dokumentacja aplikacji md4 rsa

Anna Bonikowska, Dorota Nowacka, Mateusz Goszczyński

May 20, 2024

1 MD4

- **ogólna zasada działania md4**

  MD4 jest funkcja skrótu, która przyjmuje wiadomość o dowolnej długości i generuje skrócona, stałej długości sume kontrolna 128-bitowa. Algorytm składa sie z kilku rund przekształceń, które operuja na blokach wiadomości i wewnetrznym stanie. Każda runda obejmuje przekształcenia bitowe, operacje logiczne i operacje arytmetyczne, w celu generowania skrótu na podstawie bloków danych. Wynikowy skrót można użyć jako jednoznacznego identyfikatora wiadomości lub do weryfikacji integralności danych.

- **inicjalizacja obiektu**

  Metoda __init__ w niej jest inicjalizowany nowy obiekt klasy md4 oraz jest sprawdzane czy ma on odpowiednie własności. Sa również w tym miejscu ustawiane atrybuty potrzebne do dalszego działania klasy

  def __init__(self, x):

  ```
  if len(x)   2 ** 61:
          raise Exception("Dane wejściowe sa za długie")
  if not isinstance(x, bytes) or isinstance(x, bytearray):
          raise Exception("" + x + "" + " nie jest instancja bytes.")



  self.x=x


  self.A= 1732584193
  self.B= 4023233417
  self.C= 2562383102
  self.D= 271733878
  ```

```
self.a= 1732584193
self.b= 4023233417
self.c= 2562383102
self.d= 271733878

self.y1=0
self.z1=[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
self.w1=[3,7,11,19,3,7,11,19,3,7,11,19,3,7,11,19]

self.y2=1518500249
self.z2=[0,4,8,12,1,5,9,13,2,6,10,14,3,7,11,15]
self.w2=[3,5,9,13,3,5,9,13,3,5,9,13,3,5,9,13]

self.y3=1859775393
self.z3=[0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15]
self.w3=[3,9,11,15,3,9,11,15,3,9,11,15,3,9,11,15]

self.wynik_w_bajtach=b"wartosc poczatkowa"
```

- **alternatywne sposoby inicjalizacji obiektu**

```
inicjalizacja obiektu gdy jego argument jest stringiem
@classmethod
def from_string(cls, x):
    return cls(x.encode())

inicjalizacja obiektu z pliku tekstowego
@classmethod
def from_file(cls, x):
    with open(x, "rb") as plik:
        return cls(plik.read())
```

- **główna metoda haszujaca algorytmu md4**

```
def get_hash(self):

Sprawdzenie czy już wcześniej była uruchumiona ta funkcja.
    if self.wynik_w_bajtach != b"wartosc poczatkowa":
        return ( int.from_bytes(self.wynik_w_bajtach, byteorder="big"))
    else:

        self.dopelnianie_x()
Licznik zawiera na ile modułów bedzie podzielona wiadomosc.
```

self.licznik=len(self.x)/64

for j in range(0,int(self.licznik)):
ustalanie stanu poczatkowego
self.a=self.A
self.b=self.B
self.c=self.C
self.d=self.D

W każdej z poniższych petli wiadomość bedzie przekształcana przez jedna z trzech funkcji pomocniczych F, G, H
for i in range(0,16):
self.a = ((self.a + self.F(self.b, self.c, self.d) + int.from_bytes(self.x[(j*64 + self.z1[i]*4):(j*64 + self.z1[i]*4 +4)], byteorder="little") + self.y1) %(2**32))
self.a= self.przesuniecie_bitowe_z_obrotem(self.a , self.w1[i])
self.a , self.b , self.c, self.d = self.d , self.a, self.b, self.c

for i in range(0,16):
self.a = ((self.a + self.G(self.b, self.c, self.d) + int.from$_b$$ytes(self.x[(j * 64 + self.z2[i] * 4) : (j * 64 + self.z2[i] * 4 + 4)], byteorder = "little") + self.y2)%(2 * *32))
$self.a = self.przesuniecie\_bitowe\_z\_obrotem(self.a, self.w2[i])$
$self.a, self.b, self.c, self.d = self.d, self.a, self.b, self.c$

