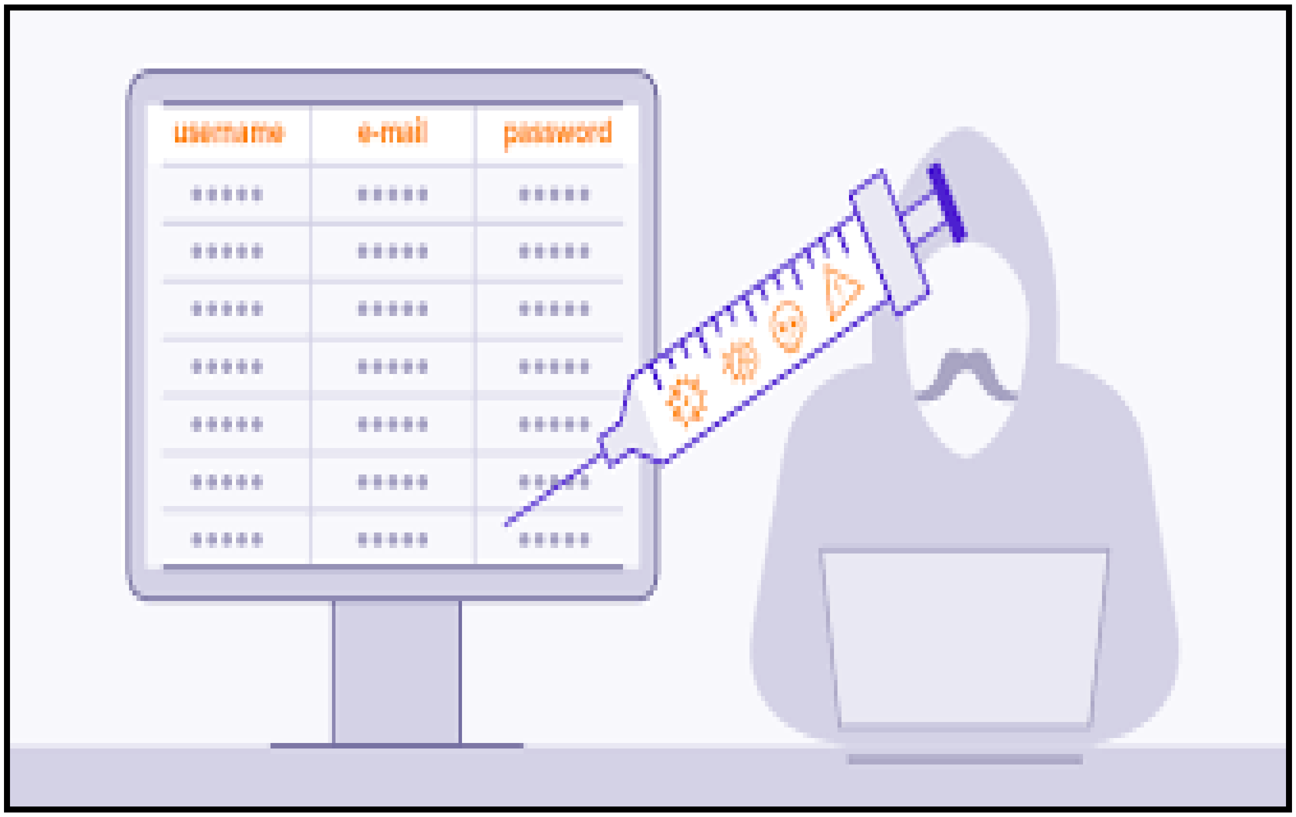
Software Security

SQL Injections

By: Audrey Waggener and Kyleen Mitchell

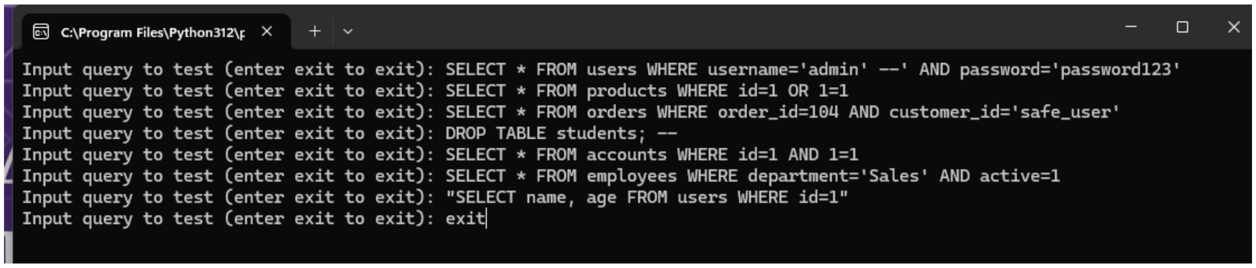


Description

For our project we developed a program to detect vulnerabilities in SQL queries. The program takes SQL queries from the user and analyzes each one. If the program detects a vulnerability in the query it logs it as a vulnerability and generates potential fixes to mitigate the vulnerability. At the end of the program it prints out each query along with if a vulnerability was found and the proposed fixes to mitigate it. The program saves all that information to a csv file.

