CS 445: Applied Cryptography and Computer Systems Security

Public Cipher Lab

Team Members

Chinwe Ibegbu
Nana Kofi Boakye

November 2022

Task 1

Code

```
import random;
from Crypto.Hash import SHA256
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from base64 import b64encode

"""
If you want to use p and g as 37 and 5 respectively, uncomment them.
The current implementation would be off of using the IETF suggestion
"""

# p and g as hex values
p_hex = "B10B8F96 A080E01D DE92DE5E AE5D54EC 52C99FBC FB06A3C6 9A6A9DCA
52D23B61 6073E286 75A23D18 9838EF1E 2EE652C0 13ECB4AE A9061123 24975C3C
D49B83BF ACCBDD7D 90C4BD70 98488E9C 219A7372 4EFFD6FA E5644738 FAA31A4F
F55BCCC0 A151AF5F 0DC8B4BD 45BF37DF 365C1A65 E68CFDA7 6D4DA708 DF1FB2BC
2E4A4371"
g_hex = "A4D1CBD5 C3FD3412 6765A442 EFB99905 F8104DD2 58AC507F D6406CFF
14266D31 266FEA1E 5C41564B 777E690F 5504F213 160217B4 B01B886A 5E91547F
9E2749F4 D7FBD7D3 B9A92EE1 909D0D22 63F80A76 A6A24C08 7A091F53 1DBF0A01
69B6A28A D662A4D1 8E73AFA3 2D779D59 18D08BC8 858F4DCE F97C2A24 855E6EEB
22B3B2E5"

# Function to return the decimal values of a string of hex values
def hex_value(p):
    p_split = p.split(" ")
    # new_p = [];
    new_p = ""
    for i in p_split:
        temp = int(i, 16)
        new_p += str(temp)
    # new_p = new_p.join()
    return int(new_p)

# p and g as decimals from the hex calc
p = hex_value(p_hex)
```

```
g = hex_value(g_hex)

print("p is ", p ,"\ng is ", g)
# p and g as 37 and 5
# p = 37
# g = 5


# Function for generating a random prime numbers in the range x and y,
lower and upper bound respectively
def primesInRange(x, y):
    prime_list = []
    for n in range(x, y):
        isPrime = True

        for num in range(2, n):
            if n % num == 0:
                isPrime = False

        if isPrime:
            prime_list.append(n)

    return prime_list


# Select a random number from the prime number list which is from 1 to
1000 for Alice and Bob || Consider making it such that the number picked
by Alice cant be the same as Bob
randomAlice = random.choice(primesInRange(1,1000));
randomBob = random.choice(primesInRange(1,1000));
print ("Alice's generated number is: ", randomAlice, "\nBob's number is:",
randomBob);


# Function that can be used to calculate g^a*mod(p) or B = g^b*mod(p)
def modCalculator(uno,exponent):
    return (uno**exponent)%p

# Alice and Bob's first modular calculation
AlicePrimaryCalc = modCalculator(g, randomAlice);
```

```
BobPrimaryCalc = modCalculator(g, randomBob);


# Alice and Bob's second modular calculation
AliceSecondaryCalc = modCalculator(BobPrimaryCalc, randomAlice);
BobSecondaryCalc = modCalculator(AlicePrimaryCalc, randomBob);

# Bytearray form of Alice and Bob's second modular calculation
byteArrayAlice = str(AliceSecondaryCalc).encode("utf8");
byteArrayBob = str(BobSecondaryCalc).encode("utf8");

# SHA 256
SHA_Alice = SHA256.new()
SHA_Bob = SHA256.new()

# Alice and Bob's "k"
Alice_k = SHA_Alice.update(byteArrayAlice);
Bob_k = SHA_Bob.update(byteArrayBob);

print ("Alice's key is: ", SHA_Alice.hexdigest() , "\nBob's key is: ",
SHA_Bob.hexdigest())

if (SHA_Alice.digest() ==  SHA_Bob.digest()):
    print ("Älice and Bob's symmetric key k are equal")
else:
    print ("Alice and Bob's symmetric key k arent equal")

# Truncating Bob and Alice to 16 bytes and in bytes unencoded
byteArrayAlice = bytearray(SHA_Alice.digest())
byteArrayBob = bytearray(SHA_Bob.digest())

byteArrayAlice = byteArrayAlice[:17]
byteArrayBob = byteArrayBob[:17]

aliceKey = byteArrayAlice[0:-1]
bobKey = byteArrayBob[0:-1]

print("Alice and Bob's truncated keys are: ", aliceKey)

# Bob and Alice's message in bytes
```

```
messageBob = b"Hi Alice!"
messageAlice = b"Hi Bob!"

# Alice and Bob's encrypted messages
cipher = AES.new(aliceKey, AES.MODE_CBC)
AliceCipherText_Bytes = cipher.encrypt(pad(messageAlice, AES.block_size))
BobCipherText_Bytes = cipher.encrypt(pad(messageBob, AES.block_size))

# Printing Bob and Alice's AES encrypted messages
AliceCipherText = b64encode(AliceCipherText_Bytes).decode('utf-8')
BobCipherText = b64encode(BobCipherText_Bytes).decode('utf-8')
print("Alice's Cipher Text is ", AliceCipherText)
print("Bob's Cipher Text is ", BobCipherText)
```

