Amity School of Engineering and Technology
Amity University Noida

Applied Cryptography
[CSE442]

Lab File

Submitted To: -
Mr. Paurush Bhulania

Submitted by: -
Ojaswin Aggarwal
7CSE 2X
A2305222103

Index

S.No	Experiment	Date	Remarks
1	Introduction to Cryptography Concepts	14/07/25	
2	Encrypt the given plaintext using Caesar Cipher	21/07/25	
3	Encrypt the given plaintext using Hill Cipher	28/07/25	
4	Decrypt the given plaintext using Brute Force on Caesar Cipher	4/08/25	
5	Encrypt the given plaintext using Playfair Cipher	11/08/25	
6	Encrypt the given plaintext using Vigenère Cipher	18/08/25	
7	Encrypt the given plaintext using Vernam Cipher	25/08/25	
8	Encrypt the given plaintext using Diffie-Hellman Key Exchange	1/09/25	
9	Encrypt the given plaintext using DES (Data Encryption Standard)	8/09/25	
10	Perform and simulate Merkle-Damgård Toy Hash and Length Extension	15/09/25	

Experiment - 1

Aim - To Understand the Fundamental Concepts of Computer Security

Procedure of the Experiment

1. Different Types of Computer Security Threats

Computer security threats are malicious acts or entities that aim to damage, steal, or disrupt digital information and systems. Understanding these threats is essential for building effective defense mechanisms.

a) Viruses, Worms, Trojans, and Other Malware

Malware stands for *malicious software*. It refers to any software intentionally designed to cause damage to a computer, server, or network.

- Virus:
  - Attaches itself to legitimate programs or files.
  - Activates when the infected file is executed.
  - Can delete files, reformat hard drives, or slow down the system.
  - *Examples:* Michelangelo virus, ILOVEYOU virus.
- Worm:
  - Replicates itself without attaching to a host.
  - Spreads across networks, consuming bandwidth and system resources.
  - Can shut down entire networks.
  - *Examples:* SQL Slammer worm, Conficker.
- Trojan Horse:
  - Disguises as legitimate software to trick users.
  - Opens backdoors for hackers to access systems remotely.
  - Often used to steal personal information or download other malware.
  - *Example:* Zeus Trojan.
- Spyware:
  - Secretly collects user data such as browsing habits or keystrokes.
  - Often bundled with free software.
  - Can lead to identity theft.
- Ransomware:
  - Encrypts files and demands payment for decryption.
  - Disrupts critical services such as hospitals or banks.
  - *Examples:* WannaCry, REvil.
- Adware:
  - Displays unwanted advertisements.
  - Redirects users to malicious websites or slows down system performance.

b) Phishing and Social Engineering

- Phishing:
  - Attackers impersonate legitimate organizations through email, SMS, or websites.

- Users are tricked into revealing personal information like login credentials or credit card details.
- *Example:* An email that looks like it's from your bank asking you to verify your password.

● Social Engineering:
- Manipulates people into giving away confidential information.
- Relies on psychological manipulation rather than technical hacking.
- *Example:* A fake IT support call asking for your password.

c) Denial of Service (DoS) Attacks

● DoS Attack:
- Overwhelms a network, server, or website with excessive traffic.
- Makes services unavailable to legitimate users.
- Often used to disrupt business operations.

● DDoS (Distributed DoS):
- An attack launched from multiple sources, usually through botnets.
- Harder to detect and defend against.

d) Unauthorized Access and Hacking

● Unauthorized Access:
- Occurs when someone gains access to data, networks, or devices without permission.
- Can lead to theft, leakage, or destruction of sensitive information.

● Hacking:
- Involves exploiting system vulnerabilities.
- *Types of Hackers:*
- Black Hat: Acts with malicious intent.
- White Hat: Ethical hackers who strengthen system security.
- Grey Hat: Breaks into systems without permission but not for malicious reasons.

2. Different Types of Security Measures

Security measures are the tools and techniques used to protect systems from these threats.

a) Firewalls
- Act as a protective barrier between an internal network and the internet.
- Filter traffic based on defined rules such as IP addresses, ports, and protocols.
- *Types include:*
    - Packet-filtering firewalls that filter based on packet headers.
    - Stateful inspection that tracks active connections.
    - Application firewalls that control access to specific programs.

b) Anti-virus Software
- Scans systems for known malware and suspicious activity.
- Provides real-time protection and alerts.
- Requires regular updates to handle new threats.
- *Examples:* Norton, Kaspersky, McAfee, Bitdefender.

c) Intrusion Detection Systems (IDS)

- Monitor and analyze network or system activity for unusual behavior.
- *Types include:*
  - NIDS (Network IDS) that monitors network traffic.
  - HIDS (Host IDS) that monitors a specific device.
- IDS systems typically trigger alerts but do not block threats directly (that role belongs to IPS – Intrusion Prevention System).

d) Encryption

- Converts readable data (plaintext) into unreadable code (ciphertext).
- Protects data during transmission and storage.
- Only authorized users with the correct key can decrypt it.
- *Types include:*
  - Symmetric Encryption – uses the same key for encryption and decryption (e.g., AES, DES).
  - Asymmetric Encryption – uses a pair of keys (public and private) (e.g., RSA).

3. What is Computer Security?

Computer security, or cybersecurity, is the protection of computer systems and networks from theft, damage, disruption, or unauthorized access. It covers the protection of hardware, software, data, networks, and users.
The goal is to maintain systems that are confidential, trustworthy, available, and accountable.

CIA Triad: The Pillars of Computer Security

The CIA Triad defines three essential principles for information security.

1. Confidentiality

Confidentiality means preventing unauthorized access to sensitive data. It ensures that data is accessible only to those with proper authority.
Techniques used: Encryption, authentication, access control, data classification, and VPNs.
Example: Only HR personnel can access employee salary details.

2. Integrity

Integrity ensures that information is accurate and unaltered. Data must remain consistent and trustworthy, preventing unauthorized or accidental modifications.
Techniques used: Checksums, hash functions (like SHA-256), digital signatures, and version control.
Example: A student's exam result in a database should not be modified without authorization.

3. Availability

Availability ensures systems and data are accessible to authorized users whenever needed. It prevents downtime and ensures business continuity.
Techniques used: Redundancy, load balancing, DoS/DDoS protection, and failover mechanisms.
Example: Online banking systems should be accessible to users around the clock.

Expanded Model: CIAAN (CIA Triad + Authentication + Non-repudiation)

4. Authentication
Authentication verifies the identity of users or systems before granting access.
Techniques used: Passwords, one-time passwords (OTPs), biometric verification, and multi-factor authentication.
Example: Logging in with a password and then confirming via a mobile verification code.

