Public Ciphers

Caleb Dennis, Riley Mete
CPE321-03

Task 1: Implement Diffie-Helman key exchange

The Diffie-Helman key exchange allows users to create a shared secret without communicating any information that would make the secret insecure. In task 1 Alice and Bob independently compute the same shared secret which can be used as a key to securely exchange information. To find the shared secret an adversary must find the prime number that Alice or Bob each chose. To find this number the adversary needs to compute the log base g of Alice's public key. An adversary could check all primes, but if Alice has chosen a reasonably large prime this process takes a lot of computing.

```
from Crypto.Util.number import getPrime
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes
from Crypto.Hash import SHA256
from Crypto.Cipher import AES
from Crypto.Cipher import AES


def main():
    p = 37
    g = 5

    Alice_prime = getPrime(16)
    Bob_prime = getPrime(16)
    print(f"Alice's prime: {Alice_prime}")
    print(f"Bob's prime: {Bob_prime}")

    Alice_public = pow(g, Alice_prime) % p
    Bob_public = pow(g, Bob_prime) % p

    Alice_shared_secret = bytes(pow(Bob_public, Alice_prime) % p)
    Bob_shared_secret = bytes(pow(Alice_public, Bob_prime) % p)

    Alice_hasher = SHA256.new()
    Alice_hasher.update(Alice_shared_secret)
    alice_key = Alice_hasher.digest()

    Bob_hasher = SHA256.new()
    Bob_hasher.update(Bob_shared_secret)
    bob_key = Bob_hasher.digest()

    iv = get_random_bytes(16)
```

```
    print(f"Bob's key {alice_key}\nAlice's key {bob_key}\n")
    Alice_msg = pad(bytes("Hi Bob, i'm Alice", "ascii"), 16)
    print(f"Alice: {unpad(Alice_msg, 16)}")
    Alice_encrypter = AES.new(alice_key, AES.MODE_CBC, iv=iv)
    Alice_sends_to_Bob = Alice_encrypter.encrypt(Alice_msg)

    Bob_encrypter = AES.new(bob_key, AES.MODE_CBC, iv=iv)
    Bob_received_from_Alice = Bob_encrypter.decrypt(Alice_sends_to_Bob)
    print(f"Bob received: {unpad(Bob_received_from_Alice, 16)}")



if __name__ == "__main__":
    main()
```

Task2:

The Diffie-Helman key exchange can be unsecure if an adversary is able to adjust the numbers in the protocol. As shown in task2, an adversary can gain access to both the public and private keys of Bob and Allice by manipulating the numbers in the protocol so that the math to generate a private key returns a specific value.

The first example of this can be observed in main1. In this scenario, the adversary adjusts Alice and Bob's public key by setting them to be the value of 'p'. In doing this, the adversary is able to predict that the private key of both Alice and Bob will be 0 since p^prime_val % p will always equate to be 0 for any prime value.

The second example can be observed in main2. In this scenario, the adversary adjusts the 'g' value to be either 1, p, or p-1.

- In the case of g = 1, Alice and Bob's public and private key will always be 1
    - Public: 1^prime_val % p will always equal 1
    - Private: 1^prime_val % p will always equal 1
- In the case of g = p, Alice and Bob's public and private key will always be zero
    - Public: p^prime_val % p will always equal 0
    - Private: 0^prime_val % p will always equal 0
- In the case of g = (p-1), Alice and Bob's public and private will always be (p-1)
    - Public: (p-1)^prime_val % p will always equal (p-1) as long as the exponent is odd (in this case since it's prime then it is always odd so this works)
    - Private: (p-1)^prime_val % p will always equal (p-1)

