TABLE OF CONTENT

	data integrity, authenticity, and non-repudiation.		
7.	To understand password auditing and cracking techniques using hashing algorithms and dictionary-based attacks, demonstrating the vulnerabilities of weak passwords.		
8.	To understand and simulate the basics of intrusion detection, learning how network-based intrusion detection systems (NIDS) detect suspicious activity and help protect systems from unauthorized access.		
9.	To understand and implement steganography by hiding a secret message within an image file, demonstrating how data can be concealed within multimedia content for secure communication.		
10.	The aim of this experiment is to implement and understand the process of text encryption using popular cryptographic algorithms such as Caesar Cipher, RSA, and AES. This will help in understanding how cryptographic algorithms are applied in securing text data.		

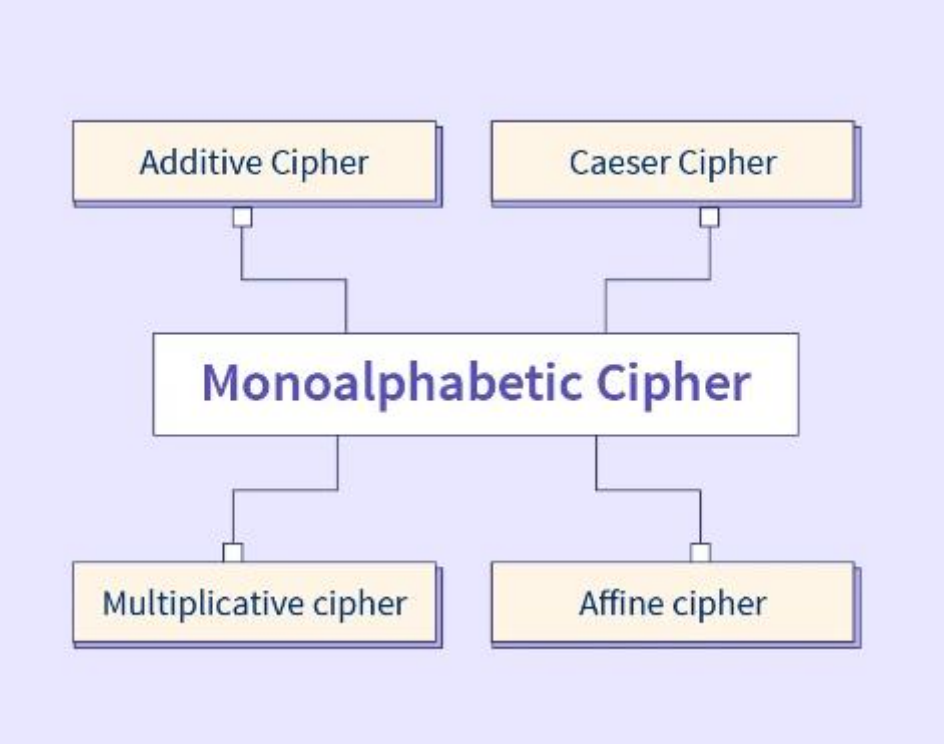
EXPERIMENT- 1

Aim

To understand the principles of monoalphabetic ciphers through the Caesar Cipher and perform cryptanalysis using frequency analysis.

Theory

1. Monoalphabetic Cipher: A monoalphabetic cipher is a substitution cipher where each letter in the plaintext is replaced with another letter from the same alphabet based on a fixed relationship. This relationship does not change throughout the message, making it more susceptible to frequency analysis compared to polyalphabetic ciphers.



2. Caesar Cipher: The Caesar Cipher, named after Julius Caesar, is a type of monoalphabetic cipher. It shifts each letter in the plaintext by a fixed number of positions down or up the alphabet. For instance, a shift of 3 would replace 'A' with 'D', 'B' with 'E', and so on.

- **Encryption formula:**
  $E(x) = (x + k) \mod 26$
  where x is the letter position, and k is the shift key.
- **Decryption formula:**
  $D(y) = (y - k) \mod 26$
  where y is the encoded letter position.

3. Cryptanalysis Using Frequency Analysis: Since English letters have typical frequencies (e.g., 'E' is the most common letter, followed by 'T' and 'A'), frequency analysis exploits these patterns to break ciphers. For example, if the most frequent letter in the cipher text corresponds

to 'E' or 'T', this can indicate the shift value, allowing the text to be decrypted without knowing the key.

