UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA

ESTRUTURAS CRIPTOGRÁFICAS

Trabalho 2

Grupo 10

Autores:

Joel Gama (A82202)

Tiago Pinheiro (A82491)

21 de Abril de 2020

Conteúdo

Introdução

O presente relatório surge no âmbito da Unidade Curricular de Estruturas Criptográficas integrada no perfil de Criptografia e Segurança da Informação.

Este trabalho é o terceiro de cinco trabalhos que ainda irão ser abordados nesta UC. O trabalho vai ter três implementações diferentes de três algoritmos. O *RSA*, o *DSA* e o *ECDSA*.

Na primeira pergunta, é pedida uma implementação de um *PKE* que seja *IND-CCA* seguro. O enunciado indica que se deve usar o *padding OAEP* para a mensagem e o sistema *RSA* para as chaves. Na segunda, é pretendido que se implemente o *DSA*.

Por fim, é pedido que se implemente o *ECDSA* utilizando curvas elípticas definidas no *FIPS186-4*.

Implementação de um esquema KEM-RSA-OAEP

2.1 Função *randomprime*

A função *randomprime* vai gerar um número primo aleatório entre dois limites, usando o número passado como argumento.

```
def randomprime(i):
    return random_prime(2**i-1,True,2**(i-1))
```

2.2 Geração dos parâmetros

Usando a função *randomprime* vão ser gerados os parâmetros **p** e **q** assim como o número **n** e o **phi**.

De seguida, é escolhido um elemento aleatório de **phi**, a que vamos chamar **e**. O **e** e o **phi** podem apenas ter um divisor em comum, o `1`. Ao menos tempo é gerado o *ring* **R**.

Por último, é gerado o **d**, que vai ser elemento da chave privada.

```
l = 2048

q = randomprime(l)
p = randomprime(l)

n = p * q
phi = (p-1)*(q-1)

e = ZZ.random_element(phi)
R = IntegerModRing(n)

while gcd(e, phi) != 1:
    e = ZZ.random_element(phi)

bezout = xgcd(e,phi)
d = Integer(mod(bezout[1],phi));
```

2.3 Chaves

```
# Public key
(n,e)

# Private key
(p,q,d)
```

2.4 OAEP *Padding*

2.4.1 Inicilização

Neste excerto de código é escolhido o tamanho do *padding* (1024 bits) e o tamanho do **k0** (256 *bits*). Também são geradas as duas funções *hash* utilizadas no *padding*.

```
nBits = 1024
k0BitsInt = 256
k0BitsFill = '0256b'
encoding = 'utf-8'

oracle1 = hashlib.sha256()
oracle2 = hashlib.sha256()
```

2.4.2 Funções de conversão

As funções de conversões de tipos e, em geral, a conversão de tipos foi um pesadelo durante este trabalho. Após algum *debug* foi possível perceber que o erro do programa prevêm destas funções, mas este não foi possível resolver.

```
def CharsToBinary(msg):
    bits = bin(int(binascii.hexlify(msg), 16))
    return bits

def BinaryToChars(bits):
    r = bin_to_ascii(bits)
    return r
```

2.4.3 *Padding*

Na função de *padding* começa-se por gerar um *string* aleatória de *bits* (em binário) com 256 *bits* de tamanho (**k0** já definido anteriormente). Depois a mensagem que irá ser enviada é passada para binário.

Na fase seguinte, é acrescentada à mensagem um conjunto de *bits* `0` de forma a que o padding tenha um tamanho de 1024 *bits*.