Code:

```python
from Cryptodome.Util.number import getPrime
from Cryptodome.Util.Padding import pad, unpad
from Cryptodome.Random import get_random_bytes
from Cryptodome.Hash import SHA256
from Cryptodome.Cipher import AES



# Part 1
def main_1():
    p = 37
    g = 5

    # Get prime for Alice and Bob
    Alice_prime = getPrime(16)
    Bob_prime = getPrime(16)
    print(f"Alice's prime: {Alice_prime}")
    print(f"Bob's prime: {Bob_prime}")

    Alice_public = pow(g, Alice_prime) % p
    Bob_public = pow(g, Bob_prime) % p

    # Mallory's modifications
    Alice_public = p
    Bob_public = p

    # Because Mallory changed public to 'p', it will always mod to 0 no matter the
prime
    Alice_shared_secret = bytes(pow(Bob_public, Alice_prime) % p)
    Bob_shared_secret = bytes(pow(Alice_public, Bob_prime) % p)

    # Mallory can now be sneaky
    Mallory_stolen_secret = bytes(0)
    Mallory_hasher = SHA256.new()
    Mallory_hasher.update(Mallory_stolen_secret)
    mallory_key = Mallory_hasher.digest()

    Alice_hasher = SHA256.new()
    Alice_hasher.update(Alice_shared_secret)
    alice_key = Alice_hasher.digest()

    Bob_hasher = SHA256.new()
    Bob_hasher.update(Bob_shared_secret)
    bob_key = Bob_hasher.digest()

    # I think the IV is also just the key in this case
    # Cuz if not, not quite sure how mallory gets this value IRL
```

```
    iv = get_random_bytes(16)

    print(f"Bob's key {alice_key}\nAlice's key {bob_key}\n")
    Alice_msg = pad(bytes("Hi Bob, i'm Alice", "ascii"), 16)
    print(f"Alice: {unpad(Alice_msg, 16)}")
    Alice_encrypter = AES.new(alice_key, AES.MODE_CBC, iv)   #using alice_key for iv
    Alice_sends_to_Bob = Alice_encrypter.encrypt(Alice_msg)

    Bob_encrypter = AES.new(bob_key, AES.MODE_CBC, iv)       #using bob_key for iv
    Bob_received_from_Alice = Bob_encrypter.decrypt(Alice_sends_to_Bob)
    print(f"Bob received: {unpad(Bob_received_from_Alice, 16)}")

    # Mallory swooping in
    Mallory_encryptor = AES.new(mallory_key, AES.MODE_CBC, iv)
    Mallory_stolen_from_Alice = Mallory_encryptor.decrypt(Alice_sends_to_Bob)
    print(f"Mallory Intercepted: {unpad(Mallory_stolen_from_Alice, 16)}")

if __name__ == "__main__":
    main_1()

def main_2():
    p = 37
    g = 5

    # Get prime for Alice and Bob
    Alice_prime = getPrime(16)
    Bob_prime = getPrime(16)
    print(f"Alice's prime: {Alice_prime}")
    print(f"Bob's prime: {Bob_prime}")

    # Mallory's Modifications
    # g = 1 # This means the public will be 1 & secret will be 1
    # Mallory_stolen_secret = bytes(1)

    # g = p # This means the public and secret will be 0
    # Mallory_stolen_secret = bytes(0)

    g = p - 1 # This means the public and secret will be p - 1
    Mallory_stolen_secret = bytes(p-1) # Ex) (p = 7, g = p-1 = 6, prime=3) => 6^3 % 7
= 6 = (p-1)

    Alice_public = pow(g, Alice_prime) % p
    Bob_public = pow(g, Bob_prime) % p

    Alice_shared_secret = bytes(pow(Bob_public, Alice_prime) % p)
    Bob_shared_secret = bytes(pow(Alice_public, Bob_prime) % p)

    # Mallory can now be sneaky
```

```
    Mallory_hasher = SHA256.new()
    Mallory_hasher.update(Mallory_stolen_secret)
    mallory_key = Mallory_hasher.digest()

    Alice_hasher = SHA256.new()
    Alice_hasher.update(Alice_shared_secret)
    alice_key = Alice_hasher.digest()

    Bob_hasher = SHA256.new()
    Bob_hasher.update(Bob_shared_secret)
    bob_key = Bob_hasher.digest()

    # I think the IV is also just the key in this case
    # Cuz if not, not quite sure how mallory gets this value IRL
    iv = get_random_bytes(16)

    print(f"Bob's key {alice_key}\nAlice's key {bob_key}\n")
    Alice_msg = pad(bytes("Hi Bob, i'm Alice", "ascii"), 16)
    print(f"Alice: {unpad(Alice_msg, 16)}")
    Alice_encrypter = AES.new(alice_key, AES.MODE_CBC, iv)   #using alice_key for iv
    Alice_sends_to_Bob = Alice_encrypter.encrypt(Alice_msg)

    Bob_encrypter = AES.new(bob_key, AES.MODE_CBC, iv)       #using bob_key for iv
    Bob_received_from_Alice = Bob_encrypter.decrypt(Alice_sends_to_Bob)
    print(f"Bob received: {unpad(Bob_received_from_Alice, 16)}")

    # Mallory swooping in
    Mallory_encryptor = AES.new(mallory_key, AES.MODE_CBC, iv)
    Mallory_stolen_from_Alice = Mallory_encryptor.decrypt(Alice_sends_to_Bob)
    print(f"Mallory intercepted: {unpad(Mallory_stolen_from_Alice, 16)}")

if __name__ == "__main__":
    main_2()
```