Code Used :

```
# Caesar Cipher Encryption, Decryption, and Cryptanalysis
def encrypt_caesar(plaintext, shift):
    ciphertext = ""
    for char in plaintext:
        if char.isalpha():
            shift_start = ord('A') if char.isupper() else ord('a')
            ciphertext += chr((ord(char) - shift_start + shift) % 26 + shift_start)
        else:
            ciphertext += char
    return ciphertext
def decrypt_caesar(ciphertext, shift):
    plaintext = ""
    for char in ciphertext:
        if char.isalpha():
            shift_start = ord('A') if char.isupper() else ord('a')
            plaintext += chr((ord(char) - shift_start - shift) % 26 + shift_start)
        else:
            plaintext += char
    return plaintext


def frequency_analysis(ciphertext):
    # Calculate frequency of each letter in the ciphertext
    freq = {}
    for char in ciphertext:
        if char.isalpha():
            if char in freq:
                freq[char] += 1
            else:
```

```
            freq[char] = 1
    # Sort letters by frequency in descending order
    sorted_freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)
    return sorted_freq
# Example run
plaintext = "HELLO WORLD"
shift_key = 3
ciphertext = encrypt_caesar(plaintext, shift_key)
print("Encrypted text:", ciphertext)
print("Decrypted text:", decrypt_caesar(ciphertext, shift_key))
print("Frequency Analysis of ciphertext:", frequency_analysis(ciphertext))
```

Output :

```
Encrypted text: KHOOR ZRUOG
Decrypted text: HELLO WORLD
Frequency Analysis of ciphertext: [('O', 3), ('R', 2), ('K', 1), ('H', 1),
    ('Z', 1), ('U', 1), ('G', 1)]

=== Code Execution Successful ===
```

Result

1. **Encryption**: The plaintext was successfully encrypted using the Caesar Cipher with a shift key of 3. For example, "HELLO WORLD" became "KHOOR ZRUOG".
2. **Decryption**: Using the same shift key, the ciphertext was decrypted back to the original message.
3. **Frequency Analysis**: The frequency analysis results showed the most common letters, helping identify possible shifts. By comparing the ciphertext letter frequencies to common English letter frequencies, we could deduce the shift key, thereby performing successful cryptanalysis.

EXPERIMENT-2

Aim

To understand the basics of OpenSSL, its cryptographic capabilities, and an introduction to the Federal Information Processing Standards (FIPS) as they relate to cryptographic security.

Theory

1. OpenSSL: OpenSSL is an open-source toolkit for implementing the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols. It provides robust cryptographic libraries that enable secure data transmission and storage. OpenSSL supports encryption, decryption, digital signatures, and certificate management and is widely used in web servers, applications, and network protocols.

Features of OpenSSL:

- **Encryption/Decryption**: Provides symmetric and asymmetric encryption algorithms.
- **Message Digests**: Supports hashing algorithms like SHA-256, MD5, etc.
- **Digital Signatures**: Verifies data integrity and authenticity.
- **Certificates**: Manages SSL/TLS certificates, allowing secure communication.

Basic OpenSSL Commands:

- openssl genpkey: Generate private keys.
- openssl rsautl: Encrypt and decrypt messages using RSA.
- openssl enc: Encrypt and decrypt data with symmetric algorithms (e.g., AES).
- openssl dgst: Generate hash values for files/messages.

2. Federal Information Processing Standards (FIPS): FIPS are standards and guidelines developed by the National Institute of Standards and Technology (NIST) in the United States to ensure high security and interoperability for federal systems. FIPS compliance is required in certain government and regulated industries to ensure data protection, cryptographic strength, and reliable implementation.

- **Key FIPS Standards in Cryptography**:
    - **FIPS 140-2**: Specifies security requirements for cryptographic modules, including approved algorithms, cryptographic keys, and data handling.
    - **FIPS 197**: Defines the Advanced Encryption Standard (AES), a symmetric encryption algorithm widely used for securing sensitive data.
    - **FIPS 180-4**: Specifies the Secure Hash Standard (SHS) and includes SHA-1, SHA-256, SHA-384, and SHA-512 hashing algorithms.

3. OpenSSL and FIPS Compliance: OpenSSL has a FIPS-compliant module to ensure the cryptographic implementation adheres to FIPS standards. This is important for organizations

Program (OpenSSL Commands)

Here are some basic OpenSSL commands for generating keys, encrypting/decrypting messages, and generating hash values. This experiment will give hands-on exposure to basic cryptographic operations.

1. **Generate an RSA Private Key:**

```
bash

openssl genpkey -algorithm RSA -out private_key.pem -aes256
```

2. **Encrypt a Message Using the RSA Key:**

```
bash

openssl rsautl -encrypt -inkey public_key.pem -pubin -in message.txt -out
```

3. **Decrypt a Message Using the RSA Key:**

```
bash

openssl rsautl -decrypt -inkey private_key.pem -in encrypted_message.bin
```

4. **Generate a SHA-256 Hash of a File:**

```
bash

openssl dgst -sha256 filename.txt
```

5. **Checking FIPS Mode in OpenSSL (If Supported):**

```
bash

openssl version -fips
```

These commands cover basic key generation, encryption, decryption, and hashing, demonstrating OpenSSL's core cryptographic capabilities.

Result

1. Successfully generated RSA keys using OpenSSL.
2. Successfully encrypted and decrypted messages using the RSA encryption commands.

3. Generated a SHA-256 hash for a given file to verify data integrity.
4. Checked FIPS mode in OpenSSL, demonstrating whether the environment supports FIPS-compliant cryptographic operations.

EXPERIMENT- 3

Aim

To understand symmetric key cryptography by generating pseudo-random numbers, creating a DES key, performing encryption and decryption using DES, and verifying file integrity using an MD5 hash.