Por último, são feitas as operações de *hash* e *XOR*. Seguindo o seguinte esquema:

- X = (mensagem+zeros) **XOR** G(randomBitString)
- Y = r **XOR** H(X)

No fim, retorna-se o **x** e o **y**, juntos.

```
def paddingOAEP(msg):

    randBitStr = format(SystemRandom()
            .getrandbits(k0BitsInt), k0BitsFill)
    binMsg = CharsToBinary(msg)

    if len(binMsg) <= (nBits-k0BitsInt):
        k1Bits = nBits - k0BitsInt - len(binMsg)
        zeroPaddedMsg = binMsg + ('0'*k1Bits)

    oracle1.update(randBitStr.encode(encoding))
    x = format(int(zeroPaddedMsg, 2) ^^
                int(oracle1.hexdigest(), 16), '0768b')
    oracle2.update(x.encode(encoding))
    y = format(int(randBitStr, 2) ^^
                int(oracle2.hexdigest(), 16), k0BitsFill)

    return x+y
```

2.4.4 *Unpad*

Começa-se por dividir o input ao separar os 256 *bits* aleatórios dos restantes e depois é seguido o seguinte processo:

- X = y **XOR** H(x)
- Y = (mensagem+zeros) **XOR** G(randomBitString)

Sendo **x** e **y** as váriaveis correspondentes no código.
Por fim, a mensagem é passada de binário para string e retornada.

```
def unpad(msg):
    x = msg[0:768]
    y = msg[768:]

    oracle2.update(x.encode(encoding))
    r = format(int(y,2) ^^
            int(oracle2.hexdigest(), 16), k0BitsFill)

    oracle1.update(r.encode(encoding))
    msgWith0s = format(int(x,2) ^^
                    int(oracle1.hexdigest(), 16), '0768b')

    msg = BinaryToChars(msgWith0s)
    return msg
```

2.5 *Encrypt*

Antes de cifrar a mensagem é feito o *padding*. A string **'0b'** foi usada para conseguir passar de inteiros para binários.

O processo da cifra começa por passar a *string* de *bits* para inteiro e depois utilizar um *IntegerModRing* para conseguir realizar as operações de exponencialização e divisão.

```
def encrypt(msg):
    msgWithPad = paddingOAEP(msg)
    m = '0b' + msgWithPad
    zz = ZZ(m)
    a = R(zz)
    ct = a**e
    return ct
```

2.6 *Decrypt*

Para decifrar, é usada a *private key* juntamente com um *Integer Mod Ring*, já definido anteriormente. Neste processo, como o objeto final não é um inteiro são feitas as devidas conversões para este ser um inteiro e, posteriormente, uma *string* de *bits*.

Com essa *string* de *bits* é realizado o *Unpad* da mensagem e,desta forma, termina o processo de decifrar a mensagem.

```
def decpryt(ct):
    b = R(ct)
    dm = b**d
    zz = ZZ(dm)
    inteiro = Integer(zz)
    m = inteiro.binary()
    msg = unpad(m)
    return msg
```

2.7 Testing

A parte de teste é bastante simples, cria-se uma mensagem que irá ser cifrada e depois o texto cifrado é passado à função de *decrypt*. Se a mensagem corresponder à mensagem retornada pela função de *decrypt*, a operação foi realizada com sucesso.

```
msg = "Hello world"

ct = encrypt(msg)
m = decpryt(ct)

print msg == m
```

Implementação do *DSA*

Uma das componentes deste trabalho era a implementação do *Digital Signature Algorithm (DSA)*. Para a implementação foram seguidos alguns passos tendo em conta os pedidos feitos no enunciado.

3.1 Parâmetros

O primeiro passo é gerar os parâmetros necessários. Os parâmetros necessários são **q, p, q** e um par de chaves.

Começamos por gerar o **q** e **p**. Para gerar o q utilizamos o valor **N** dado como argumento para gerar um *N-bits prime*. O **p** é o *modulus* é gerado a partir do **L**.

```
def generate_p_q(L, N):
    g = N
    n = (L - 1) // g
    b = (L - 1) % g
    while True:
        # gerar q
        while True:
            s = xmpz(randrange(1, 2 ** (g)))
            a = sha1(to_binary(s)).hexdigest()
            zz = xmpz((s + 1) % (2 ** g))
            z = sha1(to_binary(zz)).hexdigest()
            U = int(a, 16) ^ int(z, 16)
            mask = 2 ** (N - 1) + 1
            q = U | mask
            if is_prime(q, 20):
                break
        # gerar p
        i = 0  # contador
        j = 2  # offset
        while i < 4096:
            V = []
            for k in range(n + 1):
                arg = xmpz((s + j + k) % (2 ** g))
                zzv = sha1(to_binary(arg)).hexdigest()
                V.append(int(zzv, 16))
```

```
        W = 0
        for qq in range(0, n):
            W += V[qq] * 2 ** (160 * qq)
        W += (V[n] % 2 ** b) * 2 ** (160 * n)
        X = W + 2 ** (L - 1)
        c = X % (2 * q)
        p = X - c + 1  # p = X - (c - 1)
        if p >= 2 ** (L - 1):
            if is_prime(p, 10):
                return p, q
        i += 1
        j += n + 1
```