Question 1

"For task 1, how hard would it be for an adversary to solve the Diffie-Hellman Problem (DHP) given these parameters? What strategy might the adversary take?"

Given the parameter of p and g being 37 and 5, it would be easy for an adversary to attack. The best way would be a brute force attack that would use the first 100 values for p and g and then trying the generated key to decrypt the message.
If the parameters are the ones from the IETF suggestion then the adversary would have a really hard time. Unless the adversary uses a Man-in-the-middle attack, it would be extremely difficult for the adversary to get the number.

Question 2

"For task 1, would the same strategy used for the tiny parameters work for the p and g? Why or why not?"

We reckon that it would work for task 1. The reason why we say so is that the values are small and so by being in the middle the attacker could get some useful information to garner what p and g could be. Aside from that, if the strategy worker for large values then for smaller values with smaller permutations, it wouldn't be harder but easier.

Question 3

For task 2, why were these attacks possible? What is necessary to prevent it?
For task 2, the attacks were possible because the values that were sent between Alice and Bob had no further verification. If Alice and Bob could communicate

between the two of them, confirming the value sent then even if the value of p and g are altered, they could factor it once more.

Question 4

For task 3 part 1, while it's very common for many people to use the same value for e in their key (common values are 3, 7, 216+1), it is very bad if two people use the same RSA modulus n. Briefly describe why this is, and what the ramifications are.

if a single message were encrypted and sent to two or more entities in the network, then there is a technique by which an eavesdropper (any entity not in the network) could recover the message with high probability using only publicly available information and then both are quite available for attacking

Question 2

"Another RSA malleability attack?"

Question 3

"Explain the signature process"