Theory

1. Symmetric Key Cryptography: Symmetric key cryptography uses a single secret key for both encryption and decryption. This method is fast and efficient, suitable for securing large volumes of data, and is widely used in applications requiring data privacy and security.

2. Pseudo-Random Number Generation (PRNG): Pseudo-random number generators are algorithms that generate a sequence of numbers approximating the properties of random numbers. PRNGs are crucial for cryptographic key generation to ensure unpredictability. In OpenSSL, PRNG can be used to generate random keys or initialization vectors (IVs) for encryption.

3. Data Encryption Standard (DES): DES is a symmetric key algorithm that encrypts data in 64-bit blocks using a 56-bit key. Although DES is now considered less secure than modern algorithms like AES, it remains a foundational example of symmetric encryption.

- **DES Encryption/Decryption:**
    - **Encryption**: The DES algorithm scrambles the plaintext using a 56-bit key to produce ciphertext.
    - **Decryption**: The DES algorithm reverses the process, using the same key to transform the ciphertext back into plaintext.

5. **File Integrity with MD5 Hash:** The MD5 algorithm generates a 128-bit hash (or message digest) from input data. It is commonly used to verify file integrity by ensuring that the hash of the original file matches the hash of the received file. Any modification in the file would result in a different hash, signaling potential corruption or tampering.

Program (OpenSSL Commands)

Here are the OpenSSL commands for generating pseudo-random numbers, creating DES keys, performing DES encryption and decryption, and generating MD5 hashes for file integrity.

1. **Generate a Pseudo-Random DES Key:**

```
bash

openssl rand -hex 8 > des_key.key
```

   This command generates a pseudo-random 64-bit (8-byte) key suitable for DES.

2. **Encrypt a File Using DES:**

```bash
openssl enc -des -in plaintext.txt -out encrypted_des.bin -K $(cat des_ke
```

plaintext.txt is the file to be encrypted, and encrypted_des.bin is the output encrypted file. We use the generated DES key and an IV (initialization vector) of 0 for simplicity.

3. **Decrypt a File Using DES:**

```bash
openssl enc -des -d -in encrypted_des.bin -out decrypted.txt -K $(cat des
```

This command decrypts encrypted_des.bin back to its original form, saving it as decrypted.txt.

4.**Generate an MD5 Hash for File Integrity:**

```bash
openssl dgst -md5 plaintext.txt
```

This command generates an MD5 hash of plaintext.txt. The hash can later be compared to verify that the file remains unchanged.

Result

1. Successfully generated a pseudo-random DES key using OpenSSL's random number generator.
2. Successfully encrypted and decrypted a file using DES, demonstrating the concept of symmetric key cryptography.
3. Generated an MD5 hash of the original file, allowing for file integrity verification by comparing hash values before and after transmission or storage.

EXPERIMENT – 4

Aim

To understand the process of symmetric key distribution over a network using NetCat and Apache, and to analyse potential vulnerabilities and key compromise using the network protocol analyser, Wireshark.

Theory

1. Symmetric Key Distribution: In symmetric cryptography, both sender and receiver must share a secret key for encryption and decryption. Securely distributing this key over a network is critical because interception during transmission can lead to unauthorized access. Secure methods of key distribution often involve encryption and/or secure protocols.

2. NetCat: NetCat (often called the "Swiss Army knife" of networking) is a simple tool that reads and writes data across network connections using TCP or UDP. It's commonly used for network debugging and monitoring and can establish raw network connections for data transmission.

3. Apache Web Server: The Apache HTTP server is a widely-used web server software. In this experiment, we use Apache to simulate a web server environment for secure file sharing, including shared keys.

4. Wireshark: Wireshark is a protocol analyzer used to capture and analyze network traffic. It allows examination of data packets to detect potential vulnerabilities, including the possibility of key compromise if sensitive information, like symmetric keys, is sent without encryption.

Procedure

1. **Symmetric Key Distribution Using NetCat:**

```
bash

nc -l -p 12345 > received_key.key
```

This command opens NetCat on port 12345 and saves incoming data to received_key.key.

```
bash

nc <receiver_IP> 12345 < des_key.key
```

This command sends the contents of des_key.key (symmetric key) to the specified IP address and port, initiating the key transfer.

2. **Symmetric Key Distribution Using Apache:**

Step 1: Place the key file (des_key.key) in a secured directory on the Apache server with restricted permissions.

Step 2: Configure Apache to enable HTTPS (to encrypt the connection and protect the key during transfer).

Step 3: The client can download the key file securely over HTTPS using:

```
bash

wget https://<server_IP>/path/to/des_key.key
```

This ensures the key is transmitted over an encrypted channel

3. **Key Compromise Analysis Using Wireshark:**

Step 1: Start a Wireshark capture on the network interface used for communication.

Step 2: Monitor the traffic between the sender and receiver while the key is transmitted via NetCat (without encryption).

Step 3: Examine the captured packets in Wireshark to see if the symmetric key appears in plaintext. This highlights the vulnerability of unencrypted channels.

Step 4: Repeat the key transfer over HTTPS (with Apache) and capture the packets. Verify that the key is not visible in plaintext, showing how HTTPS secures the key exchange.