5. Non-Repudiation
Non-repudiation ensures that once a message or action is performed, it cannot later be denied.
Techniques used: Digital signatures, secure audit logs, and blockchain verification.
Example: A user who sends a digitally signed document cannot later deny sending it.

Summary Table – CIAAN Model at a Glance

Principle	Purpose	Key Technologies/Methods
Confidentiality	Prevent unauthorized data access	Encryption, Access Control, VPN
Integrity	Ensure data is accurate and unchanged	Hashing, Checksums, Digital Signatures
Availability	Keep systems accessible when needed	Redundancy, Load Balancing, Backups
Authentication	Confirm identity of users/devices	Passwords, Biometrics, MFA
Non-repudiation	Prevent denial of performed actions	Digital Signatures, Logging

Why the CIA Triad is Important

The CIA Triad forms the foundation of secure system design. It promotes balanced security — focusing on confidentiality alone can weaken availability, for instance.
Its principles apply universally, from banking systems and government databases to personal devices.
Together with the CIAAN model, it ensures that data is protected, systems are functional, users are verified, and actions are traceable.

Observations and Learnings

Observations:

1. Computer security threats are diverse and constantly evolving.
2. Malware types differ by their intent and behavior.
3. Psychological manipulation is a major security risk through phishing and social engineering.
4. DoS and DDoS attacks target availability rather than data, yet can cause massive damage.
5. Strong access control and authentication systems are essential defenses.
6. The CIA Triad provides the foundation for all security design.
7. Effective security requires multiple layers of defense working together.
8. The CIAAN model adds authentication and non-repudiation to address modern challenges.

Learnings:

1. Knowing the nature of threats helps choose the right defense tools and policies.
2. Security must balance confidentiality, integrity, and availability.
3. User awareness and education are essential for preventing human-targeted attacks.
4. Strong authentication and access control prevent many breaches.
5. Continuous monitoring and auditing help detect issues early.
6. Encryption is critical to maintaining confidentiality.
7. Cybersecurity is an ongoing process that evolves with new threats.
8. Ethical and legal accountability are vital aspects of cybersecurity.

Conclusion

In today's interconnected world, computer security is not just a technical necessity but a cornerstone of individual, organizational, and national safety. Understanding threats such as malware, phishing, DoS attacks, and unauthorized access helps users and professionals anticipate and mitigate risks.

Security measures like firewalls, antivirus programs, intrusion detection systems, and encryption form the foundation of digital defense. The CIA Triad—Confidentiality, Integrity, and Availability—along with Authentication and on-repudiation (CIAAN), ensures systems remain secure, trusted, and resilient.

Ultimately, strong computer security combines technology, policy, and human awareness to create safe and reliable digital environments that support innovation while preserving privacy and trust.

Experiment - 2

Aim - Encrypt the given plain text using the Caesar Cipher encryption algorithm.

Procedure of the Experiment

Code -

```
public class CaesarCipher {

    // Method to encrypt plain text
    public static String encrypt(String plainText, int shift) {
        StringBuilder cipherText = new StringBuilder();

        for (int i = 0; i < plainText.length(); i++) {
            char ch = plainText.charAt(i);

            if (Character.isUpperCase(ch)) {
                char c = (char) (((ch - 'A' + shift) % 26) + 'A');
                cipherText.append(c);
            } else if (Character.isLowerCase(ch)) {
                char c = (char) (((ch - 'a' + shift) % 26) + 'a');
                cipherText.append(c);
            } else {
                cipherText.append(ch);
            }
        }
        return cipherText.toString();
    }

    // Method to decrypt cipher text
    public static String decrypt(String cipherText, int shift) {
        // Decryption is just shifting backwards
        return encrypt(cipherText, 26 - (shift % 26));
    }

    // Main method
    public static void main(String[] args) {
        String message = "Hello World!";
        int shift = 3;
        String encrypted = encrypt(message, shift);
        String decrypted = decrypt(encrypted, shift);

        System.out.println("Original Message : " + message);
        System.out.println("Encrypted Message: " + encrypted);
        System.out.println("Decrypted Message: " + decrypted);
    }
}
```

Observations and Learnings

Observations:

1. The Caesar Cipher shifts each alphabetic character by a fixed number of positions.
2. Both uppercase and lowercase letters are handled separately to maintain proper casing.
3. Non-alphabetic characters such as spaces, digits, and punctuation marks remain unchanged.
4. Decryption works by reversing the shift, which is achieved by subtracting the shift from 26.
5. The modulo operation ensures smooth wrapping of characters from ‘Z’ back to ‘A’ and ‘z’ to ‘a’.

Learnings:

1. Understood how classical encryption techniques like the Caesar Cipher operate.
2. Gained practical experience in manipulating characters using ASCII values in Java.
3. Learned the importance of the modulus (%) operation for maintaining alphabetic boundaries.
4. Observed that encryption and decryption can share the same logic with minor adjustments.
5. Improved understanding of efficient string handling using the StringBuilder class.

Conclusion

The Caesar Cipher is one of the simplest and most fundamental encryption techniques. This Java program effectively demonstrates how a fixed shift can convert plain text into an encrypted format and then restore it through decryption. While it is not secure by modern cryptographic standards, implementing the Caesar Cipher provides valuable insight into encryption fundamentals. The experiment enhanced understanding of string manipulation, character arithmetic, and logical structuring in Java programming.

Experiment - 3

Aim - Encrypt the given plaintext using the Hill Cipher encryption algorithm.

