R.M.K. COLLEGE OF
ENGINEERING AND TECHNOLOGY

(An Autonomous Institution)
(Sponsored by LAKSHMIKANNTHAMMAL EDUCATION TRUST)
Approved by All India Council for Technical Education, New Delhi and
Affiliated to Anna University, Chennai

All the Eligible Programs Accredited by NBA – NAAC with "A" Grade & ISO 9001 : 2005 Certified

R.S.M. Nagar, Puduvoyal - 601 206. Gummidipoondi Tk., Thiruvallur Dist. Tamilnadu, India.

Department : **COMPUTER SCIENCE AND ENGINEERING (CYBER SECURITY)**

Laboratory :

Semester : **V**

Certified that this is a bonafide record work done by …………..……………………

with Roll / Reg. Number.. He / She is a student

of ....in

the **R.M.K. COLLEGE OF ENGINEERING AND TECHNOLOGY, Puduvoyal.**

Faculty-in-Charge **Head of the Department**

Internal Examiner Date: **External Examiner**

INDEX

Exp. No.: 1 Date:	**Implementing Core Security Principles in a Sample Application**

AIM:

To practice implementing core security principles like data encryption and input validation in a sample C++ application to protect sensitive information and prevent vulnerabilities.

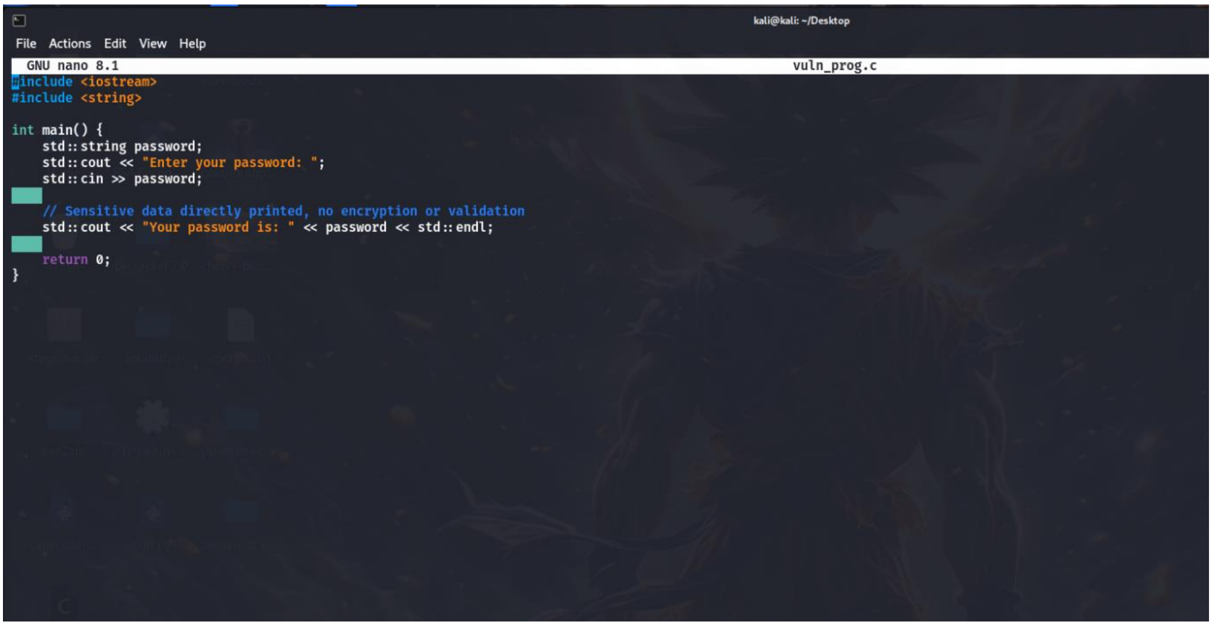
PROCEDURES:

1. Write a basic C++ program that takes user input and processes sensitive information.
2. Identify potential vulnerabilities, such as improper input validation and unencrypted data handling.
3. Implement encryption to protect sensitive data and ensure secure communication.
4. Apply input validation to prevent malicious input, such as SQL injection or buffer overflow.
5. Compile and run the program with secure code practices in place to verify proper security implementation.

PROGRAM:

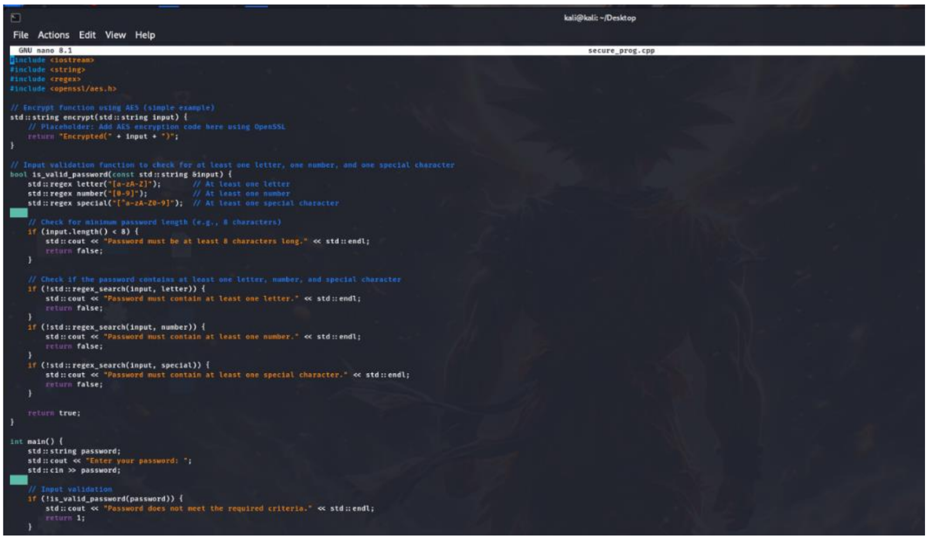
Vulnerable Program (without encryption and input validation):

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string password;
    cout << "Enter your password: ";
    cin >> password;

    // Sensitive data directly printed, no encryption or validation
    cout << "Your password is: " << password << endl;

    return 0;
}
```



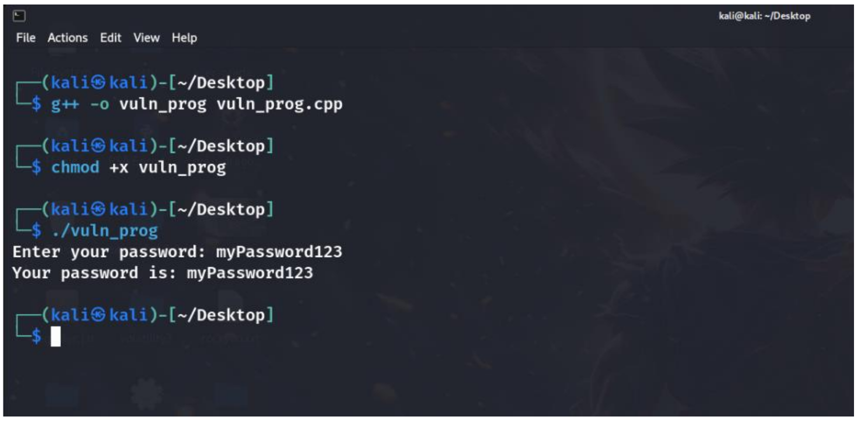
Secure Program (with encryption and input validation):

```
#include <iostream>
#include <string>
#include <regex>
#include <openssl/aes.h>
Using namespace std;

// Encrypt function using AES (simple example)

string encrypt(string input) {
    return "Encrypted(" + input + ")";
}

// Input validation function to check for at least one letter, one number, and one special character
bool is_valid_password(const string &input) {
    regex letter("[a-zA-Z]");       // At least one letter
    regex number("[0-9]");          // At least one number
    regex special("[^a-zA-Z0-9]");  // At least one special character

    // Check for minimum password length (e.g., 8 characters)
    if (input.length() < 8) {
        cout << "Password must be at least 8 characters long." << endl;
        return false;
    }

    // Check if the password contains at least one letter, number, and special character
    if (!regex_search(input, letter)) {
        cout << "Password must contain at least one letter." << endl;
        return false;
    }
    if (!regex_search(input, number)) {
        cout << "Password must contain at least one number." << endl;
        return false;
    }
    if (!regex_search(input, special)) {
        cout << "Password must contain at least one special character." << endl;
        return false;
    }

    return true;
}

int main() {
    string password;
    cout << "Enter your password: ";
    cin >> password;

