LAB # 6

Lab Assignment: Secure Software Design and Engineering (CY-321)

Instructor: Jaziya Sajid
Topic: Defensive Coding using CodeQL

Objective:

The objective of this lab is to introduce students to defensive coding practices using CodeQL. By the end of this lab, students will be able to identify vulnerabilities in code and implement security controls to mitigate threats.

What is CodeQL?

CodeQL is a static analysis tool developed by GitHub for identifying vulnerabilities in source code. It works by treating code as a database, allowing security researchers and developers to write queries to detect security flaws, coding errors, and other vulnerabilities across large codebases.

Why is CodeQL Used?

- **Automated Security Scanning:** CodeQL helps detect security vulnerabilities early in the development lifecycle.
- **Custom Query Writing:** Developers can create their own queries to find specific security issues.
- **Wide Language Support:** It supports multiple programming languages like C, C++, Java, Python, JavaScript, and more.
- **Integration with CI/CD Pipelines:** It can be integrated into GitHub Actions and other CI/CD workflows for continuous security scanning.
- **Proactive Threat Detection:** Helps organizations comply with secure coding standards by preventing common vulnerabilities like SQL injection, XSS, and buffer overflows.

Industry-Wise Goodness of CodeQL

- **Software Development:** Used for secure coding practices in software projects.
- **Cybersecurity & Penetration Testing:** Helps security teams conduct static code analysis.
- **Financial Services & Banking:** Ensures financial applications are secure against threats.

- **Healthcare & Critical Infrastructure:** Identifies vulnerabilities in sensitive and regulated software.
- **Open Source & Enterprise Applications:** Maintains security in open-source projects and large-scale enterprise applications.

Secure Software is More Than Just Secure Code

Many developers assume that avoiding common mistakes like buffer overflows or SQL injections is enough to secure their applications. However, security is much broader than just avoiding bad code. **A well-secured system integrates security into every stage of the software development lifecycle (SDLC).**

Key Aspects of Secure Software Development:

1. **Secure Design**
    - Before writing a single line of code, software architecture should be designed with security in mind.
    - Consider security principles such as **least privilege**, **defense in depth**, and **fail-safe defaults**.
    - Example: Implementing **role-based access control (RBAC)** to restrict access to sensitive functions.
2. **Threat Modeling**
    - Identifying potential threats early helps prevent vulnerabilities before they occur.
    - Developers should assess the software for **attack vectors**, such as unauthorized access, injection attacks, or denial-of-service (DoS) threats.
    - Example: In a banking application, threat modeling would include assessing risks like **session hijacking** or **data leakage**.
3. **Code Reviews and Static Analysis**
    - Automated security tools, such as **CodeQL**, **SonarQube**, or **SAST (Static Application Security Testing)**, can help detect vulnerabilities.
    - Peer code reviews ensure that security best practices are followed.
    - Example: A security review might catch an instance of **hardcoded API keys** that could be exploited by attackers.

4. **Strong Authentication and Authorization**
    - Proper authentication mechanisms, such as **multi-factor authentication (MFA)**, prevent unauthorized access.
    - Implementing secure **OAuth 2.0** or **OpenID Connect (OIDC)** mechanisms for access control.
    - Example: A banking app requiring **biometric authentication** alongside a password for secure login.
5. **Encryption and Secure Data Handling**
    - Sensitive data should always be encrypted both **at rest** and **in transit**.
    - Use secure encryption standards like **AES-256** for data storage and **TLS 1.3** for secure communications.
    - Example: Encrypting customer credit card details before storing them in a database.

The Consequences of a Single Insecure Line of Code

A **single mistake** in the code can create serious security vulnerabilities. Here are a few examples:

1. Buffer Overflow Due to Unsafe Functions

Insecure Code:

```
char buffer[10]; strcpy(buffer, "This is a long string!"); // No boundary check
```

Problem: The strcpy() function does not check the buffer size, leading to **buffer overflow** and potential **remote code execution (RCE)**.