Procedure of the Experiment

Code -

```
import java.util.Scanner;

public class HillCipher {
    static int[][] keyMatrix = new int[2][2];
    static int[][] inverseKeyMatrix = new int[2][2];
    static int[] messageVector = new int[2];
    static int[] cipherVector = new int[2];
    static int[] decryptedVector = new int[2];

    // Convert character to integer (A=0, ..., Z=25)
    static int charToInt(char c) {
        return c - 'A';
    }

    // Convert integer to character (0=A, ..., 25=Z)
    static char intToChar(int i) {
        return (char) (i + 'A');
    }

    // Calculate modular inverse of determinant mod 26
    static int modInverse(int a, int m) {
        a = a % m;
        for (int x = 1; x < m; x++)
            if ((a * x) % m == 1)
                return x;
        return -1;
    }

    // Compute inverse of 2x2 key matrix modulo 26
    static boolean findInverseKeyMatrix() {
        int det = (keyMatrix[0][0] * keyMatrix[1][1] - keyMatrix[0][1] * keyMatrix[1][0])
% 26;
        if (det < 0) det += 26;

        int detInv = modInverse(det, 26);
        if (detInv == -1) {
            System.out.println("Inverse doesn't exist (det not coprime to 26).");
            return false;
        }

        inverseKeyMatrix[0][0] = (keyMatrix[1][1] * detInv) % 26;
```

```
        inverseKeyMatrix[1][1] = (keyMatrix[0][0] * detInv) % 26;
        inverseKeyMatrix[0][1] = (-keyMatrix[0][1] + 26) * detInv % 26;
        inverseKeyMatrix[1][0] = (-keyMatrix[1][0] + 26) * detInv % 26;

        return true;
    }

    // Encrypt 2-letter block
    static String encrypt(String message) {
        message = message.toUpperCase();
        if (message.length() % 2 != 0) message += "X"; // padding

        StringBuilder cipherText = new StringBuilder();

        for (int i = 0; i < message.length(); i += 2) {
            messageVector[0] = charToInt(message.charAt(i));
            messageVector[1] = charToInt(message.charAt(i + 1));

            cipherVector[0] = (keyMatrix[0][0] * messageVector[0] + keyMatrix[0][1] *
messageVector[1]) % 26;
            cipherVector[1] = (keyMatrix[1][0] * messageVector[0] + keyMatrix[1][1] *
messageVector[1]) % 26;

            cipherText.append(intToChar(cipherVector[0]));
            cipherText.append(intToChar(cipherVector[1]));
        }
        return cipherText.toString();
    }

    // Decrypt 2-letter block
    static String decrypt(String cipher) {
        cipher = cipher.toUpperCase();
        StringBuilder plainText = new StringBuilder();

        for (int i = 0; i < cipher.length(); i += 2) {
            cipherVector[0] = charToInt(cipher.charAt(i));
            cipherVector[1] = charToInt(cipher.charAt(i + 1));

            decryptedVector[0] = (inverseKeyMatrix[0][0] * cipherVector[0] +
inverseKeyMatrix[0][1] * cipherVector[1]) % 26;
            decryptedVector[1] = (inverseKeyMatrix[1][0] * cipherVector[0] +
inverseKeyMatrix[1][1] * cipherVect
```

Observations and Learnings

Observations:

1. The Hill Cipher uses a square key matrix (2x2 or 3x3), which must be invertible modulo 26 for decryption to function correctly.
2. If the plaintext length isn't a multiple of the matrix size, the message is padded with an extra character ('X').
3. Encryption converts plaintext blocks into numeric vectors, multiplies them with the key matrix, and applies modulo 26 to get ciphertext.
4. Decryption requires finding the modular inverse of the key matrix, which is only possible if the determinant is coprime with 26.
5. The cipher accepts only uppercase alphabetic input; non-alphabetic characters are ignored or removed.

Learnings:

1. The Hill Cipher demonstrates how **linear algebra and modular arithmetic** form the foundation of classical cryptography.
2. Understanding **modular inverses** is essential to decrypt messages successfully.
3. Although it introduces polyalphabetic substitution and block encryption, the Hill Cipher is **vulnerable to known plaintext attacks**, making it insecure for modern use.
4. Implementing the algorithm in Java strengthens practical knowledge of matrix handling, modulo operations, ASCII manipulation, and error checking for key validity.
5. Encryption and decryption are **mathematical inverses**, showing clear symmetry and reversibility in classical ciphers.

Conclusion

The Hill Cipher applies linear algebra principles to encrypt text using matrix operations and modular arithmetic. In this experiment, both encryption and decryption were implemented using a 2x2 key matrix in Java. The process highlighted how plaintext can be systematically converted to ciphertext and restored using the matrix inverse. Although not secure by contemporary standards, this experiment effectively illustrates the mathematical foundation of cryptography and reinforces core programming and logical problem-solving skills.

Experiment - 4

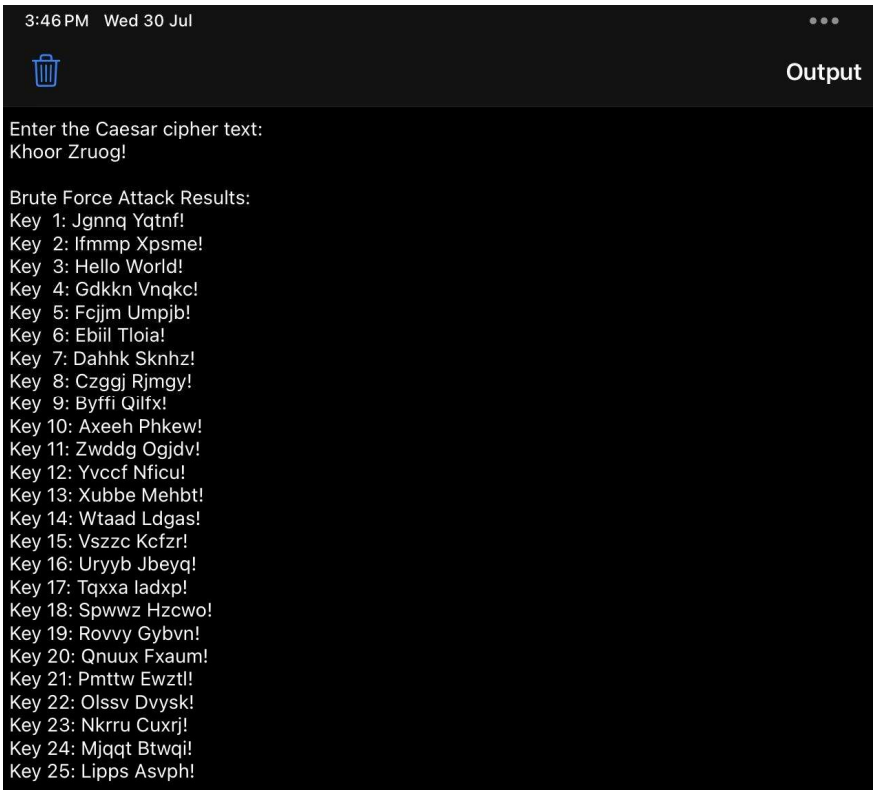
Aim - Decrypt the given plaintext using Brute Force on the Caesar Cipher encryption algorithm.

