Task I.

A. Code

```
from Crypto.Hash import SHA256
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad
from random import randrange
import hashlib

p =
int("B10B8F96A080E01DDE92DE5EAE5D54EC52C99FBCFB06A3C69A6A9DCA52D23B616073E28675
A23D189838EF1E2EE652C013ECB4AEA906112324975C3CD49B83BFACCBDD7D90C4BD7098488E9C2
19A73724EFFD6FAE5644738FAA31A4FF55BCCC0A151AF5F0DC8B4BD45BF37DF365C1A65E68CFDA7
6D4DA708DF1FB2BC2E4A4371", 16)
g =
int("A4D1CBD5C3FD34126765A442EFB99905F8104DD258AC507FD6406CFF14266D31266FEA1E5C
41564B777E690F5504F213160217B4B01B886A5E91547F9E2749F4D7FBD7D3B9A92EE1909D0D226
3F80A76A6A24C087A091F531DBF0A0169B6A28AD662A4D18E73AFA32D779D5918D08BC8858F4DCE
F97C2A24855E6EEB22B3B2E5",16)

a = randrange(p) # a < p
b = randrange(p)  # b < p

# Alice and Bob both generates A and B
A = pow(g, a, p)
B = pow(g, b, p)

# generate S based on each other's A and B
Al_s = str(pow(B, a, p)).encode()
Bob_s = str(pow(A, b, p)).encode()

# generate key
Al_k = hashlib.sha256(Al_s).hexdigest()
Bob_k = hashlib.sha256(Bob_s).hexdigest()

# trim key to 32 bit because that's the max key size for AES-256
Al_k = Al_k[:32].encode()
Bob_k = Bob_k[:32].encode()

Al_m = "Hi Bob!".encode()
```

```
Bob_m = "Hi Alice!".encode()

iv = get_random_bytes(16)

# get Alice's cipher
Al_cipher = AES.new(Al_k, AES.MODE_CBC, iv)
Al_c = Al_cipher.encrypt(pad(Al_m, AES.block_size))

# get Bob's cipher
Bob_cipher = AES.new(Bob_k, AES.MODE_CBC, iv)
Bob_c = Bob_cipher.encrypt(pad(Bob_m, AES.block_size))

print("----")
# You can't encrypt and decrypt using the same object
Al_cipher = AES.new(Al_k, AES.MODE_CBC, iv)
Bob_cipher = AES.new(Bob_k, AES.MODE_CBC, iv)
print(unpad(Al_cipher.decrypt(Bob_c), AES.block_size)) # Alice decrypts bob's
message
print(unpad(Bob_cipher.decrypt(Al_c), AES.block_size)) # Bob decrypts Alice's
message
```

B. How hard would it be for an adversary to solve the Diffie Hellman Problem (DHP) given these parameters? What strategy might the adversary take?

The adversary can exhaust the signature space by picking a random p and g. We know that both p and g is a prime. If we limit p and g to be a 16 bits integer, there are only a limited number of primes out there, so we can brute force p, q, a, and b and continue making s and k, then see if our k matches.

C. Would the same strategy used for the tiny parameters work here? Why or why not?

No. While there's also a finite number of primes in the 1024-bit space, it will take too long to brute force.

