ECDH

April 6, 2019

1 ECDH - *Elliptic Curve Diffie-Hellman*

O **ECDH** é uma variante do algoritmo de *Diffie-Hellman* mas para curvas elípticas. Mais concretamente, é um protocolo de acordo de chaves que permite que duas partes, cada uma tendo um par de chaves pública-privada sobre uma curva elíptica, estabeleçam um segredo partilhado sobre um canal inseguro.

Suponhamos que a Alice quer partilhar uma chave com Bob, mas o único canal disponível é susceptível de ser interceptado por terceiros.

Inicialmente, a Alice e o Bob geram as suas próprias chaves privadas e públicas. Nomeadamente, a chave privada d_A e a chave pública $H_A = d_A G$ estão associadas à Alice e as chaves d_B e $H_B = d_B G$ ao Bob. Observe-se que Alice e Bob estão usando os mesmos **parâmetros de domínio**: o mesmo ponto base G, na mesma curva elíptica, sobre o mesmo corpo finito.

Seguidamente, a Alice e o Bob trocam as suas chaves públicas H_A e H_B sobre um canal inseguro. Note-se ainda que uma entidade terceira ao interceptar H_A e H_B não será capaz de descobrir nem d_A, nem d_B, sem resolver o problema do logaritmo discreto (*hard problem*).

Posteriormente, a Alice calcula $S = d_A H_B$, usando a sua própria chave privada e a chave pública de Bob. E Bob calcula $S = d_B H_A$, usando a sua própria chave privada e a chave pública de Alice. Observe-se que S é igual tanto para Alice, como para o Bob, pois:

$$S = d_A H_B = d_A(d_B G) = d_B(d_A G) = d_B H_A$$

Segue-se uma classe Python que implementa o ECDH, usando curvas elípticas binárias segundo as seguintes restrições:

- a dimensão n do corpo $K = GF(2^n)$ é fornecida como parâmetro de inicialização da classe;
- a curva é definita pelas raízes em K^2 de um polinómio $\phi \equiv y^2 + xy + x^3 + x^2 + b$, sendo o parâmetro b escolhido de forma a que a curva E/ϕ tenha um grupo de torsão de ordem prima e de tamanho $\geq 2^{n-1}$.

```
In [376]: class ECDH:
              def __init__(self, n):
                  self.genCurve(n)

              def genCurve(self, n):
                  K = GF(2^n)
                  while True:
                      b=0
```

```
                while b==0: #E cannot be a singular curve
                    b = K.random_element()
                E = EllipticCurve(K,[1,1,0,0,b])
                e_ord = E.order()
                e_fact = list(factor(e_ord))[-1][0]
                if e_fact >= (2^(n-1)):
                    while True:
                        P = E.random_element()
                        p_ord = P.order()
                        if p_ord > e_fact:
                            h = Integer(p_ord/e_fact) #cofator h
                            Q = h * P
                            self.Q = Q
                            self.N = e_fact
                            return E

            def genPrivPubKeyPair(self):
                k = randint(0, self.N-1)
                kQ = k*self.Q
                return k,kQ

            def genSharedKey(self, k, sQ):
                return k*sQ
```

1.1 Alice

```
In [377]: def Alice(conn, ecdh):
              k, kQ = ecdh.genPrivPubKeyPair()
              conn.send(kQ)
              sQ = conn.recv()
              sKey = ecdh.genSharedKey(k, sQ)
              print("[Alice] Shared key: " + str(sKey))
```

1.2 Bob

```
In [378]: def Bob(conn, ecdh):
              s, sQ = ecdh.genPrivPubKeyPair()
              kQ = conn.recv()
              conn.send(sQ)
              sKey = ecdh.genSharedKey(s, kQ)
              print("[Bob] Shared key: " + str(sKey))

In [379]: from multiprocessing import Process, Pipe
          from getpass import getpass

          class Connection:
              def __init__(self, left, right, timeout=None):
                  left_end, right_end = Pipe()
```

```
            self.timeout = timeout
            ecdh = ECDH(5)
            self.lproc = Process(target = left, args=(left_end, ecdh))
            self.rproc = Process(target = right, args=(right_end, ecdh))
            self.left = lambda : left(left_end)
            self.right = lambda : right(right_end)

        def auto(self, proc=None):
            if proc == None:
                self.lproc.start()
                self.rproc.start()
                self.lproc.join(self.timeout)
                self.rproc.join(self.timeout)
            else:
                proc.start()
                proc.join(self.timeout)
        def manual(self):
            self.left()
            self.right()
```

```
In [380]: Connection(Alice, Bob, timeout=10).auto()

[Bob] Shared key: (z5^3 + z5^2 : z5^4 + z5^3 : 1)
[Alice] Shared key: (z5^3 + z5^2 : z5^4 + z5^3 : 1)
```