Result

1. Successfully demonstrated symmetric key distribution using NetCat and verified that key distribution over an unencrypted NetCat connection can lead to key compromise.
2. Successfully transferred the symmetric key over a secured HTTPS connection with Apache, demonstrating that HTTPS can protect against interception.
3. Wireshark analysis showed that keys transferred over NetCat without encryption are visible in plaintext, confirming the vulnerability of unencrypted channels. However, keys transferred over HTTPS were secure, and Wireshark did not reveal the key content, showcasing the importance of encrypted communication channels.

EXPERIMENT – 5

Aim

To understand and implement Message Authentication Codes (MAC) to verify data integrity and authenticity, ensuring that the message is unaltered and from a legitimate sender.

Theory

1. Message Authentication Code (MAC): A Message Authentication Code (MAC) is a cryptographic checksum that is generated by applying a cryptographic algorithm to a message along with a secret key. The primary purpose of a MAC is to verify the integrity and authenticity of a message, ensuring that it has not been tampered with during transmission.

- **How MAC Works:**
    - A sender generates a MAC value by applying a cryptographic algorithm (e.g., HMAC with SHA-256) on the message and the shared secret key.
    - The receiver, who has the same secret key, generates a MAC on the received message and compares it with the received MAC. If they match, the message is authentic and untampered.
- **Types of MACs:**
    - **CBC-MAC**: Uses block cipher modes like CBC (Cipher Block Chaining).
    - **HMAC (Hash-Based MAC)**: Uses a cryptographic hash function (e.g., SHA-256) along with a key.

2. HMAC (Hash-Based Message Authentication Code): HMAC is a type of MAC that combines a cryptographic hash function with a secret key to produce a fixed-size output, usually more secure than a hash alone. HMAC-SHA256, for example, is widely used for message authentication in various protocols.

Code Used :

```
import hmac
import hashlib

# Function to generate HMAC
def generate_hmac(key, message):
    # Create HMAC object with the key and SHA-256 hash function
    mac = hmac.new(key.encode(), message.encode(), hashlib.sha256)
    return mac.hexdigest()
```

```
# Function to verify HMAC
def verify_hmac(key, message, received_mac):
    # Generate MAC for comparison
    mac = hmac.new(key.encode(), message.encode(), hashlib.sha256)
    return hmac.compare_digest(mac.hexdigest(), received_mac)

# Example usage
key = "supersecretkey"
message = "This is a confidential message."

# Generate HMAC for the message
mac = generate_hmac(key, message)
print("Generated MAC:", mac)

# Verify HMAC by simulating receiver's process
is_valid = verify_hmac(key, message, mac)
print("Is the message valid?", is_valid)
```

Output :

```
Generated MAC: e9067bc217dcd5053a320d364016eec812aad4991d845a04883fb2e1c67c8e6
    5
Is the message valid? True

=== Code Execution Successful ===
```

Explanation:

generate_hmac function creates an HMAC using SHA-256 for a given message and key.

verify_hmac function compares a received MAC with the MAC generated from the received message to check its validity.

Result

1. Successfully generated an HMAC for the message using a secret key, ensuring data integrity and authenticity.
2. Verified the HMAC on the receiver's end, confirming that the message remained unchanged and was from an authenticated sender.

EXPERIMENT – 6

Aim

To understand the concept of digital signatures and implement the generation of a digital signature for a message, ensuring data integrity, authenticity, and non-repudiation.

Theory

1. Digital Signature: A digital signature is a cryptographic mechanism used to verify the authenticity and integrity of digital messages or documents. It uses asymmetric cryptography (public and private keys) to generate a unique digital code that binds the signer's identity to the message.

- **How Digital Signatures Work:**
    - The sender uses their private key to generate a signature on the message, creating a unique hash of the message that is encrypted with the private key.
    - The receiver uses the sender's public key to decrypt the signature and verify it by comparing the decrypted hash with the hash generated from the received message.
- **Properties of Digital Signatures:**
    - **Authenticity**: Confirms the identity of the sender.
    - **Integrity**: Ensures the message has not been altered.
    - **Non-Repudiation**: Prevents the sender from denying their signed message.

3. **Digital Signature Algorithm (DSA) and RSA:** Digital signatures can be implemented using various algorithms, such as the Digital Signature Algorithm (DSA) or RSA. In this experiment, we will use RSA for demonstration, as it is widely supported and commonly used in secure communications.

Code Used :

```
In [3]: pip install cryptography

Requirement already satisfied: cryptography in c:\users\dell\anaconda3\lib\site-packages (41.0.3)
Requirement already satisfied: cffi>=1.12 in c:\users\dell\anaconda3\lib\site-packages (from cryptography) (1.15.1)
Requirement already satisfied: pycparser in c:\users\dell\anaconda3\lib\site-packages (from cffi>=1.12->cryptography) (2.21)
Note: you may need to restart the kernel to use updated packages.
```

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import serialization

# Step 1: Generate RSA Private and Public Keys
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048
)
public_key = private_key.public_key()

# Step 2: Save the Private Key in PEM Format (optional)
private_pem = private_key.private_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PrivateFormat.PKCS8,
    encryption_algorithm=serialization.NoEncryption()
)
with open("private_key.pem", "wb") as key_file:
    key_file.write(private_pem)

