VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"Jnana Sangama", Belagavi-590018

SKILL DEVELOPMENT ACTIVITY REPORT ON

SOFTWARE ENGINEERING (25MCS104E)

TOPIC:

Submitted in partial fulfillment of the requirements for the award of the degree of

MASTER OF TECHNOLOGY
IN
COMPUTER SCIENCE AND ENGINEERING

Submitted by
SUMA

Under the guidance of

Dr. Sunita Chalageri
Associate Professor
Department of Computer Science and Engineering

Department of Computer Science and Engineering
K. S. INSTITUTE OF TECHNOLOGY
(An Autonomous Institution under VTU)
#14, Raghuvanahalli, Kanakapura Road, Bengaluru- 560109

2025-2026

K. S. INSTITUTE OF TECHNOLOGY

(An Autonomous Institution under VTU)

#14, Raghuvanahalli, Kanakapura Main Road, Bengaluru-560109

Department of Computer Science & Engineering

CERTIFICATE

This is to certify that the Skill Development Activity on Software Engineering entitled **" Security Best Practices: Buffer Overflow and SQL Injection"** carried out by **Name : SUMA** of First Semester M.Tech, CSE, **K. S. Institute of Technology** in the partial fulfilment for the award of the **Master of Technology** in **Computer Science & Engineering** of the **Visvesvaraya Technological University, Belagavi,** during the year 2025-26. It is certified that all corrections/suggestions indicated for Internal Assessment of Software Engineering (25MCS104E) Course assignment have been incorporated in the report. The Activity report has been approved as it satisfies the academic requirements in respect of Skill Development Activity prescribed for the said degree for the First semester.

Dr . Sunita Chalageri
Associate Professor
Dept of CSE,KSIT

Dr. Rekha B Venkatapur
Prof. & HOD
Dept. of CSE, KSIT

ACKNOWLEDGEMENT

I take this opportunity to express my sincere gratitude to my college **K. S. Institute of Technology** for providing a supportive environment that enabled me to successfully complete this Skill Development Activity and prepare this report.

I also extend my heartfelt thanks to the **Management of K. S. Institute of Technology** for providing all the necessary resources required for the Skill Development Activity.

My sincere thanks to **Dr. Dilip Kumar K**, **Principal and Director,** K. S. Institute of Technology for his continuous support and encouragement.

I am grateful to my PG Coordinator, **Dr. Krishna Gudi, Associate professor,** Department of CSE for his guidance and support throughout the course of this work.

I would also like to extend my gratitude to **Dr. Rekha B Venkatapur, Professor and Head,** Department of Computer Science and Engineering for all the support forwarded to me in completing this Skill Development Activity successfully.

I would also like to express my gratitude and appreciation to the faculty of the **Software Engineering** course **Dr, Sunita Chalageri, Associate professor,** Department of CSE for her timely guidance and valuable inputs which greatly helped me in Successfully completing my Activity.

Finally, I express my appreciation to my family and friends for their Moral support and help prodding me to complete the Skill Development Activity.

Suma

ABSTRACT

This report examines critical security best practices in software development, with a specific focus on buffer overflow and SQL injection vulnerabilities. It explores how these common, yet dangerous security flaws can compromise system integrity, confidentiality, and availability when not properly addressed. The report analyzes the causes and mechanisms behind buffer overflow attacks, including improper memory handling and lack of boundary checks, as well as SQL injection attacks that exploit insufficient input validation and insecure database queries. It further discusses preventive strategies such as secure coding standards, input validation, parameterized queries, proper memory management, code reviews, and security testing. By integrating security considerations throughout the development process, organizations can significantly reduce the risk of exploitation. This study highlights the importance of proactive security measures in building robust, reliable, and resilient software systems.

CONTENTS

SL NO.	TITLE	PAGE NO
1	**CHAPTER 1 : INTRODUCTION**	1
2	**CHAPTER 2 : FUNDAMENTALS OF BUFFER OVERFLOW** 2.1 Definition of Buffer Overflow 2.2 How Buffer Overflow Occurs 2.3 Types of Buffer Overflow 2.4 How to Prevent Buffer	4
3	**CHAPTER 3 : FUNDAMENTALS OF SQL-IJECTION** 3.1 Definition of SQL-Injection 3.2 How SQL- Injection Occurs 3.3 Types of SQL-Injection 3.4 Impact of SQL-Injection 3.5 How to Prevent SQL-Injection	9
4	**CHAPTER 4 : TOOLS AND PREREQUISITES** 4.1 Harware Requirements 4.2 Software Requirements 4.3 Python Libraries 4.4 Enviroment Setup 4.5 Database Initialization 4.6 Running the Application	14
5	**CHAPTER 5 : IMPLEMENTATION**	16
6	**CHAPTER 6: ONLINE COURSE CERTIFICATION RELATED TO THE COURSE**	29
7	**CHAPTER 7 : PRESENTATION SNAPSHOTS**	30
8	**GEO TAGGED PHOTOGRAPH DURING DEMONSTRATION**	36
9	**CONCLUSION**	37
	REFERENCES	38

CHAPTER 1

INTRODUCTION

Software engineering is a systematic, disciplined, and methodical approach to the development, design, implementation, testing, deployment, and maintenance of software systems. It combines principles from computer science, engineering, mathematics, and project management to produce high-quality software that meets user requirements and performs reliably under various conditions. In the modern digital era, software systems are deeply integrated into almost every aspect of society, including finance, healthcare, transportation, communication, education, entertainment, and government operations. Because these systems often manage sensitive data and critical services, their reliability, efficiency, scalability, and especially security are of paramount importance. As systems grow more complex and interconnected, the role of software engineering becomes increasingly essential in ensuring structured development, minimizing risks, and delivering dependable solutions.

Unlike basic programming, which primarily focuses on writing code to perform specific tasks, software engineering encompasses the entire Software Development Life Cycle (SDLC). The SDLC provides a structured framework that guides the software development process from initial concept to final product and ongoing maintenance. It typically includes phases such as planning, requirements analysis, system design, implementation, testing, deployment, and maintenance. During the planning phase, project goals, scope, feasibility, cost estimation, timelines, and risk assessments are defined to establish a clear roadmap. Requirement analysis follows, where both functional requirements (what the system should do) and non-functional requirements (such as performance, reliability, usability, maintainability, and security) are carefully gathered and documented. Clear and well-defined requirements reduce misunderstandings, prevent scope creep, and ensure that the final product aligns with user expectations.

The system design phase translates requirements into a structured blueprint for development. This includes defining the system architecture, selecting appropriate technologies, designing databases, specifying user interfaces, and outlining data flow between components. A well-

designed architecture enhances scalability, maintainability, and performance while reducing the likelihood of structural weaknesses. Implementation, or coding, is the phase where developers convert design specifications into executable programs using suitable programming languages, frameworks, and development tools.

During this stage, adherence to coding standards, documentation practices, and secure programming techniques is critical. Testing then verifies that the system functions as intended and meets specified requirements. Various testing methods—including unit testing, integration testing, system testing, regression testing, and user acceptance testing—are applied to detect and resolve defects. Deployment makes the application available to users in a production environment, ensuring proper configuration and operational readiness. Finally, maintenance ensures long-term functionality by correcting errors, improving performance, adapting to environmental changes, and applying security updates.

Among the many topics in software engineering, software security has emerged as one of the most critical. With the rapid increase in cyber threats, organizations face constant risks of data breaches, unauthorized access, financial loss, and reputational damage. Security vulnerabilities often arise from improper coding practices, insufficient validation of user input, inadequate system design, and lack of security testing. Two of the most significant and frequently exploited vulnerabilities are buffer overflow and SQL injection attacks. These vulnerabilities highlight the importance of integrating security principles into every stage of the software engineering process.

