Assignment Group 9

Arian Houshmand
Lucas Summers

Module 3 Assignment - Public Ciphers

Task 1 - Implement Diffie-Hellman Key Exchange:

```
# small parameters
#q = 37
#alpha = 5

# large parameters
q = int("B10B8F96A080E01DDE92DE5EAE5D54EC52C99FBCFB06A3C69A6A9DCA52D23B616073E286"
       "75A23D189838EF1E2EE652C013ECB4AEA906112324975C3CD49B83BFACCBDD7D90C4BD70"
       "98488E9C219A73724EFFD6FAE5644738FAA31A4FF55BCCC0A151AF5F0DC8B4BD45BF37DF"
       "365C1A65E68CFDA76D4DA708DF1FB2BC2E4A4371", 16)

alpha = int("A4D1CBD5C3FD34126765A442EFB99905F8104DD258AC507FD6406CFF14266D31266FEA1E"
           "5C41564B777E690F5504F213160217B4B01B886A5E91547F9E2749F4D7FBD7D3B9A92EE1"
           "909D0D2263F80A76A6A24C087A091F531DBF0A0169B6A28AD662A4D18E73AFA32D779D59"
           "18D08BC8858F4DCEF97C2A24855E6EEB22B3B2E5", 16)
```

We set up the large parameters q (prime) and α (generator) for the Diffie-Hellman key exchange. Both Bob and Alice will be using this to create their keys.

```
# Alice keys
X_A = random.randint(1, q - 1)  # priv
Y_A = pow(alpha, X_A, q)  # pub

# Bob keys
X_B = random.randint(1, q - 1)  # priv
Y_B = pow(alpha, X_B, q)  # pub
```

Both Alice and Bob generate their private keys (X_A , X_B) randomly and compute their public keys (Y_A,Y_B) using modular exponentiation with the shared parameters from earlier: α and q.

```
# shared secret
s_A = pow(Y_B, X_A, q)   # Alice
s_B = pow(Y_A, X_B, q)   # Bob

# check shared secrets are equal
assert s_A == s_B, "Shared secrets do not match!"
s = s_A
```

Alice and Bob compute the shared secret s using the other party's public key and their own private key. The assertion ensures that both compute the same ss, validating the protocol.

```
shared_key = hashlib.sha256(str(s).encode()).digest()[:16]
```

We then hash the shared secret s using SHA-256 and truncate it to 16 bytes to derive the symmetric key used for encryption.

```
# Alice message encrypted
message_A = "Hi Bob!"
cipher = AES.new(shared_key, AES.MODE_CBC, iv=b'1234567890123456') # 16 dig IV
ciphertext_A = cipher.encrypt(pad(message_A.encode(), AES.block_size))
```

We then encrypt Alice's message , "Hi Bob!", using AES in CBC mode. We also make a fixed 16-byte IV and pad the plaintext to fit AES's block size.

```
# Bob decrypts Alice message
cipher = AES.new(shared_key, AES.MODE_CBC, iv=b'1234567890123456')
decrypted_message_A = unpad(cipher.decrypt(ciphertext_A), AES.block_size).decode()
```

We then decrypt Alice's message for Bob using the symmetric key and IV, retrieving the original message.

```
# Bob reply encrypted
message_B = "Hi Alice!"
cipher = AES.new(shared_key, AES.MODE_CBC, iv=b'1234567890123456')
ciphertext_B = cipher.encrypt(pad(message_B.encode(), AES.block_size))

# Alice decrypts Bob reply
cipher = AES.new(shared_key, AES.MODE_CBC, iv=b'1234567890123456')
decrypted_message_B = unpad(cipher.decrypt(ciphertext_B), AES.block_size).decode()
```

We then repeat the entire process for Bob's reply. Encrypting his reply and then decrypting it for Alice to view.

```
print("Alice's Message:", message_A)
print("Ciphertext Sent from Alice to Bob:", ciphertext_A)
print("Decrypted Message at Bob's End:", decrypted_message_A)
print("Bob's Message:", message_B)
print("Ciphertext Sent from Bob to Alice:", ciphertext_B)
print("Decrypted Message at Alice's End:", decrypted_message_B)
```

Printing the output, we get the following:

```
Alice's Message: Hi Bob!
Ciphertext Sent from Alice to Bob: b'|j\xf4.H\xba\xc6\x85Pv\xb5\xc1\xbb\x17?#'
Decrypted Message at Bob's End: Hi Bob!
Bob's Message: Hi Alice!
Ciphertext Sent from Bob to Alice: b"-\xe7\x8e\xe17p\xedO\xace'\xdew\xa9S\xf9"
Decrypted Message at Alice's End: Hi Alice!
```