# Step 3: Save the Public Key in PEM Format (optional)
public_pem = public_key.public_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PublicFormat.SubjectPublicKeyInfo
)
with open("public_key.pem", "wb") as key_file:
    key_file.write(public_pem)

# Step 4: Message to Sign
message = b"This is a secure message."
```

```
# Step 5: Generate the Digital Signature
signature = private_key.sign(
    message,
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
)

print("Digital Signature (in hex):", signature.hex())

# Step 6: Verify the Digital Signature
try:
    public_key.verify(
        signature,
        message,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    print("Signature is valid. Message authenticity and integrity verified.")
except Exception as e:
    print("Signature verification failed:", e)
```

Output :

```
Digital Signature (in hex): 6e89db7ed35d3cd4533facab578adc3e1ab311bbfc6980c535ea4696559f4fa693269b1d72ea14d1c55d8296ce00185c9d8490b22c8df26
acbed392fb59e97cd821201f94559f720ad5a371feffa5a2dda9c225f74946c87fa5bfc3240039d16e573dd143382eca9f706f3ed88880ad6ff7ec26617bcdae9dfad7cf29f
2dc0a1e4a1650a726f29618f95174a75147dc5198f1e6c26549f3dfc7d2302136bc6c4a870dca8920671b82eb5fc09f619a892046d7165b920d88d2f58654598e742a87f870
6e87a619ec416add0f5060477455ff361e4dac7a7f195adf17ef1ab4dc419252e2862b2f7d79c4baf46c803ecdeff58c8a8642abac965498e265757efd4
Signature is valid. Message authenticity and integrity verified.
```

Explanation:

- The program generates RSA public and private keys.
- The private key is used to create a digital signature for the message, while the public key verifies the signature.
- The signature is created using the PSS padding scheme and SHA-256 hashing.
- If the verification succeeds, it confirms that the message is authentic and has not been altered.

Result

1. Successfully generated RSA keys and created a digital signature for the message using the private key.
2. Verified the digital signature using the public key, confirming the authenticity and integrity of the signed message.

EXPRIMENT – 7

Aim

To understand password auditing and cracking techniques using hashing algorithms and dictionary-based attacks, demonstrating the vulnerabilities of weak passwords.

Theory

1. Password Auditing and Cracking: Password auditing involves testing passwords for weaknesses to identify those that could easily be compromised. Password cracking is the process of recovering passwords from data that has been stored in hashed or encrypted form, often using various techniques like brute force, dictionary attacks, and rainbow tables.

2. Hashing Algorithms: A password is typically stored in a hashed form for security. Hashing algorithms, such as SHA-256 or MD5, convert passwords into fixed-size hash values. A hash cannot easily be reversed to retrieve the original password; however, attackers can use precomputed tables or dictionary attacks to crack weak passwords.

3. Dictionary Attack: A dictionary attack uses a list of common passwords (dictionary) to attempt cracking hashed passwords. This method is faster than brute force, especially effective against weak or common passwords.

4. Common Tools for Password Auditing: Tools such as John the Ripper, Hashcat, and Hydra are commonly used for password auditing and cracking. In this experiment, we'll use Python to simulate a basic dictionary attack on a hashed password.

Code Used :

```
import hashlib

# Simulated hashed password (e.g., hash of "password123")

hashed_password = hashlib.sha256("password123".encode()).hexdigest()


# Dictionary of possible passwords

dictionary = ["password", "123456", "password123", "qwerty", "letmein"]


# Function to perform dictionary attack

def dictionary_attack(hashed_password, dictionary):

    for password in dictionary:

        # Hash each password in the dictionary and compare

        password_hash = hashlib.sha256(password.encode()).hexdigest()

        print(f"Trying password: {password} -> Hash: {password_hash}")
```

```
        if password_hash == hashed_password:
            print("Password found:", password)
            return password
    print("Password not found in dictionary.")
    return None

# Running the dictionary attack
cracked_password = dictionary_attack(hashed_password, dictionary)
if cracked_password:
    print("Cracked Password:", cracked_password)
else:
    print("Failed to crack the password.")
```

Output :