Procedure of the Experiment

Code –

```
import java.util.Scanner;

public class BruteForceCC {

    // Function to decrypt a message using a given key
    public static String decrypt(String cipherText, int key) {
        StringBuilder decryptedText = new StringBuilder();

        for (char ch : cipherText.toCharArray()) {
            if (Character.isUpperCase(ch)) {
                char c = (char) ((ch - 'A' - key + 26) % 26 + 'A');
                decryptedText.append(c);
            } else if (Character.isLowerCase(ch)) {
                char c = (char) ((ch - 'a' - key + 26) % 26 + 'a');
                decryptedText.append(c);
            } else {
                decryptedText.append(ch); // keep punctuation/spaces unchanged
            }
        }
        return decryptedText.toString();
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter the Caesar cipher text:");
        String cipherText = scanner.nextLine();

        System.out.println("\nBrute Force Attack Results:");
        for (int key = 1; key < 26; key++) {
            String decrypted = decrypt(cipherText, key);
            System.out.printf("Key %2d: %s%n", key, decrypted);
        }

        scanner.close();
    }
}
```

Output –



Observations and Learnings

Observations:

1. The Caesar Cipher has only 25 possible keys (excluding the trivial shift of 0), which makes it highly vulnerable to brute force attacks.
2. Trying all possible keys quickly reveals the correct plaintext, as the same shift applies uniformly across all characters.
3. Uppercase and lowercase letters must be processed separately to maintain proper case during decryption.
4. Non-alphabetic characters such as spaces, punctuation, and digits remain unchanged, preserving the structure of the text.
5. The cipher can be decrypted without prior knowledge of the key, simply by testing all key possibilities.

Learnings:

1. **Caesar Cipher Weakness:** The cipher's small key space makes it easily breakable, showing why it is unsuitable for secure communication today.
2. **Programming Reinforcement:** This experiment strengthens Java programming skills, including character handling, modular arithmetic, looping constructs, and string manipulation.

3. **Cryptanalysis Foundation:** Brute force serves as a fundamental concept in cryptanalysis—effective only when the key space is small.
4. **Key Space Relevance:** Modern encryption algorithms such as AES employ vastly larger key spaces to make brute force attacks computationally infeasible.

Conclusion

The Caesar Cipher, while historically significant, is highly insecure due to its small and predictable key space. This experiment demonstrated how brute force decryption can easily break the cipher by systematically trying all possible key values. Implementing this in Java provided practical exposure to cryptographic attack methods, string handling, and logic building. The exercise highlights the importance of large key spaces and stronger encryption standards in modern secure systems.

Experiment - 5

Aim - Encrypt the given plaintext using the Playfair Cipher encryption algorithm.

Procedure of the Experiment

Code-

```
import java.util.*;

public class PlayfairCipher {
    private char[][] keyMatrix = new char[5][5];

    public void generateKeyMatrix(String key) {
        key = key.toUpperCase().replaceAll("[^A-Z]", "").replace("J", "I");
        Set<Character> used = new LinkedHashSet<>();

        // Add unique characters from the key
        for (char c : key.toCharArray()) {
            used.add(c);
        }

        // Add remaining letters A-Z (excluding J)
        for (char c = 'A'; c <= 'Z'; c++) {
            if (c != 'J') used.add(c);
        }

        Iterator<Character> it = used.iterator();
        for (int i = 0; i < 5; i++)
            for (int j = 0; j < 5; j++)
                keyMatrix[i][j] = it.next();
    }

    private String formatPlainText(String input) {
        input = input.toUpperCase().replaceAll("[^A-Z]", "").replace("J", "I");
        StringBuilder formatted = new StringBuilder();
        for (int i = 0; i < input.length(); i++) {
            formatted.append(input.charAt(i));
            if (i + 1 < input.length() && input.charAt(i) == input.charAt(i + 1)) {
                formatted.append('X');
            }
        }
        if (formatted.length() % 2 != 0) {
            formatted.append('X');
        }
        return formatted.toString();
    }

    private int[] findPosition(char c) {
```

```
        for (int i = 0; i < 5; i++)
            for (int j = 0; j < 5; j++)
                if (keyMatrix[i][j] == c)
                    return new int[]{i, j};
        return null;
    }

    public String encrypt(String plaintext) {
        String formattedText = formatPlainText(plaintext);
        StringBuilder cipher = new StringBuilder();

        for (int i = 0; i < formattedText.length(); i += 2) {
            char a = formattedText.charAt(i);
            char b = formattedText.charAt(i + 1);

            int[] posA = findPosition(a);
            int[] posB = findPosition(b);

            if (posA[0] == posB[0]) {
                cipher.append(keyMatrix[posA[0]][(posA[1] + 1) % 5]);
                cipher.append(keyMatrix[posB[0]][(posB[1] + 1) % 5]);
            } else if (posA[1]
```

Output –

Observations and Learnings

Observations:

1. Key Matrix Structure:
   The key matrix is a 5×5 grid created from the keyword followed by remaining letters of the alphabet (excluding ‘J’). The position of each character determines substitution during encryption and decryption.
2. Handling Repeated Letters and Odd Lengths:
   When a plaintext pair contains repeated letters (e.g., “HELLO”), an ‘X’ is

inserted between them. If the message length is odd, an extra ‘X’ is added at the end.

3. Digraph-Based Substitution:
   The plaintext is processed in pairs (digraphs). Depending on whether the letters appear in the same row, same column, or form a rectangle, the cipher applies specific substitution rules.
4. Symmetry of Encryption and Decryption:
   Decryption reverses the encryption steps precisely, restoring the formatted plaintext.
5. Character Replacement:
   The letter ‘J’ is replaced with ‘I’, and non-alphabetic characters are ignored before encryption.

Learnings:

1. Logical Design of Classical Ciphers:
   The Playfair Cipher demonstrates how early encryption relied on positional logic rather than mathematical complexity.
2. Significance of Preprocessing:
   Cleaning and formatting input data—removing special characters, handling duplicates, and adjusting length—is essential for correct encryption.
3. Matrix Manipulation:
   Building and navigating a 2D key matrix reinforces understanding of array structures in Java.
4. Rule-Based Logic:
   The cipher highlights condition-based operations, showing how positional rules define substitution.
5. Java Programming Practice:
   This implementation strengthens knowledge of arrays, loops, conditionals, and string handling.

Conclusion

The Playfair Cipher is a classic example of structured, rule-based encryption that uses digraph substitution for security. Through this implementation in Java, we observed how preprocessing, matrix operations, and logical substitution work together to form encrypted messages. While no longer practical for modern cryptography, the Playfair Cipher remains an excellent educational model for understanding the mechanics of classical encryption and the evolution toward modern cryptographic algorithms.

Experiment - 6

Aim - Encrypt the given plaintext using the Vigenère Cipher encryption algorithm.

