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:  0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74020
BBEA63B139B22514A08798E3404DDEF9519B3CD3A431B302B0A6DF25F14374FE1356D6D51C245E485
B576625E7EC6F44C42E9A637ED6B0BFF5CB6F406B7EDEE386BFB5A899FA5AE9F24117C4B1FE649286
651ECE45B3DC2007CB8A163BF0598DA48361C55D39A69163FA8FD24CF5F83655D23DCA3AD961C62F3
56208552BB9ED529077096966D670C354E4ABC9804F1746C08CA237327FFFFFFFFFFFFFFFF,
        14: 0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74020
BBEA63B139B22514A08798E3404DDEF9519B3CD3A431B302B0A6DF25F14374FE1356D6D51C245E485
B576625E7EC6F44C42E9A637ED6B0BFF5CB6F406B7EDEE386BFB5A899FA5AE9F24117C4B1FE649286
651ECE45B3DC2007CB8A163BF0598DA48361C55D39A69163FA8FD24CF5F83655D23DCA3AD961C62F3
56208552BB9ED529077096966D670C354E4ABC9804F1746C08CA18217C32905E462E36CE3BE39E772
C180E86039B2783A2EC07A28FB5C55DF06F4C52C9DE2BCBF6955817183995497CEA956AE515D22618
98FA051015728E5A8AACAA68FFFFFFFFFFFFFFFF,
        15: 0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74020
BBEA63B139B22514A08798E3404DDEF9519B3CD3A431B302B0A6DF25F14374FE1356D6D51C245E485
B576625E7EC6F44C42E9A637ED6B0BFF5CB6F406B7EDEE386BFB5A899FA5AE9F24117C4B1FE649286
651ECE45B3DC2007CB8A163BF0598DA48361C55D39A69163FA8FD24CF5F83655D23DCA3AD961C62F3
56208552BB9ED529077096966D670C354E4ABC9804F1746C08CA18217C32905E462E36CE3BE39E772
C180E86039B2783A2EC07A28FB5C55DF06F4C52C9DE2BCBF6955817183995497CEA956AE515D22618
98FA051015728E5A8AAAC42DAD33170D04507A33A85521ABDF1CBA64ECFB850458DBEF0A8AEA71575
D060C7DB3970F85A6E1E4C7ABF5AE8CDB0933D71E8C94E04A25619DCEE3D2261AD2EE6BF12FFA06D9
8A0864D87602733EC86A64521F2B18177B200CBBE117577A615D6C770988C0BAD946E208E24FA074E
5AB3143DB5BFCE0FD108E4B82D120A93AD2CAFFFFFFFFFFFFFFFF,
        16: 0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74020
BBEA63B139B22514A08798E3404DDEF9519B3CD3A431B302B0A6DF25F14374FE1356D6D51C245E485
B576625E7EC6F44C42E9A637ED6B0BFF5CB6F406B7EDEE386BFB5A899FA5AE9F24117C4B1FE649286
651ECE45B3DC2007CB8A163BF0598DA48361C55D39A69163FA8FD24CF5F83655D23DCA3AD961C62F3
56208552BB9ED529077096966D670C354E4ABC9804F1746C08CA18217C32905E462E36CE3BE39E772
C180E86039B2783A2EC07A28FB5C55DF06F4C52C9DE2BCBF6955817183995497CEA956AE515D22618
98FA051015728E5A8AAAC42DAD33170D04507A33A85521ABDF1CBA64ECFB850458DBEF0A8AEA71575
D060C7DB3970F85A6E1E4C7ABF5AE8CDB0933D71E8C94E04A25619DCEE3D2261AD2EE6BF12FFA06D9
8A0864D87602733EC86A64521F2B18177B200CBBE117577A615D6C770988C0BAD946E208E24FA074E
5AB3143DB5BFCE0FD108E4B82D120A92108011A723C12A787E6D788719A10BDBA5B2699C327186AF4
```