```
Trying password: password -> Hash: 5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
Trying password: 123456 -> Hash: 8d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca12020c923adc6c92
Trying password: password123 -> Hash: ef92b778bafe771e89245b89ecbc08a44a4e166c06659911881f383d4473e94f
Password found: password123
Cracked Password: password123
```

Explanation:

- The program stores a hashed version of a sample password (password123 hashed with SHA-256).
- A dictionary list contains common weak passwords to simulate a dictionary attack.
- For each password in the dictionary, it hashes the password and checks if it matches the target hash.
- If a match is found, the password is cracked; if not, the program concludes the password is not in the dictionary.

Result

1. Successfully simulated a dictionary attack to attempt cracking a hashed password.
2. Demonstrated the effectiveness of a dictionary attack for identifying weak passwords in a limited dictionary.
3. Highlighted the importance of using complex and unique passwords to prevent vulnerability to such attacks.

EXPERIMENT -8

Aim

To understand and simulate the basics of intrusion detection, learning how network-based intrusion detection systems (NIDS) detect suspicious activity and help protect systems from unauthorized access.

Theory

1. Intrusion Detection System (IDS): An Intrusion Detection System (IDS) monitors network or system activities for malicious actions or policy violations. An IDS can alert administrators about potential intrusions but does not block or prevent attacks.

- **Types of IDS**:
    - **Network-based IDS (NIDS)**: Monitors and analyzes network traffic to detect malicious activity. Examples include Snort and Suricata.
    - **Host-based IDS (HIDS)**: Monitors and analyzes the internal system operations of a single host.

2. IDS Techniques: Intrusion detection can be done using:

- **Signature-based Detection**: Matches patterns of known attacks against network data.
- **Anomaly-based Detection**: Detects unusual patterns that deviate from normal behavior.

3. Snort as an IDS Tool: Snort is an open-source NIDS that uses a set of predefined rules to detect threats. It can capture network packets and analyze them in real-time for specific patterns indicative of intrusions.

4. Role of IDS in Cybersecurity: IDS helps to identify and respond to threats in real time, reducing the risk of successful attacks. However, it is crucial to keep IDS rules updated to ensure accurate detection of emerging threats.

Procedure

1. Installing Snort (if not already installed):

- On a Linux system, install Snort using the following command:

```
bash

sudo apt-get install snort
```

- Configure Snort by editing the configuration file (typically located at /etc/snort/snort.conf).

2. Running Snort for Intrusion Detection:

- Run Snort in network intrusion detection mode to monitor real-time traffic:

```bash
sudo snort -A console -q -c /etc/snort/snort.conf -i eth0
```

- The -A console option outputs alerts to the console, -q runs in quiet mode, -c specifies the configuration file, and -i specifies the network interface (replace eth0 with your active network interface).

3. Simulating an Intrusion for Testing:

- To test Snort, you can simulate network traffic that triggers predefined Snort rules. For example:

```bash
ping -c 4 www.example.com
```

- Snort rules can be added or modified to detect specific traffic patterns, such as ICMP (ping) requests or suspicious TCP packets.

4. Analysing Snort Alerts:

- Review Snort's alert messages in the console to see if any activity has been flagged as suspicious or malicious.
- Check Snort's log file (usually located in /var/log/snort/alert) to review details of each detected intrusion.

Result

1. Successfully configured and ran Snort to monitor network traffic for potential intrusions.
2. Simulated network activity to test Snort's detection capabilities.
3. Analysed alerts and logs generated by Snort to identify suspicious network activity, demonstrating the use of a network-based intrusion detection system (NIDS) in detecting potential intrusions.

EXPERIMENT 9

Aim

To understand and implement steganography by hiding a secret message within an image file, demonstrating how data can be concealed within multimedia content for secure communication.

Theory

1. Steganography: Steganography is the technique of hiding information within other non-secret data, such as images, audio files, or video files, in a way that conceals the presence of the hidden message. Unlike encryption, where the message's content is transformed, steganography hides the message so that it appears as part of the file.

- **Types of Steganography**:
    - **Image Steganography**: Hides information within an image by altering its pixel values.
    - **Audio Steganography**: Embeds hidden data in audio files, often by modifying audio waveforms or using frequency modulation.
    - **Video Steganography**: Uses video files as cover media, embedding data in video frames.

2. Least Significant Bit (LSB) Technique: In image steganography, one common approach is the Least Significant Bit (LSB) technique. It replaces the least significant bit in the pixel data of an image with the bits of the secret message. This minimal change is typically imperceptible to the human eye, allowing for data concealment without affecting the visual quality.

3. Applications of Steganography: Steganography is used in digital watermarking, confidential communication, and avoiding detection in scenarios where secrecy is crucial. However, steganography can also be used maliciously, making it an important area of study in cybersecurity.

Code Used :

```
from PIL import Image

# Function to encode a message into an image

def encode_message(image_path, message, output_path):

    try:

        # Load the image

        image = Image.open(image_path)

        encoded_image = image.copy()

        width, height = image.size
```

```
        # Add a delimiter to mark the end of the message
        message += "###"  # Add delimiter to end the message
        message_bits = ''.join([format(ord(char), '08b') for char in message])  # Convert message to bits
        data_index = 0

        # Loop through each pixel and modify the LSB to hide the message
        for y in range(height):
            for x in range(width):
                pixel = list(encoded_image.getpixel((x, y)))
                for n in range(3):  # Loop over RGB channels
                    if data_index < len(message_bits):
                        pixel[n] = pixel[n] & ~1 | int(message_bits[data_index])  # Modify LSB
                        data_index += 1
                encoded_image.putpixel((x, y), tuple(pixel))
                if data_index >= len(message_bits):  # Stop when the entire message is encoded
                    break
            if data_index >= len(message_bits):  # Stop when the entire message is encoded
                break

        # Save the encoded image
        encoded_image.save(output_path)
        print(f"Message encoded and saved as {output_path}")

    except Exception as e:
        print(f"Error encoding the message: {e}")