Task 2

Code

```
import random;
from Crypto.Hash import SHA256
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from base64 import b64encode

"""
If you want to use p and g as 37 and 5 respectively, uncomment them.
The current implementation would be off of using the IETF suggestion
"""

# p and g as hex values
```

```
p_hex = "B10B8F96 A080E01D DE92DE5E AE5D54EC 52C99FBC FB06A3C6 9A6A9DCA
52D23B61 6073E286 75A23D18 9838EF1E 2EE652C0 13ECB4AE A9061123 24975C3C
D49B83BF ACCBDD7D 90C4BD70 98488E9C 219A7372 4EFFD6FA E5644738 FAA31A4F
F55BCCC0 A151AF5F 0DC8B4BD 45BF37DF 365C1A65 E68CFDA7 6D4DA708 DF1FB2BC
2E4A4371"
g_hex = "A4D1CBD5 C3FD3412 6765A442 EFB99905 F8104DD2 58AC507F D6406CFF
14266D31 266FEA1E 5C41564B 777E690F 5504F213 160217B4 B01B886A 5E91547F
9E2749F4 D7FBD7D3 B9A92EE1 909D0D22 63F80A76 A6A24C08 7A091F53 1DBF0A01
69B6A28A D662A4D1 8E73AFA3 2D779D59 18D08BC8 858F4DCE F97C2A24 855E6EEB
22B3B2E5"

# Function to return the decimal values of a string of hex values
def hex_value(p):
    p_split = p.split(" ")
    # new_p = [];
    new_p = ""
    for i in p_split:
        temp = int(i, 16)
        new_p += str(temp)
    # new_p = new_p.join()
    return int(new_p)

# p and g as decimals from the hex calc
p = hex_value(p_hex)
g = hex_value(g_hex)

print("p is ", p ,"\ng is ", g)
# p and g as 37 and 5
# p = 37
# g = 5



# Function for generating a random prime numbers in the range x and y,
lower and upper bound respectively
def primesInRange(x, y):
    prime_list = []
    for n in range(x, y):
        isPrime = True

        for num in range(2, n):
```

```
            if n % num == 0:
                isPrime = False

        if isPrime:
            prime_list.append(n)

    return prime_list


# Select a random number from the prime number list which is from 1 to
1000 for Alice and Bob || Consider making it such that the number picked
by Alice cant be the same as Bob
randomAlice = random.choice(primesInRange(1,1000));
randomBob = random.choice(primesInRange(1,1000));
print ("Alice's generated number is: ", randomAlice, "\nBob's number is:",
randomBob);


# Function that can be used to calculate g^a*mod(p) or B = g^b*mod(p)
def modCalculator(uno,exponent):
    return (uno**exponent)%p

# Alice and Bob's first modular calculation
AlicePrimaryCalc = modCalculator(g, randomAlice);
BobPrimaryCalc = modCalculator(g, randomBob);

"""
MALLORY'S ATTACK
"""
AlicePrimaryCalc = p
BobPrimaryCalc = p
print("\n\nATTACK DETECTED.... MALLORY HAS ATTACKED AND TAMPERED WITH THE
VALUES ALICE AND BOB SENT EACH OTHER\n\n")

# Alice and Bob's second modular calculation
AliceSecondaryCalc = modCalculator(BobPrimaryCalc, randomAlice);
BobSecondaryCalc = modCalculator(AlicePrimaryCalc, randomBob);

# Bytearray form of Alice and Bob's second modular calculation
byteArrayAlice = str(AliceSecondaryCalc).encode("utf8");
```

```
byteArrayBob = str(BobSecondaryCalc).encode("utf8");

# SHA 256
SHA_Alice = SHA256.new()
SHA_Bob = SHA256.new()

# Alice and Bob's "k"
Alice_k = SHA_Alice.update(byteArrayAlice);
Bob_k = SHA_Bob.update(byteArrayBob);

print ("Alice's key is: ", SHA_Alice.hexdigest() , "\nBob's key is: ",
SHA_Bob.hexdigest())

if (SHA_Alice.digest() ==  SHA_Bob.digest()):
    print ("Älice and Bob's symmetric key k are equal")
else:
    print ("Alice and Bob's symmetric key k arent equal")

# Truncating Bob and Alice to 16 bytes and in bytes unencoded
byteArrayAlice = bytearray(SHA_Alice.digest())
byteArrayBob = bytearray(SHA_Bob.digest())

byteArrayAlice = byteArrayAlice[:17]
byteArrayBob = byteArrayBob[:17]

aliceKey = byteArrayAlice[0:-1]
bobKey = byteArrayBob[0:-1]

print("Alice and Bob's truncated keys are: ", aliceKey)

# Bob and Alice's message in bytes
messageBob = b"Hi Alice!"
messageAlice = b"Hi Bob!"

# Alice and Bob's encrypted messages
cipher = AES.new(aliceKey, AES.MODE_CBC)
AliceCipherText_Bytes = cipher.encrypt(pad(messageAlice, AES.block_size))
BobCipherText_Bytes = cipher.encrypt(pad(messageBob, AES.block_size))

# Printing Bob and Alice's AES encrypted messages
```

```
AliceCipherText = b64encode(AliceCipherText_Bytes).decode('utf-8')
BobCipherText = b64encode(BobCipherText_Bytes).decode('utf-8')
print("Alice's Cipher Text is ", AliceCipherText)
print("Bob's Cipher Text is ", BobCipherText)
```

```
import random;
from Crypto.Hash import SHA256
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from base64 import b64encode

"""
If you want to use p and g as IETF suggestion respectively, uncomment
them.
The current implementation would be off of using the 37 and 5 respectively
"""

# p and g as hex values
p_hex = "B10B8F96 A080E01D DE92DE5E AE5D54EC 52C99FBC FB06A3C6 9A6A9DCA
52D23B61 6073E286 75A23D18 9838EF1E 2EE652C0 13ECB4AE A9061123 24975C3C
D49B83BF ACCBDD7D 90C4BD70 98488E9C 219A7372 4EFFD6FA E5644738 FAA31A4F
F55BCCC0 A151AF5F 0DC8B4BD 45BF37DF 365C1A65 E68CFDA7 6D4DA708 DF1FB2BC
2E4A4371"
g_hex = "A4D1CBD5 C3FD3412 6765A442 EFB99905 F8104DD2 58AC507F D6406CFF
14266D31 266FEA1E 5C41564B 777E690F 5504F213 160217B4 B01B886A 5E91547F
9E2749F4 D7FBD7D3 B9A92EE1 909D0D22 63F80A76 A6A24C08 7A091F53 1DBF0A01
69B6A28A D662A4D1 8E73AFA3 2D779D59 18D08BC8 858F4DCE F97C2A24 855E6EEB
22B3B2E5"

# Function to return the decimal values of a string of hex values
def hex_value(p):
    p_split = p.split(" ")
    # new_p = [];
    new_p = ""
    for i in p_split:
        temp = int(i, 16)
        new_p += str(temp)
    # new_p = new_p.join()
```

```
    return int(new_p)