for i in range(0,16):
self.a = ((self.a + self.H(self.b, self.c, self.d) + int.from$_b$$ytes(self.x[(j* 64+self.z3[i]*4) : (j*64+self.z3[i]*4+4)], byteorder = "little")+self.y3)%(2* *32))
$self.a = self.przesuniecie\_bitowe\_z\_obrotem(self.a, self.w3[i])$
$self.a, self.b, self.c, self.d = self.d, self.a, self.b, self.c$

Wyliczanie nowej wartości stanu
self.A= (self.A +self.a)%(2**32)
self.B= (self.B +self.b)%(2**32)
self.C= (self.C +self.c)%(2**32)
self.D= (self.D +self.d)%(2**32)

Konwertowanie wartości stanu do postaci bytowej i nastepne łaczenie jej i konwertowanie do liczy w systemie dziesiatkowym
self.wynik_w_bajtach = b"".join([int(self.A).to_bytes(4,'little'), int(self.B).to_bytes(4,'little'),

int(self.C).to_bytes(4,'little'), int(self.D).to_bytes(4,'little')])

return(int.from_bytes(self.wynik_w_bajtach, byteorder="big"))

- **metoda specjalna**

  Metoda specjalna pozwalajaca potraktować obiekt jako napis przedstawiajacy zapisana szesnastkowo wyliczona wartość funkcji skrótu.
  def __str__(self):

  if self.wynik_w_bajtach != b"wartosc poczatkowa" :

  return (self.wynik_w_bajtach.hex())
  else:

  self.get_hash()
  return (self.wynik_w_bajtach.hex())

- **funkcje pomocnicze**

  Funkcje te wykonuja działania bitowe na podanych wartościach.

  @staticmethod
  def F(x, y, z):
  return ((x&y)|((~x)&z))

  @staticmethod
  def G(x, y, z):
  return ((x&y)|(y&z)|(x&z))

  @staticmethod
  def H(x, y, z):
  return (x ^ y ^ z)

- **Przesuniecie bitowe**

  Operator nie oznacza przesuniecia bitowego w Pythonie, przez co jest potrzebna dodatkowa funkcja , która wykonuje ta czynność.
  def przesuniecie_bitowe_z_o$brotem(a, w)$:
  $return((a << w)|(a >> (32 - w)))$

- **dopełnianie wiadomości**

  Algorytm md4 wykonuje przekształcenia na 512 bitowych fragmentach wiadomości. Dlatego długość wiadomości w bytach musi być podzielna przez 64. Ponizsza funkcja dodaje do oryginalnej wiadomości jeden bit 1 ,jej dłygośc w bitach oraz odpowiednia ilość zer tak aby długość wiadomości w bitach była podzielna przez 512.

  def dopelnianie_x(self):

  ```
  self.dlugosc=len(self.x)*8
  self.v= 512 - ((self.dlugosc + 65)%512)
  self.k=self.v%8
  self.f=(self.v-self.k)/8
  self.f=int(self.f)
  self.dlu_bytes=struct.pack("¡Q", self.dlugosc)
  self.zapychacz=(2**self.k).to_bytes(1,'big')
  self.h= bytearray(self.f)
  self.g=(self.x + self.zapychacz + self.h + self.dlu_bytes)
  self.x=bytearray(self.g)
  ```

2 RSA

Jest to niesymetryczny algorytm szyfrujacy, którego zasadnicza cecha sa dwa klucze: publiczny do kodowania informacji oraz prywatny do jej odczytywania. Klucz publiczny (można go udostepniać wszystkim zainteresowanym) umożliwia jedynie zaszyfrowanie danych i w żaden sposób nie ułatwia ich odczytania, nie musi wiec być chroniony.Drugi klucz (prywatny, przechowywany pod nadzorem) służy do odczytywania informacji zakodowanych przy pomocy pierwszego klucza.