Procedure of the Experiment

Code-

```
import java.util.Scanner;

public class VigenereCipher {

    // Function to encrypt the text using Vigenère Cipher
    public static String encrypt(String plaintext, String key) {
        StringBuilder ciphertext = new StringBuilder();

        plaintext = plaintext.toUpperCase();
        key = key.toUpperCase();

        // Repeat the key to match the length of the plaintext
        StringBuilder fullKey = new StringBuilder();
        for (int i = 0, j = 0; i < plaintext.length(); i++) {
            if (Character.isLetter(plaintext.charAt(i))) {
                fullKey.append(key.charAt(j));
                j = (j + 1) % key.length();
            } else {
                fullKey.append(plaintext.charAt(i)); // Keep non-letter characters
            }
        }

        // Encrypt the plaintext
        for (int i = 0; i < plaintext.length(); i++) {
            char plainChar = plaintext.charAt(i);
            char keyChar = fullKey.charAt(i);

            if (Character.isLetter(plainChar)) {
                int p = plainChar - 'A';
                int k = keyChar - 'A';
                int c = (p + k) % 26;
                ciphertext.append((char) (c + 'A'));
            } else {
                ciphertext.append(plainChar); // Keep spaces or punctuation
            }
        }

        return ciphertext.toString();
    }

    // Main method to test the cipher
```

```
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the plaintext: ");
        String plaintext = scanner.nextLine();

        System.out.print("Enter the key: ");
        String key = scanner.nextLine();

        String encrypted = encrypt(plaintext, key);
        System.out.println("Encrypted text: " + encrypted);

        scanner.close();
    }
}
```

Output –

Observations and Learnings

Observations:

1. **Case Handling:**
   The cipher converts all characters to uppercase to ensure uniformity since it operates only on A–Z.
2. **Key Repetition:**
   The input key is repeated cyclically to match the length of the plaintext. Non-letter characters are skipped in key application but retained in output.
3. **Use of Modular Arithmetic:**
   The encryption uses (p + k) % 26 to maintain letter shifts within the alphabet range.
4. **Interactive User Input:**
   The use of Scanner enables users to provide custom plaintext and keys, enhancing usability.
5. **Preservation of Non-Alphabetic Characters:**
   Spaces, digits, and punctuation remain unchanged, keeping the ciphertext structured and readable.

Learnings:

1. **Understanding the Vigenère Cipher:**
   Learned that it is a *polyalphabetic substitution cipher* where each character in the plaintext is shifted by a different amount, depending on the key.
2. **String Handling in Java:**
   Gained practical experience using StringBuilder, charAt(), and case conversion (toUpperCase()).
3. **Efficient Key Management:**
   Understood how to repeat and align the key correctly with the plaintext while skipping non-letter characters.
4. **Role of Modular Arithmetic:**
   Reinforced the use of modular operations to ensure cyclic alphabet shifting within 26 letters.
5. **Security Perspective:**
   The Vigenère cipher provides better security than Caesar Cipher due to multiple shifting patterns, but it is still susceptible to frequency analysis if the key is short or reused.

Conclusion

The Java implementation of the Vigenère Cipher successfully demonstrates the concept of polyalphabetic encryption, where each letter of the plaintext is encrypted using a different shift determined by a repeating key. The program effectively handles uppercase conversion, key repetition, and non-alphabetic characters. While the cipher is a significant improvement over monoalphabetic systems like Caesar Cipher, it remains weak against advanced cryptanalysis and is not suitable for modern secure communication. This experiment provides valuable insight into how classical ciphers evolved toward more robust encryption methods.

Experiment - 7

Aim - Encrypt the given plaintext using the Vernam Cipher encryption algorithm.

Procedure of the Experiment

Code-

```
import java.nio.charset.StandardCharsets;
import java.security.SecureRandom;
import java.util.Base64;

public class VernamCipher {

    // Core XOR operations
    public static byte[] xorOTP(byte[] data, byte[] key) {
        if (key.length < data.length) {
            throw new IllegalArgumentException("Key must be at least as long as data
(OTP).");
        }
        byte[] out = new byte[data.length];
        for (int i = 0; i < data.length; i++) {
            out[i] = (byte) (data[i] ^ key[i]);
        }
        return out;
    }

    public static byte[] xorRepeat(byte[] data, byte[] key) {
        if (key.length == 0) throw new IllegalArgumentException("Key must not be
empty.");
        byte[] out = new byte[data.length];
        for (int i = 0; i < data.length; i++) {
            out[i] = (byte) (data[i] ^ key[i % key.length]);
        }
        return out;
    }

    // Helper functions
    public static byte[] randomKey(int length) {
        byte[] k = new byte[length];
        new SecureRandom().nextBytes(k);
        return k;
    }

    public static String toHex(byte[] bytes) {
        StringBuilder sb = new StringBuilder(bytes.length * 2);
        for (byte b : bytes) sb.append(String.format("%02x", b));
        return sb.toString();
    }
```

```
    public static byte[] fromHex(String hex) {
        if ((hex.length() & 1) == 1) throw new IllegalArgumentException("Hex string
must have even length.");
        byte[] out = new byte[hex.length() / 2];
        for (int i = 0; i < out.length; i++) {
            int hi = Character.digit(hex.charAt(2 * i), 16);
            int lo = Character.digit(hex.charAt(2 * i + 1), 16);
            if (hi < 0 || lo < 0) throw new IllegalArgumentException("Invalid hex.");
            out[i] = (byte) ((hi << 4) + lo);
        }
        return out;
    }

    // CLI Implementation
    public static void main(String[] args) {
        if (args.length == 0) {
            printHelp();
            demo();
            return;
        }

        switch (args[0].toLowerCase()) {
            case "otp":
                if (args.length == 2) {
                    byte[] pt = args[1].getBytes(StandardCharsets.UTF_8);
                    byte[] key = randomKey(pt.length);
                    byte[] ct = xorOTP(pt, key);
                    System.out.println("key-hex=" + toHex(key));
                    System.out.println("ciphertext-hex=" + toHex(ct));
                } else if (args.length == 3) {
                    byte[] pt = args[1].getBytes(StandardCharsets.UTF_8);
                    byte[] key = fromHex(args[2]);
                    byte[] ct = xorOTP(pt, key);
                    System.out.println("ciphertext-hex=" + toHex(ct));
                } else {
                    System.out.println("Usage: java VernamCipher otp <plaintext>
[keyHex]");
                }
                break;

            case "decrypt":
                if (args.length == 3) {
                    byte[] ct = fromHex(args[1]);
                    byte[] key = fromHex(args[2]);
                    byte[] pt = xorOTP(ct, key);
                    System.out.println("plaintext=" + new String(pt,
StandardCharsets.UTF_8));
                } else {
```

```
                System.out.println("Usage: java VernamCipher decrypt <cipherHex>
<keyHex>");
            }
            break;

        case "repeat":
            if (args.length == 3) {
                byte[] data = args[1].getBytes(StandardCharsets.UTF_8);
                byte[] key = args[2].getBytes(StandardCharsets.UTF_8);
                byte[] ct = xorRepeat(data, key);
                System.out.println("ciphertext-base64=" +
Base64.getEncoder().encodeToString(ct));
                byte[] dec = xorRepeat(ct, key);
                System.out.println("decrypted=" + new String(dec,
StandardCharsets.UTF_8));
            } else {
                System.out.println("Usage: java VernamCipher repeat <plaintext>
<keyText>");
            }
            break;

        default:
            printHelp();
      }
   }

   private static void printHelp() {
      System.out.println("VernamCipher - OTP and repeating-key XOR");
      System.out.println("Commands:");
      System.out.println("  otp <plaintext> [keyHex]       Encrypt with one-time
pad.");
      System.out.println("                                 If keyHex omitted, a random key is
generated and printed.");
      System.out.println("  decrypt <cipherHex> <keyHex>    Decrypt OTP (hex
inputs).");
      System.out.println("  repeat <plaintext> <keyText>    Repeating-key XOR
demo (not secure).");
   }

   private static void demo() {
      String msg = "HELLO VERNAM!";
      byte[] key = randomKey(msg.getBytes(StandardCharsets.UTF_8).length);
      byte[] ct = xorOTP(msg.getBytes(StandardCharsets.UTF_8), key);
      byte[] pt = xorOTP(ct, key);

      System.out.println("\n--- Quick Demo (OTP) ---");
      System.out.println("plaintext     = " + msg);
      System.out.println("key-hex       = " + toHex(key));
      System.out.println("ciphertext-hex= " + toHex(ct));
```

```
        System.out.println("decrypted     = " + new String(pt,
StandardCharsets.UTF_8));
    }
}
```