    // Input validation
    if (!is_valid_password(password)) {
        cout << "Password does not meet the required criteria." << endl;
        return 1;
    }

    string encrypted_password = encrypt(password);
    cout << "Your encrypted password is: " << encrypted_password << endl;

    return 0;
}
```

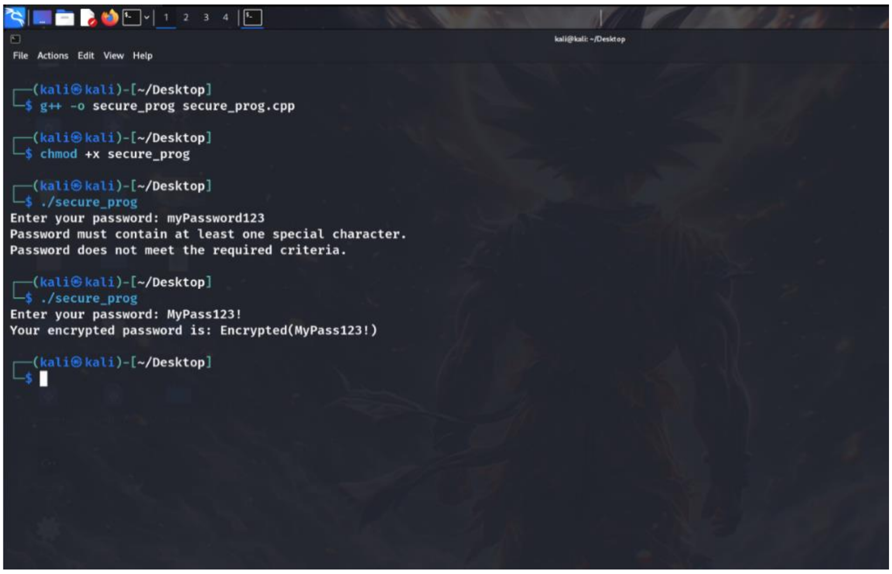


OUTPUT:

Vulnerable Program Output:



Enter your password: myPassword123
Your password is: myPassword123



Enter your password: MyPass123!
Your encrypted password is: Encrypted(MyPass123!)

RESULT:

The vulnerable version of the program allowed sensitive data (the password) to be exposed in plaintext, and there was no input validation to prevent malicious input. By implementing encryption and input validation, the secure version successfully protects sensitive information and prevents invalid inputs from being processed.

Ex. No. : 2 **Date:**	**Integrating Security Practices into the SDLC Phases**

AIM:

To understand and integrate essential security practices at each stage of the Software Development Life Cycle (SDLC) to reduce vulnerabilities early in development.

PROCEDURES:

Requirements Gathering and Analysis:

- Identify security requirements alongside functional requirements.
- Perform **threat modeling** to anticipate security threats and plan mitigations early.
- Example: If the application handles sensitive data (e.g., passwords), plan for encryption and secure storage.

Design Phase:

- Implement secure design principles such as **least privilege**, **separation of duties**, and **fail-safe defaults**.
- Design using **input validation**, **authentication**, and **authorization controls**.
- Example: For password handling, design an architecture that includes hashing (e.g., SHA-256 with salt) rather than plain text storage.

Development Phase:

- Follow **secure coding practices** such as data validation, output encoding, and avoiding hard-coded secrets.
- Use libraries that offer built-in security mechanisms, like cryptography libraries for data encryption.
- Example: Integrate **parameterized queries** to prevent SQL Injection and sanitize user inputs to avoid XSS attacks.

Testing Phase:

- Conduct **security-focused testing** like vulnerability assessments and **penetration testing**.
- Perform **code review** and use tools for static analysis to detect vulnerabilities.
- Example: Test the application against common security risks (e.g., SQL Injection, XSS) using tools like OWASP ZAP.

Deployment Phase:

- Ensure secure configuration of servers, databases, and application environments.
- Use **environment-specific configurations** for sensitive information, and ensure sensitive data is encrypted during deployment.
- Example: Use HTTPS for data transmission and secure server configurations, like disabling unused ports.

Maintenance Phase:

- Monitor and apply security patches regularly.
- Conduct periodic security audits and update the application to address new vulnerabilities.
- Example: Review logs for unusual activity and patch dependencies to their latest secure versions.

PROGRAM:

Development Phase:

```
// Account Creation
function createAccount(username, password):
    if not isValidInput(username, password):
        return "Invalid input"
    hashedPassword = hashPassword(password)  // Hash password with SHA-256
    storeInDatabase(username, hashedPassword)

// Login
function login(username, password):
    if not isValidInput(username, password):
        return "Invalid input"
    storedHash = getPasswordFromDatabase(username)
    if verifyPassword(password, storedHash):
        sessionToken = createSessionToken(username)
        storeSessionToken(sessionToken)
        setSecureCookie("session_token", sessionToken)
        return "Login successful"
    else:
        return "Invalid username or password"

// Input Validation
function isValidInput(input):
    return regexCheck(input, "[safe_characters_only]")  // Prevents SQL Injection, XSS

// Password Hashing
function hashPassword(password):
    salt = generateRandomSalt()
    return SHA256(password + salt)  // Simple example of hashing with salt
```

RESULT:

By integrating security practices at each phase of the SDLC, potential vulnerabilities can be mitigated early, leading to a secure and resilient application

EX. NO. : 3 **DATE:**	**Recognizing and Mitigating Common String Manipulation Errors**

AIM:

To identify common string manipulation errors in C programs, such as buffer overflows, and understand methods to mitigate these vulnerabilities through secure coding practices.

PROCEDURES:

1. Write a vulnerable C program with common string manipulation errors (like a buffer overflow).

2. Compile and run the vulnerable program.

3. Exploit the vulnerability by providing specific input from the terminal that triggers the vulnerability (e.g., buffer overflow).

4. Write a secure version of the program by applying secure coding practices.

5. Compile and run the secure version of the program to confirm that the vulnerability is mitigated..

PROGRAM:

<u>Vulnerable Program</u>

```
#include <stdio.h>
#include <string.h>

void vulnerable_function(char *input) {
    char buffer[10];
    strcpy(buffer, input);  // Vulnerable to buffer overflow
    printf("Buffer content: %s\n", buffer);
}

int main(int argc, char *argv[]) {
    if (argc > 1) {
        vulnerable_function(argv[1]);
    } else {
        printf("Please provide an argument\n");
    }
    return 0;
}
```

Exploiting the Vulnerability

1. Compile the program:

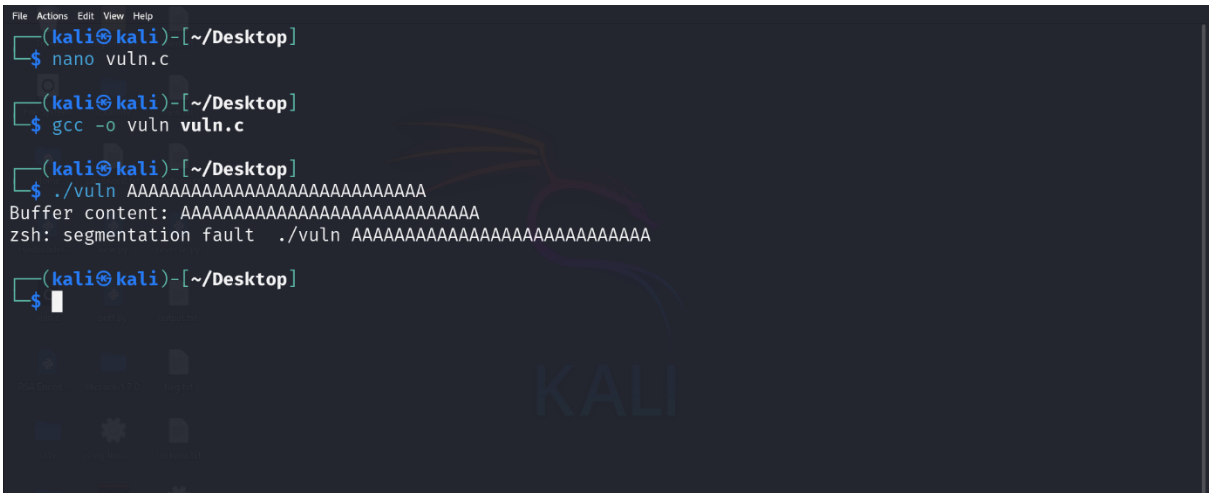
gcc -o vuln vuln.c

2.Exploit the program by providing a long string as input:

./vuln AAAAAAAAAAAAAAAAAAAAAAAAAAAA

3.This will trigger a buffer overflow, as the input is larger than the allocated buffer (10 bytes).

OUTPUT:



Secure Coding Version:

