**Practical Exam**

**Aim: Issue of Key Sharing**

Refer to the attached figure (c) here. Bob is preparing to send message to Alice. Bob and Alice both secretly computes the code (s) without sharing on any communication channel. Suggest key exchange algorithm to Bob and Alice for securely exchange information without sharing actual key. Once they form secret code, Bob applies SHA256 hash algorithm on original message (M) plus code (s) and send hash of original message and code (M||s) to Alice. Alice will receive bundle of H(M||s) and first append code (s) with received message (M) and produce hash of the message (H) that compare with H(M||s) to make sure that message is not altered by any attackers.

Task to perform:

1) Use some key exchange algorithm to calculate value of s (secret code) which must be unique at sender and receiver side.

2) Implementation can be done using any programming language such as c, c++, java, python, c#, javascript, php etc.

3) Apply SHA256 on message and secret code and display it on output screen.

Verify the hash value at receiver end [Verificaton is an optional for implementation]

**Code :**

#!/usr/bin/env python

"""

PyDHE - Diffie-Hellman Key Exchange in Python

Copyright (C) 2015 by Mark Loiseau

This program is free software: you can redistribute it and/or modify

it under the terms of the GNU General Public License as published by

the Free Software Foundation, either version 3 of the License, or

(at your option) any later version.

This program is distributed in the hope that it will be useful,

but WITHOUT ANY WARRANTY; without even the implied warranty of

MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the

GNU General Public License for more details.

You should have received a copy of the GNU General Public License

along with this program.  If not, see <http://www.gnu.org/licenses/>.

"""

import hmac

import base64

import hashlib

from hashlib import sha256

from binascii import hexlify # For debug output

# If a secure random number generator is unavailable, exit with an error.

try:

    import ssl

    random\_function = ssl.RAND\_bytes

    random\_provider = "Python SSL"

except (AttributeError, ImportError):

    import OpenSSL

    random\_function = OpenSSL.rand.bytes

    random\_provider = "OpenSSL"

class DiffieHellman(object):

    """

    A reference implementation of the Diffie-Hellman protocol.

    By default, this class uses the 6144-bit MODP Group (Group 17) from RFC 3526.

    This prime is sufficient to generate an AES 256 key when used with

    a 540+ bit exponent.

    """

    def \_\_init\_\_(self, generator=2, group=17, keyLength=540):

        """

        Generate the public and private keys.

        """

        min\_keyLength = 180

        default\_keyLength = 540

        default\_generator = 2

        valid\_generators = [ 2, 3, 5, 7 ]

        # Sanity check fors generator and keyLength

        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):

        """

        Given a group number, return a prime.

        """

        default\_group = 17

        primes = {

        5:  0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74020BBEA63B139B22514A08798E3404DDEF9519B3CD3A431B302B0A6DF25F14374FE1356D6D51C245E485B576625E7EC6F44C42E9A637ED6B0BFF5CB6F406B7EDEE386BFB5A899FA5AE9F24117C4B1FE649286651ECE45B3DC2007CB8A163BF0598DA48361C55D39A69163FA8FD24CF5F83655D23DCA3AD961C62F356208552BB9ED529077096966D670C354E4ABC9804F1746C08CA237327FFFFFFFFFFFFFFFF,

        14: 0x

        15: 

        16: 

        17:

        18:

        }

        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):

        """

        Generate a random number with the specified number of bits

        """

        \_rand = 0

        \_bytes = bits // 8 + 8

        while(\_rand.bit\_length() < bits):

            try:

                # Python 3

                \_rand = int.from\_bytes(random\_function(\_bytes), byteorder='big')

            except:

                # Python 2

                \_rand = int(OpenSSL.rand.bytes(\_bytes).encode('hex'), 16)

        return \_rand

    def genPrivateKey(self, bits):

        """

        Generate a private key using a secure random number generator.

        """

        return self.genRandom(bits)

    def genPublicKey(self):

        """

        Generate a public key X with g\*\*x % p.

        """

        return pow(self.generator, self.privateKey, self.prime)

    def checkPublicKey(self, otherKey):

        """

        Check the other party's public key to make sure it's valid.

        Since a safe prime is used, verify that the Legendre symbol == 1

        """

        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):

        """

        Check to make sure the public key is valid, then combine it with the

        private key to generate a shared secret.

        """

        if(self.checkPublicKey(otherKey) == True):

            sharedSecret = pow(otherKey, privateKey, self.prime)

            return sharedSecret

        else:

            raise Exception("Invalid public key.")

    def genKey(self, otherKey):

        """

        Derive the shared secret, then hash it to obtain the shared key.

        """

        self.sharedSecret = self.genSecret(self.privateKey, otherKey)

        # Convert the shared secret (int) to an array of bytes in network order

        # Otherwise hashlib can't hash it.

        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 the shared secret key

        """

        return self.key

    def showParams(self):

        """

        Show the parameters of the Diffie Hellman agreement.

        """

        print("Parameters:")

        print("Prime[{0}]: {1}".format(self.prime.bit\_length(), self.prime))

        print("Generator[{0}]: {1}\n".format(self.generator.bit\_length(),

            self.generator))

        print("Private key[{0}]: {1}\n".format(self.privateKey.bit\_length(),

            self.privateKey))

        print("Public key[{0}]: {1}".format(self.publicKey.bit\_length(),

            self.publicKey))

    def showResults(self):

        """

        Show the results of a Diffie-Hellman exchange.

        """

        print("Results:")

        print("Shared secret[{0}]: {1}".format(self.sharedSecret.bit\_length(),

            self.sharedSecret))

        print("Shared key[{0}]: {1}".format(len(self.key), hexlify(self.key)))

def verify(receivedHashed, message):

    ourHashed = hashFunction(message)

    if receivedHashed == ourHashed:

        print("\nMessage Hash verification successful!", )

        print(receivedHashed, " = ", ourHashed)

    else:

        print("\nMessage Hash verification failed!")

        print(receivedHashed, " != ", ourHashed)

def hashFunction(message):

    hashed = sha256(message.encode("UTF-8")).hexdigest()

    return hashed

if \_\_name\_\_=="\_\_main\_\_":

    """

    Run an example Diffie-Hellman exchange

    """

    a = DiffieHellman()

    b = DiffieHellman()

    a.genKey(b.publicKey)

    b.genKey(a.publicKey)

    #a.showParams()

    #a.showResults()

    #b.showParams()

    #b.showResults()

    if(a.getKey() == b.getKey()):

        print("Shared keys match.")

        print("Key hash:", hexlify(a.key))

        secret = hexlify(a.key)

        msg = input("\nEnter Message  : ")

        hash = hashFunction(msg)

        print("Message Hash : " + hash)

        verify(hash,msg)

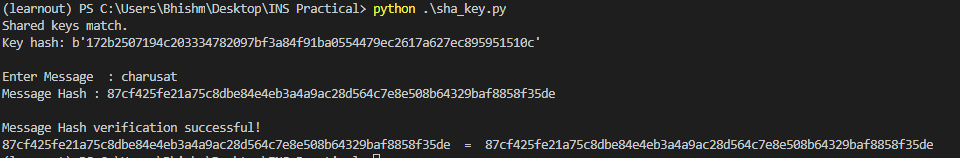
    else:

        print("Shared secrets didn't match!")

        print("Shared secret A: ", a.genSecret(b.publicKey))

        print("Shared secret B: ", b.genSecret(a.publicKey))

**Output:**

****