Depois de ter o **q** e **p** podemos gerar o **g** utilizando a seguinte formula.

```
#g = h^exp mod p
def generate_g(p, q):
    while True:
        h = randrange(2, p - 1)
        exp = xmpz((p - 1) // q)
        g = powmod(h, exp, p)
        if g > 1:
            break
    return g
```

3.2 Chave pública e chave privada

Os parâmetros que faltam são o par de chaves. A chave privada é um número aleatório entre 1 e **q** (chamado de **x**) e a chave pública (chamada de **y**) é o resultado de:

```
def generate_keys(g, p, q):
    x = randrange(2, q) # aleatório entre 2 e q
    y = powmod(g, x, p) # g^x mod p
    return x, y
```

O passo seguinte é assinar. Para isso são realizadas as seguinte operações: verificação dos parâmetros, calculo de **k**, **r**, **m** e por fim **s**.

3.3 Assinatura

```
def sign(M, p, q, g, x):
    if not validate_params(p, q, g):
        raise Exception("Invalid params")
    while True:
        k = randrange(2, q)  # 0 < k < q
        r = powmod(g, k, p) % q # (g^k mod p) mod q
```

```
        m = int(sha1(M).hexdigest(), 16)
        try:
         # 1/k (H + x*r) mod q
            s = (invert(k, q) * (m + x * r)) % q # invmod(k, q) * (H + x*
            return r, s
        except ZeroDivisionError:
            pass
```

3.4 Verificação

A última parte do algoritmo *DSA* corresponde à verificação. Nesta é utilizado a seguinte técnica:

```
- Verificar se o r e s estão entre 0 e q (excluindo)
- Calcular w = invmod(s, q).
- u1 = (H * w) mod q.
- u2 = (r * w) mod q.
- u1 = g^u1 mod p.
- u2 = y^u2 mod p.
- v = u1 * u2 mod q.
- Se v == r, a mensagem verificasse
```

Assim, utilizando estes passos foram aplicados da seguinte forma:

```
def verify(M, r, s, p, q, g, y):
    if not validate_params(p, q, g):
        raise Exception("Invalid params")
    if not validate_sign(r, s, q):
        return False
    try:
        w = invert(s, q)
    except ZeroDivisionError:
        return False
    m = int(sha1(M).hexdigest(), 16)
    u1 = (m * w) % q
    u2 = (r * w) % q

    # v = g^u1 * y^u2 mod p  mod q
    v = (powmod(g, u1, p) * powmod(y, u2, p)) % p % q
    if v == r:
        return True
    return False
```

Implementação do *ECDSA*

Na última alínea foi pedido que o grupo implementasse o *Elliptic Curve Digital Signature Algorithm (ECDSA)*. De entre as curvas elípticas primas definidas no *FIPS186-4*, escolhemos a curva *P-192*.