```
#include <stdio.h>
#include <string.h>

void secure_function(char *input) {
    char buffer[10];
    // Use strncpy to prevent buffer overflow
    strncpy(buffer, input, sizeof(buffer) - 1);
    buffer[9] = '\0';  // Ensuring null termination
    printf("Buffer content: %s\n", buffer);
}

int main(int argc, char *argv[]) {
    if (argc > 1) {
        secure_function(argv[1]);
    } else {
        printf("Please provide an argument\n");
    }
    return 0;
}
```

Running the Secure Program

1.Compile the secure version:

gcc -o sec sec.c

2. Test with a long input:

./sec AAAAAAAAAAAAAAAAAAAAAAAAAAAA

3. The program will truncate the input to fit within the buffer size, mitigating the buffer overflow.

OUTPUT:

RESULT:

The vulnerable program allowed for a buffer overflow when an oversized input was provided. By using `strncpy` and ensuring proper null termination, the secure version successfully mitigated the overflow vulnerability.

Ex. No. : 4 Date:	**Understanding Pointer Vulnerabilities and Applying Mitigation Strategies**

AIM:

To identify and understand common pointer vulnerabilities in C programs and apply secure coding practices to mitigate these vulnerabilities.

.

PROCEDURES:

1. Write a vulnerable C program that demonstrates pointer-related vulnerabilities.
2. Compile and run the vulnerable program on a Linux terminal and observe how exploitation is possible.
3. Analyze the root cause of the vulnerability.
4. Modify the code using secure coding practices.
5. Compile and run the secure version of the program.
6. Compare the outputs to demonstrate the effectiveness of mitigation strategies.

PROGRAM:

<u>Vulnerable Program (vuln.c)</u>

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *ptr = (int*)malloc(sizeof(int)); // Allocate memory
   *ptr = 42; // Assign a value to the allocated memory
    printf("Value: %d\n", *ptr);
    free(ptr);
    printf("Value after free: %d\n", *ptr); // Use-After-Free
    *ptr = 100; // Writing to freed memory
    // Program may still proceed, leading to undefined behavior
    printf("New value: %d\n", *ptr); // This could crash the program or give garbage
    return 0;
}
```

Exploiting the Vulnerability

1. Compile the vulnerable program:

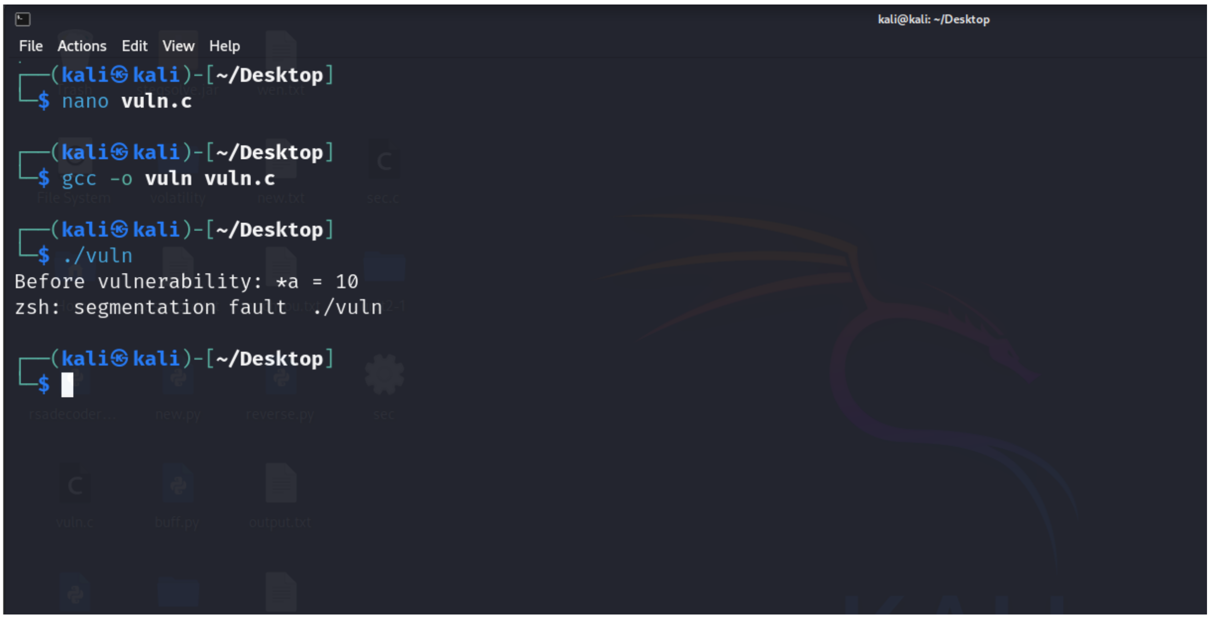
   gcc vuln.c -o vuln

2. Run the program with a short input:

   ./vuln

3. The program may produce undefined behavior or crash after attempting to dereference a dangling pointer result in either a segmentation fault or corrupted data being printed.

OUTPUT:



Secure Code

```
#include <stdio.h>
#include <stdlib.h>
void secure_function(int **ptr) {
    if (*ptr != NULL) {
        free(*ptr);  // Free the pointer only if it's not already NULL
        *ptr = NULL; // Set the pointer to NULL after freeing
    }
}
int main() {
    int *a = (int *)malloc(sizeof(int));
    *a = 10;
    printf("Before secure function: *a = %d\n", *a);
```

secure_function(&a);

    if (a == NULL)

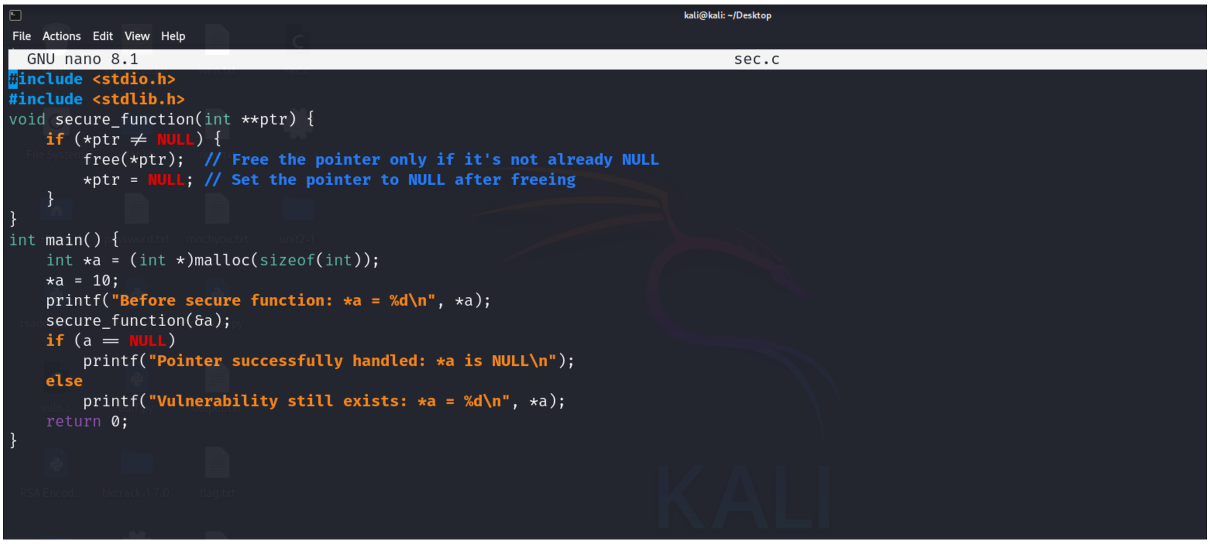
        printf("Pointer successfully handled: *a is NULL\n");

    else

        printf("Vulnerability still exists: *a = %d\n", *a);

    return 0;

}



Secure Code Output

1. Compile the secure program

   gcc sec.c -o sec

2. Run the secure program

   ./sec

3. The message `"Pointer successfully handled: *a is NULL"` confirms that the double pointer vulnerability has been mitigated by ensuring the pointer is nullified after being freed.

