na validação")
        w = self.calculate_w(s)
        u1 = self.calculate_u1(w, msg)
        u2 = self.calculate_u2(r, w)
        v = self.calculate_v(u1, u2, pub_key)
        return v == r

    def validate_sign_vars(self,r ,s):
        return 0 < ZZ(r) < self.q and 0 < ZZ(s) < self.q

    def calculate_w(self, s):
        return inverse_mod(ZZ(s), self.q)

    def calculate_u1(self, w, msg):
        hash_m = hash_sha256(msg)
        hash_as_int = int.from_bytes(hash_m, 'big')
        hw = hash_as_int * w
        return mod(hw, self.q)

    def calculate_u2(self, r, w):
        rw = ZZ(r) * w
        return mod(rw, self.q)

    def calculate_v(self, u1, u2, pub_key):
        g_u1 = self.g ^ u1
        y_u2 = pub_key ^ u2
        g_y = g_u1 * y_u2
        v1 = mod(g_y, self.p)
        return mod(v1, self.q)

    def validate_params(self):
        return ((power_mod(self.g, self.q, self.p) == 1)
                and (self.g > 1)
                and ((self.p - 1) % self.q == 0)
                )
```

Teste

In [1...

```
dsa = DSA(224,2048)

msg = b"Mensagem Secreta"

sign_vars = dsa.sign(msg)

dsa.verify(sign_vars[0], sign_vars[1], dsa.pub_key, msg)
```

Out[1... True

ECDSA

Neste exercício vamos implementar um ECDSA segundo uma curva elíptica definida no **FIPS186-4**. Os valores dos parâmetros neste caso já se encontram gerados, tendo optado pela curva **P192**

In [1...

```
EC_P192 = { 'p' :  6277101735386680763835789423207666416083908700390324961
            'n' :  6277101735386680763835789423176059013767194773182842284
           'seed' : '3045ae6fc8422f64ed579528d38120eae12196',
           'c' : '3099d2bbbfcb2538542dcd5fb078b6ef5f3d6fe2c745de65',
           'b' : '64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1',
           'gx' : '188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012',
           'gy' :  '07192b95ffc8da78631011ed6b24cdd573f977a11e794811'}
```

Gerador

Para calcular o ponto gerador, precisamos dos parâmetros **b** e **p** tabelados para obter a curva elíptica correspondente.
Sabendo que a curva tem o formato $y^2 = x^3 - 3x + b \mod p$ a nossa curva pode ser obtida através: $E = EllipticCurve(GF(p), [-3, b])$

In [1...

```
class ECDSA:
    def __init__(self, Curve):
        p = Curve['p']
        self.n = Curve['n']
        b = ZZ(Curve['b'], 16)
        gx = ZZ(Curve['gx'], 16)
        gy = ZZ(Curve['gy'], 16)

        EC = EllipticCurve(GF(p),[-3,b])
        self.g = EC(gx,gy)

        priv, pub = self.gen_key()
        self.priv_key = priv
        self.pub_key = pub
```

Chave privada e Chave pública

- **Chave privada** : É necessário que seja um valor aleatório entre 0 e n, onde **n** representa a ordem do ponto de base (**g**)
- **Chave pública** : É obtida a partir do ponto de base e da chave privada $(q = d \times G)$

In [1...

```
class ECDSA(ECDSA):
    def gen_key(self):
        # private key
        d = ZZ.random_element(1, self.n)
        # public key
        q =  d * self.g
        return d, q
```

Assinatura de uma mensagem

- 1ª passo: Calcular **e** -> e = hash(mensagem)
- 2º passo: Calcular k e r tal que r > 0 s > 0
- 3º passo: Calcular **s** -> $((e + d \times r) \div k) \mod n$
- 4ª passo: Assinar (r,s)

In [1...

```
class ECDSA(ECDSA):
    def sign(self, msg):
        r = 0
        k = 0
        s = 0
        e = hash_sha256(msg)
        e_as_int = int.from_bytes(e, 'big')
        while s == 0:
            while r == 0:
                k = ZZ.random_element(1, self.n-1)
                (x1, y1, z1) =  k * self.g
                r = mod(x1, self.n)

            dr = r * self.priv_key
            p2 = e_as_int + dr

            s1 = power_mod(k, -1, self.n)
            s2 = mod(p2,self.n)
            s = s1*s2

        return r, s
```

Verificar a assinatura

Este processo também pode ser divido em algumas etapas:

- Tamanho de **r** e de **s**
  - 0 < r < n
  - 0 < s < n
- Calcular a hash da mensagem: e = hash(m)
- Calcular **w** : $s^{-1} \mod n$
- Calcular **u1** : $(e \times w) \mod n$
- Calcular **u2** : $(r \times w) \mod n$
- Calcular **o** : $(u1 \times G) + (u2 \times q)$
- A partir deste elemento identidade **o**, pegamos na sua componente x: ox = o[0]
- Verificar $r == o_x \mod n$

In [1...

```
class ECDSA(ECDSA):
    def verify(self, r, s, pub_key, msg):
        if self.validate_sign_vars(r, s) == False:
            print("Erro na validação")

        e = hash_sha256(msg)
        e_as_int = int.from_bytes(e, 'big')
        w = power_mod(ZZ(s),-1,self.n)

        u1 = ZZ(mod(e_as_int * w, self.n))
        u2 = ZZ(mod(r * w, self.n))

        u1_q = u1 * self.g
        u2_q = u2 * pub_key

        o = u1_q + u2_q

        return r == mod(o[0], self.n)

    def validate_sign_vars(self,r ,s):
        return 0 < ZZ(r) < self.n and 0 < ZZ(s) < self.n
```

In [1...

```
ecdsa = ECDSA(EC_P192)
pub_key = ecdsa.pub_key
r, s = ecdsa.sign(b"Hello")
ecdsa.verify(r, s, pub_key, b"Hello")
```

Out[1... True

In [ ]:

In [ ]: