Ryan Arveseth, Noah Cook, Tyler DeFreitas, Logan Holland, Scott Malin, Joseph Olsen
02/11/2021

SQL Injection Lab

Test Cases

Valid Input

Ryan Arveseth	username guessMyPassword
Noah Cook	Jigglyname Jigglypassword!
Tyler DeFreitas	Tyler DeFreitas
Logan Holland	__sqdwrd__ c1@RIne7
Scott Malin	FireFrog22 1stick0fgum
Joseph Olsen	secureUser c2VjdXJlUGFzc3dvcmQ

Vulnerabilities

Tautology

Ryan Arveseth	username whatever' OR 'a'='a
Noah Cook	Jiggly Jigglypassword!' OR 1=1
Tyler DeFreitas	Tyler DeFreitas' OR 'True' = 'True;
Logan Holland	spngbb qwerty' or 'a'='a';
Scott Malin	JSmith wrongPass' OR ' 'r' = 'r

Joseph Olsen	hackerlvl1 coolio\’ OR 1=1; --

Union

Ryan Arveseth	username password’ UNION SELECT * FROM USERS
Noah Cook	Jigglyname Jigglypassword’ UNION SELECT * FROM passwords;
Tyler DeFreitas	Tyler DeFreitas’ UNION SELECT * FROM users;
Logan Holland	mrkrbs’ FULL OUTER JOIN confidential_info; -- qwerty
Scott Malin	FireFrog22 fakePass’ UNION SELECT * FROM users
Joseph Olsen	get hacked\’ UNION SELECT * FROM users;

AddStatement

Ryan Arveseth	username something’; INSERT INTO products (product_id, product_name, price) VALUES (nextval(product_sequence), Awesome Sauce’, ‘1.00’);,
Noah Cook	Jigglyname Jigglypassword!’; INSERT INTO passwords (Jigglyname, JigglyPassword);
Tyler DeFreitas	Tyler’; INSERT INTO passwords (Tyler, DeFreitas); DeFreitas
Logan Holland	ptrckstr’; ALTER TABLE users DROP COLUMN passwords; mypwd
Scott Malin	ByeByeData beGone’; DROP TABLE users;
Joseph Olsen	hackervoice Im_In\’;INSERT INTO users (username, password) Values (\‘hackervoice\’\,’ Im_In\’);

Comment

Ryan Arveseth	something‘ OR USERNAME LIKE ‘ryan%’;-- password
Noah Cook	Jigglyname’; /* Jigglypassword!
Tyler DeFreitas	Tyler’; /* DeFreitas
Logan Holland	sndychks’; /* pwd_removed
Scott Malin	FireFrog22’; -- wrongPass
Joseph Olsen	admin\’; -- admin

Code Fragments

RunStrongProgram:

```
// To run the strong program, first it filters the query through TestQueryStrong. If that fails, it
// runs it through TestQueryWeak which will help to hone in on what the potential issues are.
void RunStrongProgram() {
    login_creds creds = GetLoginCreds();

    if (TestQueryStrong(creds)) {
        cout << "Test passed, valid credentials! \n";
    }
    else {
        TestQueryWeak(GenerateQuery(creds));
        cout << "Test failed, invalid credentials. \n";
    }
}
```

.

Function Explanation	This function will run the query strong function. If it passes, then the credentials supplied were valid. If it doesn't, more insight is given for why it didn't work by testing through the weak queries, which will specify what caused the strong test to fail

TestQueryStrong:

```
// To test the query strongly, both the username and password are filtered through a regex expression.
// The regex expression is formatted so that any SQL injection attacks will be detected and rejected.
bool TestQueryStrong(login_creds creds) {
    return(TestValidUsername(creds.username) && TestValidPassword(creds.password));
}
```

Function Explanation	TestQueryStrong tests against the inputted username and password. These functions use regex to test the username and password inputs. It returns true if it passes, and false if it does not.

TestValidUsername:

```
// Runs a regex expression to see if username conforms with an acceptable format
bool TestValidUsername(string str) {
    return regex_match(str, regex("[a-zA-Z0-9_]*"));
}
```

Function Explanation	This function tests whether the user’s username is valid or not with regex. Most usernames cannot contain many, if any special characters - especially not any scripting characters “-”, “;” “/”, etc.

TestValidPassword:

```
// Runs a regex expression to see if password conforms with an acceptable format
bool TestValidPassword(string str) {
    return regex_match(str, regex("[a-zA-Z0-9!@#%&*()$^_=+]*"));
}
```

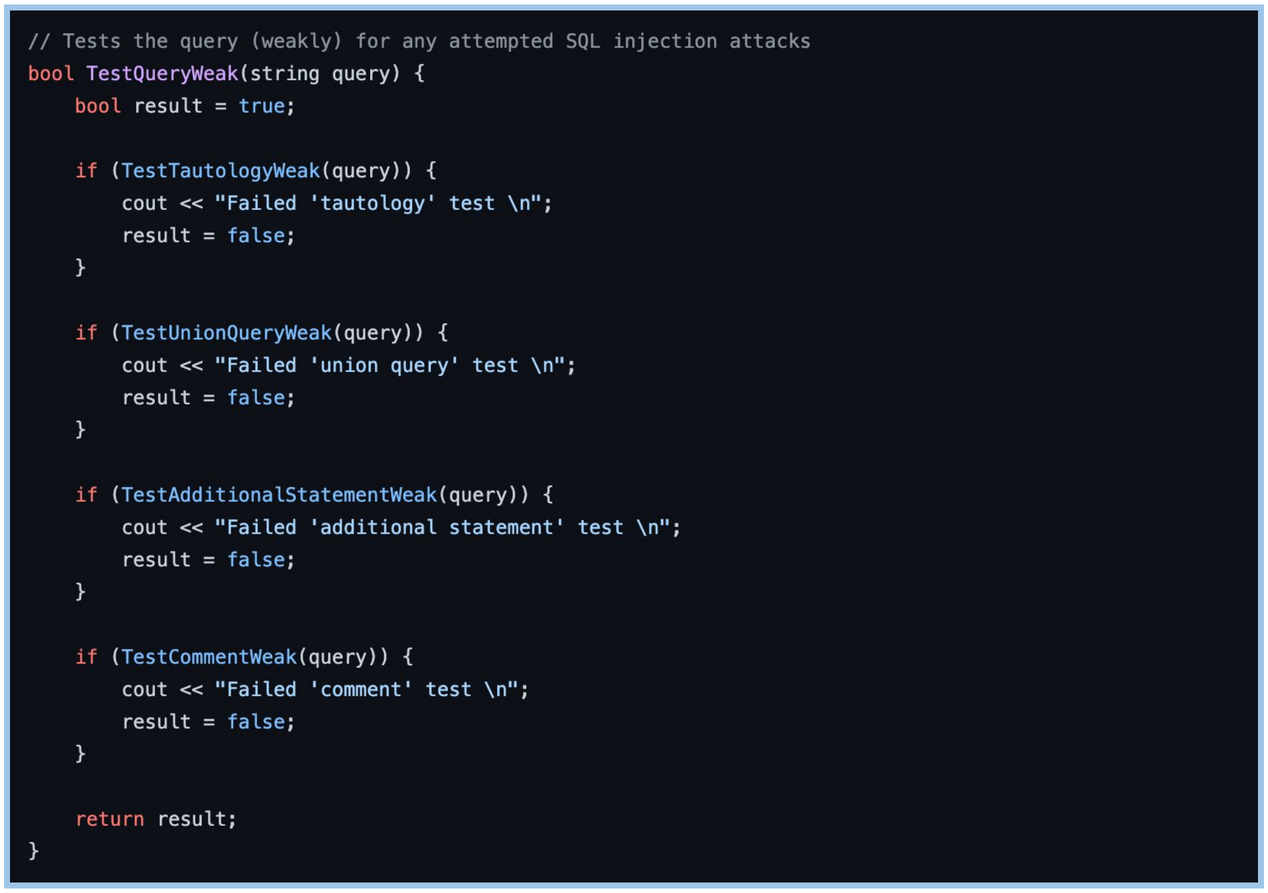
Function Explanation	This function tests whether the supplied password is valid or not with regex. Most of these will not allow you to enter spaces or semicolons - classic injection characters.

RunWeakProgram:

```
// Start of weak test.
void RunWeakProgram() {
    string query = GenerateQuery(GetLoginCreds());

    (TestQueryWeak(query)) ?
        cout << "Test passed, valid credentials! \n"
    :
        cout << "Test failed, invalid credentials. \n";
}
```

Function Explanation	RunWeakProgram will call the TestQueryWeak function, which returns a boolean to determine what should be outputted to the user, like “Valid/Invalid Credentials”.

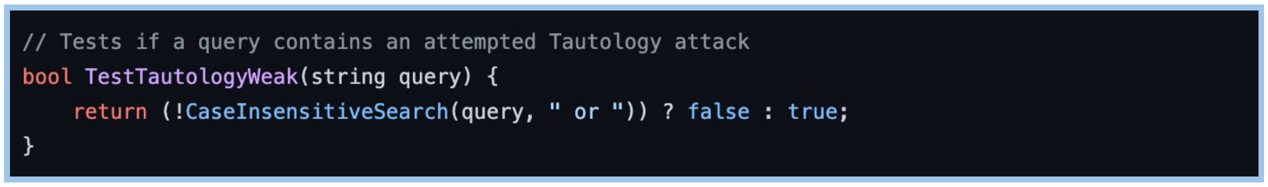
TestQueryWeak:



```
// Tests the query (weakly) for any attempted SQL injection attacks
bool TestQueryWeak(string query) {
    bool result = true;

    if (TestTautologyWeak(query)) {
        cout << "Failed 'tautology' test \n";
        result = false;
    }

    if (TestUnionQueryWeak(query)) {
        cout << "Failed 'union query' test \n";
        result = false;
    }

    if (TestAdditionalStatementWeak(query)) {
        cout << "Failed 'additional statement' test \n";
        result = false;
    }

    if (TestCommentWeak(query)) {
        cout << "Failed 'comment' test \n";
        result = false;
    }

    return result;
}
```

Function Explanation	TestQueryWeak tests the 4 different types of sql injection attacks and returns false if the string fails the check.

TestTautologyWeak:



```
// Tests if a query contains an attempted Tautology attack
bool TestTautologyWeak(string query) {
    return (!CaseInsensitiveSearch(query, " or ")) ? false : true;
}
```

Function Explanation	Checks for any occurrences of “ or “ in the supplied string. This would indicate the presence of a tautology attack.

TestUnionQueryWeak:

```
// Tests if a query contains a Union Query attack
bool TestUnionQueryWeak(string query) {
    return (!CaseInsensitiveSearch(query, " union ") && !CaseInsensitiveSearch(query, " join "))
        ? false : true;
}
```

Function Explanation	Checks for a " union " or " join " in the supplied string. Either of these could mean a union query

TestAdditionalStatementWeak:

```
// Tests if a query contains an attempted Additional Statement attack
bool TestAdditionalStatementWeak(string query) {
    return (query.substr(0, query.length() - 2).find(';') == -1) ? false : true;
}
```

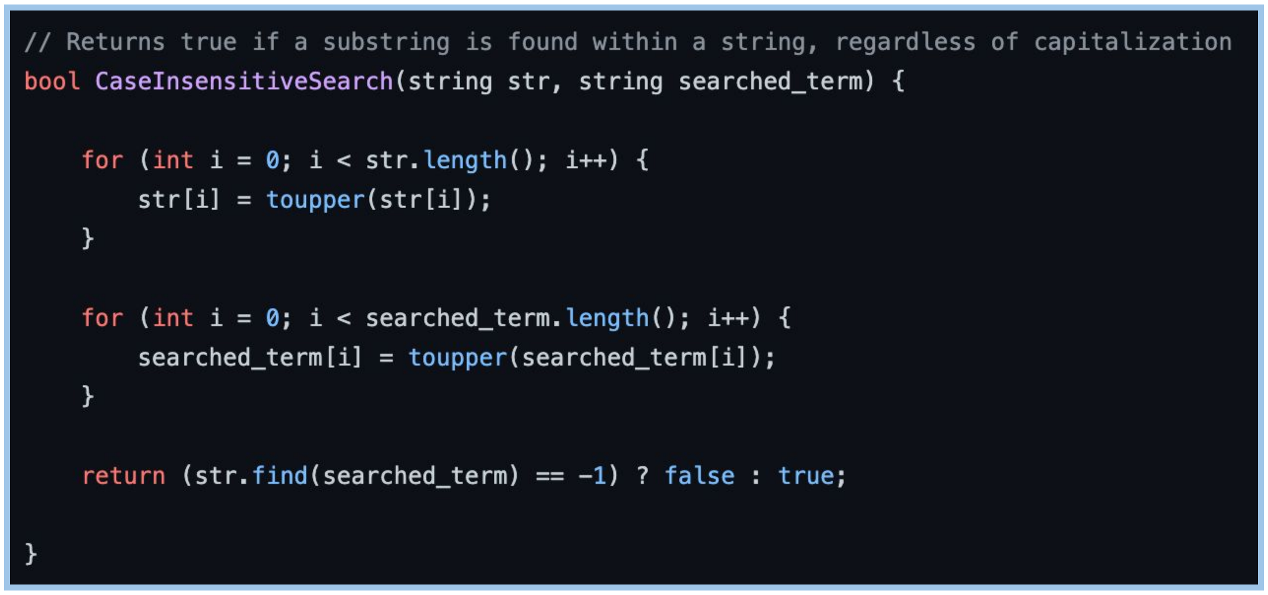
Function Explanation	Searches for the presence of a semicolon, which would indicate a termination of a sql statement, and allow the user to add additional sql to the existing query.

TestCommentWeak:

```
bool TestCommentWeak(string query) {
    return (query.find("--") == -1 and query.find("/*") == -1) ? false : true;
}
```

Function Explanation	Searches the supplied string for two dashes, which would suggest the presence of a comment injection attack.

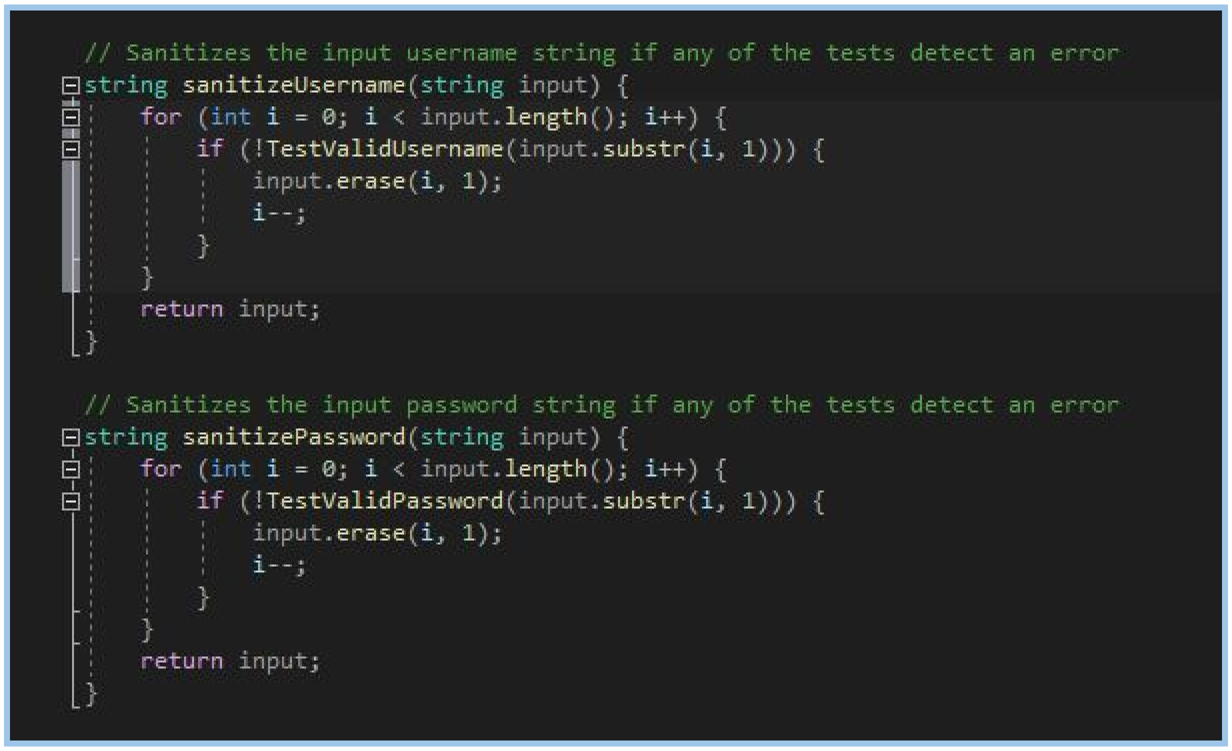
CaseInsensitiveSearch:



```
// Returns true if a substring is found within a string, regardless of capitalization
bool CaseInsensitiveSearch(string str, string searched_term) {

    for (int i = 0; i < str.length(); i++) {
        str[i] = toupper(str[i]);
    }

    for (int i = 0; i < searched_term.length(); i++) {
        searched_term[i] = toupper(searched_term[i]);
    }

    return (str.find(searched_term) == -1) ? false : true;

}
```