RESULT:

In this experiment, we observed the dangers of dangling pointers and how they can lead to undefined behavior. By implementing secure coding practices, such as nullifying pointers after freeing, we successfully mitigated this vulnerability.

Ex. No. : 5 Date:	**Identifying and Fixing Common Dynamic Memory Management Errors**

AIM:

To identify common dynamic memory management errors, such as memory leaks and invalid memory accesses in C programs, and apply techniques to fix these issues using proper memory allocation and deallocation practices.

.

PROCEDURES:

1. Write a C program that deliberately includes common dynamic memory management errors (e.g., memory leaks, double free, or invalid memory access).
2. Compile and run the program.
3. Analyze the program's behavior to identify the memory management issues using tools like valgrind or built-in sanitizers.
4. Refactor the program to fix the errors by applying proper memory management practices (such as freeing allocated memory correctly, avoiding invalid accesses, and preventing double-free errors).
5. Compile and run the refactored program to verify that the memory errors are resolved.

PROGRAM:

<u>Vulnerable Program</u>

```
#include <stdio.h>
#include <stdlib.h>

void memory_leak() {
    int *leak = (int *)malloc(sizeof(int) * 10);  // Memory allocated but not freed
}

void invalid_access() {
    int *array = (int *)malloc(sizeof(int) * 5);
    array[5] = 10;  // Invalid access outside the allocated memory
}

int main() {
    memory_leak();
    invalid_access();
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

void memory_leak() {
    int *leak = (int *)malloc(sizeof(int) * 10);  // Memory allocated but not freed
}

void invalid_access() {
    int *array = (int *)malloc(sizeof(int) * 5);
    array[5] = 10;  // Invalid access outside the allocated memory
}

int main() {
    memory_leak();
    invalid_access();
    return 0;
}

```

Identifying Errors:

1. Compile the program:

gcc -g -o vuln vuln.c

2.Run the program with valgrind to detect memory management issues:

valgrind --leak-check=yes ./vuln

OUTPUT:

```
┌──(kali㉿kali)-[~/Desktop/securecodeing/unit-4]
└─$ gcc -g -o vuln vuln.c

┌──(kali㉿kali)-[~/Desktop/securecodeing/unit-4]
└─$ valgrind --leak-check=yes ./vuln
==36368== Memcheck, a memory error detector
==36368== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==36368== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==36368== Command: ./vuln
==36368==
==36368== Invalid write of size 4
==36368==    at 0x109170: invalid_access (vuln.c:10)
==36368==    by 0x109190: main (vuln.c:15)
==36368==  Address 0x4a530c4 is 0 bytes after a block of size 20 alloc'd
==36368==    at 0x4840808: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==36368==    by 0x109163: invalid_access (vuln.c:9)
==36368==    by 0x109190: main (vuln.c:15)
==36368==
==36368==
==36368== HEAP SUMMARY:
==36368==     in use at exit: 60 bytes in 2 blocks
==36368==   total heap usage: 2 allocs, 0 frees, 60 bytes allocated
==36368==
==36368== 20 bytes in 1 blocks are definitely lost in loss record 1 of 2
==36368==    at 0x4840808: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==36368==    by 0x109163: invalid_access (vuln.c:9)
==36368==    by 0x109190: main (vuln.c:15)
==36368==
==36368== 40 bytes in 1 blocks are definitely lost in loss record 2 of 2
==36368==    at 0x4840808: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==36368==    by 0x10914A: memory_leak (vuln.c:5)
==36368==    by 0x109186: main (vuln.c:14)
==36368==
==36368== LEAK SUMMARY:
==36368==    definitely lost: 60 bytes in 2 blocks
==36368==    indirectly lost: 0 bytes in 0 blocks
==36368==      possibly lost: 0 bytes in 0 blocks
==36368==    still reachable: 0 bytes in 0 blocks
==36368==         suppressed: 0 bytes in 0 blocks
==36368==
==36368== For lists of detected and suppressed errors, rerun with: -s
==36368== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
```

Secure Coding Version:

```
#include <stdio.h>
#include <stdlib.h>

void fixed_memory_leak() {
    int *leak = (int *)malloc(sizeof(int) * 10);
    if (leak != NULL) {
        // Perform operations with leak
        free(leak);  // Free allocated memory to prevent leaks
    }
}

void fixed_invalid_access() {
    int *array = (int *)malloc(sizeof(int) * 5);
    if (array != NULL) {
        array[4] = 10;  // Corrected access within allocated memory
        free(array);  // Free the memory after use
    }
}

int main() {
    fixed_memory_leak();
    fixed_invalid_access();
    return 0;
}
```

Running the Secure Program

1.Compile the secure version:

gcc -o fixed fixed.c

2. Run the program with valgrind to detect memory management issues:

valgrind --leak-check=yes ./fixed

```
┌──(kali㉿kali)-[~/Desktop/securecodeing/unit-4]
└─$ gcc -g -o fixed fixed.c

┌──(kali㉿kali)-[~/Desktop/securecodeing/unit-4]
└─$ valgrind --leak-check=yes ./fixed
==38168== Memcheck, a memory error detector
==38168== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==38168== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==38168== Command: ./fixed
==38168==
==38168==
==38168== HEAP SUMMARY:
==38168==     in use at exit: 0 bytes in 0 blocks
==38168==   total heap usage: 2 allocs, 2 frees, 60 bytes allocated
==38168==
==38168== All heap blocks were freed -- no leaks are possible
==38168==
==38168== For lists of detected and suppressed errors, rerun with: -s
==38168== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

RESULT:

The original program had issues with memory leaks and invalid memory access. After applying proper memory management, these issues were resolved, making the program safe and efficient..

.

Ex. No. : 6 Date:	**Understanding and Implementing Safe Dynamic Memory Management Practices in C++ and Java**

AIM:

To understand dynamic memory management in C++ and Java, recognize common errors such as memory leaks, and implement secure coding practices to manage memory efficiently and safely.

PROCEDURE:

For C++:

I)Write a program that uses dynamic memory allocation.

Allocate memory dynamically using new or malloc().

Create a scenario where memory is allocated but not freed (memory leak).

ii)Compile and run the program.

Observe the effects of memory mismanagement.

iii)Modify the program to properly free allocated memory.

Use delete or free() to deallocate memory.

iv)Use smart pointers for automatic memory management.

Implement the program using smart pointers (std::unique_ptr, std::shared_ptr) to automatically manage memory and prevent leaks.

v)Compile and run the improved version.

Confirm that memory is properly managed without leaks.

For Java:

I)Write a Java program that creates objects dynamically.

Simulate memory mismanagement by continuously creating objects and holding references.

ii)Run the program and monitor memory usage.

Observe how the Java garbage collector handles memory when objects are not explicitly freed.

iii)Use best practices for memory management in Java.

Dereference unused objects by setting their references to null.

Implement finalize() and try-with-resources for automatic resource management.

iv)Run the optimized program to observe improved memory management.

Ensure that memory usage is reduced, and objects are freed efficiently.

OUTPUT:

<u>Vulnerable C++ Program (Memory Leak Example):</u>

```
#include <iostream>

void createLeak() {
   int* ptr = new int[10];  // Dynamically allocated memory
   // No delete operation to free memory
}

int main() {
   createLeak();
   std::cout << "Memory leak created.\n";
   return 0;
}
```

```cpp
#include <iostream>

void createLeak() {
    int* ptr = new int[10];  // Dynamically allocated memory
    // No delete operation to free memory
}

int main() {
    createLeak();
    std::cout << "Memory leak created.\n";
    return 0;
}
//Memory Leak Example
```

<u>Running the vulnerable Program:</u>

1.Compile the Program:

gcc -o vuln vuln.cpp

2. Run the program:

./vuln

OUTPUT:

```
┌──(kali㉿kali)-[~/Desktop/securecodeing]
└─$ g++ -o vuln vuln.cpp

┌──(kali㉿kali)-[~/Desktop/securecodeing]
└─$ ./vuln
Memory leak created.
```

<u>Improved C++ Program (Using Delete):</u>