# Function to decode a message from an image
def decode_message(image_path):
    try:
        image = Image.open(image_path)
```

```
        width, height = image.size
        message_bits = []

        # Loop through each pixel to extract the LSB of the RGB values
        for y in range(height):
            for x in range(width):
                pixel = image.getpixel((x, y))
                for n in range(3):  # Loop over RGB channels
                    message_bits.append(pixel[n] & 1)  # Extract the LSB

        # Convert the bits to characters
        message = ''.join(chr(int(''.join(map(str, message_bits[i:i+8])), 2)) for i in range(0, len(message_bits), 8))

        # Extract the message before the delimiter
        message = message.split("###")[0]  # Extract the message up to the delimiter
        print("Decoded message:", message)

    except Exception as e:
        print(f"Error decoding the message: {e}")

# Example usage
image_path = "input_image.png"  # Path to input image
output_path = "encoded_image.png"  # Path to save encoded image
message = "Secret message here"

# Encode and decode the message
encode_message(image_path, message, output_path)
decode_message(output_path)
```

Result

1. Successfully encoded a secret message within an image using the LSB technique, hiding data without visible alterations to the image.
2. Retrieved the hidden message accurately by decoding the modified image, demonstrating the effectiveness of steganography in secure communication.

EXPERIMENT – 10

Aim:

The aim of this experiment is to implement and understand the process of text encryption using popular cryptographic algorithms such as Caesar Cipher, RSA, and AES. This will help in understanding how cryptographic algorithms are applied in securing text data.

Theory:

1. Cryptography Basics: Cryptography is the practice of securing information by transforming it into an unreadable format. This can only be reverted to its original form using a specific key. It is widely used for securing data in communication, financial transactions, and more.

2. Types of Cryptographic Algorithms:

- **Symmetric Cryptography:** The same key is used for both encryption and decryption. Examples include AES (Advanced Encryption Standard), DES (Data Encryption Standard), and RC4.
- **Asymmetric Cryptography:** Different keys are used for encryption and decryption. RSA (Rivest-Shamir-Adleman) is a popular example.

3. Cryptographic Algorithms:

- **Caesar Cipher:** A simple substitution cipher where each letter in the plaintext is shifted a certain number of places down or up the alphabet.
- **RSA (Rivest-Shamir-Adleman):** An asymmetric encryption algorithm that uses a public key for encryption and a private key for decryption. It is widely used for secure data transmission.
- **AES (Advanced Encryption Standard):** A symmetric key algorithm that encrypts data in blocks of 128 bits using key sizes of 128, 192, or 256 bits. It is highly secure and widely used in practice.

Program:

1. Caesar Cipher Implementation (Symmetric Encryption)

```
def caesar_cipher_encrypt(plain_text, shift):
    cipher_text = ""
    for char in plain_text:
        if char.isalpha():  # Encrypt only alphabetic characters
            shift_base = 65 if char.isupper() else 97
            cipher_text += chr((ord(char) - shift_base + shift) % 26 + shift_base)
        else:
```

```
            cipher_text += char  # Non-alphabetic characters remain unchanged
    return cipher_text

def caesar_cipher_decrypt(cipher_text, shift):
    return caesar_cipher_encrypt(cipher_text, -shift)
# Example usage
message = "Hello World!"
shift_value = 3
encrypted_message = caesar_cipher_encrypt(message, shift_value)
decrypted_message = caesar_cipher_decrypt(encrypted_message, shift_value)
print("Original Message:", message)
print("Encrypted Message:", encrypted_message)
print("Decrypted Message:", decrypted_message)
```

Output:

```
Original Message: Hello World!
Encrypted Message: Khoor Zruog!
Decrypted Message: Hello World!
```

2. RSA Implementation (Asymmetric Encryption)

```
# Install pycryptodome if not already installed (run this in a Jupyter cell)
!pip install pycryptodome

from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto.Random import get_random_bytes

# Generate RSA keys (public and private)
def generate_rsa_keys():
    key = RSA.generate(2048)  # Generate RSA keys with 2048 bits
    private_key = key.export_key()  # Export the private key
    public_key = key.publickey().export_key()  # Export the public key
```

```
    return private_key, public_key

# Encrypt message using RSA public key
def rsa_encrypt(public_key, message):
    key = RSA.import_key(public_key)  # Import public key
    cipher = PKCS1_OAEP.new(key)  # Create cipher using public key
    encrypted_message = cipher.encrypt(message.encode())  # Encrypt the message
    return encrypted_message

# Decrypt message using RSA private key
def rsa_decrypt(private_key, encrypted_message):
    key = RSA.import_key(private_key)  # Import private key
    cipher = PKCS1_OAEP.new(key)  # Create cipher using private key
    decrypted_message = cipher.decrypt(encrypted_message)  # Decrypt the message
    return decrypted_message.decode()  # Decode back to string

# Example usage
private_key, public_key = generate_rsa_keys()  # Generate RSA keys

message = "Hello RSA Encryption!"  # The message to be encrypted
encrypted_message = rsa_encrypt(public_key, message)  # Encrypt the message
decrypted_message = rsa_decrypt(private_key, encrypted_message)  # Decrypt the message