Task II

1. Code

a. Change A and B to p

```
from Crypto.Hash import SHA256
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad
from random import randrange
import hashlib

p =
int("B10B8F96A080E01DDE92DE5EAE5D54EC52C99FBCFB06A3C69A6A9DCA52D23B616073E28675
A23D189838EF1E2EE652C013ECB4AEA906112324975C3CD49B83BFACCBDD7D90C4BD7098488E9C2
19A73724EFFD6FAE5644738FAA31A4FF55BCCC0A151AF5F0DC8B4BD45BF37DF365C1A65E68CFDA7
6D4DA708DF1FB2BC2E4A4371", 16)
g =
int("A4D1CBD5C3FD34126765A442EFB99905F8104DD258AC507FD6406CFF14266D31266FEA1E5C
41564B777E690F5504F213160217B4B01B886A5E91547F9E2749F4D7FBD7D3B9A92EE1909D0D226
3F80A76A6A24C087A091F531DBF0A0169B6A28AD662A4D18E73AFA32D779D5918D08BC8858F4DCE
F97C2A24855E6EEB22B3B2E5",16)

a = randrange(p) # a < p
b = randrange(p)  # b < p

# Alice and Bob generates A and B
A = pow(g, a, p)
B = pow(g, b, p)

# Here, mallory modifies A and B to p
A = p
B = p

# Alice and Bob creates s based on each other's A and B
Al_s = str(pow(B, a, p)).encode()
Bob_s = str(pow(A, b, p)).encode()

# Alice and Bob creates a key
Al_k = hashlib.sha256(Al_s).hexdigest()
Bob_k = hashlib.sha256(Bob_s).hexdigest()
```

```python
# Only use the first 32 bits because AES-256 only accepts 32-bit key
Al_k = Al_k[:32].encode()
Bob_k = Bob_k[:32].encode()

# Encode their message to bytes
Al_m = "Hi Bob!".encode()
Bob_m = "Hi Alice!".encode()

iv = get_random_bytes(16)
# get Alice's cipher
Al_cipher = AES.new(Al_k, AES.MODE_CBC, iv)
Al_c = Al_cipher.encrypt(pad(Al_m, AES.block_size))

# get Bob's cipher
Bob_cipher = AES.new(Bob_k, AES.MODE_CBC, iv)
Bob_c = Bob_cipher.encrypt(pad(Bob_m, AES.block_size))

# -- As mallory
# Mallory needs to guess "a" , it really doesn't matter which one
guess_a = 1
while True:
   try:
       guess_s = str(pow(p, guess_a, p)).encode()
       guess_k = hashlib.sha256(guess_s).hexdigest()[:32].encode()
       # try decrypting Bob's text
       mallory_cipher_1 = AES.new(guess_k, AES.MODE_CBC, iv)
       print("AS MALLORY -- try deciphering Alice's and Bob's message")
       print(unpad(mallory_cipher_1.decrypt(Bob_c), AES.block_size))
       mallory_cipher_2 = AES.new(guess_k, AES.MODE_CBC, iv)
       print(unpad(mallory_cipher_2.decrypt(Al_c), AES.block_size))
       break
   except:
       print("Unsuccesful")
       guess_a += 1 # get another a

print("AS ALICE AND BOB -- decipher each other's message")
# You can't encrypt and decrypt using the same eobject
Al_cipher = AES.new(Al_k, AES.MODE_CBC, iv)
Bob_cipher = AES.new(Bob_k, AES.MODE_CBC, iv)
print(unpad(Al_cipher.decrypt(Bob_c), AES.block_size)) # Alice decrypts bob's
message
```

```
print(unpad(Bob_cipher.decrypt(Al_c), AES.block_size)) # Bob decrypts Alice's
message
```