```cpp
#include <iostream>

void noLeak() {
    int* ptr = new int[10];  // Dynamically allocated memory
    delete[] ptr;  // Freeing memory
}

int main() {
    noLeak();
    std::cout << "Memory safely managed.\n";
    return 0;
}
```

```cpp
#include <iostream>

void noLeak() {
    int* ptr = new int[10];  // Dynamically allocated memory
    delete[] ptr;  // Freeing memory
}

int main() {
    noLeak();
    std::cout << "Memory safely managed.\n";
    return 0;
}
//Using Delete
```

<u>Running the secured Program(using delete):</u>

1.Compile the Program:

g++ -o improveddelete improveddelete.cpp

2. Run the program:

./improveddelete

OUTPUT:

```
┌──(kali㉿kali)-[~/Desktop/securecodeing]
└─$ g++ -o improveddelete improveddelete.cpp

┌──(kali㉿kali)-[~/Desktop/securecodeing]
└─$ ./improveddelete
Memory safely managed.
```

<u>C++ Program with Smart Pointers:</u>

```cpp
#include <iostream>
#include <memory>

void safeMemory() {
    std::unique_ptr<int[]> ptr = std::make_unique<int[]>(10);  // Automatically managed memory
}

int main() {
    safeMemory();
    std::cout << "Memory managed using smart pointers.\n";
    return 0;
}
```

```cpp
#include <iostream>
#include <memory>

void safeMemory() {
    std::unique_ptr<int[]> ptr = std::make_unique<int[]>(10);  // Automatically managed memory
}

int main() {
    safeMemory();
    std::cout << "Memory managed using smart pointers.\n";
    return 0;
}
//smart pointer
```

Running the secured Program(using smart pointer):

1.Compile the Program:

g++ -o improvedsmart improvedsmart.cpp

2. Run the program:

./improvedsmart

OUTPUT:

```
┌──(kali㉿kali)-[~/Desktop/securecodeing]
└─$ g++ -o improvedsmart improvedsmart.cpp

┌──(kali㉿kali)-[~/Desktop/securecodeing]
└─$ ./improvedsmart
Memory managed using smart pointers.
```

Java Program (Without Memory Management):

```java
public class MemoryLeak {
    public static void main(String[] args) {
        for (int i = 0; i < 1000; i++) {
            String str = new String("Memory leak");  // Objects created without dereferencing
        }
        System.out.println("Potential memory leak.");
    }
}
```

```java
public class MemoryLeak {
    public static void main(String[] args) {
        for (int i = 0; i < 1000; i++) {
            String str = new String("Memory leak");  // Objects created without dereferencing
        }
        System.out.println("Potential memory leak.");
    }
}
//Without Memory Management
```

Running the vuln Program:

1.Compile the Program:

javac MemoryLeak.java

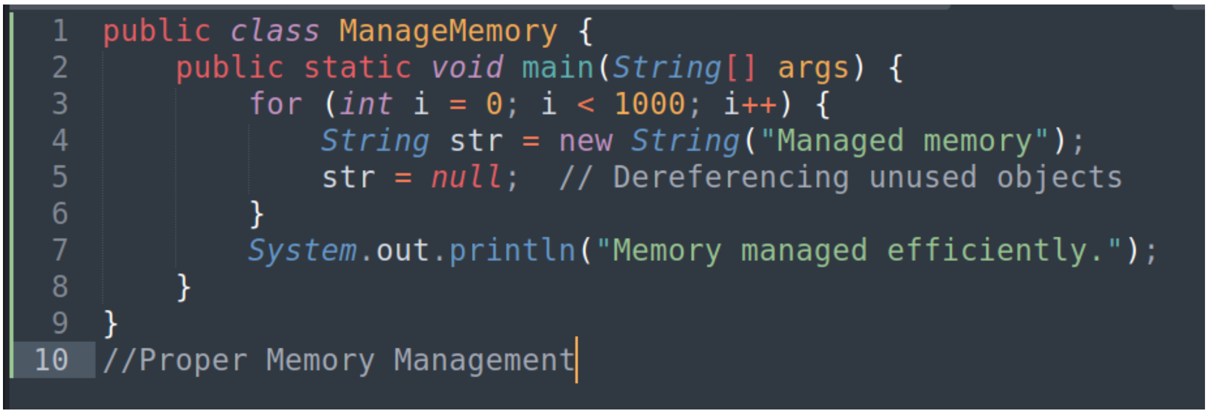
2. Run the program:

java MemoryLeak

```
┌──(kali㉿kali)-[~/Desktop/securecodeing]
└─$ java MemoryLeak.java
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Potential memory leak.
```

Java Program (With Memory Management):

```
public class ManageMemory {
    public static void main(String[] args) {
        for (int i = 0; i < 1000; i++) {
            String str = new String("Managed memory");
            str = null;  // Dereferencing unused objects
        }
        System.out.println("Memory managed efficiently.");
    }
}
//Proper Memory Management
```



```
public class ManageMemory {
    public static void main(String[] args) {
        for (int i = 0; i < 1000; i++) {
            String str = new String("Managed memory");
            str = null;  // Dereferencing unused objects
        }
        System.out.println("Memory managed efficiently.");
    }
}
//Proper Memory Management
```

Running the Secure Program:

1.Compile the Program:

javac Managememory.java

2. Run the program:

java ManageMemory

OUTPUT:

```
┌──(kali㉿kali)-[~/Desktop/securecodeing]
└─$ java ManageMemory.java
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Memory managed efficiently.
```

RESULT:

By employing proper memory management techniques, both C++ and Java programs were able to mitigate the risks associated with dynamic memory mismanagement

Ex. No. : 07 Date:	**How improper quoting can lead to SQL injection and how to prevent it using parameterized queries**

AIM:

To understand how improper quoting can lead to SQL injection and how to prevent it using parameterized queries.

PROCEDURE:

1. **Set Up Development Environment**:
   - Ensure Node.js and npm are installed on your Linux machine.
   - Create a new project directory named sql-injection-project.
2. **Initialize Node.js Project**:
   - Run npm init -y to create a package.json file.
   - Install required packages with:

     npm install express sqlite3 body-parser
3. **Create the Server File**:
   - Create a file named server.js.
   - Write the server code to handle SQL queries, both vulnerable and secure.
4. **Create the Public Directory and HTML File**:
   - Create a directory named public.
   - Inside this directory, create index.html to serve as the front-end interface for login.
5. **Create CSS File**:
   - In the public directory, create a style.css file to style the HTML page.
6. **Create SQLite Database**:
   - Create the SQLite database file database.db.
   - Set up a table for users with sample data.
7. **Run the Application**:
   - Start the Node.js server with node server.js.
   - Open a web browser and access the application at **http://localhost:3000**.

PROGRAM:

1. server.js

```
const express = require('express');
const bodyParser = require('body-parser');
const sqlite3 = require('sqlite3').verbose();

const app = express();
const db = new sqlite3.Database('./database.db');
app.use(bodyParser.urlencoded({ extended: true }));
app.use(express.static('public'));
app.get('/', (req, res) => {
  res.sendFile(__dirname + '/public/index.html');
})
app.post('/login-vulnerable', (req, res) => {
  const username = req.body.username;
  const password = req.body.password;
  const query = `SELECT * FROM users WHERE username = '${username}' AND password = '${password}'`;
  db.all(query, [], (err, rows) => {
    if (err) {
      throw err;
    }
```

```
 if (rows.length > 0) {
            res.send('Login successful!');
        } else {
            res.send('Invalid credentials');
        }
    });
});
app.post('/login-secure', (req, res) => {
    const username = req.body.username;
    const password = req.body.password;
    const query = `SELECT * FROM users WHERE username = ? AND password = ?`;
    db.all(query, [username, password], (err, rows) => {
        if (err) {
            throw err;
        }
        if (rows.length > 0) {
            res.send('Login successful!');
        } else {
            res.send('Invalid credentials');
        }
    });
});
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`);
});
```

2. public/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>SQL Injection Test</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <h1>SQL Injection Test</h1>
    <h2>Vulnerable Login</h2>
    <form action="/login-vulnerable" method="POST">
        <input type="text" name="username" placeholder="Username" required>
        <input type="password" name="password" placeholder="Password" required>
        <button type="submit">Login (Vulnerable)</button>
    </form>

    <h2>Secure Login</h2>
    <form action="/login-secure" method="POST">
        <input type="text" name="username" placeholder="Username" required>
        <input type="password" name="password" placeholder="Password" required>
        <button type="submit">Login (Secure)</button>
    </form>
</body>
```

