Министерство образования Республики Беларусь

Учреждение Образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Кафедра ИИТ

Лабораторная работа № 3

«ОТКРЫТОЕ РАСПРОСТРАНЕНИЕ КЛЮЧЕЙ»

|  |  |  |
| --- | --- | --- |
| Проверил: |  | Выполнил: |
| Захаров В. В. |  | Казимов Л. З. |

Минск 2021

**1. Цель работы**

Изучить алгоритмы открытого распространения ключей – алгоритм Диффи-Хельмана и определить их стойкость

**2. Ход работы**

**Алгоритм**

1. Существуют два абонента. Имеются два несекретных числа g и P, известные всем заинтересованным сторонам. P – большое простое число, g – является первообразным корнем по модулю P. Первообразным корнем по модулю P называют такое число g, что все его степени по модулю P принимают значения всех чисел, взаимно простых с P. Также у первого абонента есть большое случайное число , у второго большое случайное число .

2. Абоненты вычисляют числа (для первого абонента) и (для второго абонента)

3. Абоненты обмениваются числами, полученными на втором шаге по открытому каналу (защищённому от модификации).

4. Абоненты вычисляют общий секретный ключ

**Код**

|  |
| --- |
| import hashlib import ssl from binascii import hexlify  random\_function = ssl.RAND\_bytes random\_provider = "Python SSL"   class DiffieHellman(object):  def \_\_init\_\_(self, generator=2, group=17, keyLength=540):  min\_keyLength = 180  default\_keyLength = 540   default\_generator = 2  valid\_generators = [2, 3, 5, 7]   if (generator not in valid\_generators):  print("Error: Invalid generator. Using default.")  self.generator = default\_generator  else:  self.generator = generator   if (keyLength < min\_keyLength):  print("Error: keyLength is too small. Setting to minimum.")  self.keyLength = min\_keyLength  else:  self.keyLength = keyLength   self.prime = self.getPrime(group)   self.privateKey = self.genPrivateKey(keyLength)  self.publicKey = self.genPublicKey()   def getPrime(self, group=17):  default\_group = 17   primes = {  5: 0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74020BBEA63B139B22514A08798E3404DDEF9519B3CD3A431B302B0A6DF25F14374FE1356D6D51C245E485B576625E7EC6F44C42E9A637ED6B0BFF5CB6F406B7EDEE386BFB5A899FA5AE9F24117C4B1FE649286651ECE45B3DC2007CB8A163BF0598DA48361C55D39A69163FA8FD24CF5F83655D23DCA3AD961C62F356208552BB9ED529077096966D670C354E4ABC9804F1746C08CA237327FFFFFFFFFFFFFFFF,  14: 0x}   if group in primes.keys():  return primes[group]  else:  print("Error: No prime with group %i. Using default." % group)  return primes[default\_group]   def genRandom(self, bits):  \_rand = 0  \_bytes = bits // 8 + 8   while (\_rand.bit\_length() < bits):  try:  \_rand = int.from\_bytes(random\_function(\_bytes), byteorder='big')  except:  pass   return \_rand   def genPrivateKey(self, bits):  return self.genRandom(bits)   def genPublicKey(self):  return pow(self.generator, self.privateKey, self.prime)   def checkPublicKey(self, otherKey):  if (otherKey > 2 and otherKey < self.prime - 1):  if (pow(otherKey, (self.prime - 1) // 2, self.prime) == 1):  return True  return False   def genSecret(self, privateKey, otherKey):  if self.checkPublicKey(otherKey) == True:  sharedSecret = pow(otherKey, privateKey, self.prime)  return sharedSecret  else:  raise Exception("Invalid public key.")   def genKey(self, otherKey):  self.sharedSecret = self.genSecret(self.privateKey, otherKey)   try:  \_sharedSecretBytes = self.sharedSecret.to\_bytes(  self.sharedSecret.bit\_length() // 8 + 1, byteorder="big")  except AttributeError:  \_sharedSecretBytes = str(self.sharedSecret)   s = hashlib.sha256()  s.update(bytes(\_sharedSecretBytes))  self.key = s.digest()   def getKey(self):  return self.key   def showParams(self):  print("Параметры:")  print("Простое число[{0}]: {1}".format(self.prime.bit\_length(), self.prime))  print("Генератор[{0}]: {1}".format(self.generator.bit\_length(),  self.generator))  print("Закрытый ключ[{0}]: {1}".format(self.privateKey.bit\_length(),  self.privateKey))  print("Открытый ключ[{0}]: {1}".format(self.publicKey.bit\_length(),  self.publicKey))   def showResults(self):  print("Итого:")  print("Общий секрет[{0}]: {1}".format(self.sharedSecret.bit\_length(),  self.sharedSecret))  print("Общий секретный ключ[{0}]: {1}".format(len(self.key), hexlify(self.key)))   def main():  a = DiffieHellman()  b = DiffieHellman()   *# a.genKey(b.publicKey)  # b.genKey(a.publicKey)* a.genKey(b.publicKey)  x = 0  while True:  try:  b.genKey(x)  if a.getKey() == b.getKey():  print("Ключи совпали")  print("Ключ:", b.publicKey)  break  except:  pass  x += 1   *# print("A:")  # a.showParams()  # a.showResults()  #  # print("\nB:")  # b.showParams()  # b.showResults()   # if (a.getKey() == b.getKey()):  # print("Shared keys match.")  # print("Key:", hexlify(a.key))  # else:  # print("Shared secrets didn't match!")  # print("Shared secret A: ", a.genSecret(b.publicKey))  # print("Shared secret B: ", b.genSecret(a.publicKey))* if \_\_name\_\_ == "\_\_main\_\_":  main() |

**Взлом**

Моё вычислительное устройство не смогло подобрать правильный секретный ключ на основе открытых ключей в течении часа.

На основе информации, находящейся в открытом доступе: реализации алгоритма использующего числа длиной более 300 цифр (в моём случае длина закрытого ключа – 180 цифр) могут быть взломаны в течение года.

**3. Выводы**

Алгоритм Диффи-Хельмана будет считаться безопасным, если сам алгоритм использует ключи длиной более 2048 бит, а обмен ключами происходит по каналу, защищенному от изменений (защищённому от атак человек посередине).

Также по доступной в открытых источниках информации алгоритм может быть взломан квантовым компьютером