Task 2 Output:

As we can see based on the output, Mallory was able to intercept the message sent to Bob by manipulating either Alice and Bob's public key or by manipulating the 'g' value.

```
Alice's prime: 63647
Bob's prime: 49603
Bob's key
b"\xe3\xb0\xc4B\x98\xfc\x1c\x14\x9a\xfb\xf4\xc8\x99o\xb9$'\xaeA\xe4d\x9b\x93L\xa4\
x95\x99\x1bxR\xb8U"
Alice's key
b"\xe3\xb0\xc4B\x98\xfc\x1c\x14\x9a\xfb\xf4\xc8\x99o\xb9$'\xaeA\xe4d\x9b\x93L\xa4\
x95\x99\x1bxR\xb8U"

Alice: b"Hi Bob, i'm Alice"
Bob received: b"Hi Bob, i'm Alice"
Mallory Intercepted: b"Hi Bob, i'm Alice"
Alice's prime: 50051
Bob's prime: 63367
Bob's key
b'm\xb6_\xd5\x9f\xd3V\xf6r\x91@W\x1b[\xcdk\xb3\xb84\x92\xa1n\x1b\xf0\xa3\x88DB\
xfc<\x8a\x0e'
Alice's key
b'm\xb6_\xd5\x9f\xd3V\xf6r\x91@W\x1b[\xcdk\xb3\xb84\x92\xa1n\x1b\xf0\xa3\x88DB\
xfc<\x8a\x0e'

Alice: b"Hi Bob, i'm Alice"
Bob received: b"Hi Bob, i'm Alice"
Mallory intercepted: b"Hi Bob, i'm Alice"
```

Task 3: RSA and MITM key fixing

Part 1: Man In the middle attack

If Mallory can intercept Bobs encrypted message, and modify it, then she can trick Allice into using her own key for securing future communications. The below code simulates a key exchange between Alice and Bob, which is then interfered with by Mallory. It produces the following output:

```
n: 475855918583
L: 7674932880
d: 357297713
Bobs message as integer 1114595841
Bobs message in bytes: Bob
Alice's decrypted integer: 1114595841
Alice's decrypted message: Bob
if Mallory subsitutes her own key '4d616c01' which is: 1298230273 for Bob's
message
Allice receives 1298230273 and will decrypt the intercepted message to Mal
Alice is now using Mallory's key
```

Another attack Mallory could execute which takes advantage of RSA Malleability is to wait for Alice and Bob to both exchange keys and keep copies of both. She could then pretend to be Alice to Bob, pretend to be Bob to Alice, and listen to all the information they exchange.