4.1 Inicialização

Definimos então a classe que vai implementar essa curva, essa classe vai conter o tabelamento da curva *P-192*, a sua respetiva inicialização que está presente no início da função *verify* e também possui as verificações de modo a comprovar as *car*. Essas verificações são as seguintes: verificar se **G** tem ordem n, verificar a estrutura de grupo abeliano na órbita de **G** e verificar se **P** aleatório está na órbita de **G** é equivalente a resolver o problema do logaritmo discreto nesta curva.

```
# Curve P-192 from FIPS 186-4

class MyECDSA():

# Curve table
  global NIST
  NIST = dict()
  NIST['P-192'] =  {
    'p': 6277101735386680763835789423207666416083908700390324961279,
    'n': 6277101735386680763835789423176059013767194773182842284081,
    'seed' : '3045ae6fc8422f64ed579528d38120eae12196d5',
    'c': '3099d2bbbfcb2538542dcd5fb078b6ef5f3d6fe2c745de65',
    'b': '64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1',
    'Gx' : '188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012',
    'Gy' : '07192b95ffc8da78631011ed6b24cdd573f977a11e794811'
  }

# E : y^2 = x^3 - 3*x + b (mod p)

    def verify(self):
        # init
        c = NIST['P-192']
        p = c['p']
```

```
n = c['n']
b = ZZ(c['b'],16)
Gx = ZZ(c['Gx'],16)
Gy = ZZ(c['Gy'],16)

E = EllipticCurve(GF(p),[-3,b])
G = E((Gx,Gy))
print(E)
print("G = ",G)

# Verificar se G tem ordem n
print(G * n)

# Verificar a estrutura de grupo abeliano na órbita de G
i = ZZ.random_element(1,n-1)
j = ZZ.random_element(1,n-1)
print(G*i, G*j)
print(G*i + G*j)
print(G*(i+j))

P = E.random_point()
# Verificar se P aleatório está na órbita de G é
equivalente a resolver o problema do logaritmo
discreto nesta curva
# Mas pode-se ver algumas propriedades
n = P.order()
# Conjunto dos pontos P tais que  G * m == P
m=7
G.division_points(m)
```

4.2 Par de chaves

O processo para gerar a chave privada e a chave pública a utilizar no algoritmo é:

- Depois de selecionar uma curva elíptica, selecionamos um ponto base $G \in E(\mathbb{Z}p)$ de ordem r.
- Gerar um inteiro aleatório s contido entre 1 e (r-1).
- Computar $W = sG$
- Assim, a chave privada é s e a chave pública é (E, G, r, W).

4.3 Assinatura

Para a assinatura da mensagem m os passos a seguir são:

- Calcular a *hash* criptográfica da mensagem m.
- Gerar um inteiro aleatório u contido entre 1 e (r-1)
- Computar $V = uG = (xV, yV)$ e $c = xV \mod r$ (voltar ao segundo ponto se c = 0)
- Computar $d = u^-1.(f + s.c) \mod r$ (se d = 0 voltar ao segundo ponto)
- A assinatura da mensagem m é o par de inteiros (c,d)

4.4 Verificação

O passo final do algoritmo é a verificação.

- Primeiro obter uma cópia autenticada da chave pública (verificar de c e d estão entre 1 e r-1)
- Computar $f = H(m)$ e $h = d^-1 \mod r$
- Computar $h1 = f.h \mod r$ e $h2 = c.h \mod r$
- Por fim computar $h1G + h2W = (x1, y1)$ e $c1 = x1 \mod r$
- A assinatura é aceite se e só se $c1 = c$

Conclusão

Este trabalho foi importante para aplicar a matéria dada nas aulas ao longo deste semestre.

Foi possível aplicar conceitos relacionados *RSA*, *DSA* e *ECDSA*. Mais especificamente geração de chaves públicas e privadas, funções de encapsulamento, utilização de números primos e de curvas elípticas.

Analisando o trabalho realizado, apesar de todo o trabalho do grupo, não foi possível terminar o exercício 1 e o exercício 3.

No exercício 1 faltou resolver os problemas com as conversões de tipos e, uma vez corrigidos esses problemas, testar o restante código. Com base, em testes intermédios é possível afirmar que o restante código do exercício 1 se encontra funciona mas apenas é possível afirmar com os problemas anteriormente enunciados corrigidos.

Por outro lado, no exercício 3 faltou implementar os três últimos passos referidos no capítulo anterior: gerar o par de chaves, assinatura da mensagem m e verificação da assinatura.