Task 1:

```
from Crypto.Hash import SHA256
from Crypto.Cipher import AES
import random

blockSizeBytes = 16

def padByteArray(array):
   if(len(array) % blockSizeBytes != 0):
       newLen = (len(array)//blockSizeBytes + 1) * blockSizeBytes
       addedBytes = newLen - len(array)
       padding = bytearray(addedBytes.to_bytes(1, 'little')) * addedBytes
       # print(padding)
       # print(array)
       return array + padding
   else:
       return array

# public key
# p = 37
p =
0xB10B8F96A080E01DDE92DE5EAE5D54EC52C99FBCFB06A3C69A6A9DCA52D23B616073E28675A23D189838EF1E2EE652C013ECB4AEA906112324975
C3CD49B83BFACCBDD7D90C4BD7098488E9C219A73724EFFD6FAE5644738FAA31A4FF55BCCC0A151AF5F0DC8B4BD45BF37DF365C1A65E68CFDA76D4D
A708DF1FB2BC2E4A4371

# large prime number
g =
0xA4D1CBD5C3FD34126765A442EFB99905F8104DD258AC507FD6406CFF14266D31266FEA1E5C41564B777E690F5504F213160217B4B01B886A5E915
47F9E2749F4D7FBD7D3B9A92EE1909D0D2263F80A76A6A24C087A091F531DBF0A0169B6A28AD662A4D18E73AFA32D779D5918D08BC8858F4DCEF97C
2A24855E6EEB22B3B2E5

# private keys
a = random.randint(0, p)
b = random.randint(0, p)

# Alice
A = (g ** a) % p

# Bob
B = (g ** b) % p

# A, B exchanged between Alice and Bob

# Alice calculates private combined key with B
sa = (B ** a) % p

# Bob calculates private combined key with A
sb = (A ** b) % p

# print(sa == sb)

aliceKeyHash = SHA256.new()
aliceKeyHash.update(sa.to_bytes(8, 'big'))
```

```python
bobKeyHash = SHA256.new()
bobKeyHash.update(sb.to_bytes(8, 'big'))

# print(aliceKeyHash.hexdigest()[0:16])
# print(bobKeyHash.hexdigest()[0:16])

aliceSend = padByteArray(b'Hello There')

aliceCipher = AES.new(aliceKeyHash.hexdigest()[0:16], AES.MODE_CBC, aliceKeyHash.hexdigest()[16:32])
aliceMessage = aliceCipher.encrypt(aliceSend)

bobCipher = AES.new(bobKeyHash.hexdigest()[0:16], AES.MODE_CBC, bobKeyHash.hexdigest()[16:32])
bobRecieved = bobCipher.decrypt(aliceMessage)

print(bobRecieved) # bobRecieved is the same as aliceSend except with padded bytes on the end
```

Task 2:

// copy the first 50 lines from task 1

```python
aliceKeyHash = SHA256.new()
aliceKeyHash.update(sa.to_bytes(8, 'big'))

bobKeyHash = SHA256.new()
bobKeyHash.update(sb.to_bytes(8, 'big'))

# print(aliceKeyHash.hexdigest()[0:16])
# print(bobKeyHash.hexdigest()[0:16])

aliceSend = padByteArray(b'Hello There')

aliceCipher = AES.new(aliceKeyHash.hexdigest()[0:16], AES.MODE_CBC,
aliceKeyHash.hexdigest()[16:32])
aliceMessage = aliceCipher.encrypt(aliceSend)

# print(aliceMessage)

bobCipher = AES.new(bobKeyHash.hexdigest()[0:16], AES.MODE_CBC,
bobKeyHash.hexdigest()[16:32])
bobRecieved = bobCipher.decrypt(aliceMessage)

print(bobRecieved)


# Mallory decryption
# if g = 1, shared secret will always be 1, hash will always be SHA256(1)
# if g = p-1, shared secret will always be 1, hash will always be SHA256(1)
```

```
# mallorySecret = 1
# malloryKeyHash = SHA256.new()
# malloryKeyHash.update(mallorySecret.to_bytes(8, 'big'))
# malloryCipher = AES.new(malloryKeyHash.hexdigest()[0:16], AES.MODE_CBC,
malloryKeyHash.hexdigest()[16:32])
# malloryRecieved = malloryCipher.decrypt(aliceMessage)

# print(malloryRecieved)

# if g = p, shared secret will always be 0, hash will always be SHA256(0)
mallorySecret = 0
malloryKeyHash = SHA256.new()
malloryKeyHash.update(mallorySecret.to_bytes(8, 'big'))
malloryCipher = AES.new(malloryKeyHash.hexdigest()[0:16], AES.MODE_CBC,
malloryKeyHash.hexdigest()[16:32])
malloryRecieved = malloryCipher.decrypt(aliceMessage)

print(malloryRecieved)
```