Function Explanation	Parses both supplied strings, then compares them to each other in upper-case.

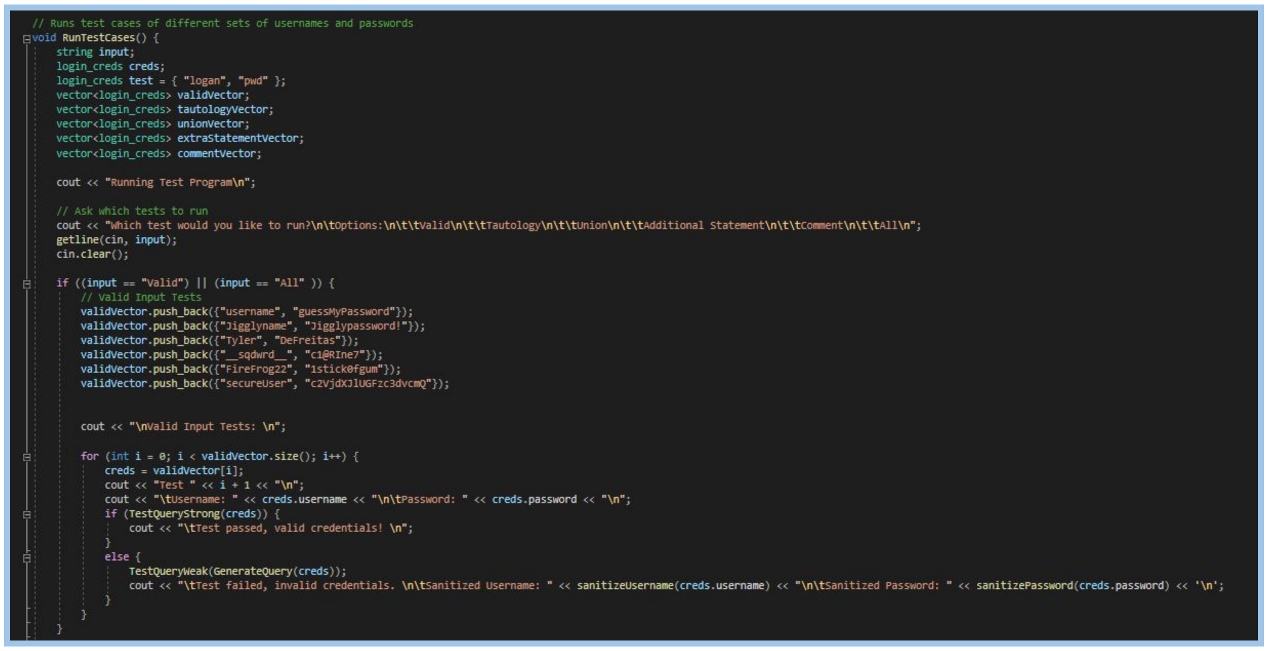
SanitizeUsername and sanitizePassword:



```cpp
// Sanitizes the input username string if any of the tests detect an error
string sanitizeUsername(string input) {
    for (int i = 0; i < input.length(); i++) {
        if (!TestValidUsername(input.substr(i, 1))) {
            input.erase(i, 1);
            i--;
        }
    }
    return input;
}

// Sanitizes the input password string if any of the tests detect an error
string sanitizePassword(string input) {
    for (int i = 0; i < input.length(); i++) {
        if (!TestValidPassword(input.substr(i, 1))) {
            input.erase(i, 1);
            i--;
        }
    }
    return input;
}
```

Function Explanation	Takes either the username or password and returns a sanitized version of it.