Buffer overflow is a memory management vulnerability that occurs when a program writes more data to a buffer than it was allocated to hold. A buffer is a temporary storage area in memory used for processing data. When excess data exceeds the buffer's capacity, it can overwrite adjacent memory locations, potentially corrupting data or altering the execution flow of a program. In severe cases, attackers can exploit buffer overflow vulnerabilities to inject and execute malicious code, gain unauthorized access, or crash entire systems. This type of vulnerability is commonly associated with low-level programming languages that allow direct memory manipulation without automatic boundary checking. Preventing buffer overflow requires careful memory management, bounds checking, secure coding practices, compiler-level protections, and thorough testing.

SQL injection, in contrast, is a vulnerability that affects database-driven applications. It occurs when user inputs are not properly validated or sanitized before being incorporated into SQL queries. Attackers can inject malicious SQL statements into input fields such as login forms or search boxes, altering the intended database command. This can result in unauthorized data retrieval, modification, deletion, or bypassing of authentication mechanisms. SQL injection attacks exploit weaknesses in how applications construct database queries dynamically. Preventive measures include using parameterized queries (prepared statements), input validation, stored procedures, least-privilege database access, and secure configuration practices. Regular security assessments and code reviews further reduce the risk of such vulnerabilities.

The implementation of secure software engineering practices requires a proactive and integrated approach. Security should be embedded into the SDLC rather than treated as a final step. During planning and requirement analysis, security requirements such as authentication, authorization, encryption, and data protection should be clearly defined. During design, secure architecture principles such as defense in depth, least privilege, and separation of concerns should be applied. During implementation, developers must follow secure coding guidelines and avoid unsafe functions or improper input handling. During testing, security testing methods such as vulnerability scanning, static code analysis, dynamic testing, and penetration testing should be performed to identify weaknesses. After deployment, continuous monitoring, patch management, and updates are necessary to address newly discovered vulnerabilities and evolving threats.

In conclusion, software engineering serves as the foundation for building reliable, maintainable, and secure applications in an increasingly interconnected world. As cyber threats continue to evolve, the integration of robust security measures—particularly against vulnerabilities such as buffer overflow and SQL injection—remains essential. Through disciplined methodologies, comprehensive testing, secure design principles, and continuous improvement, software engineers can develop systems that not only meet functional requirements but also withstand modern cybersecurity challenges, ensuring long-term reliability and trustworthiness.

CHAPTER 2

FUNDAMENTALS OF BUFFER OVERFLOW

2.1 Definition of Buffer Overflow

A buffer overflow occurs when a program writes more data into a buffer than the allocated memory space can hold. A buffer is a temporary memory storage area used to store data while it is being processed. When excessive data exceeds the buffer's boundary, it overwrites adjacent memory locations, potentially corrupting data or altering program execution.

2.2 How Buffer Overflow Occurs

Buffer overflows typically occur in programming languages such as C and C++ that allow direct memory access without automatic boundary checking. Buffer overflow vulnerabilities primarily arise from insecure programming practices, improper memory management, and insufficient input validation. In low-level programming languages such as C and C++, developers have direct access to memory, and arrays or buffers do not automatically enforce boundary limits. When programs accept user input without verifying its size, an attacker can provide more data than the allocated memory can hold, causing an overflow.

The use of unsafe standard library functions, such as strcpy, gets, or sprintf, which do not perform bounds checking, is a major contributor to buffer overflow vulnerabilities. Another common cause is improper pointer handling, where incorrect arithmetic or references can inadvertently overwrite adjacent memory locations. Integer overflows or miscalculations of memory size can also lead to buffer overflows indirectly, as the program may allocate insufficient memory and allow data to exceed buffer boundaries. Human factors, such as insufficient developer training in secure coding practices, lack of awareness of memory safety issues, and pressure to meet tight deadlines, increase the likelihood of these errors occurring. Inadequate testing and failure to perform stress tests, fuzz testing, or static and dynamic code analysis may leave buffer overflow vulnerabilities undetected until they are exploited.

Additionally, software complexity can exacerbate the problem; assumptions about input size or memory allocation in one module may not hold true in interactions with other parts of the system, creating unintentional overflow conditions. Ultimately, buffer overflow vulnerabilities are caused by a combination of technical limitations in programming languages, unsafe coding practices, and human oversight during software development

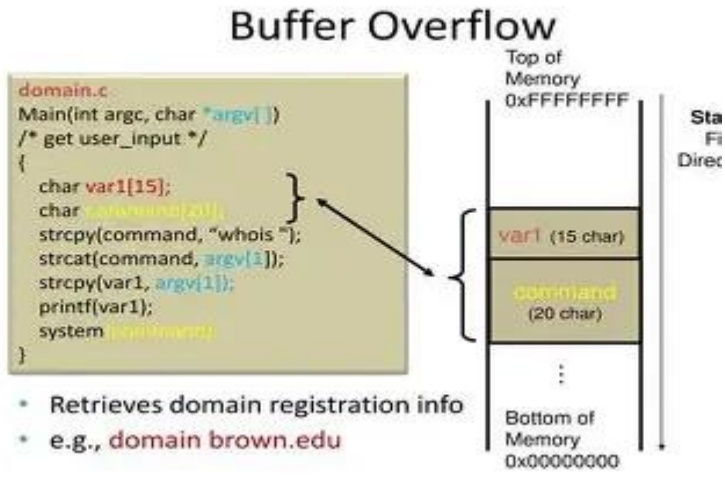


Figure 2.1: Buffer Overflow Attack

2.3 Types of Buffer Overflow

2.3.1 Stack-based buffer overflow

Occurs in memory allocated on the call stack and may overwrite return addresses, leading to execution of arbitrary code. Stack-based buffer overflow occurs when data exceeds the allocated space of a buffer in the call stack, a special memory region used to store function parameters, local variables, and return addresses. When a program does not validate input size, attackers can provide excessive data that overwrites these critical memory locations. Overwriting a function's return address is the most serious consequence because it allows the attacker to redirect program execution to malicious code, often referred to as shellcode. For example, in a vulnerable login program, if a username field stored on the stack does not limit input length, an attacker can input a string long enough to overwrite the return address and inject executable code that the program runs upon function return.

Stack-based overflows can lead to program crashes, denial of service, or complete system

compromise if administrative privileges are exploited. Historically, many high-profile malware and worm attacks, such as the Code Red and Blaster worms, exploited stack-based buffer overflows to spread and execute code remotely. Prevention requires strict input validation, the use of safe functions that check bounds, compiler protections such as stack canaries, and memory security techniques like Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP).

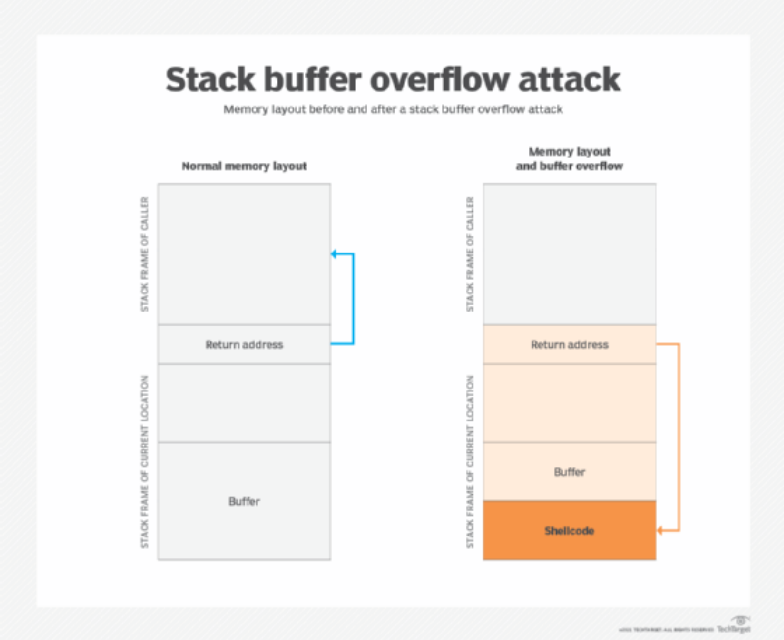


Figure 3.1 : Stack Based Overflow

2.3.2 Heap-based buffer overflow

Heap-based buffer overflow occurs in dynamically allocated memory, known as the heap, which is used for objects, buffers, and data structures that persist throughout the program's execution. Unlike stack overflows, which immediately affect function control flow, heap based overflows often corrupt adjacent memory blocks, object metadata, or pointer structures, indirectly influencing program behavior.

For instance, a web application that stores user-uploaded files in dynamically allocated buffers may be vulnerable if it does not validate the size of the uploaded data. An attacker can overflow the buffer and overwrite a function pointer, redirecting program execution to malicious code.

Heap overflows are also commonly used in use-after-free and double-free vulnerabilities, where previously freed memory is manipulated to execute arbitrary code.

Real-world incidents include vulnerabilities in software like Adobe Flash and Internet Explorer, where heap overflows were leveraged to execute remote code and compromise systems without user interaction. Preventing heap overflows requires strict input validation, careful memory allocation, and avoiding unsafe functions such as sprintf or strcpy. Modern operating systems also implement heap protection mechanisms, including heap metadata integrity checks, guard pages, and runtime heap analysis tools. Developers are encouraged to use memory-safe programming techniques and leverage automatic memory management where possible, reducing the risk of exploitation.

2.3.3 Integer overflow

Occurs when arithmetic operations exceed the maximum value that can be stored in a variable, potentially leading to memory allocation errors and further buffer overflow vulnerabilities. Integer overflow is a subtle vulnerability that occurs when an arithmetic operation produces a value exceeding the maximum storage capacity of the variable type, causing the value to wrap around or behave unexpectedly. For example, adding two large integers in a 32-bit variable may produce a negative or wrapped value, which can then be used to allocate a buffer smaller than intended. Subsequent writes to this buffer may exceed its allocated size, resulting in a buffer overflow. Integer overflows can also cause incorrect array indexing, pointer miscalculations, or logic bypasses, enabling attackers to manipulate program behavior or access restricted memory areas. A real-world example is the 2003 Internet Explorer vulnerability, where integer overflow in image dimension calculations allowed heap buffer overflows, which attackers exploited to execute arbitrary code. Integer overflow is particularly dangerous because it is often overlooked during testing, as arithmetic calculations appear correct during normal operation. Preventing integer overflow requires careful input validation, ensuring that arithmetic operations do not exceed variable limits, using larger or safer data types when necessary, and implementing explicit overflow checks in the code.

Automated static analysis tools can help detect potential integer overflows before deployment. Additionally, modern compilers offer built-in protections such as runtime overflow detection,

which can alert developers when arithmetic operations exceed safe ranges.

2.4 Prevention of Buffer Overflow

Preventing buffer overflow requires a combination of secure coding practices, proper memory management, and system-level protections. The first line of defense is input validation, which ensures that all user-supplied data is checked for size, type, and format before being processed or stored in memory buffers. Developers should replace unsafe functions, such as strcpy, gets, or sprintf, with safer alternatives like strncpy, fgets, or functions that explicitly enforce boundary checks. Bounds checking is essential for all memory operations, guaranteeing that no data exceeds the allocated buffer size.

Modern compilers provide protections such as stack canaries, which insert special values in memory to detect overflow before a function returns, preventing attackers from hijacking execution flow. Operating system-level protections, including Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP), make it significantly harder for attackers to predict memory addresses or execute injected code. Additional preventive measures include thorough code reviews, static and dynamic analysis, and fuzz testing, which can detect potential overflow vulnerabilities during development and testing phases. Using programming languages that provide automatic memory management or built-in bounds checking, such as Java or Python, further reduces the risk of buffer overflows. Finally, continuous developer training, adherence to secure coding standards, and ongoing software maintenance are critical to ensure that buffer overflow vulnerabilities are identified, mitigated, and prevented throughout the software lifecycle.

By applying these strategies collectively, organizations can protect applications against one of the most dangerous and historically exploited security vulnerabilities.

CHAPTER 3

FUNDAMENTALS OF AND SQL INJECTION

3.1 Definition of SQL Injection

SQL injection is a vulnerability that occurs when user input is improperly validated and directly included in SQL queries. Instead of being treated as data, malicious input is interpreted as part of the SQL command, allowing attackers to manipulate database operations.

3.2 How SQL Injection Occurs

SQL injection vulnerabilities arise when applications fail to properly validate, sanitize, or handle user input before incorporating it into SQL queries. At its core, SQL injection is caused by the improper separation of data and code, allowing attackers to manipulate the logic of database queries by submitting specially crafted input. One of the most common technical causes is dynamic SQL query construction through string concatenation. In many legacy and poorly designed applications, developers construct SQL statements by directly appending user input to query strings. For example, a login form might construct a query like SELECT * FROM users WHERE username = ' + userInput + ' AND password = ' + passwordInput + '. If there is no input validation or escaping, an attacker can inject SQL keywords or operators to bypass authentication, extract data, or even alter database contents.

Another critical cause is insufficient input validation. When applications fail to restrict the length, type, or format of user input, attackers can supply data containing malicious characters, such as single quotes ('), semicolons (;), or SQL comments (--). These characters can break the intended query syntax, allowing unauthorized manipulation of database commands. For instance, entering ' OR '1'='1 in an unvalidated login field can transform a legitimate authentication query into a condition that always evaluates as true, granting unauthorized access. In addition, lack of proper sanitization or escaping of special characters exacerbates the problem, as databases may interpret these characters as SQL code instead of user data.

Poor authentication and session management practices also contribute to SQL injection vulnerabilities.

Complex application logic and integration points can also contribute to SQL injection. Large-scale applications often involve multiple modules, third-party APIs, and microservices, each potentially interacting with a database. A vulnerability in one module can propagate, allowing attackers to access other components or escalate the impact of an injection attack. For example, if an input from one service is passed unchecked to another module that constructs SQL queries, an attacker may exploit this trust to inject SQL commands. Similarly, applications that rely on client-side input validation alone are particularly vulnerable because client-side checks can be bypassed entirely.

In summary, the causes of SQL injection are multifaceted, encompassing technical flaws, human errors, legacy system issues, and organizational shortcomings. Dynamic SQL construction, lack of input validation, improper sanitization, weak authentication, excessive database privileges, poor error handling, and outdated software are all direct contributors. Indirect causes include developer inexperience, insufficient code reviews, insecure legacy systems, and inadequate organizational security policies.