# Output the results
print("Original Message:", message)
print("Encrypted Message:", encrypted_message)  # This will show as a byte object
print("Decrypted Message:", decrypted_message)  # This should match the original message
```

Output :

```
Collecting pycryptodome
  Downloading pycryptodome-3.21.0-cp36-abi3-win_amd64.whl.metadata (3.4 kB)
Downloading pycryptodome-3.21.0-cp36-abi3-win_amd64.whl (1.8 MB)
   ---------------------------------------- 0.0/1.8 MB ? eta -:--:--
   ---------------------------------------- 0.0/1.8 MB ? eta -:--:--
    --------------------------------------- 0.0/1.8 MB 653.6 kB/s eta 0:00:03
   --- ------------------------------------ 0.1/1.8 MB 1.7 MB/s eta 0:00:01
   ----------- ---------------------------- 0.6/1.8 MB 4.0 MB/s eta 0:00:01
   ---------------------- ---------------- 1.1/1.8 MB 5.6 MB/s eta 0:00:01
   ---------------------------------- ------ 1.5/1.8 MB 7.0 MB/s eta 0:00:01
   ---------------------------------------- 1.8/1.8 MB 7.2 MB/s eta 0:00:00
Installing collected packages: pycryptodome
Successfully installed pycryptodome-3.21.0
Original Message: Hello RSA Encryption!
Encrypted Message: b'\x14T]\xf4\xbe\xdf\xc5\xed\x8e\x88>\x1a;\xf7\xb0\xcb\x15.\xd1&\x08Q2~\xa4K\xb3\xce\xda"\xfdqL\xe8^.^\x9fC\xb8\x16A\xd5
\xb2\x1b\xcf,\t\xc41\xcd\xbf\xc2_v\x85\t0\xc4\xb1*[Q\r&d\x98\xec\xbf\x8b\xb0\xb6\x1e\xb0\xfbz;\xb84\x15\xef\xab\xfbXm[Na\xbeh\xaf\x81=UH\xb
crB\xcfc\xe3b%:!\x11\x0c\x9c\x81\x0cO\x9a\\\xbd\x83n\x13\xba\xe2\x8d\x98\xdf\xe8s\xbb8\x0b\xe0X)\xd5\xcfc\xda`\x97\x88\xebYI\xf5\x0c\xe1Qy
\x83\xc3\xf8T\xc1\x8e2\xbc\xde\x8el=d\x8a$\x12`a\xf5u\x17\x1d\r\x01H\xeb\x13;\\\xc4{\x80\xb8p\xdd\xbf\x13d\xecs\xb98\xe0\x85^Z\xf4P\xe5\xf8
PE\xe1/]\x04\xd3\xbf\x9c9\xb6C\x05\xe9\xfef6\xc6*|/\x04\xf9\xa4%\xd1c\x9f\xbaw,3\xf5\xf39\x073\x95\x89\x08\x17\x14i\xb9\xca\x9bp\xfb\xd7\xb
7X\xc6\xad\x02\xa5\xd8\xde\xd6\xc3c\x08'
Decrypted Message: Hello RSA Encryption!
```

3. AES Implementation (Symmetric Encryption)

```
# Install pycryptodome if it's not already installed
!pip install pycryptodome
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
import base64
# AES encryption with CBC mode
def aes_encrypt(plain_text, key):
    cipher = AES.new(key, AES.MODE_CBC)
    # Pad the plain text to make it a multiple of 16 bytes (AES block size)
    plain_text_padded = plain_text + (16 - len(plain_text) % 16) * ' '
    cipher_text = cipher.encrypt(plain_text_padded.encode())
    # Return the IV and ciphertext encoded in base64
    return base64.b64encode(cipher.iv + cipher_text).decode()
# AES decryption
def aes_decrypt(cipher_text, key):
    cipher_data = base64.b64decode(cipher_text)  # Decode base64 to get the cipher data
    iv = cipher_data[:16]  # Extract the IV
    cipher_text = cipher_data[16:]  # Extract the encrypted message
```

```
    cipher = AES.new(key, AES.MODE_CBC, iv)  # Recreate cipher using IV
    decrypted_text = cipher.decrypt(cipher_text).decode().strip()  # Decrypt and remove padding
    return decrypted_text
# Example usage
key = get_random_bytes(16)  # AES key (128 bits, 16 bytes)
message = "Hello AES Encryption!"  # Original message
# Encrypt and decrypt the message
encrypted_message = aes_encrypt(message, key)
decrypted_message = aes_decrypt(encrypted_message, key)
# Output the results
print("Original Message:", message)
print("Encrypted Message:", encrypted_message)  # Encrypted message in base64
print("Decrypted Message:", decrypted_message)  # Decrypted message should match the original message
```

Output :

```
Requirement already satisfied: pycryptodome in c:\users\dell\anaconda3\lib\site-packages (3.21.0)
Original Message: Hello AES Encryption!
Encrypted Message: xl9KKnPoPPEva+8EPVh+xj2wSJIO2+ZX9Zk7TuuKa6aXrSuu/2PgNCrypAmo+N7o
Decrypted Message: Hello AES Encryption!
```