RunTestCases:

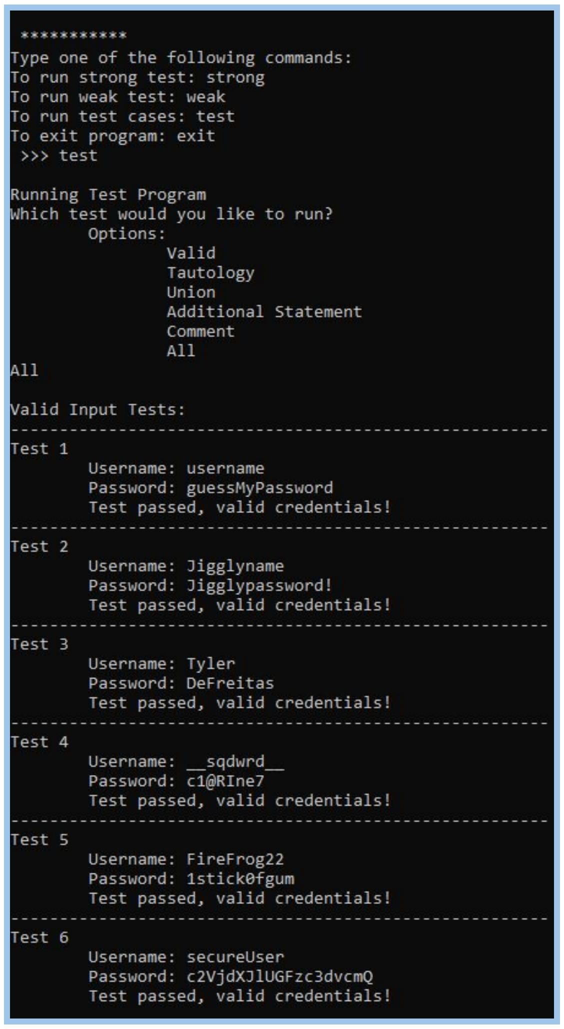


```cpp
// Runs test cases of different sets of usernames and passwords
void RunTestCases() {
    string input;
    login_creds creds;
    login_creds test = { "logan", "pwd" };
    vector<login_creds> validVector;
    vector<login_creds> tautologyVector;
    vector<login_creds> unionVector;
    vector<login_creds> extraStatementVector;
    vector<login_creds> commentVector;

    cout << "Running Test Program\n";

    // Ask which tests to run
    cout << "Which test would you like to run?\n\tOptions:\n\t\tValid\n\t\tTautology\n\t\tUnion\n\t\tAdditional Statement\n\t\tComment\n\t\tAll\n";
    getline(cin, input);
    cin.clear();

    if ((input == "Valid") || (input == "All" )) {
        // Valid Input Tests
        validVector.push_back({"username", "guessMyPassword"});
        validVector.push_back({"Jigglyname", "Jigglypassword!"});
        validVector.push_back({"Tyler", "DeFreitas"});
        validVector.push_back({"__sqdwrd__", "c1@RIne7"});
        validVector.push_back({"FireFrog22", "1stick0fgum"});
        validVector.push_back({"secureUser", "c2VjdXJlUGFzc3dvcmQ"});


        cout << "\nValid Input Tests: \n";

        for (int i = 0; i < validVector.size(); i++) {
            creds = validVector[i];
            cout << "Test " << i + 1 << "\n";
            cout << "\tUsername: " << creds.username << "\n\tPassword: " << creds.password << "\n";
            if (TestQueryStrong(creds)) {
                cout << "\tTest passed, valid credentials! \n";
            }
            else {
                TestQueryWeak(GenerateQuery(creds));
                cout << "\tTest failed, invalid credentials. \n\tSanitized Username: " << sanitizeUsername(creds.username) << "\n\tSanitized Password: " << sanitizePassword(creds.password) << '\n';
            }
        }
    }
```

Function Explanation	This function runs different sets of test cases against the verification tests. It then returns whether or not each test case passed. If the test fails, it returns a sanitized version of the inputs.

Justification

Valid Tests

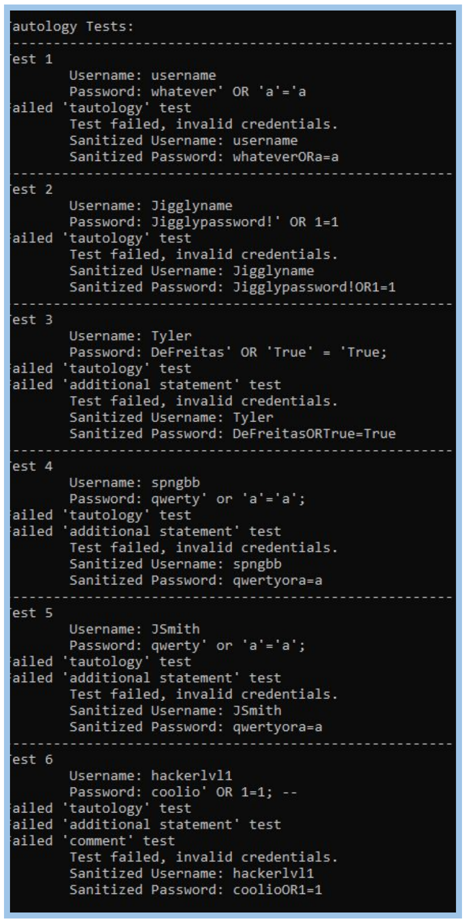


```
 ***********
Type one of the following commands:
To run strong test: strong
To run weak test: weak
To run test cases: test
To exit program: exit
 >>> test

Running Test Program
Which test would you like to run?
        Options:
                Valid
                Tautology
                Union
                Additional Statement
                Comment
                All
All

Valid Input Tests:
-------------------------------------------------------
Test 1
        Username: username
        Password: guessMyPassword
        Test passed, valid credentials!
-------------------------------------------------------
Test 2
        Username: Jigglyname
        Password: Jigglypassword!
        Test passed, valid credentials!
-------------------------------------------------------
Test 3
        Username: Tyler
        Password: DeFreitas
        Test passed, valid credentials!
-------------------------------------------------------
Test 4
        Username: __sqdwrd__
        Password: c1@RIne7
        Test passed, valid credentials!
-------------------------------------------------------
Test 5
        Username: FireFrog22
        Password: 1stick0fgum
        Test passed, valid credentials!
-------------------------------------------------------
Test 6
        Username: secureUser
        Password: c2VjdXJlUGFzc3dvcmQ
        Test passed, valid credentials!
```