- **inicjalizacja obiektu**
  Metoda __init__ w niej jest inicjalizowany nowy obiekt klasy rsa.

- **generowanie kluczy**
  Chcemy wygenerowac pare kluczy - publiczny i prywatny. Klucz publiczny to (N,e), klucz prywatny to (d,p,q) W tym celu generujemy dwie liczby pierwsze p i q oraz e spełniajace równanie :
  gcd {e, O} = 1. Gdzie O = (p-1)(q-1). Liczba N = p*q. Liczby p, q ,e powinny byc duze aby zapewnic bezpieczensto szyfrowania jednak wpływa to na czas działania algorytmu. Aby wyliczyć klucz prywatny, wyznaczamy stała d spełniajaca równanie: e d = 1 mod (p 1)(q 1).
  Funkcja "klucze" korzysta z metod statycznych do wygenerowania potrzebnych wartosci.

- **metody**

1. czy_pierwsza - sprawdzenie czy wybrana liczba jest pierwsza,bedzie potrzebna do wygenerowania liczb pierwszych
```
@staticmethod
def czy_pierwsza(liczba):
    x=1
    if liczba ¡ 2:
        return False
    for i in range(2, int(liczba**(1/2) +1)):
        x+=1
        print(x)
        if liczba % i == 0:
            return False
    return True
```

2. generuj_pierwsza - generuje całkowita liczbe pierwsza z zadanego przedziału przy pomocy funkcji random. W petli while wykorzystuje metode statyczna czy_pierwsza by sprawdzic czy wygenerowana jest pierwsza
```
@staticmethod
def generuj_pierwsza():
    pierwsza= random.randint(2**7+1,2**8)
    while not RSA.czy_pierwsza(pierwsza):
        pierwsza= random.randint(2**7+1,2**8)
    return pierwsza
```

3. gcd - algorytm euklidesa służy do znalezienia najwiekszego wspólnego dzielnika dwóch liczb
```
@staticmethod
def gcd(a,b):
    while b != 0:
        a,b = b, a%b
    return a
```

4. generuj_wzglednie_pierwsza(O) - generuje liczbe całkowita e z podanego przedziału, która jest wzglednie pierwsza z O=(p-1)(q-1) . Wykorzystuje do tego random oraz metode statyczna gcd.
```
@staticmethod
def generuj_wzglednie_pierwsza(O):
    x=1
    wzglednie_pierwsza = random.randint(2, O - 1)
    while not RSA.gcd(wzglednie_pierwsza,O)==1:
        wzglednie_pierwsza = random.randint(2,O - 1)
        x+=1
        print(x)
    return wzglednie_pierwsza
```

5. znajdz_d - za pomoca petli znajduje liczbe d spełniajaca zadane równanie.
   @staticmethod
   def znajdz_d(e, O):
       for d in range (3, O):
           if (d*e) % O == 1:
               return d

- **klucze** Funkcja klucze za pomoca metod statycznych generuje klucze.
  def klucze(self):
      self.p=RSA.generuj_pierwsza()
      self.q=RSA.generuj_pierwsza()
      while self.p==self.q:
          self.q=RSA.generuj_pierwsza()
      self.N = self.p*self.q
      self.O = (self.p-1)*(self.q-1)
      self.e = self.generuj_wzglednie_pierwsza(self.O)
      self.klucz_publiczny=[self.N, self.e]
      self.d= RSA.znajdz_d(self.e,self.O)
      self.klucz_prywatny=[self.d, self.p, self.q]

- **szyfrowanie wiadomości**
  Majac klucz publiczny możemy zaszyfrować wiadomość m
  spełniajaca $m \leq 0 < N$. wartosc zaszyfrowana c wynosi c = m^e mod N.
  def szyfruj(self,m):
      c = (m**self.e) % self.N
          return c

- **odszyfrowanie wiadomosci**
  Zaszyfrowana wiadomość c odszyfrujemy kluczem prywatnym korzystajac ze wzoru m = c^d mod p*q
  def odszyfruj(self,c):
      m = (c**self.d) % (self.p*self.q)
          return m

3 diagram UML