Task 2:

1. Tampering with Keys

```
# Alice's private and public keys
X_A = random.randint(1, q - 1)
Y_A = pow(alpha, X_A, q)

# Bob's private and public keys
X_B = random.randint(1, q - 1)
Y_B = pow(alpha, X_B, q)
```

Alice's and Bob's key get generated normally

```
# Mallory intercepts and replaces public keys
Y_A_tampered = q
Y_B_tampered = q
```

Mallory intercepts the public keys and send q as the public keys instead to both Bob and Alice

```
# Alice and Bob compute shared secrets
```

```
s_A = pow(Y_B_tampered, X_A, q)
s_B = pow(Y_A_tampered, X_B, q)

# Mallory computes the same shared secret (s = 0 due to tampered keys)
mallory_shared_secret = 0
```

Shared secret is easily known to Mallory because s = q ^ X mod q will always be 0.

```
# Generate keys
shared_key = hashlib.sha256(str(s_A).encode()).digest()[:16]
mallory_key =
hashlib.sha256(str(mallory_shared_secret).encode()).digest()[:16]
```

Both shared keys must be the same because of the same shared secret.

```
# Alice encrypts a message
message_A = "Hi Bob!"
cipher = AES.new(shared_key, AES.MODE_CBC, iv=b'1234567890123456')
ciphertext_A = cipher.encrypt(pad(message_A.encode(), AES.block_size))
```

```
# Mallory decrypts the message
cipher = AES.new(mallory_key, AES.MODE_CBC, iv=b'1234567890123456')
mallory_decrypted_message = unpad(cipher.decrypt(ciphertext_A),
AES.block_size).decode()

print("Mallory Decrypted Message from Alice to Bob:",
mallory_decrypted_message)
```

Mallory is able to successfully decrypt the message to Bob because from the shared secret she was able to generate the same shared AES key as Bob and Alice.

Printing the output, we get the following:

```
Mallory Decrypted Message from Alice to Bob: Hi Bob!
```

2. Tampering with Alpha

```
for case in [1, 2, 3]:  # Test tampering with alpha = 1, q, q-1
       if case == 1:
           tampered_alpha = 1
```

```
        elif case == 2:
            tampered_alpha =  q
        elif case == 3:
            tampered_alpha =  q - 1
```

Loops through each possible value that can be used as a tampered alpha

```
# Alice's private and public keys
X_A = random.randint(1, q - 1)
Y_A = pow(tampered_alpha, X_A, q)

# Bob's private and public keys
X_B = random.randint(1, q - 1)
Y_B = pow(tampered_alpha, X_B, q)

# Shared secrets
s_A = pow(Y_B, X_A, q)
s_B = pow(Y_A, X_B, q)
```

Alice and Bobs public and private keys and shared secrets are generated normally, but the public keys are now generated with the tampered value for alpha sent from Mallory.

```
if tampered_alpha == 1:
        mallory_shared_secret = 1
elif tampered_alpha == q:
        mallory_shared_secret = 0
elif tampered_alpha == q - 1:
        mallory_shared_secret = 1 if X_A % 2 == 0 and X_B % 2 == 0 else q -
1
```

For case 1, because both public keys must be 1 (1^X mod q = 1), the shared secret s = 1 ^ X mod q = 1
For case 2, because both public keys must be 1 (q^X mod q = 0), the shared secret s = 0^X mod q = 0.
For case 3, because both public keys must be -1 ((q-1)^X mod q = -1), the shared secret s = (-1)^X mod q, which is either 1 if X is even and q-1 if X is odd.

```
# Generate keys
shared_key = hashlib.sha256(str(s_A).encode()).digest()[:16]
```

```
mallory_key =
hashlib.sha256(str(mallory_shared_secret).encode()).digest()[:16]
```

Again, Mallory can easily reproduce the same shared key because she can predict the shared secret s.

```
# Alice encrypts a message
message = "Hi Bob!"
cipher = AES.new(shared_key, AES.MODE_CBC, iv=b'1234567890123456')
ciphertext_A = cipher.encrypt(pad(message_A.encode(), AES.block_size))
```

```
# Mallory decrypts the message
cipher = AES.new(mallory_key, AES.MODE_CBC, iv=b'1234567890123456')
mallory_decrypted_message = unpad(cipher.decrypt(ciphertext_A),
AES.block_size).decode()

print(f"Case {case} (alpha = {tampered_alpha}): Mallory Decrypted
Message:", mallory_decrypted_message)
```

For each case, Mallory can easily decrypt the message to Bob because she generated the same shared AES key.