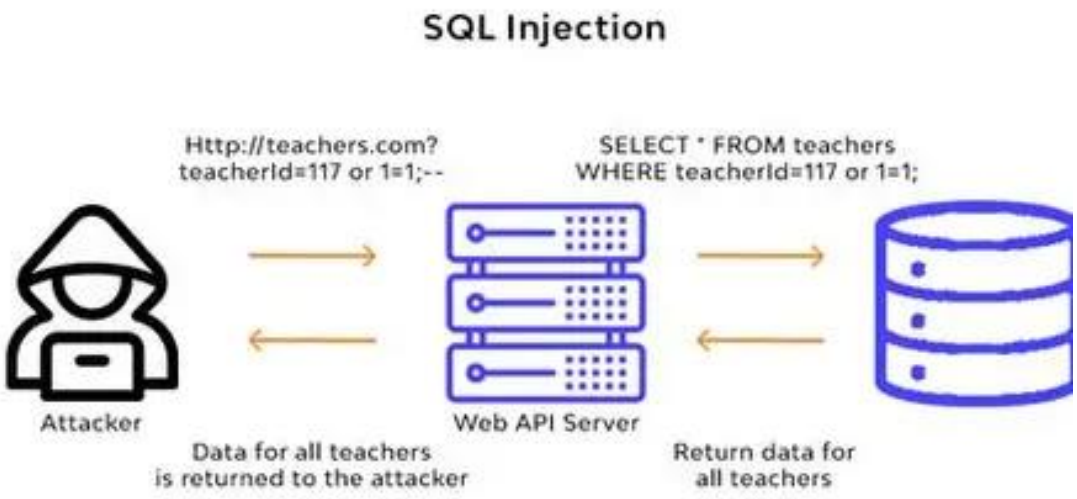


Figure 3.2: SQL Injection

3.3Types of SQL Injection

3.3.1 In-band SQL injection

In-Band SQL Injection is the most common type of SQL injection attack, where attackers use the same communication channel to both send malicious SQL queries and retrieve the results. It exploits vulnerabilities in applications that directly process user input within SQL statements. Within in-band injection, error-based SQL injection leverages detailed database error messages to gather information about the database structure, such as table names, column names, and data types. Attackers can use this information to construct further attacks and extract sensitive data. Union-based SQL injection is another variant where attackers use the SQL UNION operator to combine malicious queries with legitimate ones, allowing them to retrieve additional data from other tables without modifying the application logic.

In-band attacks are dangerous because they provide immediate feedback to the attacker, making them relatively easy to execute and highly effective in stealing data.Blind SQL injection: The application does not directly display results, so attackers infer information based on system responses.

3.3.2 Out-of-band SQL injection

Out-of-Band SQL Injection is less common and typically used when the attacker cannot retrieve results through the same channel or when in-band methods are blocked by security controls. In this type, the malicious query forces the database to communicate with an external system controlled by the attacker, such as a DNS or HTTP server, to send data. Out-of-band attacks are slower and more complex, but they are effective against secure systems that block error messages or direct responses. They demonstrate the attacker's ability to bypass traditional protections and still exfiltrate sensitive information.

3.3.3 Blind SQL Injection

Blind SQL Injection occurs when an application does not directly display database output or suppresses error messages, forcing attackers to infer information indirectly. There are two common methods of blind injection: boolean-based blind SQL injection, where attackers send queries that evaluate to true or false and observe application responses, and time-based blind SQL injection, where queries include deliberate delays to infer information based on response times. Blind SQL injection is often harder to detect and exploit than in-band injection, but it is

widely used against applications with strict error handling.

3.4 Impact of SQL Injection

SQL injection (SQLi) is one of the most serious security vulnerabilities in web applications, with wide-ranging consequences for organizations and users. One of the primary impacts is unauthorized access to sensitive data. Attackers can exploit vulnerable queries to retrieve confidential information such as usernames, passwords, financial records, and personal identification details. For example, union-based SQL injection allows malicious users to combine their input with legitimate queries to extract additional data from databases, bypassing authentication and access controls. Large-scale breaches can expose millions of records, resulting in severe operational and legal consequences.

SQL injection also enables data manipulation and deletion, which affects the integrity of databases. Attackers can alter transaction records, update user privileges, or delete critical data, potentially causing operational disruption or financial loss. For instance, modifying account balances in a banking system or deleting product records in an e-commerce platform can compromise business operations and decision-making. In addition, SQL injection can lead to system compromise and remote code execution in cases where databases allow execution of stored procedures or operating system commands. Exploiting such vulnerabilities may grant attackers control over servers, enabling installation of malware, persistent access, or lateral movement within a network.

SQL injection can additionally cause operational disruption, including downtime, slow performance, or service outages. Blind SQL injection attacks that force repeated or complex queries can degrade system resources, affecting critical services for e-commerce platforms, financial applications, or public portals. This can lead to lost revenue, frustrated users, and reduced service reliability.

In summary, SQL injection has far-reaching effects, including unauthorized data access, data corruption, system compromise, financial loss, reputational harm, operational disruption, and legal consequences. Its prevalence and exploitability make it critical for organizations to implement strong security measures to prevent attacks, protect sensitive information, and maintain operational integrity.

3.5 Prevention of SQL Injection

Preventing SQL injection requires a combination of secure coding practices, proper input handling, and system-level protections. One of the most effective techniques is the use of parameterized queries or prepared statements, which separate user input from SQL commands. By treating all input as data rather than executable code, parameterized queries prevent attackers from altering the structure of SQL statements. Similarly, stored procedures can provide controlled access to the database when implemented securely, though they must still include input validation to avoid injection.

Input validation and sanitization are fundamental preventive measures. Applications should enforce strict checks on user input, including acceptable characters, length limits, and data types. Special characters such as single quotes ('), semicolons (;), and comment markers (--) should either be escaped or blocked. Server-side validation is critical, as client-side checks can be bypassed by attackers. Additionally, escaping input when parameterization is not possible provides another layer of defense, ensuring that special characters do not interfere with query execution.

Principle of least privilege is a key security strategy. Database accounts used by applications should have only the minimum permissions required for their functionality. Limiting access reduces the potential impact of a successful SQL injection attack. For example, read-only accounts should not have the ability to modify tables, and administrative privileges should be reserved for essential maintenance tasks only.

Web Application Firewalls (WAFs) and other monitoring tools can help detect and block SQL injection attempts. WAFs analyze incoming requests for suspicious patterns and malicious payloads, providing an additional layer of defense even if the application code contains vulnerabilities. Automated tools can help developers detect unsafe query constructions and unvalidated inputs during development.

In conclusion, preventing SQL injection requires a multi-layered approach combining parameterized queries, input validation, least privilege, WAFs, secure error handling, regular testing, and developer education. By implementing these measures, organizations can significantly reduce the risk of SQL injection attacks, safeguard sensitive data, and maintain the integrity and availability of their applications.