Output –

Observations and Learnings

Observations:

1. The program supports both One-Time Pad (OTP) and repeating-key XOR encryption methods.
    - OTP is perfectly secure if the key is random, as long as the plaintext, and used only once.
    - Repeating-key XOR is insecure because repeating short keys allows frequency analysis.
2. The program uses hexadecimal and Base64 encodings to handle ciphertext and keys effectively.
3. SecureRandom ensures the generation of cryptographically strong random keys.
4. The XOR operation's self-inverse property is demonstrated — applying XOR with the same key decrypts the ciphertext.
5. Input validation prevents errors like invalid hex, short keys, or empty inputs.
6. The CLI interface allows flexible usage for both encryption and decryption, and the demo mode simplifies testing.

Learnings:

1. Gained a clear understanding of the Vernam Cipher and its reliance on XOR operations.
2. Recognized the critical importance of key management — key randomness, length, and one-time usage directly affect security.

3. Understood why the One-Time Pad is theoretically unbreakable yet practically limited by key distribution and storage challenges.
4. Learned how cryptographic data is represented using hex and Base64 for reliable transfer across systems.
5. Observed how SecureRandom differs from standard Random and why cryptographically secure randomness is essential.

Conclusion

This experiment demonstrates the working of the Vernam Cipher using both **One-Time Pad (OTP)** and **repeating-key XOR** methods. It effectively shows how XOR operations can provide perfect secrecy under ideal key conditions. However, it also highlights that OTP's security depends entirely on the randomness and secrecy of the key, making it impractical for large-scale use. The repeating-key XOR, though functional, is not secure due to key reuse vulnerabilities. This exercise enhances understanding of classical cryptography, randomness, and data encoding techniques essential for secure encryption systems.

Experiment - 8

Aim - Encrypt the given plaintext using the Diffie-Hellman Key Exchange encryption algorithm.

Procedure of the Experiment

Code –

```
import javax.crypto.Cipher;
import javax.crypto.KeyAgreement;
import javax.crypto.SecretKey;
import javax.crypto.spec.GCMParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import javax.crypto.interfaces.DHPublicKey;
import javax.crypto.spec.DHParameterSpec;
import java.security.*;
import java.security.spec.X509EncodedKeySpec;
import java.util.Arrays;

public class DiffieHellmanDemo {
    public static void main(String[] args) throws Exception {
        SecureRandom rnd = new SecureRandom();

        // Alice generates DH key pair
        KeyPairGenerator aliceKpairGen = KeyPairGenerator.getInstance("DH");
        aliceKpairGen.initialize(2048, rnd);
        KeyPair aliceKpair = aliceKpairGen.generateKeyPair();
        byte[] alicePubEnc = aliceKpair.getPublic().getEncoded();

        // Bob creates key pair using Alice's public key parameters
        KeyFactory kf = KeyFactory.getInstance("DH");
        X509EncodedKeySpec x509KeySpec = new
X509EncodedKeySpec(alicePubEnc);
        PublicKey alicePubFromEnc = kf.generatePublic(x509KeySpec);
        DHPublicKey aliceDhPub = (DHPublicKey) alicePubFromEnc;
        DHParameterSpec dhParams = aliceDhPub.getParams();

        KeyPairGenerator bobKpairGen = KeyPairGenerator.getInstance("DH");
        bobKpairGen.initialize(dhParams, rnd);
        KeyPair bobKpair = bobKpairGen.generateKeyPair();
        byte[] bobPubEnc = bobKpair.getPublic().getEncoded();
        // Both compute shared secret using KeyAgreement
        KeyAgreement aliceKeyAgree = KeyAgreement.getInstance("DH");
        aliceKeyAgree.init(aliceKpair.getPrivate());
        PublicKey bobPubFromEnc = kf.generatePublic(new
X509EncodedKeySpec(bobPubEnc));
        aliceKeyAgree.doPhase(bobPubFromEnc, true);
        byte[] aliceShared = aliceKeyAgree.generateSecret();
```

```
        KeyAgreement bobKeyAgree = KeyAgreement.getInstance("DH");
        bobKeyAgree.init(bobKpair.getPrivate());
        bobKeyAgree.doPhase(alicePubFromEnc, true);
        byte[] bobShared = bobKeyAgree.generateSecret();
        System.out.println("Shared secret equal: " + Arrays.equals(aliceShared,
bobShared));
        System.out.println("Shared secret (hex, truncated): " +
toHex(Arrays.copyOf(aliceShared, 32)));
        // Derive AES-128 key from shared secret
        byte[] aesKey = deriveAes128FromSharedSecret(aliceShared);
        SecretKeySpec aesKeySpec = new SecretKeySpec(aesKey, "AES");
        System.out.println("Derived AES-128 key (hex): " + toHex(aesKey));
        // AES-GCM encrypt/decrypt demo
        String plaintext = "Hello Diffie-Hellman + AES-GCM!";
        byte[] iv = new byte[12];
        rnd.nextBytes(iv);
        byte[] ciphertext = aesGcmEncrypt(plaintext.getBytes("UTF-8"), aesKeySpec,
iv);
        byte[] decrypted = aesGcmDecrypt(ciphertext, aesKeySpec, iv);
        System.out.println("Plaintext: " + plaintext);
        System.out.println("Ciphertext (hex): " + toHex(ciphertext));
        System.out.println("Decrypted: " + new String(decrypted, "UTF-8"));
    }
    private static byte[] deriveAes128FromSharedSecret(byte[] shared) throws
Exception {
        MessageDigest sha256 = MessageDigest.getInstance("SHA-256");
        byte[] hash = sha256.digest(shared);
        return Arrays.copyOf(hash, 16); // AES-128
    }
    private static byte[] aesGcmEncrypt(byte[] plain, SecretKey key, byte[] iv) throws
Exception {
        Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
        GCMParameterSpec spec = new GCMParameterSpec(128, iv);
        cipher.init(Cipher.ENCRYPT_MODE, key, spec);
        return cipher.doFinal(plain);
    }
    private static byte[] aesGcmDecrypt(byte[] cipherText, SecretKey key, byte[] iv)
throws Exception {
        Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
        GCMParameterSpec spec = new GCMParameterSpec(128, iv);
        cipher.init(Cipher.DECRYPT_MODE, key, spec);
        return cipher.doFinal(cipherText);
    }

    private static String toHex(byte[] b) {
        StringBuilder sb = new StringBuilder();
        for (byte x : b) sb.append(String.format("%02x", x & 0xff));
        return sb.toString();
    }
}
```