Printing the output, we get the following:

```
Case 1 (alpha = 1): Mallory Decrypted Message: Hi Bob!
Case 2 (alpha =
124325339146889384540494091085456630009856882741872806181731279018491820800119
460022367403769795008250021191767583423221479185609066059226301250167164084041
279837566626881119772675984258163062926954046545485368458404445166682380071370
274810671501916789361956272226105723317679562001235501455748016154805420913):
Mallory Decrypted Message: Hi Bob!
Case 3 (alpha =
124325339146889384540494091085456630009856882741872806181731279018491820800119
460022367403769795008250021191767583423221479185609066059226301250167164084041
279837566626881119772675984258163062926954046545485368458404445166682380071370
274810671501916789361956272226105723317679562001235501455748016154805420912):
Mallory Decrypted Message: Hi Bob!
```

Task 3:

1. Implementing RSA key gen + encryption/decryption:

```python
# RSA Key Generation
def generate_rsa_keypair(bits=2048, e=65537):
   while True:
       p = getPrime(bits // 2)
       q = getPrime(bits // 2)
       n = p * q
       phi_n = (p - 1) * (q - 1)
       if phi_n % e != 0:
           break
   d = inverse(e, phi_n)
   return (n, e), (n, d)  # Public key, Private key
```

NOTE: we must continue picking a new p and q until e is coprime to phi_n (i.e. no common factors other than 1)

```python
# RSA Encryption
def rsa_encrypt(m, public_key):
   n, e = public_key
   return pow(m, e, n)

# RSA Decryption
def rsa_decrypt(c, private_key):
   n, d = private_key
   return pow(c, d, n)
```

```python
public_key, private_key = generate_rsa_keypair()
message = "Hello, RSA!"
m_int = int(binascii.hexlify(message.encode()), 16)
print(f"Original message as integer: {m_int}")

ciphertext = rsa_encrypt(m_int, public_key)
print(f"Encrypted ciphertext: {ciphertext}")

decrypted_int = rsa_decrypt(ciphertext, private_key)
print(f"Decrypted integer: {decrypted_int}")

decrypted_message = binascii.unhexlify(hex(decrypted_int)[2:]).decode()
```

```
print(f"Decrypted message: {decrypted_message}")
```

Printing the result, we get the following:
```
Original message as integer: 87521618088895491219865889
Encrypted ciphertext:
4543242063016721498882248303861397579780373162811246195626276704104220455572473433153417813668478801655466410694068761623774302126071912657877741472592171312025359963576532813365054930853461048496155857425230274254358943393093666053862299880114575320349287573786045010102151556973217239491885445115294745468413439305927770404513324816016048453754245061641407501852883455692855314463332210072883295882406616612263831745787179792516250636447484048256809160213260838532270779553726821168376027925381745058559089598229364716852225059025581319016068997454099356108710227139117252523389343425940106258657687632840379534345
Decrypted integer: 87521618088895491219865889
Decrypted message: Hello, RSA!
```
IT WORKS!

2. RSA Malleability Attacks

```
# Generate keys
public_key, private_key = generate_rsa_keypair()
n, e = public_key

# Alice chooses a symmetric key (s) and encrypts it
s = random.randint(2, n - 1)  # Random key less than n
c = rsa_encrypt(s, public_key)  # Encrypted symmetric key
```

```
# Mallory modifies the ciphertext
c_prime = (c * pow(2, e, mod=n))  # Multiply ciphertext by 2^e mod n

# Alice decrypts c_prime
s_prime = rsa_decrypt(c_prime, private_key)  # Alice computes new "s"

# Mallory computes s from s_prime
recovered_s = s_prime // 2  # Divide the modified plaintext by 2

# Verify Mallory's attack
print("Original symmetric key (s):", s)
```

```
print("Recovered symmetric key by Mallory:", recovered_s)
print("Attack Successful:", s == recovered_s)
```