CHAPTER 4

TOOLS AND PREREQUISITES

4.1. Hardware Requirements

- **Processor:** Minimum 1 GHz CPU (dual-core recommended)
- **RAM:** Minimum 2 GB (4 GB or more recommended for smooth operation)
- **Storage:** At least 500 MB free disk space for Python, libraries, and database
- **Internet Connection:** Required for downloading dependencies and libraries

4.2. Software Requirements

- **Operating System:** Windows 10/11, Linux (Ubuntu 18+), or macOS
- **Python:** Version 3.8 or higher

  Verify installation with: python --version

- **SQLite3:** Embedded database (comes with Python by default)
- **Flask Framework:** Lightweight Python web framework

  Install using pip: pip install flask

- **Text Editor / IDE:** Any editor for Python development (e.g., VS Code, PyCharm, Sublime Text, or Notepad++)

4.3. Python Libraries

The application requires the following Python modules

- flask – for creating web routes, forms, and sessions
- sqlite3 – for database operations (built into Python)
- hashlib – for password hashing (built into Python)
- logging – for security logging (built into Python)

4.4. Environment Setup

Step 1: Create a project folder (e.g., SecurityDemo)

Step 2: Place the Python file app.py in the folder

Step 3: Create a virtual environment (optional but recommended): python -m venv venv

Step 4: Activate the virtual environment:

- **Windows:** venv\Scripts\activate
- **Linux/macOS:** source venv/bin/activate

Step 5: Install Flask within the virtual environment: pip install flask

4.5. Database Initialization

- The application automatically creates the SQLite database security_demo.db on first run
- It creates the users table and adds a default admin user:
  - **Username:** admin
  - **Password:** admin123
- No manual database setup is required unless you want to pre-populate users

4.6. Running the Application

Step 1: Navigate to the project folder containing app.py

Step 2 : Run the application: python app.py

Step 3 : Open a web browser and visit: http://127.0.0.1:5000

Step 4 : Use the application to test registration, login (vulnerable and secure), admin panel, and buffer overflow demo

CHAPTER 5

IMPLEMENTATION

```
from flask import Flask, request, redirect, render_template_string, session
import sqlite3
import os
import hashlib
import logging


app = Flask(__name__)
app.secret_key = "super_secret_key_for_demo"
DATABASE = "security_demo.db"
# Logging Configuration
logging.basicConfig(
    filename="security.log",
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s - %(message)s"
)
```

```
# Database Setup

def init_db():

    conn = sqlite3.connect(DATABASE)

    cursor = conn.cursor()

    cursor.execute("""

        CREATE TABLE IF NOT EXISTS users (

            id INTEGER PRIMARY KEY AUTOINCREMENT,

            username TEXT,

            password TEXT,

            role TEXT

        )

    """)



    # Insert default admin if not exists

    cursor.execute("SELECT * FROM users WHERE username='admin'")

    if not cursor.fetchone():

        hashed = hashlib.sha256("admin123".encode()).hexdigest()

        cursor.execute(

            "INSERT INTO users (username, password, role) VALUES (?, ?, ?)",

            ("admin", hashed, "admin")
```

```
        )


    conn.commit()
    conn.close()
init_db()
# Utility Functions
def hash_password(password):
    return hashlib.sha256(password.encode()).hexdigest()
def log_attack(message):
    logging.warning(f"SECURITY ALERT: {message}")
# Home Page
@app.route("/")
def home():
    return """
    <h1>Security Demo Application</h1>
    <ul>
        <li><a href='/register'>Register</a></li>
        <li><a href='/login_vulnerable'>Login (Vulnerable)</a></li>
        <li><a href='/login_secure'>Login (Secure)</a></li>
        <li><a href='/buffer_demo'>Buffer Overflow Demo</a></li>
```

```
    </ul>
    """


# Registration
@app.route("/register", methods=["GET", "POST"])
def register():
    if request.method == "POST":
        username = request.form["username"]
        password = hash_password(request.form["password"])

        conn = sqlite3.connect(DATABASE)
        cursor = conn.cursor()
        cursor.execute(
            "INSERT INTO users (username, password, role) VALUES (?, ?, ?)",
            (username, password, "user")
        )
        conn.commit()
        conn.close()
        return "User Registered Successfully"
    return """
```

```
    <h2>Register</h2>
    <form method="post">
        Username: <input name="username"><br>
        Password: <input name="password" type="password"><br>
        <input type="submit">
    </form>
# SQL Injection - Vulnerable Login
@app.route("/login_vulnerable", methods=["GET", "POST"])
def login_vulnerable():
    if request.method == "POST":
        username = request.form["username"]
        password = request.form["password"]
        conn = sqlite3.connect(DATABASE)
        cursor = conn.cursor()
        # ❌ Vulnerable Query (String Concatenation)
        query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{hash_password(password)}'"
        print("Executing Query:", query)
        try:
            cursor.execute(query)
```

```
            user = cursor.fetchone()
            if user:
                session["user"] = user[1]
                return redirect("/dashboard")
            else:
                return "Invalid Credentials"
        except Exception as e:
            log_attack(f"SQL Injection Attempt: {str(e)}")
            return "Error occurred"
    return """
    <h2>Login (Vulnerable to SQL Injection)</h2>
    <form method="post">
        Username: <input name="username"><br>
        Password: <input name="password" type="password"><br>
        <input type="submit">
    </form>
# SQL Injection - Secure Login
@app.route("/login_secure", methods=["GET", "POST"])
def login_secure():
    if request.method == "POST":
```

```
    username = request.form["username"]
    password = hash_password(request.form["password"])

    conn = sqlite3.connect(DATABASE)
    cursor = conn.cursor()
    # ✅ Secure Parameterized Query
    cursor.execute(
        "SELECT * FROM users WHERE username = ? AND password = ?",
        (username, password)
    )
    user = cursor.fetchone()
    if user:
        session["user"] = user[1]
        return redirect("/dashboard")
    else:
        return "Invalid Credentials"
return """
<h2>Login (Secure Version)</h2>
<form method="post">
    Username: <input name="username"><br>
```

```
        Password: <input name="password" type="password"><br>
        <input type="submit">
    </form>
# Dashboard
@app.route("/dashboard")
def dashboard():
    if "user" not in session:
        return redirect("/")
    return f"""
    <h2>Welcome {session['user']}</h2>
    <a href='/logout'>Logout</a>
# Logout
@app.route("/logout")
def logout():
    session.clear()
    return redirect("/")
# Simulated Buffer Overflow Demo
@app.route("/buffer_demo", methods=["GET", "POST"])
def buffer_demo():
    if request.method == "POST":
```

```
    data = request.form["data"]

    buffer_size = 20  # Simulated fixed buffer

    if len(data) > buffer_size:
        log_attack("Simulated Buffer Overflow Detected")
        return "⚠ Buffer Overflow Detected! Input too large."

    # Simulated buffer allocation
    buffer = [""] * buffer_size
    for i in range(len(data)):
        buffer[i] = data[i]

    return f"Stored in buffer: {''.join(buffer).strip()}"

return """
<h2>Simulated Buffer Overflow Demo</h2>
<p>Enter text (Max 20 characters)</p>
<form method="post">
    Data: <input name="data"><br>
```

```
        <input type="submit">
    </form>
# Admin Panel (View Users)
@app.route("/admin")
def admin():
    if "user" not in session:
        return redirect("/")
    conn = sqlite3.connect(DATABASE)
    cursor = conn.cursor()
    cursor.execute("SELECT id, username, role FROM users")
    users = cursor.fetchall()
    conn.close()
    output = "<h2>All Users</h2><ul>"
    for u in users:
        output += f"<li>ID: {u[0]} | Username: {u[1]} | Role: {u[2]}</li>"
    output += "</ul>"

    return output
if __name__ == "__main__":
app.run(debug=True)
```

.