Output –

```
Shared secret equal: true
Shared secret (hex, truncated): cb62c07963785494f853aa3ad9065a9a371b91b57e460375daf855e9894aa88d
Derived AES-128 key (hex): 98e97851ecbf16998a78dc35f87434cc
Plaintext: Hello Diffie-Hellman + AES-GCM!
Ciphertext (hex): cf205b08a868dd0781342fae29f46579b61d3d0a55bcad0ab5c96798f3a8223efb2de6c8a5dcb92d8a48b46ea319d
7
Decrypted: Hello Diffie-Hellman + AES-GCM!


...Program finished with exit code 0
Press ENTER to exit console.
```

Observations and Learnings

Observations:

1. Both parties successfully generated private and public key pairs.
2. Public keys were exchanged securely without revealing private keys.
3. The shared secret computed by Alice and Bob was identical.
4. The raw shared secret appeared random and suitable for cryptographic use.
5. A key derivation function (KDF) was necessary to transform the shared secret into a usable AES key.

Learnings:

1. Diffie–Hellman relies on the difficulty of the discrete logarithm problem.
2. Private keys remain confidential, never leaving the party.
3. Public keys can be shared over insecure channels safely.
4. KDFs are essential to convert shared secrets into symmetric encryption keys.
5. Modern implementations often use elliptic-curve variants (ECDH/ECDHE) for better efficiency and security.

Conclusion

The experiment demonstrated that two parties can securely establish a shared secret over an insecure channel without directly transmitting the key. By exchanging only public keys and keeping private keys secret, both parties independently derived the same secret key. Applying a KDF converted this shared secret into a usable AES-128 key for encryption and decryption. This experiment validates Diffie–Hellman as a foundational cryptographic protocol, highlighting its practical relevance in securing modern communication channels such as HTTPS, VPNs, and encrypted messaging systems.

Experiment - 9

Aim - Encrypt the given plaintext using the DES (Data Encryption Standard) encryption algorithm.

Procedure of the Experiment

Code –

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;
import java.security.SecureRandom;
import java.util.Base64;

public class DesDemo {
    public static void main(String[] args) throws Exception {
        SecureRandom rnd = new SecureRandom();
        // Generate a DES key (56-bit effective key)
        KeyGenerator keyGen = KeyGenerator.getInstance("DES");
        keyGen.init(56, rnd);
        SecretKey desKey = keyGen.generateKey();
        // Prepare IV for CBC mode (DES block size = 8 bytes)
        byte[] iv = new byte[8];
        rnd.nextBytes(iv);
        IvParameterSpec ivSpec = new IvParameterSpec(iv);
        // Plaintext to encrypt
        String plaintext = "Hello DES! This is a sample message.";
        System.out.println("Plaintext: " + plaintext);
        // Encrypt with DES/CBC/PKCS5Padding
        Cipher encryptCipher = Cipher.getInstance("DES/CBC/PKCS5Padding");
        encryptCipher.init(Cipher.ENCRYPT_MODE, desKey, ivSpec);
        byte[] ciphertext = encryptCipher.doFinal(plaintext.getBytes("UTF-8"));
        // Decrypt with DES/CBC/PKCS5Padding
        Cipher decryptCipher = Cipher.getInstance("DES/CBC/PKCS5Padding");
        decryptCipher.init(Cipher.DECRYPT_MODE, desKey, ivSpec);
        byte[] recovered = decryptCipher.doFinal(ciphertext);
        String decryptedText = new String(recovered, "UTF-8");
        // Print results
        System.out.println("DES key (Base64): " +
Base64.getEncoder().encodeToString(desKey.getEncoded()));
        System.out.println("IV (Base64):      " +
Base64.getEncoder().encodeToString(iv));
        System.out.println("Ciphertext (Base64): " +
Base64.getEncoder().encodeToString(ciphertext));
        System.out.println("Decrypted text: " + decryptedText);
    }
}
```

Output –

```
Plaintext: Hello DES! This is a sample message.
DES key (Base64): m0ACs4x6I84=
IV (Base64):      xVAbx4e++LU=
Ciphertext (Base64): lf15mmnZBUssSNo6ReG2M5UYselvckJsGvNGwSp00gfQr7mluYLMKw==
Decrypted text: Hello DES! This is a sample message.


...Program finished with exit code 0
Press ENTER to exit console.
```

Observations and Learnings

Observations:

1. A 56-bit DES secret key was successfully generated.
2. An 8-byte Initialization Vector (IV) was created for CBC mode.
3. Plaintext was encrypted into unreadable ciphertext.
4. Ciphertext differed on each execution due to the random IV.
5. Decryption successfully recovered the original plaintext.

Learnings:

1. DES operates on 64-bit blocks using a 56-bit effective key.
2. CBC mode requires an IV to produce different ciphertexts for identical plaintexts.
3. Java's Cipher API simplifies encryption and decryption with DES/CBC/PKCS5Padding.
4. Base64 encoding is useful for representing binary keys and ciphertext in text form.
5. DES is considered insecure today due to its small key size; modern systems prefer AES or 3DES.

Conclusion

The experiment successfully demonstrated DES encryption and decryption using CBC mode. The original message was securely transformed into ciphertext and accurately recovered with the same key and IV. This exercise reinforced concepts of block cipher encryption, IV usage, and key handling. It also highlighted DES's limitations: its short 56-bit key is vulnerable to brute-force attacks, making it unsuitable for modern cryptography. DES remains useful for educational purposes and understanding classical block cipher operations.

Experiment - 10

Aim - To perform and simulate Merkle–Damgård toy hash and length extension.

Procedure of the Experiment

Code –