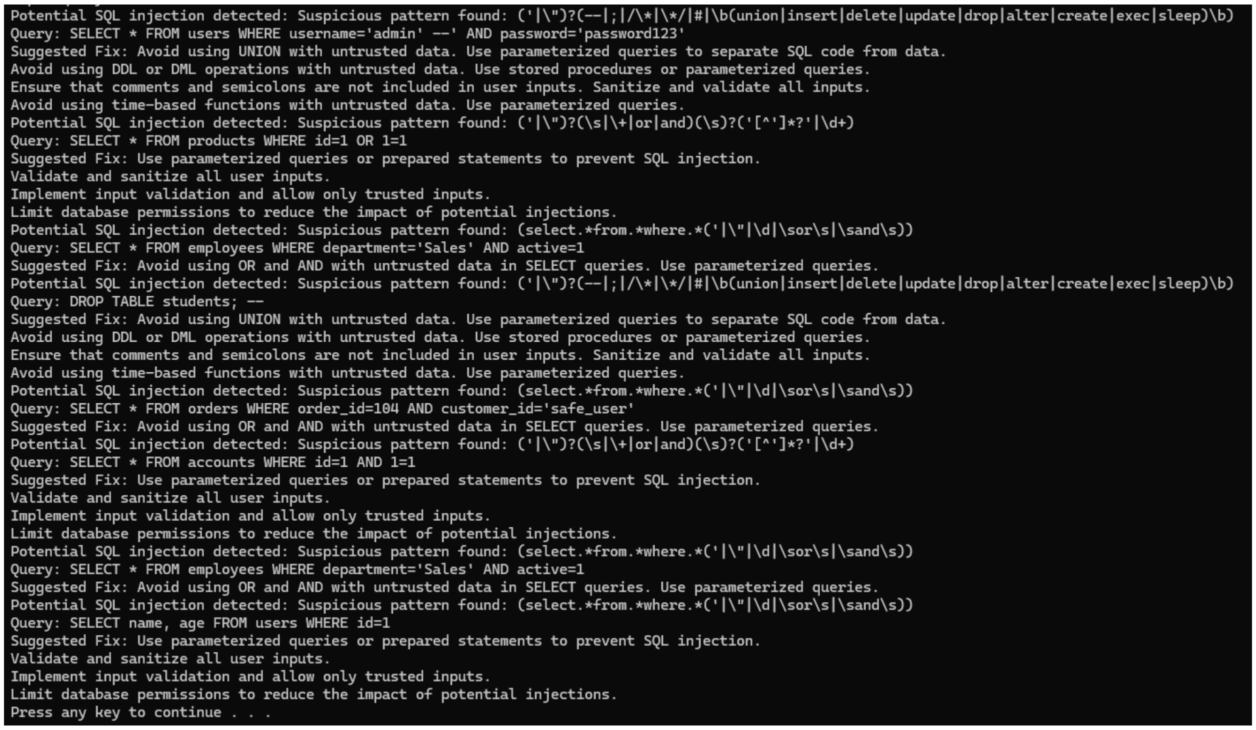
Step by Step

1. Input queries into program:



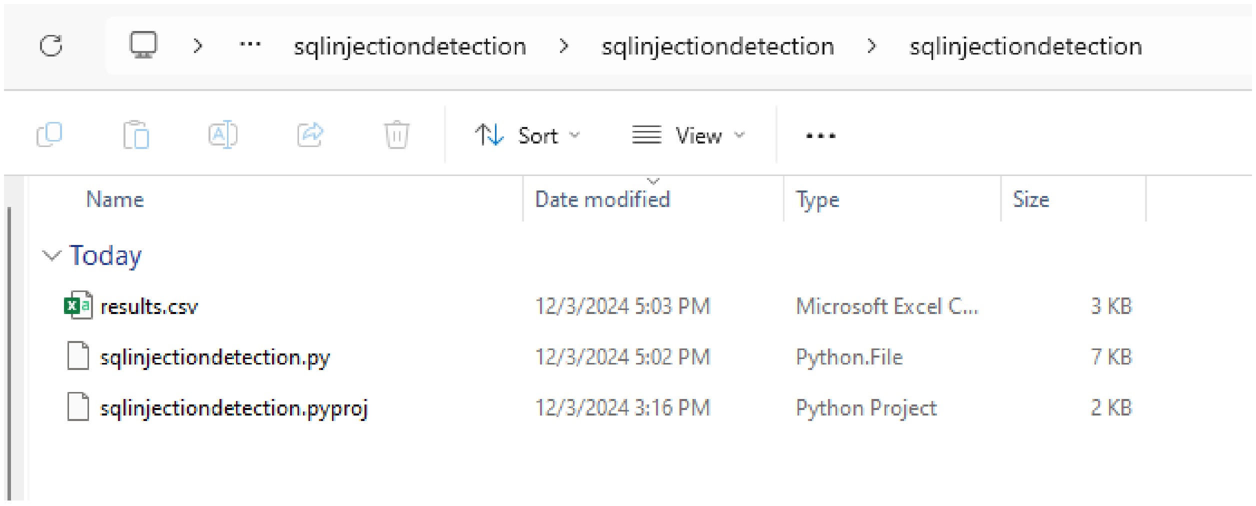
```
Input query to test (enter exit to exit): SELECT * FROM users WHERE username='admin' --' AND password='password123'
Input query to test (enter exit to exit): SELECT * FROM products WHERE id=1 OR 1=1
Input query to test (enter exit to exit): SELECT * FROM orders WHERE order_id=104 AND customer_id='safe_user'
Input query to test (enter exit to exit): DROP TABLE students; --
Input query to test (enter exit to exit): SELECT * FROM accounts WHERE id=1 AND 1=1
Input query to test (enter exit to exit): SELECT * FROM employees WHERE department='Sales' AND active=1
Input query to test (enter exit to exit): "SELECT name, age FROM users WHERE id=1"
Input query to test (enter exit to exit): exit
```