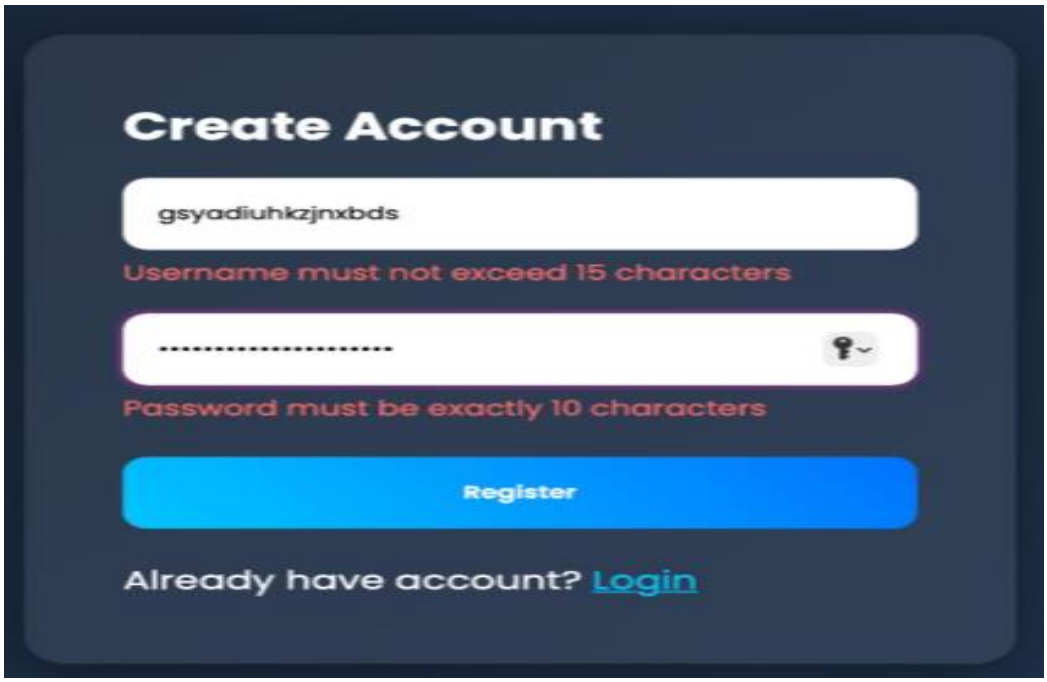


Figure 5.1: Violation of credentials during registrations

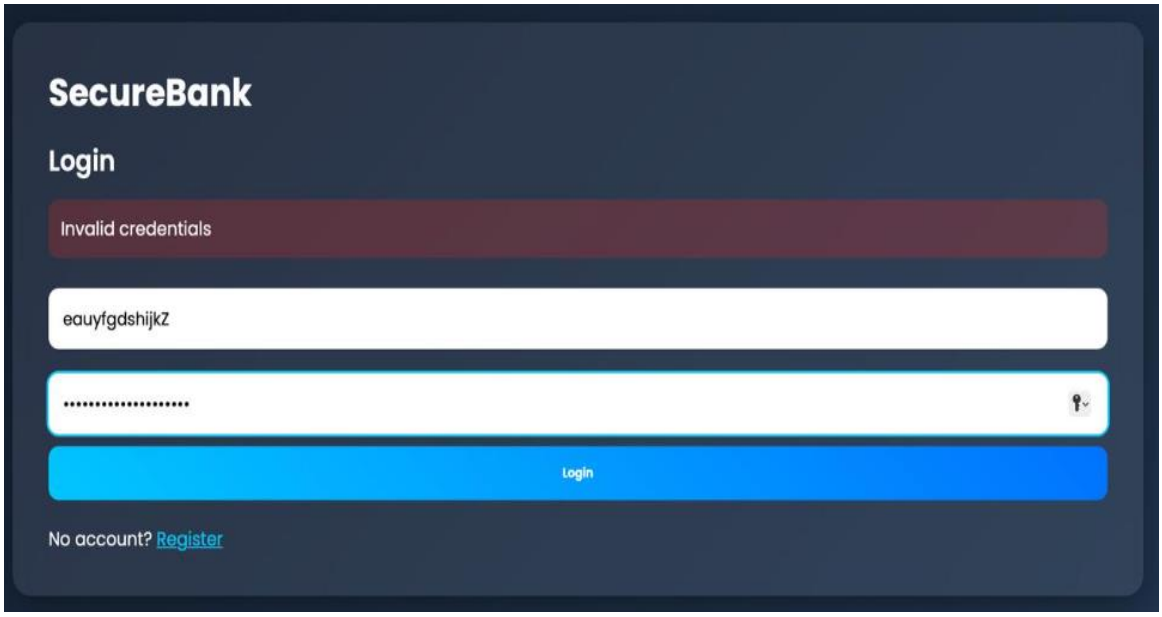


Figure 5.2: Violation of credentials during Login

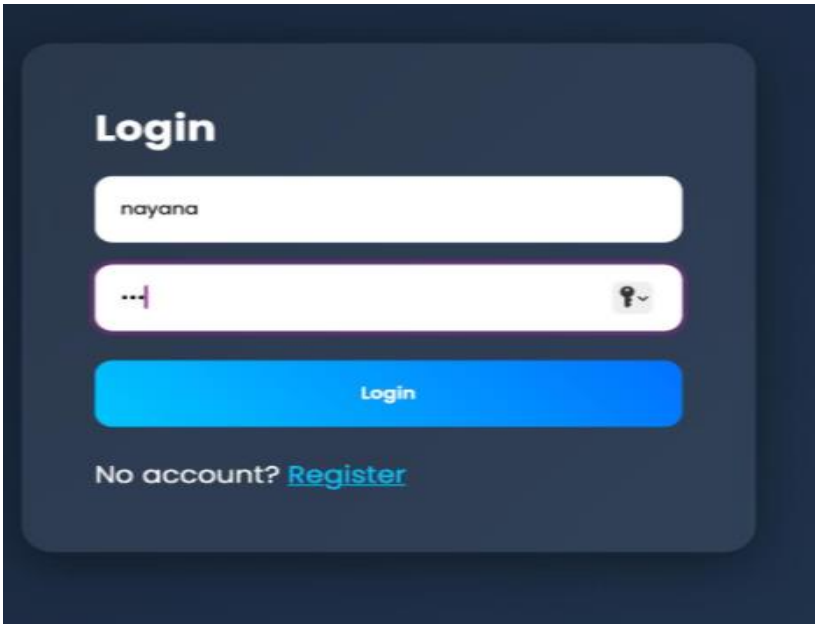


Figure 5.3: Successful Login

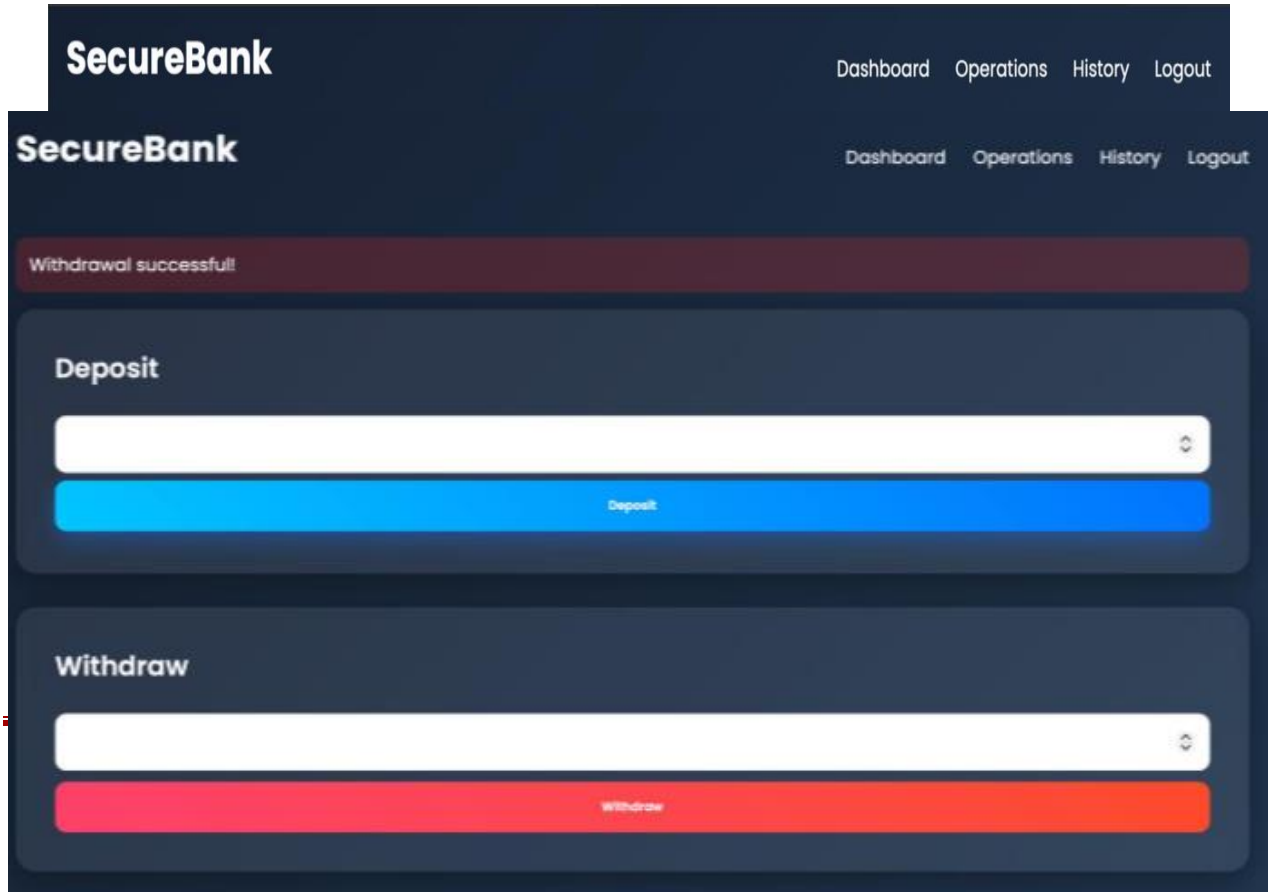


Figure 5.5: Operations page – withdraw and deposit

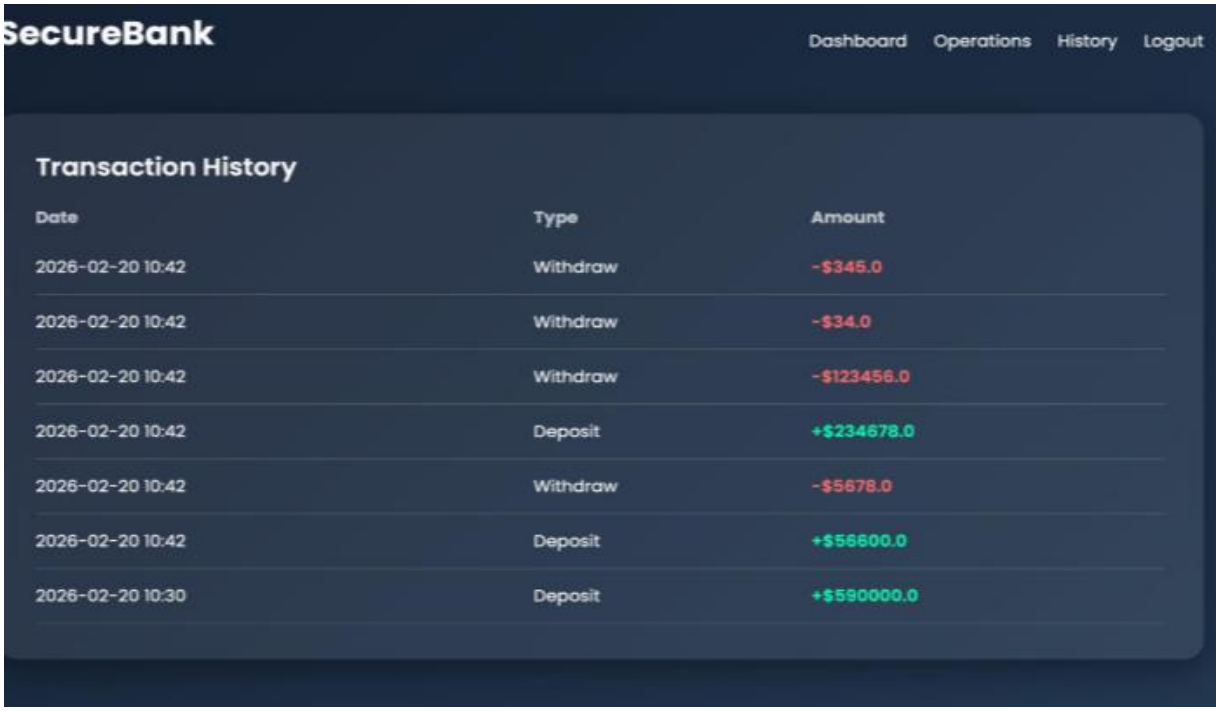


Figure 5.6 : Transactions Page

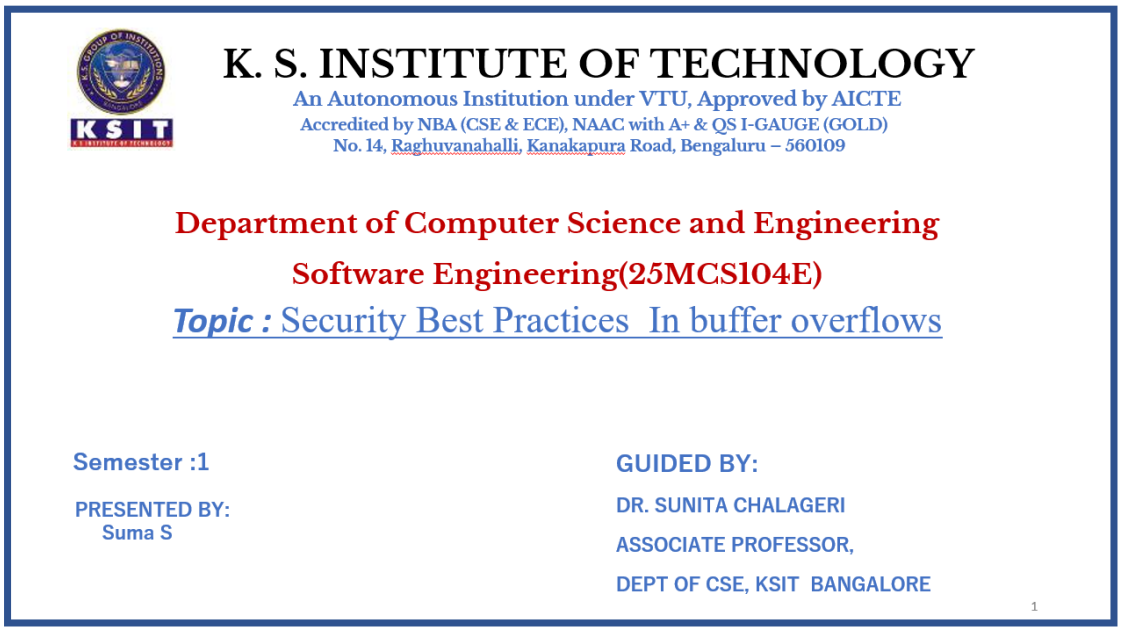
CHAPTER 6

COURSE COMPLETION CERTIFICATE

Figure 6.1: Course Completion Certificate

CHAPTER 7

PRESENTATION IMAGES



INTRODUCTION

- Modern software applications store and process highly sensitive data such as user credentials, financial records, and personal information.
- Security best practices are essential to protect applications from cyberattacks and unauthorized access.
- Many security breaches occur due to common programming mistakes and poor security design.

