Part B:

1- a- this is the code of the program in python:

```python
# Mapping characters to their respective numbers
charstr = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
chars = list(charstr)
nums = [str(i) for i in range(0, 26)]

# Function to convert character to its corresponding number
def char_to_num(char):
    return chars.index(char)

# Function to convert number to its corresponding character
def num_to_char(num):
    return chars[num]

# Encryption function
def encrypt(plaintext, key):
    # Ensure the key length is at least as long as the plaintext
    if len(key) < len(plaintext):
        raise ValueError("Key length must be greater than or equal to plaintext length")

    # Perform modular addition
    cipher_text = ''
    for i in range(len(plaintext)):
        plain_num = char_to_num(plaintext[i])
        key_num = char_to_num(key[i])
        cipher_num = (plain_num + key_num) % 26
        cipher_text += num_to_char(cipher_num)

    return cipher_text
```

```python
# Taking input from the user
plaintext = input("Enter the plaintext (only capital letters): ")
key = "THISISANEXAMPLEKEYINCOMPUTERSECURITYEXAM"

cipher = encrypt(plaintext, key)
print(f"Plaintext: {plaintext}")
print(f"Key: {key}")
print(f"Cipher: {cipher}")
```

When I execute it, it asks for plaintext input:

```
Enter the plaintext (only capital letters): NANCY
```

Then when I enter the input it gives this output:

```
Enter the plaintext (only capital letters): NANCY
Plaintext: NANCY
Key: THISISANEXAMPLEKEYINCOMPUTERSECURITYEXAM
Cipher: GHVUG
```

b- this is the code of the program in python, I have used some of the functions I already defined in 1- a:

```python
# Decryption function
def decrypt(cipher_text, key):
    # Ensure the key length is at least as long as the ciphertext
    if len(key) < len(cipher_text):
        raise ValueError("Key length must be greater than or equal to ciphertext

    # Perform modular subtraction
    plaintext = ''
    for i in range(len(cipher_text)):
        cipher_num = char_to_num(cipher_text[i])
        key_num = char_to_num(key[i])
        plain_num = (cipher_num - key_num) % 26
        plaintext += num_to_char(plain_num)

    return plaintext
```

```python
# Decrypt the cipher text
cipher = input("Enter the cipher text (only capital letters): ")
decrypted_text = decrypt(cipher, key)
print(f"Decrypted text: {decrypted_text}")

# Verify the decryption
if decrypted_text == plaintext:
    print("Decryption successful: The decrypted text matches the original plaintext.")
else:
    print("Decryption failed: The decrypted text does not match the original plaintext.")
```

when I run the code it asks for the cipher text input which in this case I chose it to be the cipher text that was the output from the previous program in 1-a to be able to test the results later:

```
Enter the cipher text (only capital letters):
GHVUG
```

This is the output of the decryption and the testing with the previous program:

```
Enter the cipher text (only capital letters): GHVUG
Decrypted text: NANCY
Decryption successful: The decrypted text matches the original plaintext.
```

c- The first condition is if there is poor key management, a cipher can be broken even with a robust encryption method if the encryption keys are handled carelessly. This includes recycling keys between sessions, utilizing weak or simple-to-guess keys, and improperly safeguarding keys. Example: If users select simple, readily guessed passwords (like "password123"), attackers can swiftly decrypt the data using dictionary attacks. One common reason for Wi-Fi network compromises is the use of weak passwords for WPA/WPA2 encryption by users. The second

condition could be known-plaintext and chosen-plaintext attacks as these attacks happen when an attacker can choose which plaintexts to encrypt and subsequently obtain the matching ciphertexts (chosen-plaintext attack) or when the attacker has access to both the plaintext and its corresponding ciphertext (known-plaintext attack). An attacker can figure out the encryption process's key or other secrets if they have enough instances. Example: Known-plaintext attacks can be used to readily break older ciphers, such as the Caesar encryption or basic substitution ciphers. An attacker may typically determine the shift used in a Caesar cipher, for example, if they obtain the associated ciphertext and know that the plaintext contains a common phrase like "THE".

d- Firstly, key length as the number of combinations that can be made depends on the length of the key, making brute-force attacks exponentially more challenging. Greater security is achieved by longer keys because guessing the key requires more computing power. Secondly, key management as key compromise is avoided by the use of appropriate key management procedures, such as frequent rotation, safe storage, and distribution. By ensuring that keys are protected at every stage of their lifecycle, effective management lowers the possibility of unwanted access. Lastly, key confidentiality as preventing unwanted decryption requires keeping the key secret. Strict access controls and secure routes for key exchange guarantee that the key is confidential and accessible only to those who are permitted.