</html>

3. public/style.css

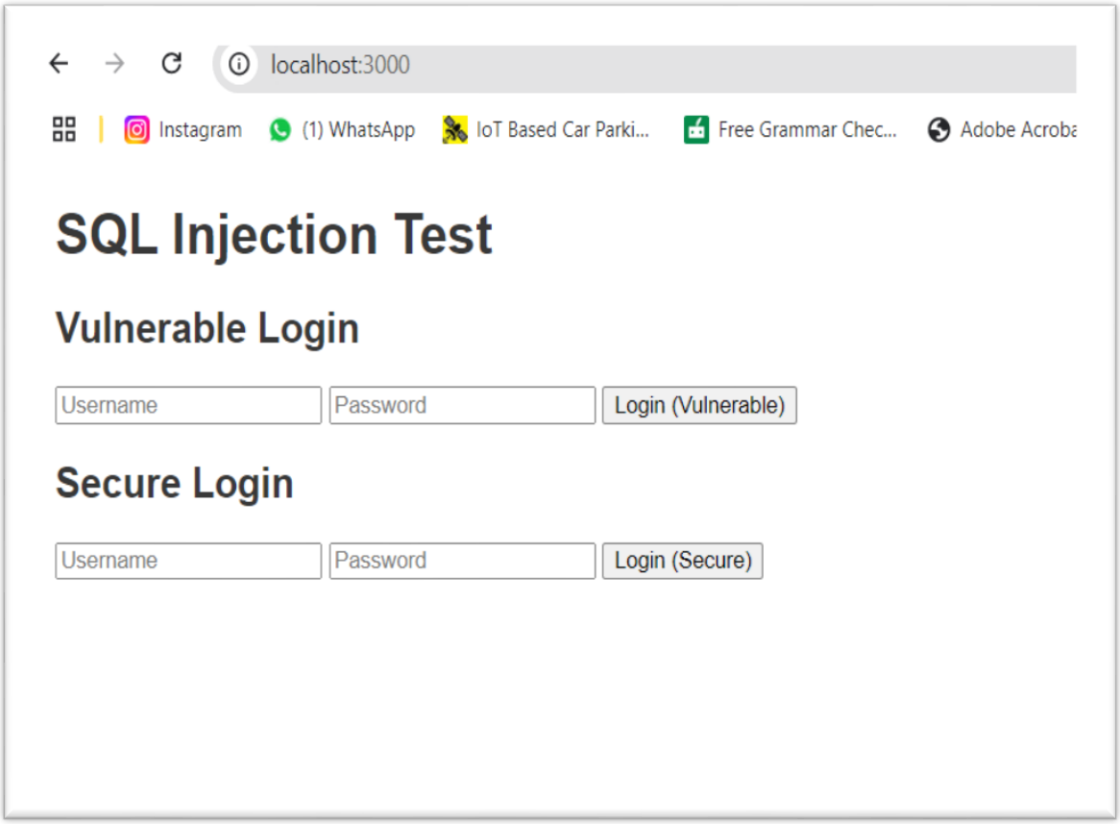
```
body {
   font-family: Arial, sans-serif;
   margin: 20px;
}
h1, h2 {
   color: #333;
}
form {
   margin-bottom: 20px;
}
```

4. SQLite Commands

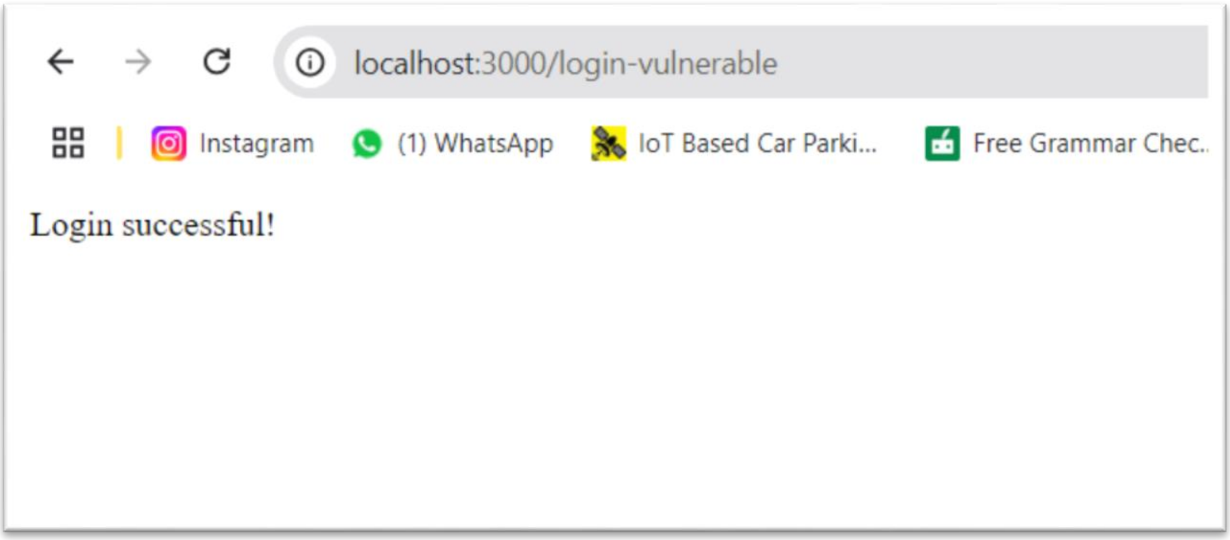
```
CREATE TABLE users (
   id INTEGER PRIMARY KEY AUTOINCREMENT,
   username TEXT NOT NULL,
   password TEXT NOT NULL
);
INSERT INTO users (username, password) VALUES ('admin', 'password123');
INSERT INTO users (username, password) VALUES ('user', 'user123');
.exit
```

OUTPUT:

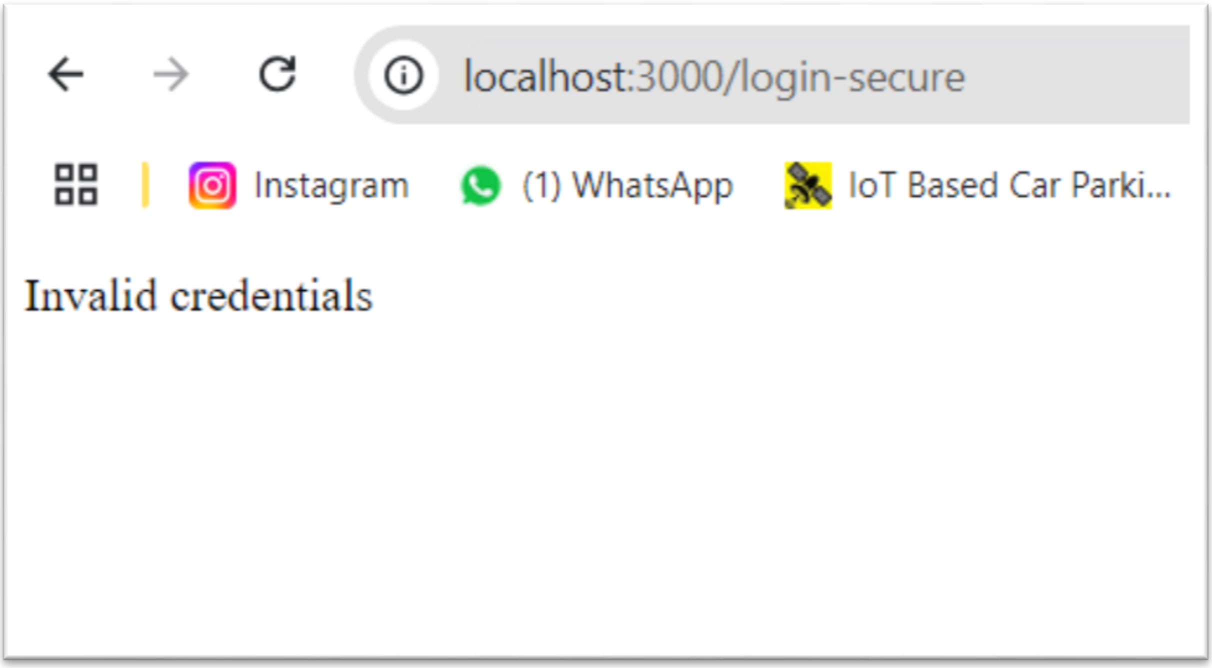
1. **Application Access**:
   - Navigate to **http://localhost:3000** in a web browser.