Solution: Use strncpy() to specify a limit:

```
strncpy(buffer, "This is a long string!", sizeof(buffer) - 1);
```

2: Hardcoded Credentials in Source Code

Insecure Code:

```
DB_PASSWORD = "mysecretpassword"
```

Problem: Attackers can easily find this credential if the code is leaked.

Solution: Store credentials securely in **environment variables** or a secrets manager.

3: Insecure API Calls Leading to Injection Attacks

Insecure Code:

```
user_input = input("Enter username: ")
query = "SELECT * FROM users WHERE username = '" + user_input + "';"
execute_query(query)
```

Problem: If the user inputs admin' --, the SQL query becomes:

```
SELECT * FROM users WHERE username = 'admin' --';
```

This results in SQL Injection, bypassing authentication.

Solution: Use parameterized queries:

```
query = "SELECT * FROM users WHERE username = ?"
execute_query(query, (user_input,))
```

Using Regular Expressions (RegEx) for Input Validation

1. **Validating Email Addresses**

```
import re
email_pattern = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"
email = "user@example.com"
if re.match(email_pattern, email):
    print("Valid email")
else:
    print("Invalid email")
```

2. **Validating Passwords**

```
password_pattern = r"^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$"
password = "Strong@123"
```

```
if re.match(password_pattern, password):
    print("Valid password")
else:
    print("Invalid password")
```

Canonicalization (C14N)

- Ensure that equivalent representations of data are treated consistently.
- Implement canonicalization for strings, URLs, file paths, and JSON.

Example:

```
from urllib.parse import urlparse, urlunparse

def canonicalize_url(url):
    parsed = urlparse(url)
    scheme = parsed.scheme.lower()
    netloc = parsed.netloc.lower()
    path = parsed.path.rstrip('/')
    return urlunparse((scheme, netloc, path, "", "", ""))

print(canonicalize_url("HTTPS://Example.COM/Home/"))
```

Input Sanitization

- Strip harmful characters from user input.
- Substitute unsafe characters with safe alternatives.
- Use literalization to neutralize malicious input.

Example:

```
import re


def strip_html_tags(input_text):
    return re.sub(r'<[^>]*>', '', input_text)


user_input = "Hello! <script>alert('Hacked!');</script>"
sanitized_input = strip_html_tags(user_input)
print(sanitized_input)
```

SQL Injection Prevention using CodeQL

- Use CodeQL to analyze code for potential SQL injection vulnerabilities.
- Implement parameterized queries to prevent SQL injection.

Example:

```
import sqlite3
conn = sqlite3.connect(":memory:")
cursor = conn.cursor()
cursor.execute("CREATE TABLE users (id INTEGER PRIMARY KEY, username TEXT)")

def get_user(username):
    query = "SELECT * FROM users WHERE username = ?"
    cursor.execute(query, (username,))
    return cursor.fetchall()

user_input = "admin' OR 1=1 --"
result = get_user(user_input)
```

```
print(result)
```

Step-by-Step Code Analysis using CodeQL in VS Code

Install Extensions

1. Open **VS Code**.
2. Go to **Extensions** (Ctrl + Shift + X).
3. Search for **"CodeQL"** and install it.
4. Search for **"CodeQL CLI" or "CodeQL Agent"** and install it.

Install Docker (If Not Installed)

1. Download **Docker Desktop** from here.
2. Install and open Docker.
3. Ensure it's **running** before proceeding.

Write Python Code (To Be Analyzed)

1. Open **VS Code** and create a **new folder** (e.g., codeql_analysis).
2. Inside the folder, create a **new Python file**:
   app.py and add this code:

Detect SQL Injection in Python Code using CodeQL

```
def login(username, password):
    if username == "admin" and password == "1234":  # Insecure hardcoded password
        print("Login successful!")
    else:
        print("Invalid credentials.")

```

```
user = input("Enter username: ")
pwd = input("Enter password: ")
login(user, pwd)
```

Create CodeQL Database

1. Open **VS Code Terminal** (Ctrl + ~).
2. Navigate to your project folder:

```
cd path/to/codeql_analysis
```

Run this command to create a CodeQL database:

```
codeql database create my-python-db --language=python --source-root=.
```

my-python-db is the name of the database.