Task 3 pt 1:

```
from Crypto.Util import number
from math import gcd

def RSA_encrypt(message:str, e, n):
   byteMessage = bytearray(message, 'ascii')
   intMessage = int.from_bytes(byteMessage, "big")
   return pow(intMessage, e, n)

def RSA_decrypt(ciphertext:int, d, n):
   messageInt = pow(ciphertext, d, n)
   bytetext = messageInt.to_bytes(messageInt.bit_length()//8 + 1, 'big')
   return bytetext.decode('ascii')

# source =
https://stackoverflow.com/questions/4798654/modular-multiplicative-inverse-function-i
n-python
def multiplicativeInverse(x, p):
   return pow(x, -1, p)

def generateKeypairs(bitwidth, e):
   # choose 2 prime numbers p, q
   p = number.getPrime(bitwidth)
   q = number.getPrime(bitwidth)
```

```
    n = p*q

    # num coprimes with N = (p-1)(q-1) = Phi(N)
    phi = (p-1)*(q-1)

    # choose d such that (d * e) % Phi(N) = 1
    inverse = multiplicative_inverse(e, phi)
    d = pow(inverse, 1, phi)

    # public key = (N, e)
    return (n, d)

e = 65537

n, BobPrivateKey = generateKeypairs(2048, e)

AliceMessage = "YY"
AliceEncrypted = RSA_encrypt(AliceMessage, e, n)

BobDecrypted = RSA_decrypt(AliceEncrypted, BobPrivateKey, n)
print(BobDecrypted)
```

Task 3 pt 2:
// copy methods from above

```
def AES_Encrypt(msg, key, iv):
    _AES = AES.new(key, AES.MODE_CBC, IV=iv)
    cipher = _AES.encrypt(msg)
    return cipher

def AES_Decrypt(cipher, key, iv):
    _AES = AES.new(key, AES.MODE_CBC, IV=iv)
    msg = _AES.decrypt(cipher)
    return msg

def padString(array):
    if(len(array) % 16 != 0):
        newLen = (len(array)//16 + 1) * 16
        addedBytes = newLen - len(array)
        padding = chr(newLen) * addedBytes
        print(padding)
        print(array)
        return array + padding
    else:
        return array
```

```python
def hack(cipherText):
   return 1

e = 3

n, d = generateKeypairs(8, e)

BobMessage = str(random.randint(1, n))
c = RSA_encrypt(BobMessage, e, n)

cPrime = hack(c)     # what does this need to be such that mallory can decrypt
encryptedMessage?
                   # would need initVector and aliceDecrypted
                   # knows n, e, c, message
                   # doesn't know d

aliceDecrypted = RSA_decrypt(cPrime, d, n)
aliceSHA256 = SHA256.new()
aliceSHA256.update(bytearray(aliceDecrypted, 'ascii'))

initVector = os.urandom(16)
encryptedMessage = AES_Encrypt(padString("Hi Bob!"), aliceSHA256.hexdigest()[0:16],
initVector)

malloryKey = RSA_decrypt(1, 1, n)
mallorySHA256 = SHA256.new()
mallorySHA256.update(bytearray(malloryKey, 'ascii'))

decryptedMessage = AES_Decrypt(encryptedMessage, mallorySHA256.hexdigest()[0:16],
initVector)
print(decryptedMessage)
```

Questions:

1. If an attacker only sees the messages passing by in the network, it would take millions of years to crack a DHP that follows the correct protocols. One way an attacker can gain access to the messages being sent back and forth would be to act as a middle man; intercept the initial trading of keys such that the attacker has a shared secret with both parties.
2. No. The private keys can be guessed easily through brute force when the prime numbers are small.
3. This attack is possible because of the nature of exponents and the modulus operator and the possibility of a man in the middle. It can be prevented if strong prime numbers are selected as keys.
4. If two separate RSA conversations share the same n, this means that both of those conversations share the same p and q private keys. Therefore the people in one of the conversations can easily decrypt the messages of the other assuming a shared e value.