2. **Testing Vulnerable Login**:
   - Enter the following in the username field of the **Vulnerable Login**:
     admin' OR '1'='1
   - The application should return "Login successful!" indicating that SQL injection was successful.



3. **Testing Secure Login**:
   - Use the same input in the **Secure Login** section.
   - The application should return "Invalid credentials," showing that the parameterized query protects against SQL injection.



RESULT :

The program was successfully executed and verified with the output.

Ex. No. : 08 Date:	**Understanding and Preventing Cross-Site Scripting (XSS) Attacks through Input Validation and Output Encoding**

AIM:

To understand the mechanisms of Cross-Site Scripting (XSS) attacks and to demonstrate effective prevention techniques by implementing input validation and output encoding.

PROCEDURE:

1. **Set Up the Development Environment**:
    - Ensure Node.js and npm are installed on your Linux machine.
    - Create a new project directory named xss-attack-project.
2. **Initialize Node.js Project**:
    - Run npm init -y to create a package.json file.
    - Install the required packages:

      npm install express body-parser
3. **Create the Server File**:
    - Create a file named server.js to handle the server logic and display both a vulnerable and a secure implementation.
4. **Create the Public Directory and HTML File**:
    - Create a public directory.
    - Inside this directory, create an index.html file that will serve as the front-end interface for user input.
5. **Create CSS File**:
    - In the public directory, create a style.css file to style the HTML page.
6. **Run the Application**:
    - Start the Node.js server with node server.js.
    - Open a web browser and access the application at **http://localhost:4000.**

PROGRAM:

1. server.js

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();
app.use(bodyParser.urlencoded({ extended: true }));
app.use(express.static('public'));

app.post('/vulnerable', (req, res) => {
  const userInput = req.body.userInput;
  res.send(`<h1>Vulnerable Response</h1><p>${userInput}</p>`);
});
app.post('/secure', (req, res) => {
  const userInput = req.body.userInput;
  const escapedInput = userInput.replace(/</g, "&lt;").replace(/>/g, "&gt;");
  res.send(`<h1>Secure Response</h1><p>${escapedInput}</p>`);
});
app.get('/', (req, res) => {
  res.sendFile(__dirname + '/public/index.html');
});
const PORT = process.env.PORT || 4000;
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

2. public/index.html

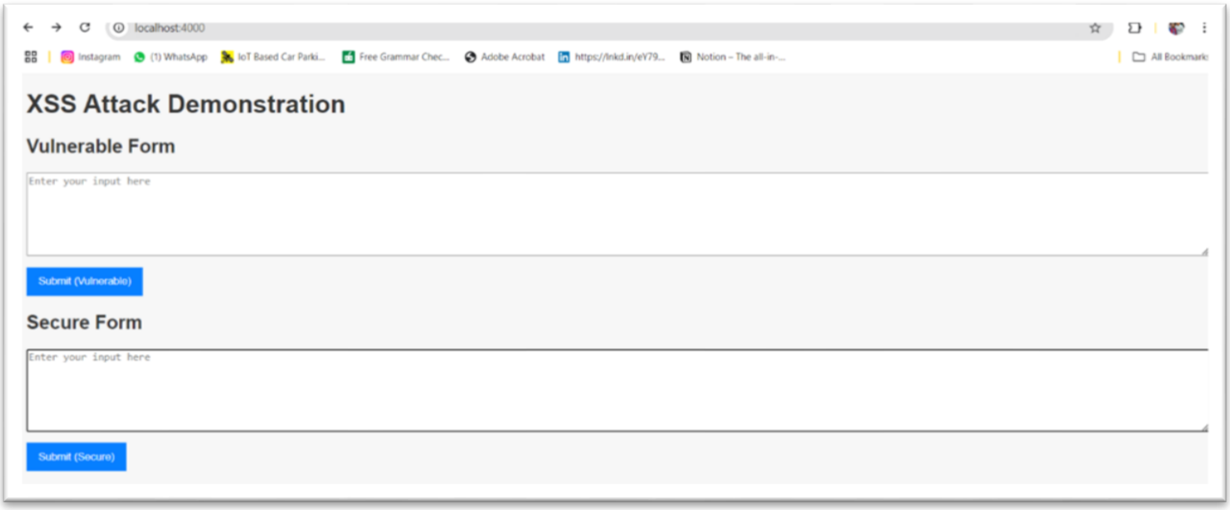
```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>XSS Demonstration</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <h1>XSS Attack Demonstration</h1>

    <h2>Vulnerable Form</h2>
    <form action="/vulnerable" method="POST">
        <textarea name="userInput" placeholder="Enter your input here" required></textarea>
        <button type="submit">Submit (Vulnerable)</button>
    </form>

    <h2>Secure Form</h2>
    <form action="/secure" method="POST">
        <textarea name="userInput" placeholder="Enter your input here" required></textarea>
        <button type="submit">Submit (Secure)</button>
    </form>
</body>
</html>
```

3. public/style.css

```css
body {
    font-family: Arial, sans-serif;
    margin: 20px;
    background-color: #f5f5f5;
}
h1, h2 {
    color: #333;
}
form {
    margin-bottom: 20px;
}
textarea {
    width: 100%;
    height: 100px;
    margin-bottom: 10px;
}
button {
    background-color: #007BFF;
    color: white;
    border: none;
    padding: 10px 15px;
    cursor: pointer;
}
button:hover {
    background-color: #0056b3;
}
```

OUTPUT:

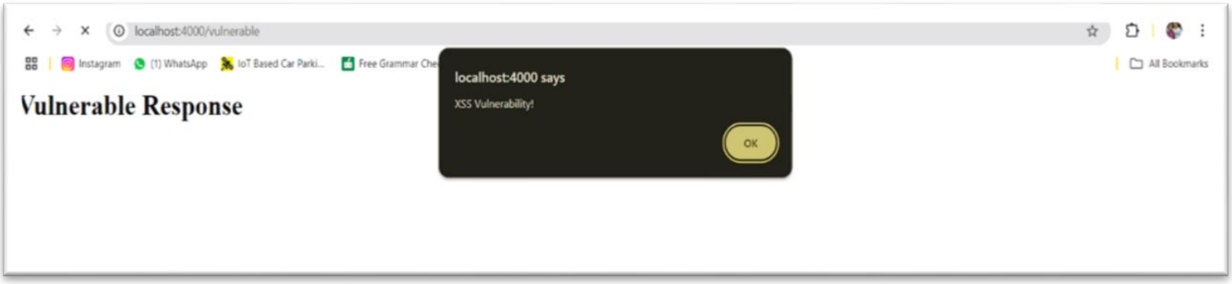
1. **Application Access:**
    - Users can navigate to http://localhost:4000 in their web browser to access the input interface.



2. **Testing the Vulnerable Form:**
    - Enter the following input into the Vulnerable Form text area:

      <script>alert('XSS Vulnerability!');</script>

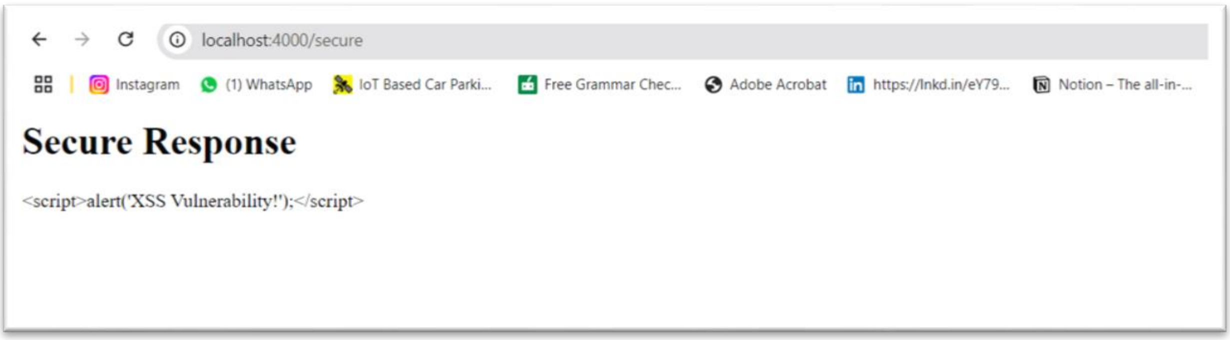
    - Upon submission, an alert box will appear, indicating that the input was not properly sanitized and is vulnerable to XSS.