This modification of c to c’ (2^e mod n) ensures that when Alice decrypts c′, she will recover twice the original key (s’ = 2s). Of course, Mallory wouldn’t necessarily have access to s’ in a real scenario, so she would either have to get direct access to it or needs some way to reveal information about the resulting decrypted value. This is why RSA malleability attacks usually need to be combined with other vulnerabilities to be successful.

```
# Encrypt a message with AES using the original symmetric key
shared_key = hashlib.sha256(str(s).encode()).digest()[:16]
message = "Hi Bob!"
cipher = AES.new(shared_key, AES.MODE_CBC, b'1234567890123456')
ciphertext = cipher.encrypt(pad(message.encode(), AES.block_size))

# Mallory decrypts the message using the recovered symmetric key
mallory_key = hashlib.sha256(str(recovered_s).encode()).digest()[:16]
mallory_cipher = AES.new(mallory_key, AES.MODE_CBC, b'1234567890123456')
recovered_message = unpad(mallory_cipher.decrypt(ciphertext),
AES.block_size).decode()

print("Original Message:", message)
print("Recovered Message by Mallory:", recovered_message)
```

Printing the output, we get the following:

```
Original symmetric key (s):
3891952829524689348320315503325993971914583061525432464874252995120858326830590256968610148748435777549169339567856016848423297946423195326455548252596069873559186161056584584088203102105245062542849088729766863086210794820832583014974541882393655044575223096689931156371439842509226001765621349038159703597282216835189205669214491680324001485035218363227656751035137218663641845706391190754130933394251414924619611934859790128812961680672169311334648878992797161899892053688447265095060754724492963936660664406653299381249608182699206073800910159907059420838725861909422257120449493783633002437689490594481345002 23
Recovered symmetric key by Mallory:
389195282952468934832031550332599397191458306152543246487425299512085832683059
025696861014874843577754916933956785601684842329794642319532645554825259606987
355918616105658458408820310210524506254284908872976686308621079482083258301497
```

454188239365504457522309668993115637143984250922600176562134903815970359728221683518920566921449168032400148503521836322765675103513721866364184570639119075413093339425141492461961193485979012881296168067216931133464887899279716189989205368844726509506075472449296393666066440665329938124960818269920607380091015990705942083872586190942225712044949378363300243768949059448134500223
Attack Successful: True
Original Message: Hi Bob!
Recovered Message by Mallory: Hi Bob!
```

Thus, Mallory was successfully able to recover the shared key s and it can be used to decrypt all messages that use that key.

## RSA Signature Malleability

```
# Generate keys
public_key, private_key = generate_rsa_keypair()
n, d = private_key
e = public_key[1]
```

```
# Mallory sees signatures for two messages
m1 = int.from_bytes("Hello".encode(), 'big')
m2 = int.from_bytes("World".encode(), 'big')
sig1 = rsa_encrypt(m1, private_key)  # Signature for m1
sig2 = rsa_encrypt(m2, private_key)  # Signature for m2
```

```
# Mallory creates a signature for m3 = m1 * m2
m3 = (m1 * m2) % n
sig3 = (sig1 * sig2) % n  # Signature for m3
```

Because sign(m,d) = m^d mod n, using sig3 = sig1 * sig2 mod n = sig3 (m1*m2)^d mod n and thus sig3 is actually a signature for m3 = (m1*m2) mod n

```
# Verify Mallory's attack
verified_m3 = rsa_decrypt(sig3, public_key)
print("Message m3 (as integer):", m3)
print("Recovered m3 from signature:", verified_m3)
print("Attack Successful:", m3 == verified_m3)
```

Printing the output, we get the following:
```
Message m3 (as integer): 116767614895467081969500

Recovered m3 from signature: 1167676148954670819695OO
Attack Successful: True
```

Thus, the predicted m3 matches what the signature produces when decrypted.

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?**
   It would be very hard for an adversary to solve the DHP with the given parameters because q is a 1024-bit prime, making brute force practically impossible. The adversary would need to compute the private key X_A or X_B from Y_A=α^(X_A) mod q, which requires solving the DLP. Man in the middle could be a potential strategy, but that is not solving the DHP algorithm.

2. **For task 1, would the same strategy used for the tiny parameters work for the large values of q and ? Why or why not?**
   No, because for smaller values like q = 37 there are only 36 possible values for X_A or X_B so it would be very easy to brute force this. However, that is not the case when q is a large 1024-bit prime.

3. **For task 2, why were these attacks possible? What is necessary to prevent it?**
   These MITM attacks are possible because DH alone does not authenticate the communicating parties or validate exchange values, which allows and attacker to intercept values undetected. Preventing them requires an authenticated key exchange, certificate-based validation, and checking DH parameters for malicious values like used in the task (q, alpha, and Y).

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.**
   Two people using the same RSA modulus n is very bad because once you know p and q (from d), you can factorize n and compute the private key for any other user using that same n. Thus, all their messages can be decrypted without user's private key ever being directly leaked. It can also lead to digital signature forgery by mixing the valid signatures from one user into another's messages, similar to the RSA signature malleability attack.
```