2- a- DES and other encryption algorithms usually work with fixed-size data blocks. Each block in DES is 64 bits (8 bytes). Pad is required to fill the empty space if the plaintext does not precisely fit this block size. Block size limitations are avoided by padding, which guarantees that any plaintext messages regardless of their initial length can be encrypted and decoded without any problems. Without padding, plaintext that does not fit inside the block size would be rejected or changed, which might damage the message's integrity. Example: Assume that the message "HELLO" needs to be encrypted using DES, which has a block size of 64 bits (8 bytes). "HELLO" is represented in ASCII as follows: ‘H’ = 72, ‘E’ = 69, ‘L’ = 76, ‘O’= 79, in binary these ASCII values are: 'H' = 01001000, 'E' = 01000101, 'L' = 01001100, 'L' = 01001100, 'O' = 01001111. We obtain a total of 40 bits (5 bytes) by concatenating these binary representations, which is fewer than the 64 bits needed for DES block size. In this case we could for example use zero padding process which is adding zeros at the end of the message to fill up the remanding space in the block so the message would be like this 01001000 01000101 01001100 01001100 01001111 00000000 00000000 00000000. The message is padded with three bytes (24 bits) of zeros to reach the 64-bit block size required by DES.

b- assuming that my name is nancy, we would take the first three letter so the message would be “NANComputerSecurity”. Then convert the message to ASCII: N = 78, A = 65, N = 78, C = 67, o = 111, m = 109, p = 112, u = 117, t = 116, e = 101, r = 114, S = 83, e = 101, c = 99, u = 117, r = 114, i = 105, t = 116, y = 12 . then turn the ASCII into binary and combine them: 01001110 01000001 01001110 01000011 01101111 01101101 01110000 01110101 01110100 01100101 01110010 01010011 01100101 01100011 01110101 01110010 01101001 01110100 01111001. It has total length 152 bits (19 bytes), which is less than the required 64-bit block size for DES. Then do the zero padding process which is adding zeros (0s) at the end of the message to fill up the remaining space in the block, padding to reach the 64-bit block size: 01001110 01000001 01001110 01000011 01101111 01101101 01110000 01110101 01110100 01100101 01110010 01010011 01100101 01100011 01110101 01110010 01101001 01110100 01111001 00000000 00000000 00000000

00000000. Lastly, converting it to hexadecimal: 4E 41 4E 43 6F 6D 70 75 74 65 72 53 65 63 75 72 69 74 79 00 00 00 00 00.

c- First we will need to convert the string to bytes this could be done by using ASCII representation for the plaintext: C = 67, o = 111, m = 109, p = 112, u = 117, t = 116, e = 101, r = 114, S = 83, e = 101, c = 99, u = 117, r = 114, i = 105, t = 116, y = 121. Then combine bytes into a large number, the byte representation is 67 111 109 112 117 116 101 114 83 101 99 117 114 105 116 121, convert the byte representation into binary 01000011 01101111 01101101 01110000 01110101 01110100 01100101 01110010 01010011 01100101 01100011 01110101 01110010 01101001 01110100 01111001 and lastly, convert binary into a single large integer by calculating this 67 * 256^15 + 111 * 256^14 + 109 * 256^13 + 112 * 256^12 + 117 * 256^11 + 116 * 256^10 + 101 * 256^9 + 114 * 256^8 + 83 * 256^7 + 101 * 256^6 + 99 * 256^5 + 117 * 256^4 + 114 * 256^3 + 105 * 256^2 + 116 * 256^1 + 121 * 256^0 which would result a really huge number which is 8.963684036*10^37. Converting the text into a padded message using PKSC#1, Assuming a simplified padding scheme: Start with a fixed header '00 || 02', add padding bytes until the total length of the block matches the block size, append a terminating byte '00' followed by the ASCII values of the plaintext. For our example it would be Padded message: 00 || 02 || PS || 00 || 67 || 111 || 109 || ... || 105 || 116 || 121.

d- A digital signature is a cryptographic method used to guarantee the integrity and validity of digital documents or messages. It offers a means of confirming that the message was sent by the person who claimed to be the sender and that it hasn't been altered since it was signed. It's the digital equivalent of a handwritten signature or stamped seal, but with significantly greater inherent security. A digital signature is intended to address the problem of tampering and impersonation in digital communications. Digital signatures can be used to verify the origin, identity, and status of electronic documents, transactions, and messages. Signers might also use them to confirm that they have provided informed consent. RSA can be used for digital signatures as it could generate a private key for signing and a corresponding public key for verification, sign a message or document by hashing the message to generate a fixed-size message, encrypting the hash value using the sender's private key, the encrypted hash value serves as the digital signature and verify the signature by decrypting the received digital signature using the sender's public key, which yields the original hash value, computing the hash of the received message and comparing the computed hash with the decrypted hash value. If they match, the signature is valid and the message hasn't been altered. DES is not typically used for digital signatures because DES requires both parties to have the same secret key. For digital signatures, a sender needs a private key and a receiver needs the corresponding public key, it produces fixed-size outputs (64 bits), which are not suitable for generating secure digital signatures. Digital signatures typically require larger hash values and signatures to ensure security against attacks and it is considered relatively insecure by modern standards due to its small key size (56 bits), making it vulnerable to brute-force attacks. Digital signatures require stronger security guarantees, which DES cannot adequately provide.