```
E23C1A946834B6150BDA2583E9CA2AD44CE8DBBBC2DB04DE8EF92E8EFC141FBECAA6287C59474E6BC
05D99B2964FA090C3A2233BA186515BE7ED1F612970CEE2D7AFB81BDD762170481CD0069127D5B05A
A993B4EA988D8FDDC186FFB7DC90A6C08F4DF435C934063199FFFFFFFFFFFFFFFF,
        17:
        0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74020BBEA
63B139B22514A08798E3404DDEF9519B3CD3A431B302B0A6DF25F14374FE1356D6D51C245E485B576
625E7EC6F44C42E9A637ED6B0BFF5CB6F406B7EDEE386BFB5A899FA5AE9F24117C4B1FE649286651E
CE45B3DC2007CB8A163BF0598DA48361C55D39A69163FA8FD24CF5F83655D23DCA3AD961C62F35620
8552BB9ED529077096966D670C354E4ABC9804F1746C08CA18217C32905E462E36CE3BE39E772C180
E86039B2783A2EC07A28FB5C55DF06F4C52C9DE2BCBF6955817183995497CEA956AE515D2261898FA
051015728E5A8AAAC42DAD33170D04507A33A85521ABDF1CBA64ECFB850458DBEF0A8AEA71575D060
C7DB3970F85A6E1E4C7ABF5AE8CDB0933D71E8C94E04A25619DCEE3D2261AD2EE6BF12FFA06D98A08
64D87602733EC86A64521F2B18177B200CBBE117577A615D6C770988C0BAD946E208E24FA074E5AB3
143DB5BFCE0FD108E4B82D120A92108011A723C12A787E6D788719A10BDBA5B2699C327186AF4E23C
1A946834B6150BDA2583E9CA2AD44CE8DBBBC2DB04DE8EF92E8EFC141FBECAA6287C59474E6BC05D9
9B2964FA090C3A2233BA186515BE7ED1F612970CEE2D7AFB81BDD762170481CD0069127D5B05AA993
B4EA988D8FDDC186FFB7DC90A6C08F4DF435C93402849236C3FAB4D27C7026C1D4DCB2602646DEC97
51E763DBA37BDF8FF9406AD9E530EE5DB382F413001AEB06A53ED9027D831179727B0865A8918DA3E
DBEBCF9B14ED44CE6CBACED4BB1BDB7F1447E6CC254B332051512BD7AF426FB8F401378CD2BF5983C
A01C64B92ECF032EA15D1721D03F482D7CE6E74FEF6D55E702F46980C82B5A84031900B1C9E59E7C9
7FBEC7E8F323A97A7E36CC88BE0F1D45B7FF585AC54BD407B22B4154AACC8F6D7EBF48E1D814CC5ED
20F8037E0A79715EEF29BE32806A1D58BB7C5DA76F550AA3D8A1FBFF0EB19CCB1A313D55CDA56C9EC
2EF29632387FE8D76E3C0468043E8F663F4860EE12BF2D5B0B7474D6E694F91E6DCC4024FFFFFFFFF
FFFFFFF,
        18:
        0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74020BBEA
63B139B22514A08798E3404DDEF9519B3CD3A431B302B0A6DF25F14374FE1356D6D51C245E485B576
625E7EC6F44C42E9A637ED6B0BFF5CB6F406B7EDEE386BFB5A899FA5AE9F24117C4B1FE649286651E
CE45B3DC2007CB8A163BF0598DA48361C55D39A69163FA8FD24CF5F83655D23DCA3AD961C62F35620
8552BB9ED529077096966D670C354E4ABC9804F1746C08CA18217C32905E462E36CE3BE39E772C180
E86039B2783A2EC07A28FB5C55DF06F4C52C9DE2BCBF6955817183995497CEA956AE515D2261898FA
051015728E5A8AAAC42DAD33170D04507A33A85521ABDF1CBA64ECFB850458DBEF0A8AEA71575D060
C7DB3970F85A6E1E4C7ABF5AE8CDB0933D71E8C94E04A25619DCEE3D2261AD2EE6BF12FFA06D98A08
64D87602733EC86A64521F2B18177B200CBBE117577A615D6C770988C0BAD946E208E24FA074E5AB3
143DB5BFCE0FD108E4B82D120A92108011A723C12A787E6D788719A10BDBA5B2699C327186AF4E23C
1A946834B6150BDA2583E9CA2AD44CE8DBBBC2DB04DE8EF92E8EFC141FBECAA6287C59474E6BC05D9
9B2964FA090C3A2233BA186515BE7ED1F612970CEE2D7AFB81BDD762170481CD0069127D5B05AA993
B4EA988D8FDDC186FFB7DC90A6C08F4DF435C93402849236C3FAB4D27C7026C1D4DCB2602646DEC97
51E763DBA37BDF8FF9406AD9E530EE5DB382F413001AEB06A53ED9027D831179727B0865A8918DA3E
DBEBCF9B14ED44CE6CBACED4BB1BDB7F1447E6CC254B332051512BD7AF426FB8F401378CD2BF5983C
A01C64B92ECF032EA15D1721D03F482D7CE6E74FEF6D55E702F46980C82B5A84031900B1C9E59E7C9
7FBEC7E8F323A97A7E36CC88BE0F1D45B7FF585AC54BD407B22B4154AACC8F6D7EBF48E1D814CC5ED
20F8037E0A79715EEF29BE32806A1D58BB7C5DA76F550AA3D8A1FBFF0EB19CCB1A313D55CDA56C9EC
2EF29632387FE8D76E3C0468043E8F663F4860EE12BF2D5B0B7474D6E694F91E6DBE115974A3926F1
2FEE5E438777CB6A932DF8CD8BEC4D073B931BA3BC832B68D9DD300741FA7BF8AFC47ED2576F6936B
```

```
A424663AAB639C5AE4F5683423B4742BF1C978238F16CBE39D652DE3FDB8BEFC848AD922222E04A40
37C0713EB57A81A23F0C73473FC646CEA306B4BCBC8862F8385DDFA9D4B7FA2C087E879683303ED5B
DD3A062B3CF5B3A278A66D2A13F83F44F82DDF310EE074AB6A364597E899A0255DC164F31CC508468
51DF9AB48195DED7EA1B1D510BD7EE74D73FAF36BC31ECFA268359046F4EB879F924009438B481C6C
D7889A002ED5EE382BC9190DA6FC026E479558E4475677E9AA9E3050E2765694DFC81F56E880B96E7
160C980DD98EDD3DFFFFFFFFFFFFFFFFF
        }

        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:

```
(learnout) PS C:\Users\Bhishm\Desktop\INS Practical> python .\sha_key.py
Shared keys match.
Key hash: b'172b2507194c203334782097bf3a84f91ba0554479ec2617a627ec895951510c'

Enter Message  : charusat
Message Hash : 87cf425fe21a75c8dbe84e4eb3a4a9ac28d564c7e8e508b64329baf8858f35de

Message Hash verification successful!
87cf425fe21a75c8dbe84e4eb3a4a9ac28d564c7e8e508b64329baf8858f35de  =  87cf425fe21a75c8dbe84e4eb3a4a9ac28d564c7e8e508b64329baf8858f35de
```