The Login credentials are taken from the input given to the user and ran through two tests to check for a valid username and password. The TestValidUsername function returns true if each character in the string matches a list of valid characters which are: a-zA-Z0-9_ If the string contains characters outside of this range the function returns true and the query will not be created and sent to the database.

The function to testValidPassword performs the same operation but the list of it allows a few more characters: a-zA-Z0-9!@#%&*()$^_=+

Tautology Tests



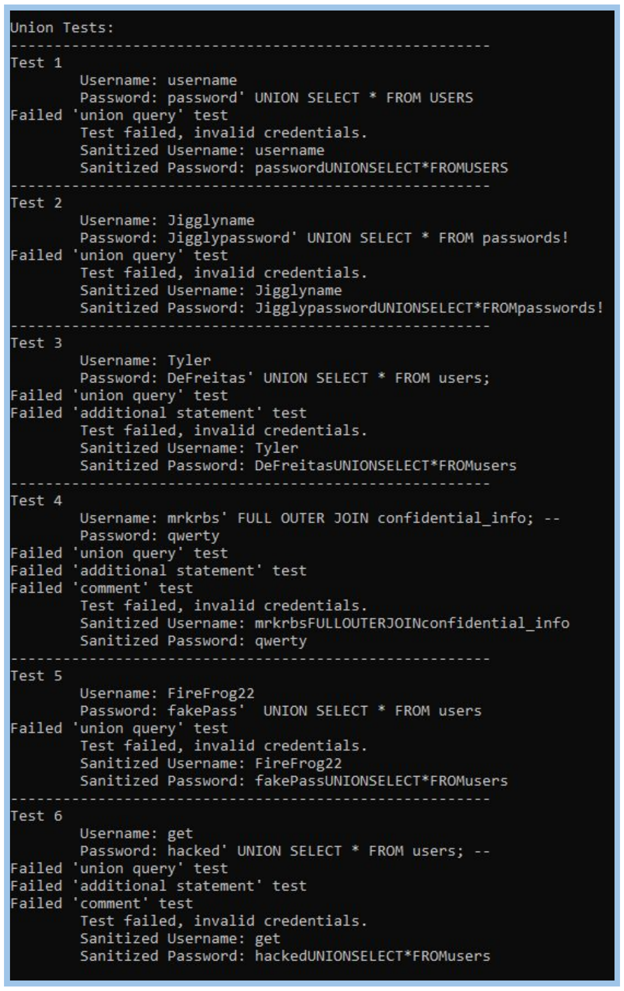
```
autology Tests:
----------------------------------------------------
est 1
        Username: username
        Password: whatever' OR 'a'='a
ailed 'tautology' test
        Test failed, invalid credentials.
        Sanitized Username: username
        Sanitized Password: whateverORa=a
----------------------------------------------------
est 2
        Username: Jigglyname
        Password: Jigglypassword!' OR 1=1
ailed 'tautology' test
        Test failed, invalid credentials.
        Sanitized Username: Jigglyname
        Sanitized Password: Jigglypassword!OR1=1
----------------------------------------------------
est 3
        Username: Tyler
        Password: DeFreitas' OR 'True' = 'True;
ailed 'tautology' test
ailed 'additional statement' test
        Test failed, invalid credentials.
        Sanitized Username: Tyler
        Sanitized Password: DeFreitasORTrue=True
----------------------------------------------------
est 4
        Username: spngbb
        Password: qwerty' or 'a'='a';
ailed 'tautology' test
ailed 'additional statement' test
        Test failed, invalid credentials.
        Sanitized Username: spngbb
        Sanitized Password: qwertyora=a
----------------------------------------------------
est 5
        Username: JSmith
        Password: qwerty' or 'a'='a';
ailed 'tautology' test
ailed 'additional statement' test
        Test failed, invalid credentials.
        Sanitized Username: JSmith
        Sanitized Password: qwertyora=a
----------------------------------------------------
est 6
        Username: hackerlvl1
        Password: coolio' OR 1=1; --
ailed 'tautology' test
ailed 'additional statement' test
ailed 'comment' test
        Test failed, invalid credentials.
        Sanitized Username: hackerlvl1
        Sanitized Password: coolioOR1=1
```

Vulnerable: The vulnerabilities in this test include user input containing the ‘or’ keyword. The vulnerability is caused by a boolean expression equating to true and therefore allowing access to data in the database.