3

Why is Security Best Practice?

- Guidelines and techniques to protect software from attacks
- Aim to prevent, detect, and respond to security threats
- Integrated throughout the Software Development Life Cycle (SDLC)
- Important for web, mobile, cloud, and enterprise application
- Protects sensitive user and business data
- Prevents cyberattacks and unauthorized access
- Minimizes system downtime and service disruption

4

Buffer Overflow - Overview

- Occurs when a program writes more data to a buffer than it can handle
- Extra data overwrites adjacent memory locations
- Common in low-level languages like C and C++
- Can allow attackers to execute malicious code
- Often caused by lack of input validation
- Difficult to detect without proper testing
- Exploited in many historical cyberattacks

6

Buffer Overflow - Example

- Example: Writing unchecked user input into a fixed-size array
- Impact:
  - Application crashes
  - Data corruption
  - Remote code execution
- Can lead to full system compromise
- Attackers may gain administrator privileges
- Serious risk in critical systems

7

SQL Injection

- A web security vulnerability
- Occurs when malicious SQL queries are injected via user input
- Exploits poor input validation
- Targets backend databases
- One of the most common web attacks
- Listed in OWASP Top 10 vulnerabilities

8

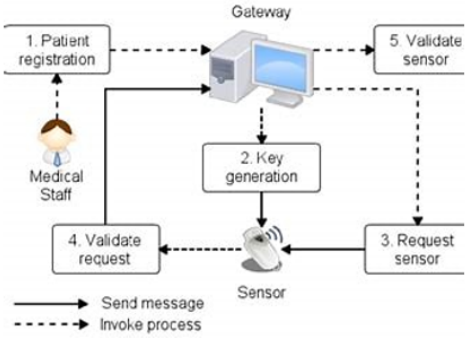
SQL Injection - Example

- Example:
  - Manipulating login forms to bypass authentication
- Impact:
  - Unauthorized data access
  - Data deletion or modification
- Use ORM frameworks
- Apply least-privilege database access
- Proper error handling to hide database details

9

Secure Authorization Practices

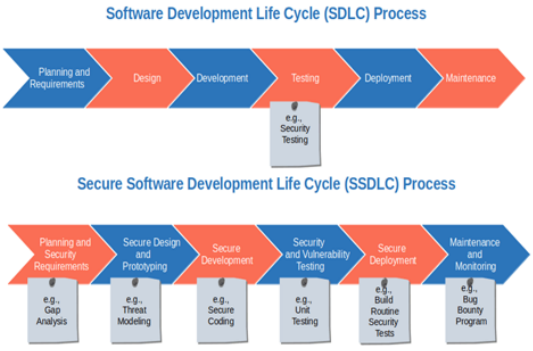
- Role-Based Access Control (RBAC)
- Principle of Least Privilege
- Server-side access checks
- Regularly review user permissions
- Separate admin and user roles
- Log authorization failures



17

Security in SDLC

- Security must be planned during requirement analysis
- Threat modeling during design phase
- Secure coding during development
- Security testing during QA phase
- Regular vulnerability scanning
- Secure deployment configurations
- Continuous monitoring and patching
- Incident response planning



19

IMPLEMENTATION SINGLRTON PATTERN

```python
MAX_BUFFER_SIZE = 64  # Define safe buffer size

def secure_input(user_input):
    """
    This function safely handles input data
    by applying bounds checking and validation.
    """

    # 1. Bounds Checking
    if len(user_input) > MAX_BUFFER_SIZE:
        raise ValueError("Input exceeds allowed buffer size")

    # 2. Input Validation
    if not user_input.isalnum():
        raise ValueError("Invalid input: only alphanumeric
            characters allowed")
```

```
Enter input data: HelloWorld123
Safe input stored successfully: HelloWorld123

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

8

Conclusion

- Security is essential in modern software systems
- Best practices help prevent common vulnerabilities
- Secure design reduces attack surfaces
- Continuous security improvement is necessary
- Developers play a crucial role in security
- Secure software builds user trust
- Security should never be an afterthought

21

GEO TAGGED PHOTO DURING PRESENTATION

CONCLUSION

Security in the Software Development Life Cycle (SDLC) is no longer optional—it is a fundamental requirement for delivering reliable, resilient, and trustworthy software systems. Integrating security practices throughout every phase of the SDLC—from requirements gathering and design to development, testing, deployment, and maintenance—significantly reduces vulnerabilities, minimizes risk, and lowers the cost of remediation.

Adopting a **Secure SDLC (SSDLC)** approach ensures that security is treated as a shared responsibility among stakeholders, including developers, security teams, project managers, and business leaders. Early threat modeling, secure coding standards, code reviews, vulnerability assessments, penetration testing, and continuous monitoring all contribute to building security into the foundation of the software rather than adding it as an afterthought.

Modern methodologies such as DevSecOps emphasize automation, continuous integration, and continuous security testing to detect and address vulnerabilities in real time. By leveraging industry-recognized frameworks and standards such as OWASP guidelines and ISO/IEC 27001, organizations can align their development processes with global best practices and regulatory requirements.

Furthermore, fostering a security-aware culture through regular training, policy enforcement, and compliance audits strengthens organizational resilience against evolving cyber threats. Secure design principles, least privilege access control, encryption, and proper incident response planning are critical components in protecting sensitive data and maintaining stakeholder trust.

In conclusion, embedding security across the entire software development lifecycle enhances software quality, protects organizational assets, ensures regulatory compliance, and builds long-term customer confidence. Organizations that proactively integrate security into their SDLC processes are better positioned to mitigate risks, respond to emerging threats, and deliver secure, high-quality software solutions in an increasingly complex digital landscape.

REFERENCES

1. **Survey of Protections from Buffer-Overflow Attacks**
   Krerk Piromsopa & R.J. Enbody — *Engineering Journal* (2011) — A detailed survey on attack types and protections against buffer overflows. Survey of Protections from Buffer-Overflow Attacks (PDF)

2. **An In-Depth Survey of Bypassing Buffer Overflow Mitigation Techniques**
   Muhammad Arif Butt et al. (2022) — Examines mitigation techniques and how attackers adapt to them. An In-Depth Survey of Bypassing Buffer Overflow Mitigation (MDPI)

3. **Fundamentals of Buffer Overflow Attacks and Detection Techniques**
   Bogdan Barchuk — *World Journal of Advanced Research and Reviews* (2025) — Recent paper covering basics, detection, and analysis. Fundamentals of Buffer Overflow Attacks and Detection Techniques (PDF)

4. **Vulnerability Detection and Prevention of SQL Injection**
   Santhosh Kumar & Anaswara (2018) — Discussion on detection and prevention approaches. Vulnerability Detection & Prevention of SQL Injection (SciencePubCo)

5. **A Review on SQL Injection**
   Vidushi & S. Niranjan (2015) — Review paper summarizing SQL injection vulnerability and research. A Review on SQL Injection (IJERT)

6. **A Survey of SQL Injection Attack Detection and Prevention**
   Khaled Elshazly et al. (2014) — Extensive survey article with techniques and analysis. A Survey of SQL Injection Attack Detection & Prevention (SCIRP)