--language=python tells CodeQL that the project is in Python.

--source-root=. sets the root folder for analysis.

Analyze the Code with CodeQL

1. Open **Command Palette** (Ctrl + Shift + P).
2. Type **"CodeQL: Run Query"** and select it.
3. Choose the **CodeQL query** related to **security vulnerabilities**.
4. Run the query, and **check the results**.

Understanding the CodeQL Output

Your output shows the **results of running a CodeQL security analysis** using pre-defined security queries for Python. Here's a breakdown of what it means:

Evaluation Progress

- [1/51 eval 1.9s] Evaluation done; writing results to codeql/python-queries/Diagnostics/ExtractedFiles.bqrs.

- This means **CodeQL executed query 1 out of 51**, took **1.9 seconds**, and stored results in a .bqrs file.

💡 **What's Happening?**

- CodeQL is running a **total of 51 security queries**.
- Each line shows the progress of executing individual queries.

Security Vulnerability Checks

Example:

```
[18/51 eval 11.8s] Evaluation done; writing results to codeql/python-queries/Security/CWE-089/SqlInjection.bqrs.
```

- **Query #18** detected potential **SQL Injection vulnerabilities** in your code.
- The results are saved in Security/CWE-089/SqlInjection.bqrs.

◇ **Common Security Checks in Your Output:**

CWE ID	Vulnerability Type
CWE-079	Cross-Site Scripting (XSS)
CWE-089	SQL Injection
CWE-078	Command Injection
CWE-117	Log Injection
CWE-295	Missing Host Key Validation
CWE-327	Broken Crypto Algorithm
CWE-611	XML External Entity (XXE)
CWE-918	Server-Side Request Forgery (SSRF)
CWE-798	Hardcoded Credentials
CWE-943	NoSQL Injection

These are common vulnerabilities found in real-world applications!

Query Results Stored in .bqrs Files

- **CodeQL saves all detected issues in .bqrs files.**
- You need to extract these results using:

```
codeql bqrs decode --format=csv --output=results.csv codeql/python-queries/Security/CWE-089/SqlInjection.bqrs
```

- **This converts the results into a readable CSV file**.

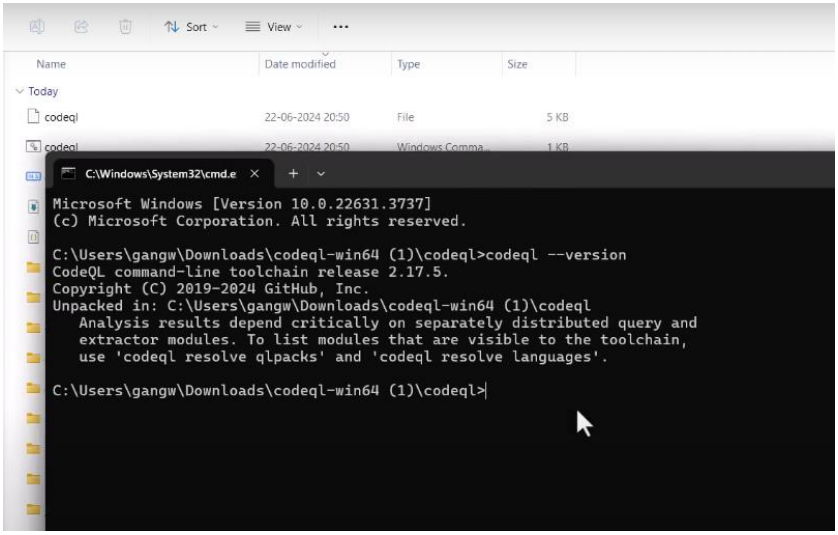
Key Points in the Output

Query Execution:

- The scan runs a set of security-related queries from the python-security-extended.qls query suite.
- These queries check for common vulnerabilities such as:
  - SQL injection
  - Command injection
  - Path injection
  - Cross-site scripting (XSS)
  - Insecure deserialization
  - Hardcoded credentials
  - And many others.