2. Program prints report of found vulnerabilities and suggested fixes



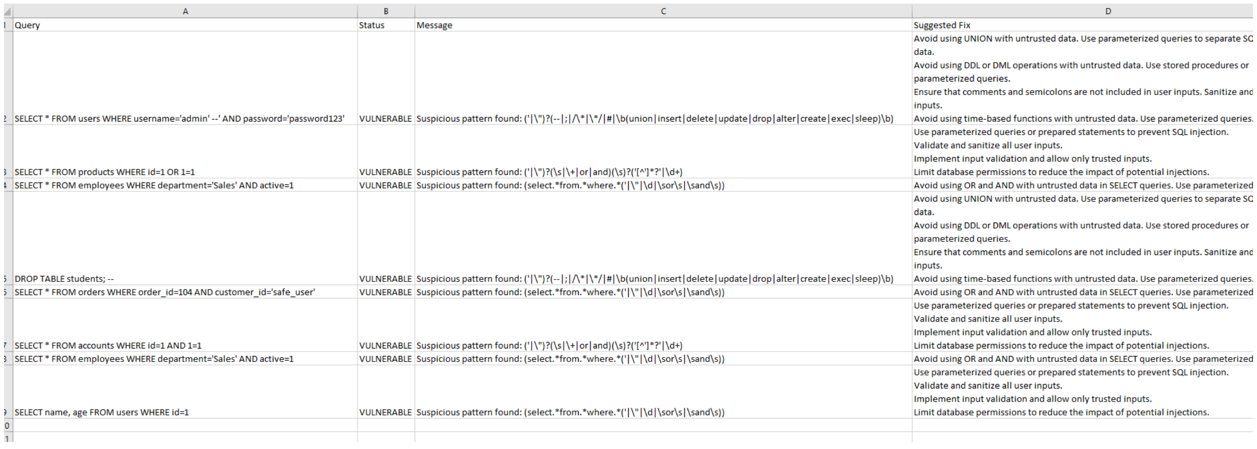
```
Potential SQL injection detected: Suspicious pattern found: ('|\")?(--|;|/\*|\*/|#|\b(union|insert|delete|update|drop|alter|create|exec|sleep)\b)
Query: SELECT * FROM users WHERE username='admin' --' AND password='password123'
Suggested Fix: Avoid using UNION with untrusted data. Use parameterized queries to separate SQL code from data.
Avoid using DDL or DML operations with untrusted data. Use stored procedures or parameterized queries.
Ensure that comments and semicolons are not included in user inputs. Sanitize and validate all inputs.
Avoid using time-based functions with untrusted data. Use parameterized queries.
Potential SQL injection detected: Suspicious pattern found: ('|\")?(\s|\+|or|and)(\s)?('[^']*?'|\d+)
Query: SELECT * FROM products WHERE id=1 OR 1=1
Suggested Fix: Use parameterized queries or prepared statements to prevent SQL injection.
Validate and sanitize all user inputs.
Implement input validation and allow only trusted inputs.
Limit database permissions to reduce the impact of potential injections.
Potential SQL injection detected: Suspicious pattern found: (select.*from.*where.*('|\"|\d|\sor\s|\sand\s))
Query: SELECT * FROM employees WHERE department='Sales' AND active=1
Suggested Fix: Avoid using OR and AND with untrusted data in SELECT queries. Use parameterized queries.
Potential SQL injection detected: Suspicious pattern found: ('|\")?(--|;|/\*|\*/|#|\b(union|insert|delete|update|drop|alter|create|exec|sleep)\b)
Query: DROP TABLE students; --
Suggested Fix: Avoid using UNION with untrusted data. Use parameterized queries to separate SQL code from data.
Avoid using DDL or DML operations with untrusted data. Use stored procedures or parameterized queries.
Ensure that comments and semicolons are not included in user inputs. Sanitize and validate all inputs.
Avoid using time-based functions with untrusted data. Use parameterized queries.
Potential SQL injection detected: Suspicious pattern found: (select.*from.*where.*('|\"|\d|\sor\s|\sand\s))
Query: SELECT * FROM orders WHERE order_id=104 AND customer_id='safe_user'
Suggested Fix: Avoid using OR and AND with untrusted data in SELECT queries. Use parameterized queries.
Potential SQL injection detected: Suspicious pattern found: ('|\")?(\s|\+|or|and)(\s)?('[^']*?'|\d+)
Query: SELECT * FROM accounts WHERE id=1 AND 1=1
Suggested Fix: Use parameterized queries or prepared statements to prevent SQL injection.
Validate and sanitize all user inputs.
Implement input validation and allow only trusted inputs.
Limit database permissions to reduce the impact of potential injections.
Potential SQL injection detected: Suspicious pattern found: (select.*from.*where.*('|\"|\d|\sor\s|\sand\s))
Query: SELECT * FROM employees WHERE department='Sales' AND active=1
Suggested Fix: Avoid using OR and AND with untrusted data in SELECT queries. Use parameterized queries.
Potential SQL injection detected: Suspicious pattern found: (select.*from.*where.*('|\"|\d|\sor\s|\sand\s))
Query: SELECT name, age FROM users WHERE id=1
Suggested Fix: Use parameterized queries or prepared statements to prevent SQL injection.
Validate and sanitize all user inputs.
Implement input validation and allow only trusted inputs.
Limit database permissions to reduce the impact of potential injections.
Press any key to continue . . .
```