# p and g as decimals from the hex calc
# p = hex_value(p_hex)
# g = hex_value(g_hex)

# p and g as 37 and 5
p = 37
g = 5
g_prime = g

"""
MALLORY'S ATTACK attack when she makes g=p-1
"""
g = p - 1
print("ATTACK DETECTED.... MALLORY HAS ATTACKED AND MADE g = p-1\n")


print("Printing the value of p and g")
print("p is currently", p ,"\ng is p-1 which is", g)

# Function for generating a random prime numbers in the range x and y,
lower and upper bound respectively
def primesInRange(x, y):
    prime_list = []
    for n in range(x, y):
        isPrime = True

        for num in range(2, n):
            if n % num == 0:
                isPrime = False

        if isPrime:
            prime_list.append(n)

    return prime_list
```

```
# Select a random number from the prime number list which is from 1 to
1000 for Alice and Bob || Consider making it such that the number picked
by Alice cant be the same as Bob
randomAlice = random.choice(primesInRange(1,1000));
randomBob = random.choice(primesInRange(1,1000));
print ("Alice's generated random number is: ", randomAlice, "\nBob's
random number is:", randomBob);


# Function that can be used to calculate g^a*mod(p) or B = g^b*mod(p)
def modCalculator(uno,exponent):
    return (uno**exponent)%p

# Alice and Bob's first modular calculation
AlicePrimaryCalc = modCalculator(g, randomAlice);
BobPrimaryCalc = modCalculator(g, randomBob);
print("Alice's A is ", AlicePrimaryCalc)
print("Bob's B is ", BobPrimaryCalc)
print("Alice and Bob have the same number for A and B which is p - 1 and
that is what they are exchanging")


# Alice and Bob's second modular calculation
AliceSecondaryCalc = modCalculator(BobPrimaryCalc, randomAlice);
BobSecondaryCalc = modCalculator(AlicePrimaryCalc, randomBob);
print("Alice's S is ", AliceSecondaryCalc)
print("Bob's B S ", BobSecondaryCalc)
print("Alice and Bob have the same number for S as well which is p - 1 and
that is what they are encrypting")


print("\n\nMallory is aware that Bob and Alice have the same A,B and S and
she knows it is p-1. Thus she can SHA encrypt p-1 and get the same key as
them. Thus when they exchange their messages, Mallory can decrypt it\n")
print("\n\nAttack when Mallory makes g = 1\n\n")



"""
```

```
g = 1
"""
p = 37
g = 5
g_prime = g

"""
MALLORY'S ATTACK
"""
g = 1
print("\n\nATTACK DETECTED.... MALLORY HAS ATTACKED AND made g = 1\n")


print("Printing the value of p and g")
print("p is currently", p ,"\nand g is", g)


# Select a random number from the prime number list which is from 1 to
1000 for Alice and Bob || Consider making it such that the number picked
by Alice cant be the same as Bob
randomAlice = random.choice(primesInRange(1,1000));
randomBob = random.choice(primesInRange(1,1000));
print ("Alice's generated random number is: ", randomAlice, "\nBob's
random number is:", randomBob);


# Function that can be used to calculate g^a*mod(p) or B = g^b*mod(p)
def modCalculator(uno,exponent):
    return (uno**exponent)%p

# Alice and Bob's first modular calculation
AlicePrimaryCalc = modCalculator(g, randomAlice);
BobPrimaryCalc = modCalculator(g, randomBob);
print("Alice's A is ", AlicePrimaryCalc)
print("Bob's B is ", BobPrimaryCalc)
print("Alice and Bob have the same number for A and B,which is 1, and that
is what they are exchanging")


# Alice and Bob's second modular calculation
```

```
AliceSecondaryCalc = modCalculator(BobPrimaryCalc, randomAlice);
BobSecondaryCalc = modCalculator(AlicePrimaryCalc, randomBob);
print("Alice's S is ", AliceSecondaryCalc)
print("Bob's B S ", BobSecondaryCalc)
print("Alice and Bob have the same number for S as well which is 1 and
that is what they are encrypting")


print("\n\nMallory is aware that Bob and Alice have the same A,B and S and
she knows it is 1. Thus she can SHA encrypt 1 and get the same key as
them. Thus when they exchange their messages, Mallory can decrypt it\n\n")




"""
g = p
"""
p = 37
g = 5
g_prime = g

"""
MALLORY'S ATTACK
"""
g = p
print("\n\nATTACK DETECTED.... MALLORY HAS ATTACKED AND made g = p\n")


print("Printing the value of p and g")
print("p is currently", p ,"\nand g is", g)


# Select a random number from the prime number list which is from 1 to
1000 for Alice and Bob || Consider making it such that the number picked
by Alice cant be the same as Bob
randomAlice = random.choice(primesInRange(1,1000));
randomBob = random.choice(primesInRange(1,1000));
print ("Alice's generated random number is: ", randomAlice, "\nBob's
random number is:", randomBob);
```

```
# Function that can be used to calculate g^a*mod(p) or B = g^b*mod(p)
def modCalculator(uno,exponent):
    return (uno**exponent)%p

# Alice and Bob's first modular calculation
AlicePrimaryCalc = modCalculator(g, randomAlice);
BobPrimaryCalc = modCalculator(g, randomBob);
print("Alice's A is ", AlicePrimaryCalc)
print("Bob's B is ", BobPrimaryCalc)
print("Alice and Bob have the same number for A and B which is 0 and that
is what they are exchanging")


# Alice and Bob's second modular calculation
AliceSecondaryCalc = modCalculator(BobPrimaryCalc, randomAlice);
BobSecondaryCalc = modCalculator(AlicePrimaryCalc, randomBob);
print("Alice's S is ", AliceSecondaryCalc)
print("Bob's B S ", BobSecondaryCalc)
print("Alice and Bob have the same number for S as well which is 0 and
that is what they are encrypting")


print("\n\nMallory is aware that Bob and Alice have the same A,B and S and
she knows it is 0. Thus she can SHA encrypt 0 and get the same key as
them. Thus when they exchange their messages, Mallory can decrypt it\n\n")
```