Tampering by setting g to 1, p, or p - 1

```
from Crypto.Hash import SHA256
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad
import hashlib
from random import randrange

# g is set to 1
p =
int("B10B8F96A080E01DDE92DE5EAE5D54EC52C99FBCFB06A3C69A6A9DCA52D23B616073E28675A23D189
838EF1E2EE652C013ECB4AEA906112324975C3CD49B83BFACCBDD7D90C4BD7098488E9C219A73724EFFD6F
AE5644738FAA31A4FF55BCCC0A151AF5F0DC8B4BD45BF37DF365C1A65E68CFDA76D4DA708DF1FB2BC2E4A4
371", 16)
g = 1 # manually adjust this to p or p - 1

a = randrange(p) # a < p
b = randrange(p)  # b < p

A = pow(g, a, p)
B = pow(g, b, p)

Al_s = str(pow(B, a, p)).encode()
Bob_s = str(pow(A, b, p)).encode()

Al_k = hashlib.sha256(Al_s).hexdigest()
Bob_k = hashlib.sha256(Bob_s).hexdigest()

Al_k = Al_k[:32].encode()
Bob_k = Bob_k[:32].encode()

Al_m = "Hi Bob!".encode()
Bob_m = "Hi Alice!".encode()

iv = get_random_bytes(16)
# get Alice's cipher
Al_cipher = AES.new(Al_k, AES.MODE_CBC, iv)
```

```
Al_c = Al_cipher.encrypt(pad(Al_m, AES.block_size))

# get Bob's cipher
Bob_cipher = AES.new(Bob_k, AES.MODE_CBC, iv)
Bob_c = Bob_cipher.encrypt(pad(Bob_m, AES.block_size))

print("AS ALICE AND BOB -- decipher each other's message----")
# You can't encrypt and decrypt using the sam eobject
Al_cipher = AES.new(Al_k, AES.MODE_CBC, iv)
Bob_cipher = AES.new(Bob_k, AES.MODE_CBC, iv)
print(unpad(Al_cipher.decrypt(Bob_c), AES.block_size)) # Alice decrypts bob's message
print(unpad(Bob_cipher.decrypt(Al_c), AES.block_size)) # Bob decrypts Alice's message

# As mallory
# if g = 1, then 1 mod anything = 1. A and b == 1
# if g = p, then p^(anything) mod p = 0 (because it'd be divisible by p)
# if g = p - 1, then (p-1)^(x)mod p = ((p-1)mod p)^x mod p = ((p mod p) - (1 mod p))^x
mod p = (-1) ^x mod p = 1

# So, A and B is either 0 or 1

# 1. Assume A and B = 0, then s and k = sha0
print("As Mallory ----")
try:
   key_sha0 = hashlib.sha256("0".encode()).hexdigest()[:32].encode()
   Al_cipher = AES.new(key_sha0, AES.MODE_CBC, iv)
   Bob_cipher = AES.new(key_sha0, AES.MODE_CBC, iv)
   print(unpad(Al_cipher.decrypt(Bob_c), AES.block_size)) # Alice decrypts bob's
message
   print(unpad(Bob_cipher.decrypt(Al_c), AES.block_size)) # Bob decrypts Alice's
message
except ValueError: #incorrect padding because message is gibberish
   key_sha0 = hashlib.sha256("1".encode()).hexdigest()[:32].encode()
   Al_cipher = AES.new(key_sha0, AES.MODE_CBC, iv)
   Bob_cipher = AES.new(key_sha0, AES.MODE_CBC, iv)
   print(unpad(Al_cipher.decrypt(Bob_c), AES.block_size)) # Alice decrypts bob's
message
   print(unpad(Bob_cipher.decrypt(Al_c), AES.block_size)) # Bob decrypts Alice's
message
```

2. Why is the attack possible? What is necessary to prevent it?

Part 1 - Changing A and B to p,

Alice’s and Bob’s formula for s becomes
s = p^(x) mod p where x is an arbitrary number

We know that it always returns 0 because p is divisible by itself. Raising p by an exponent x means multiplying p with itself x times. Since p is multiplied to itself, modulus of p is still 0.

Therefore Alice’s and Bob’s has to be SHA256 of 0. Mallory can then decrypt the message

Part 2 - Tampering with g

if g = 1, then 1 mod anything = 1. Alice’s and Bob’s s = **1**
if g = p, then p^(anything) mod p = **0** (As explained in Part 1)
if g = p - 1, then
= (p-1)^(x) mod p
= ((p-1)mod p)^x mod p
= ((p mod p) - (1 mod p))^x mod p
= (-1) ^x mod p = **1**

Alice’s and Bob’s key is either SHA256(1) or SHA256(0). Since there’s only 2 options, Mallory can brute force and see which key produces a tangible message.

Part 3 - To prevent it
A way to work around this is to add an additional logic before generating s. Alice and Bob should check that P and q are both strong primes. A definition of strong prime = https://en.wikipedia.org/wiki/Strong_prime#Definition_in_cryptography.

Task III

1. Why is it bad to share modulus n?

Say Alice and Bob have the same n, that means that Alice and Bob have the same p and q. Since phi is (p-1, q-1), and e is a publicly available key, Alice can easily compute Bob's private key (d).

2. Another RSA malleability attack?

e^{th} root attack;
We know that encryption works this way:
$C = p^e \bmod(n)$
If our p and e is small relative to n ($p^e < n$), then a plaintext can be retrieve by calculating the e-th root of c
$P = \sqrt[e]{c}$ because if p^e is smaller than n, then $p^e \bmod n$ is p^e itself.
In this case, we have a violation of integrity because a man in the middle can decrypt the cipher.

3. Explain the signature process

To forge a new m, Mallory needs to create $M^d \bmod n$ -- which is equivalent to breaking RSA encryption (because Mallory needs to know the private key d)

Assume Mallory knows M1 and M2 from Alice and have 2 valid signatures. Mallory wants to encrypt M3 such that M3 = M1 * M2
$S1 = M1^d \bmod n$
$S2 = M2^d \bmod n$

$S1 * S2 = (M1^d \bmod n) * (M2^d \bmod n) = (M1 * M2)^d \bmod n$
(S1 * S2) is now a valid signature fo M3