3. Go to program files:


sqlinjectiondetection > sqlinjectiondetection > sqlinjectiondetection
Sort
View

Name	Date modified	Type	Size
Today			
results.csv	12/3/2024 5:03 PM	Microsoft Excel C...	3 KB
sqlinjectiondetection.py	12/3/2024 5:02 PM	Python.File	7 KB
sqlinjectiondetection.pyproj	12/3/2024 3:16 PM	Python Project	2 KB

4. Full report stored in results.csv:



Query	Status	Message	Suggested Fix
SELECT * FROM users WHERE username='admin' --' AND password='password123'	VULNERABLE	Suspicious pattern found: ('|\")?(--|;|/*|*/|#|\b(union|insert|delete|update|drop|alter|create|exec|sleep)\b)	Avoid using UNION with untrusted data. Use parameterized queries to separate SC data. Avoid using DDL or DML operations with untrusted data. Use stored procedures or parameterized queries. Ensure that comments and semicolons are not included in user inputs. Sanitize and inputs. Avoid using time-based functions with untrusted data. Use parameterized queries.
SELECT * FROM products WHERE id=1 OR 1=1	VULNERABLE	Suspicious pattern found: ('|\")?(\s|\+|or|and)(\s)?('[^']*?'|\d+)	Use parameterized queries or prepared statements to prevent SQL injection. Validate and sanitize all user inputs. Implement input validation and allow only trusted inputs. Limit database permissions to reduce the impact of potential injections.
SELECT * FROM employees WHERE department='Sales' AND active=1	VULNERABLE	Suspicious pattern found: (select.*from.*where.*('|\"|\d|\sor\s|\sand\s))	Avoid using OR and AND with untrusted data in SELECT queries. Use parameterized
DROP TABLE students; --	VULNERABLE	Suspicious pattern found: ('|\")?(--|;|/*|*/|#|\b(union|insert|delete|update|drop|alter|create|exec|sleep)\b)	Avoid using UNION with untrusted data. Use parameterized queries to separate SC data. Avoid using DDL or DML operations with untrusted data. Use stored procedures or parameterized queries. Ensure that comments and semicolons are not included in user inputs. Sanitize and inputs. Avoid using time-based functions with untrusted data. Use parameterized queries.
SELECT * FROM orders WHERE order_id=104 AND customer_id='safe_user'	VULNERABLE	Suspicious pattern found: (select.*from.*where.*('|\"|\d|\sor\s|\sand\s))	Avoid using OR and AND with untrusted data in SELECT queries. Use parameterized
SELECT * FROM accounts WHERE id=1 AND 1=1	VULNERABLE	Suspicious pattern found: ('|\")?(\s|\+|or|and)(\s)?('[^']*?'|\d+)	Use parameterized queries or prepared statements to prevent SQL injection. Validate and sanitize all user inputs. Implement input validation and allow only trusted inputs. Limit database permissions to reduce the impact of potential injections.
SELECT * FROM employees WHERE department='Sales' AND active=1	VULNERABLE	Suspicious pattern found: (select.*from.*where.*('|\"|\d|\sor\s|\sand\s))	Avoid using OR and AND with untrusted data in SELECT queries. Use parameterized
SELECT name, age FROM users WHERE id=1	VULNERABLE	Suspicious pattern found: (select.*from.*where.*('|\"|\d|\sor\s|\sand\s))	Use parameterized queries or prepared statements to prevent SQL injection. Validate and sanitize all user inputs. Implement input validation and allow only trusted inputs. Limit database permissions to reduce the impact of potential injections.

Outcomes

The outcomes of these SQL injection detection and mitigation solutions are anticipated to enhance the security of systems that process the user inputs significantly. Identifying these harmful patterns in queries, the code will provide a proactive defense mechanism to detect SQL injection attempts before they can be compromised in the database. Logging the functionality will ensure the detailed tracking of suspicious activities, aiding in understanding attack vectors and improving the future defenses. Combining this with best practices like the input sanitization and parameterized queries, this solution will minimize the risks of database breaches. This overall will ensure data integrity and strengthen the overall security posture of the system. The insights gained from flagged queries can inform ongoing security assessments and system enhancements.