Question

“For task 2, why were these attacks possible? What is necessary to prevent it?”

Task 3

Code

```python
import random
from typing import Tuple

# Find n^k % p
def mod_calcualtion(n: int, k: int, p: int):
    if k == 0:
        return 1
    tmp = mod_calcualtion(n, k // 2, p) ** 2
    if k % 2 == 0:
        return tmp % p
    else:
        return tmp * n % p


# Check if a given number is prime
def is_prime(num, bases=None):
    if bases is None:
        bases = [2, 3, 5, 7, 11, 13, 17, 19, 23, 31]
    for base in bases:
        if mod_calcualtion(base, num, num) != base:
            return False
    return True

# Generate a prime number with a given length in base 2
def generate_prime(length):
    assert length >= 5
    start = 1 << length
    end = 1 << (length + 1)
    while True:
        num = random.randint(start, end - 1)
        if is_prime(num):
            return num

# Generate n = p * q === (n, phi(n))
def generate_n_and_phi_n(size):
    p, q = generate_prime(size // 2), generate_prime(size // 2 - 2)
```

```
    return p * q, (p - 1) * (q - 1)


# Find the solution of ax + by = gcd(x, y) === (x, y, gcd(x, y))
def extended_gcd(a, b):
    if b == 0:
        return 1, 0, a
    x, y, gcd = extended_gcd(b, a % b)
    return y, x - (a // b) * y, gcd

# Generate keys, returns (e, d, n), where e is the public key and d is the
private key.
def generate_keys(size: int = 512):
    n, phi_n = generate_n_and_phi_n(size)
    e = generate_prime(size // 2)
    d, _, _ = extended_gcd(e, phi_n)
    if d < 0:
        d += phi_n
    return e, d, n

# Encrypt a text with a public key and n
def encrypt(text, public_key, n):
    return mod_calcualtion(text, public_key, n)

# Decrypt a text with a private key and n
def decrypt(ciphertext, private_key, n):
    return mod_calcualtion(ciphertext, private_key, n)


# Testing
print("Generate keys:")
public_key, private_key, n = generate_keys()
print(f"public key: {public_key}")
print(f"private key: {private_key}")
print(f"n: {n}\n\n")

text = 123456789
print(f"Encrypting text {text}:")
ciphertext = encrypt(text, public_key, n)
print(f"ciphertext {ciphertext}:\n\n")
```

```
print(f"Decrypting text {ciphertext}:")
decrypted_text = decrypt(ciphertext, private_key, n)
print(f"Original text {decrypted_text}\n\n")
```