Weak Mitigation: Weak mitigation has been reached by simply searching for the ‘or’ keyword. If it is found the function will return true and the query string will not be created. This is still vulnerable to an injection because the weak mitigation only creates a blocklist that includes lowercase ‘or’

Strong Mitigation: Strong mitigation has been achieved by filtering the users input for all special characters. If they are found the query string will not be created and passed to the db.

Union Tests

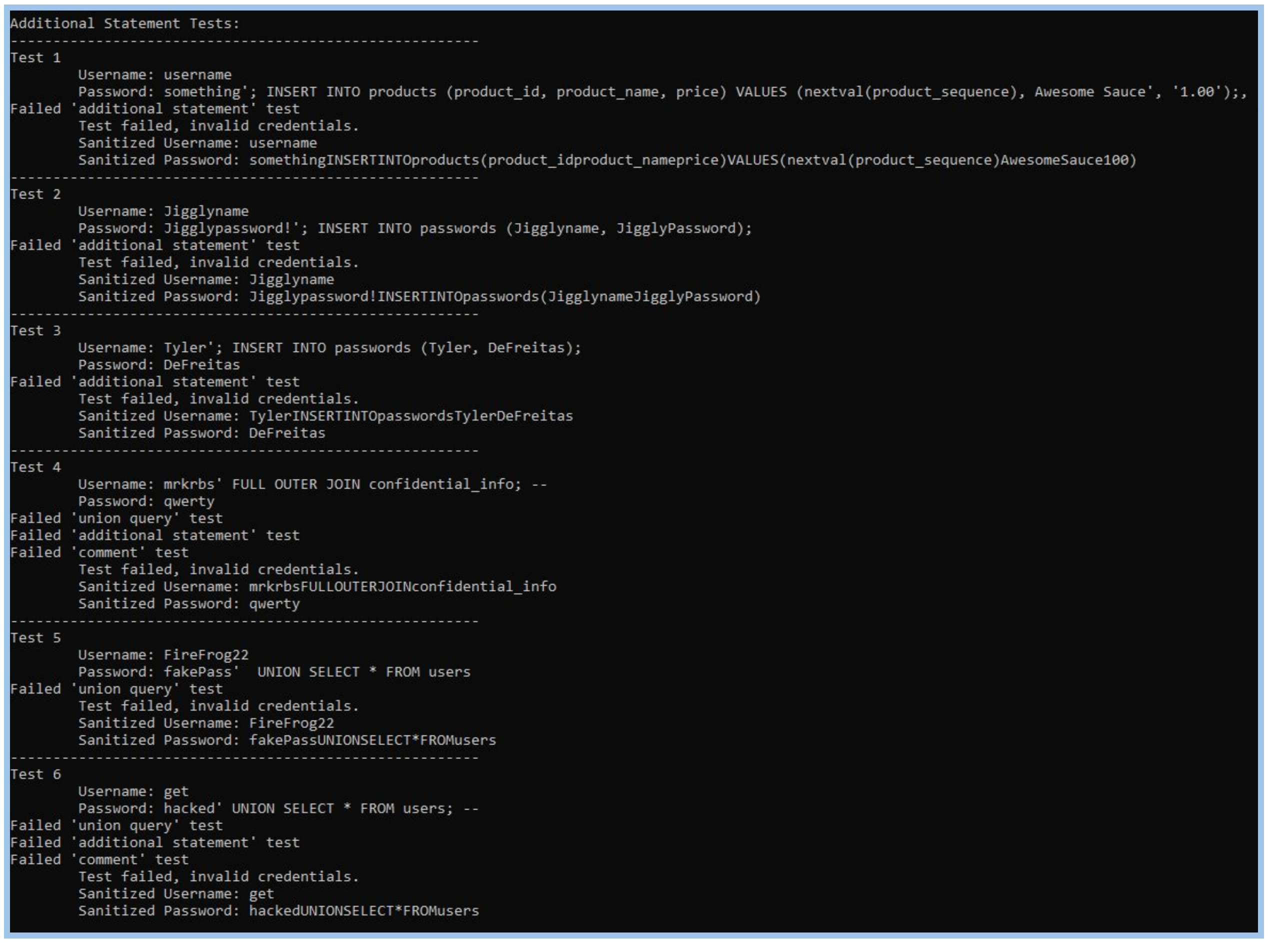

```
Union Tests:
-------------------------------------------------------
Test 1
        Username: username
        Password: password' UNION SELECT * FROM USERS
Failed 'union query' test
        Test failed, invalid credentials.
        Sanitized Username: username
        Sanitized Password: passwordUNIONSELECT*FROMUSERS
-------------------------------------------------------
Test 2
        Username: Jigglyname
        Password: Jigglypassword' UNION SELECT * FROM passwords!
Failed 'union query' test
        Test failed, invalid credentials.
        Sanitized Username: Jigglyname
        Sanitized Password: JigglypasswordUNIONSELECT*FROMpasswords!
-------------------------------------------------------
Test 3
        Username: Tyler
        Password: DeFreitas' UNION SELECT * FROM users;
Failed 'union query' test
Failed 'additional statement' test
        Test failed, invalid credentials.
        Sanitized Username: Tyler
        Sanitized Password: DeFreitasUNIONSELECT*FROMusers
-------------------------------------------------------
Test 4
        Username: mrkrbs' FULL OUTER JOIN confidential_info; --
        Password: qwerty
Failed 'union query' test
Failed 'additional statement' test
Failed 'comment' test
        Test failed, invalid credentials.
        Sanitized Username: mrkrbsFULLOUTERJOINconfidential_info
        Sanitized Password: qwerty
-------------------------------------------------------
Test 5
        Username: FireFrog22
        Password: fakePass'  UNION SELECT * FROM users
Failed 'union query' test
        Test failed, invalid credentials.
        Sanitized Username: FireFrog22
        Sanitized Password: fakePassUNIONSELECT*FROMusers
-------------------------------------------------------
Test 6
        Username: get
        Password: hacked' UNION SELECT * FROM users; --
Failed 'union query' test
Failed 'additional statement' test
Failed 'comment' test
        Test failed, invalid credentials.
        Sanitized Username: get
        Sanitized Password: hackedUNIONSELECT*FROMusers
```