Method 1: Download CodeQL CLI (Recommended)

1. **Download CodeQL CLI**
    - Go to: CodeQL CLI Releases
    - Download the latest **Windows (zip)** release.
2. **Extract the ZIP File**
    - Extract it to C:\codeql (or any preferred location).
    - Inside the folder, find codeql.exe.



3. **Add CodeQL to System PATH**
    - Open **Command Prompt (cmd)** as **Administrator** and run:

```
setx PATH "%PATH%;C:\codeql"
```

Close and reopen CMD to apply changes.

Verify Installation

Run:

```
codeql –version
```

Step 2: Prepare a Vulnerable Python Script

1. **Create a Project Folder**
   Open **CMD** and run:

```
mkdir C:\codeql-lab
cd C:\codeql-lab
```

Create a File **(vuln.py)** with SQL Injection Save this Python script in C:\codeql-lab\vuln.py:

import sqlite3

```
def get_user(username):
    conn = sqlite3.connect("users.db")
    cursor = conn.cursor()
    query = f"SELECT * FROM users WHERE username = '{username}'"
    cursor.execute(query)
    return cursor.fetchall()
user_input = "admin' OR 1=1 --"
print(get_user(user_input))
```

This code is vulnerable to SQL injection due to string concatenation.

Step 3: Create a CodeQL Database

1. **Navigate to the Project Folder**:

```
cd C:\codeql-lab
```

2. **Create a CodeQL Database**:

```
codeql database create my-python-db --language=python --source-root=.
```

- This will **analyze** the Python files and create a database for CodeQL queries.

Step 4: Run CodeQL to Detect SQL Injection

1. **Navigate to the CodeQL Queries Folder**
   - Clone the **official CodeQL repository** (if not already downloaded):

git clone https://github.com/github/codeql C:\codeql-repo

   - **Move to the Python Security Queries Folder**:

```
cd C:\codeql-repo\python\ql\src\Security\CWE-089\
```

2. **Run the SQL Injection Query**:

```
codeql query run --database=C:\codeql-lab\my-python-db SqlInjection.ql
```

   - This will scan the vuln.py script for SQL injection.

Step 5: Analyze CodeQL Output

- If a vulnerability is found, you will see:

```
Alert: Possible SQL Injection detected at vuln.py:5
```

Step 6: Fix the Vulnerability

Modify vuln.py to use **parameterized queries**:

```
import sqlite3

def get_user(username):
    conn = sqlite3.connect("users.db")
    cursor = conn.cursor()
    query = "SELECT * FROM users WHERE username = ?"
    cursor.execute(query, (username,))
    return cursor.fetchall()

```

```
user_input = "admin' OR 1=1 --"
print(get_user(user_input))
```

Step 7: Re-run CodeQL to Verify Fix

1. **Recreate the CodeQL Database**:

```
codeql database create my-python-db --language=python --source-root=.
```

2. **Run the Query Again**:

```
codeql query run --database=C:\codeql-lab\my-python-db SqlInjection.ql
```

☑ **No vulnerabilities should be detected now!**

Student Task:

Detect and Fix Improper Input Validation using CodeQL

Objective:

Use **CodeQL** to analyze a Python script for **improper input validation vulnerabilities**, identify the issues, and apply a secure fix.

Steps:

1. **Create a Python script (student_validation_task.py) with improper input validation.**
    - Write a function that accepts user input without proper **data type, format, or range validation**.
2. **Set up a CodeQL database for analysis.**
    - Initialize and create a **CodeQL database for Python**.
3. **Run CodeQL to detect vulnerabilities.**
    - Use CodeQL to scan for **improper input validation issues**.
4. **Fix the vulnerability.**
    - Implement **data type checks, range validation, and regular expressions (RegEx)** to properly validate user input.

5. **Re-run CodeQL to verify the fix.**
    - Ensure that no vulnerabilities are detected after applying the fix.

Example of an Improperly Validated Input (Before Fixing)

```
def process_age_input():
    age = input("Enter your age: ")  # No validation
    if int(age) > 0:
        print("Valid age")
    else:
        print("Invalid age")
process_age_input()
```