```
# Toy Merkle-Damgård Hash & Length-Extension Demo
# Python 3 required.
from typing import Tuple

# Parameters
IV = 0x1A2B                    # 16-bit initial value
CHUNK_SIZE = 1                 # block size in bytes
WORD_MASK = 0xFFFF             # 16-bit mask
def compress(chaining: int, block_byte: int) -> int:
    """Toy compression function: non-linear but simple."""
    rot = ((chaining << 3) | (chaining >> (16-3))) & WORD_MASK
    out = (rot ^ (block_byte & 0xFF)) + ((chaining & 0x00FF) * (block_byte &
0x00FF))
    return out & WORD_MASK
def merkle_damgard_hash(message: bytes, iv: int = IV, chunk_size: int =
CHUNK_SIZE) -> int:
    msg = bytearray(message)
    orig_len = len(msg)
    msg.append(0x80)
    while (len(msg) + 2) % chunk_size != 0:
        msg.append(0x00)
    msg += (orig_len).to_bytes(2, 'little')
    chaining = iv & WORD_MASK
    for i in range(0, len(msg), chunk_size):
        block = msg[i:i+chunk_size]
        for b in block:
            chaining = compress(chaining, b)
    return chaining
def merkle_damgard_mac(secret: bytes, message: bytes) -> int:
    """Keyed MAC = hash(secret || message)."""
    return merkle_damgard_hash(secret + message)
def glue_padding_for_length(total_message_len: int, chunk_size: int =
CHUNK_SIZE) -> bytes:
    pad = bytearray()
    pad.append(0x80)
    while (total_message_len + len(pad) + 2) % chunk_size != 0:
        pad.append(0x00)
    pad += (total_message_len).to_bytes(2, 'little')
    return bytes(pad)
def continue_hash_from_state(chain_state: int, additional: bytes) -> int:
    chaining = chain_state & WORD_MASK
```

```
    for b in additional:
        chaining = compress(chaining, b)
    return chaining
def demo():
    secret = b"SECRET"
    message = b"comment1=10;uid=1000"
    suffix = b";admin=true"
    real_mac = merkle_damgard_mac(secret, message)
    guessed_secret_len = len(secret)
    total_len = guessed_secret_len + len(message)
    glue = glue_padding_for_length(total_len, CHUNK_SIZE)
    forged_message = message + glue + suffix
    new_total_len = total_len + len(glue) + len(suffix)
    final_padding = glue_padding_for_length(new_total_len, CHUNK_SIZE)
    forged_mac = continue_hash_from_state(real_mac, suffix + final_padding)
    server_mac_on_forged = merkle_damgard_mac(secret, forged_message)
    print(f"Original MAC = 0x{real_mac:04x}")
    print(f"Forged MAC  = 0x{forged_mac:04x}")
    print(f"Server MAC  = 0x{server_mac_on_forged:04x}")
    print("SUCCESS: Length extension attack works!" if forged_mac ==
server_mac_on_forged else "FAIL")
if __name__ == "__main__":
    demo()
```

Output –

Output Clear

```
=== Toy Merkle-Damgård Hash & Length Extension Demo ===
IV = 0x1a2b, chunk size = 1 byte(s), digest width = 16 bits

Server computes MAC = H(secret || message) = 0xa335
Attacker (demo) guesses secret length = 6 bytes
Glue padding (hex) for total length 26 bytes: 801a00 (len = 3 bytes)

Attacker forges message = message || glue || suffix (len = 34 bytes)
Forged message (hex):
    636f6d6d656e74313d31303b7569643d31303030801a003b61646d696e3d74727
    565
Attacker computes forged MAC = 0x6e79
Server's MAC for secret || forged_message = 0x6e79

SUCCESS: Length extension attack works — attacker computed a valid
    MAC without knowing the secret.

Server actually hashed bytes (hex):
    534543524554636f6d6d656e74313d31303b7569643d31303030801a003b61646
    d696e3d74727565 (len = 40 bytes)

--- Brute-force secret-length demonstration (attacker tries lengths 1
    ..12) ---
```

```
SUCCESS: Length extension attack works — attacker computed a valid
    MAC without knowing the secret.

Server actually hashed bytes (hex):
    534543524554636f6d6d656e74313d31303b7569643d31303030801a003b61646
    d696e3d74727565 (len = 40 bytes)

--- Brute-force secret-length demonstration (attacker tries lengths 1
    ..12) ---
guess secret_len= 1 -> forged_mac=0x6339 server_mac=0xb946 : BAD
guess secret_len= 2 -> forged_mac=0x6579 server_mac=0x704f : BAD
guess secret_len= 3 -> forged_mac=0x67b9 server_mac=0xa847 : BAD
guess secret_len= 4 -> forged_mac=0x69f9 server_mac=0x297a : BAD
guess secret_len= 5 -> forged_mac=0x6c39 server_mac=0xdd40 : BAD
guess secret_len= 6 -> forged_mac=0x6e79 server_mac=0x6e79 : OK
guess secret_len= 7 -> forged_mac=0x70b9 server_mac=0x3a6b : BAD
guess secret_len= 8 -> forged_mac=0x72f9 server_mac=0xe559 : BAD
guess secret_len= 9 -> forged_mac=0x7539 server_mac=0x2d56 : BAD
guess secret_len=10 -> forged_mac=0x7779 server_mac=0x494c : BAD
guess secret_len=11 -> forged_mac=0x79b9 server_mac=0x5e57 : BAD
guess secret_len=12 -> forged_mac=0x7bf9 server_mac=0x3173 : BAD

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

Observations and Learnings

Observations:

1. The demo showed that an attacker can append data (suffix) to a message and forge a valid MAC without knowing the secret.
2. This works because Merkle–Damgård hash structure allows continuing from the internal state (observed MAC).
3. The only unknown for the attacker is the secret length, which can be brute-forced if short.

Learnings:

1. Naive constructions like MAC = H(secret || message) are vulnerable to length-extension attacks.
2. HMAC prevents this by using a two-layer hash structure.
3. Even strong hashes (MD5, SHA-1, SHA-2) are unsafe if used incorrectly.
4. Security requires both a strong algorithm and correct construction.

Conclusion

Length-extension attacks exploit predictable padding and chaining of Merkle–Damgård hashes. They allow an attacker to forge a valid MAC without knowing the secret, provided the internal state is known. To prevent such attacks, always use HMAC or authenticated encryption schemes like AES-GCM or ChaCha20-Poly1305 rather than custom MAC constructions.