3. **Testing the Secure Form:**
    - Use the same input in the Secure Form section.
    - Upon submission, the script tags will be escaped and displayed as plain text:

      <script>alert('XSS Vulnerability!');</script>

    - This demonstrates that the secure implementation prevents the XSS attack by escaping special characters.



RESULT : The program was successfully executed and verified with the output

Ex. No. : 09 Date:	**Analyzing and Mitigating Misuse of User Authentication to Identify Security Threats**

AIM:

To analyze a scenario where user authentication is misused due to weak password policies and apply mitigation strategies by implementing strong password validation.

PROCEDURES:

1. Develop a user authentication system in C++ that initially allows weak passwords.
2. Simulate an abuse case where a user can create an account with a weak password, making the system vulnerable to brute force attacks.
3. Modify the program to enforce strong password policies.
4. Test the program to ensure weak passwords are no longer accepted, and only strong, secure passwords are allowed.

PROGRAM:

<u>Vulnerable Program (Weak Password Policy):</u>

```
#include <iostream>
#include <string>
using namespace std;

bool login(string username, string password) {
    string storedPassword = "1234";  // Weak password
    return password == storedPassword;
}

int main() {
    string username, password;
    cout << "Enter username: ";
    cin >> username;
    cout << "Enter password: ";
    cin >> password;

    if (login(username, password)) {
        cout << "Login successful!" << endl;
    } else {
        cout << "Login failed. Weak password used." << endl;
    }

    return 0;
}
```

This version has a weak password ("1234") and no validation, making it vulnerable to brute-force attacks.

Commands to Compile and Run the Program

1. Create a File

Save the vulnerable or secure version of the code as auth.cpp.

2. Compile the Program

Open a terminal or command prompt and run:

g++ -o vulndemo vulndemo.cpp

3. **run the program**

./vulndemo

OUTPUT:

Secure Version(Strong Password Policy):

```
#include <iostream>
#include <string>
#include <cctype>
using namespace std;

bool isValidPassword(const string &password) {
    if (password.length() < 8) return false;
    bool hasUpper = false, hasLower = false, hasDigit = false;

    for (char c : password) {
        if (isupper(c)) hasUpper = true;
        if (islower(c)) hasLower = true;
        if (isdigit(c)) hasDigit = true;
    }
    return hasUpper && hasLower && hasDigit;
}

bool login(string username, string password) {
    string storedPassword = "Str0ngP@ssw0rd";
    return password == storedPassword;
}

int main() {
    string username, password;
    cout << "Enter username: ";
    cin >> username;
    cout << "Enter password: ";
    cin >> password;
```

```
if (!isValidPassword(password)) {
        cout << "Weak password. Use at least 8 characters with upper, lower, and digit." << endl;
        return 0;
    }
    if (login(username, password)) {
        cout << "Login successful!" << endl;
    } else {
        cout << "Login failed." << endl;
    }
    return 0;
}
```

Commands to Compile and Run the Program

1. **Create a File**

   Save the vulnerable or secure version of the code as auth.cpp.

2. **Compile the Program**

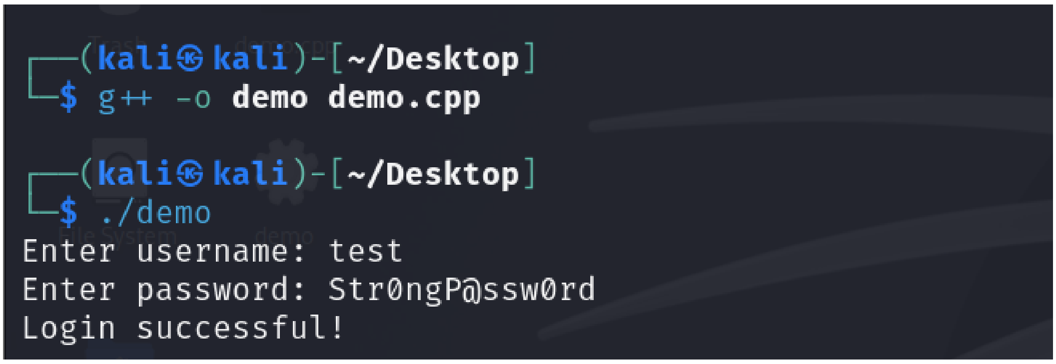
   Open a terminal or command prompt and run:

   g++ -o demo demo.cpp

3. **Run the program**

   ./demo

OUTPUT:



RESULT:

In the initial version, the system allowed weak passwords, creating a security vulnerability. In the secure version, only passwords with a minimum of 8 characters, including uppercase, lowercase, and numeric characters, are accepted, reducing the risk of brute-force attacks.

Ex. No. : 10 Date:	**Integrating Security Practices into the Software Architecture and Design Phase**

AIM:

To understand how security practices can be integrated into the software architecture and design phase to prevent vulnerabilities early in the development lifecycle.

PROCEDURE:

1. Identify potential risks, such as storing sensitive information insecurely.
2. Design a solution to mitigate these risks using security principles like encryption and input validation.
3. Develop a C++ program that initially demonstrates poor security practices.
4. Modify the program to incorporate secure coding techniques.
5. Test the program to verify that the security issues are resolved.

PROGRAM

<u>Vulnerable Program (Storing Passwords in Plain Text):</u>

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

void storeUserData(const string& username, const string& password) {
    ofstream file("user_data.txt", ios::app);
    file << username << " " << password << endl;  // Storing passwords in plain text
    file.close();
    cout << "User data stored." << endl;
}

int main() {
    string username, password;
    cout << "Enter username: ";
    cin >> username;
    cout << "Enter password: ";
    cin >> password;

    storeUserData(username, password);
    return 0;
}
```

<u>Running the vulnerable Program</u>

1.Compile the Program:

g++ -o vulnversion vulnversion.cpp

2. Run the program:

./vulnversion

OUTPUT:

```
┌──(kali㉿kali)-[~/Desktop]
└─$ g++ -o vulnversion vulnversion.cpp

┌──(kali㉿kali)-[~/Desktop]
└─$ ./vulnversion
Enter username: test
Enter password: test@123
User data stored.

┌──(kali㉿kali)-[~/Desktop]
└─$ cat user_data.txt
test test@123
```

Improved C++ Program (Secure Password Handling using Hashing):

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <openssl/sha.h>  // OpenSSL library for hashing
using namespace std;

string hashPassword(const string& password) {
    unsigned char hash[SHA256_DIGEST_LENGTH];
    SHA256((unsigned char*)password.c_str(), password.length(), hash);

    string hashedPassword;
    for (int i = 0; i < SHA256_DIGEST_LENGTH; i++) {
        char buffer[3];
        sprintf(buffer, "%02x", hash[i]);
        hashedPassword += buffer;
    }
    return hashedPassword;
}

void storeUserData(const string& username, const string& hashedPassword) {
    ofstream file("user_data.txt", ios::app);
    file << username << " " << hashedPassword << endl;  // Storing hashed password
    file.close();
    cout << "User data stored securely." << endl;
}

int main() {
    string username, password;
    cout << "Enter username: ";
    cin >> username;
    cout << "Enter password: ";
    cin >> password;

    string hashedPassword = hashPassword(password);
    storeUserData(username, hashedPassword);
    return 0;
}
```

<u>Running the secured Program:</u>

1.Install Crypto++ on your system if not installed:

- On Linux:

  sudo apt-get install libcrypto++-dev

2.Compile the Program:

g++ -o secureversion secureversion.cpp -lcryptopp

3. Run the program:

./secureversion

OUTPUT:

RESULT:

This experiment demonstrates the importance of integrating security practices during the design phase to prevent vulnerabilities.