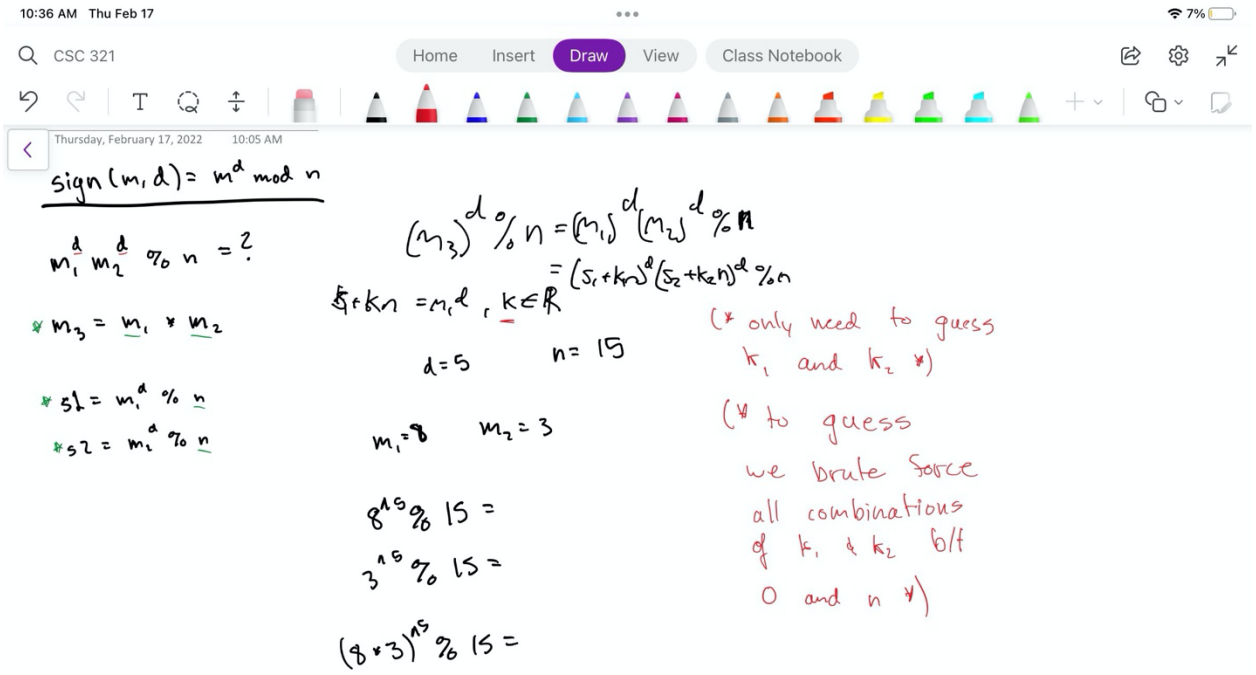
```python
from Crypto.Util.number import getPrime
from Crypto.Util.number import GCD
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes
from Crypto.Hash import SHA256
from Crypto.Cipher import AES


def modular_inverse(a, b):
    (old_r, r) = (a, b)
    (old_s, s) = (1, 0)
    (old_t, t) = (0, 1)
    while r != 0:
        quotient = old_r // r
        (old_r, r) = (r, old_r - quotient * r)
        (old_s, s) = (s, old_s - quotient * s)
        (old_t, t) = (t, old_t - quotient * t)
    return (old_s, old_t)


def main():
```

```
    e = 65537
    p = getPrime(24)
    q = getPrime(16)
    n = p * q
    print(f"n: {n}")
    L = (p - 1)*(q - 1) // GCD(p - 1, q - 1)
    print(f"L: {L}")
    if GCD(e, L ) != 1:
        print("Error: L and e are not coprime")
        quit()

    d = modular_inverse(e, L)[0]
    print(f"d: {d}")

    Bob_says = pad(bytes("Bob", "ascii"), 4).hex()
    print(f"Bobs message as integer {int(Bob_says, 16)}")
    Bob_says_bytes = bytes([int(Bob_says[idx:idx+2], 16) for idx in range(0,
len(Bob_says), 2)])
    print(f"Bobs message in bytes: {unpad(Bob_says_bytes, 4).decode('ascii')}")

    Bob_encrypts = pow(int(Bob_says, 16), e, mod=n)
    # print(Bob_encrypts)
    Alice_decrypts = pow(Bob_encrypts, d, mod=n)

    print(f"Alice's decrypted integer: {Alice_decrypts}")
    Alice_decrypts_bytes = Alice_decrypts.to_bytes(4, "big")
    print(f"Alice's decrypted message: {unpad(Alice_decrypts_bytes,
4).decode('ascii')}")

    Mallory_key = pad(bytes("Mal", "ascii"), 4).hex()
    print(f"if Mallory subsitutes her own key '{Mallory_key}' which is:
{int(Mallory_key, 16)} for Bob's message")
    Mallory_encrypts = pow(int(Mallory_key, 16), e, mod=n)
    Alice_decrypts = pow(Mallory_encrypts, d, mod=n)
    Alice_decrypts_bytes = Alice_decrypts.to_bytes(4, "big")
    print(f"Allice receives {Alice_decrypts} and will decrypt the intercepted
message to {unpad(Alice_decrypts_bytes, 4).decode('ascii')}")
    print("Alice is now using Mallory's key")



if __name__ == "__main__":
    main()
```

```python
def main1():
    e = 65537       # Alice sends e
    p = getPrime(24)
    q = getPrime(16)
    n = p * q       # Alice sends n

    # Bob computes s, select s to be n -1
    bob_s = n -1

    # Bob computes c using defined s from above and sends it back to Alice
    c = (bob_s ** e) % n

    # # Create a key for Bob using selected s value
    # bob_hasher = SHA256.new(data = str(bob_s).encode())
    # bob_key = bob_hasher.digest()[:16]

    # Mallory modifies c
    # By setting c to 1, Mallory knows that Alice's key will also have to be 1
    # Note: Mallory also knows the value of n and e
    Mallory_c = 1

    # Alice computes s using the modified c from Mallory
    # Because c is now 1, Mallory knows that s will compute to be 1
    L = (p - 1)*(q - 1) // GCD(p - 1, q - 1)
    d = modular_inverse(e, L)[0]
    s = (Mallory_c ** d) % n

    # Alice creates an encriptor
    alice_hasher = SHA256.new()
    alice_hasher.update(bytes(s))
    alice_key = alice_hasher.digest()[:16]
    alice_iv = alice_hasher.digest()[16:32]
    alice_encrypter = AES.new(alice_key, AES.MODE_CBC, alice_iv)

    # Alice sends encripted message c
    alice_msg = pad(bytes("Hi Bob!", "ascii"), 16)
    c0 = alice_encrypter.encrypt(alice_msg)

    # Mallory swooping in again, knowing that s will be 1
    # Because of this info, mallory can create their own key and encryption
    # Mallory can now decrpyt messages as well as send them
    mallory_hasher = SHA256.new()
    mallory_hasher.update(bytes(1))
    mallory_key = mallory_hasher.digest()[:16]
    mallory_iv = mallory_hasher.digest()[16:32]
```

```
    mallory_encrypter = AES.new(mallory_key, AES.MODE_CBC, mallory_iv)
    intercepted_msg = unpad(mallory_encrypter.decrypt(c0), 16).decode()
    print(f"Mallory intercepted the message {intercepted_msg}")



if __name__ == "__main__":
    main1()
```

Part 2: Explain Sign



Since the attacker knows that m3 = m1 * m2, and they also know what the modulus n is, they are able to create a valid signature for a third message by deconstructing the formula Sign(m,d) = m^d mod n. Since Mallory knows both the signatures for m1 and m2 as well as they modulus n, Mallory is able to find values k1 and k2 such that m3^k1 mod n = signature for m1 and m3^k2 mod n = signature for m2.

Questions:

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?

- . To find the shared secret an adversary must find the prime number that Alice or Bob each chose. To find this number the adversary needs to compute the log base g of Alice's public key. An adversary could check all primes, but if Alice has chosen a reasonably large prime this process takes a lot of computing.

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

- No, the above strategy used for tiny parameters would not work for large p and g values since the computation to check all the primes of a very large number would take too long to compute. In theory, you would be able to get an answer eventually, however in practice this wouldn't work because it would take many years.

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

- These kinds of attacks are possible because Mallory can predict what the secret key is going to be by adjusting g to be a certain value. Even though Mallory doesn't what the a and b values are, by adjusting g Mallory can adjust the g value so that the computation of Bob and Alice always compute to the same predicted value. This exploitation is possible since Mallory can modify the message without the sender or receiver knowing that it was modified. To prevent this type of attack, Bob or Alice can utilize a MAC (Message Authentication Code) to determine whether a message has been tampered with.

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

- Because n is the product of two primes p, and q, if two people have the same n they know that they must also share the same p and q. This means that they also share, and can compute, each other's private keys easily.