Vulnerable: Vulnerabilities exist in the user input because they attempt to join multiple SQL statements together by passing SQL statements and special characters through the username or password input field.

Weak Mitigation: The weak mitigation is accomplished by searching for the words union or join in the strings captured by the user input field. If found the query string will not be created. This is still vulnerable to an injection because the weak mitigation only creates a blocklist that includes lowercase ‘union’ and ‘join’.

Strong Mitigation: This was accomplished specifying the range of valid characters. The individual inputs are checked to make sure they can not submit sql commands and use special characters.

Add Statement Tests



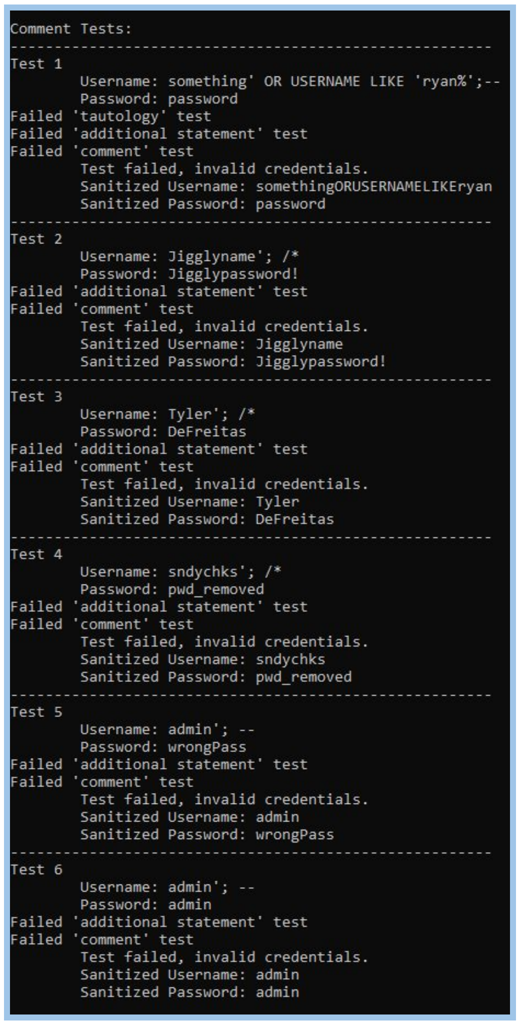
```
Additional Statement Tests:
-------------------------------------------------------
Test 1
        Username: username
        Password: something'; INSERT INTO products (product_id, product_name, price) VALUES (nextval(product_sequence), Awesome Sauce', '1.00');,
Failed 'additional statement' test
        Test failed, invalid credentials.
        Sanitized Username: username
        Sanitized Password: somethingINSERTINTOproducts(product_idproduct_nameprice)VALUES(nextval(product_sequence)AwesomeSauce100)
-------------------------------------------------------
Test 2
        Username: Jigglyname
        Password: Jigglypassword!'; INSERT INTO passwords (Jigglyname, JigglyPassword);
Failed 'additional statement' test
        Test failed, invalid credentials.
        Sanitized Username: Jigglyname
        Sanitized Password: Jigglypassword!INSERTINTOpasswords(JigglynameJigglyPassword)
-------------------------------------------------------
Test 3
        Username: Tyler'; INSERT INTO passwords (Tyler, DeFreitas);
        Password: DeFreitas
Failed 'additional statement' test
        Test failed, invalid credentials.
        Sanitized Username: TylerINSERTINTOpasswordsTylerDeFreitas
        Sanitized Password: DeFreitas
-------------------------------------------------------
Test 4
        Username: mrkrbs' FULL OUTER JOIN confidential_info; --
        Password: qwerty
Failed 'union query' test
Failed 'additional statement' test
Failed 'comment' test
        Test failed, invalid credentials.
        Sanitized Username: mrkrbsFULLOUTERJOINconfidential_info
        Sanitized Password: qwerty
-------------------------------------------------------
Test 5
        Username: FireFrog22
        Password: fakePass'  UNION SELECT * FROM users
Failed 'union query' test
        Test failed, invalid credentials.
        Sanitized Username: FireFrog22
        Sanitized Password: fakePassUNIONSELECT*FROMusers
-------------------------------------------------------
Test 6
        Username: get
        Password: hacked' UNION SELECT * FROM users; --
Failed 'union query' test
Failed 'additional statement' test
Failed 'comment' test
        Test failed, invalid credentials.
        Sanitized Username: get
        Sanitized Password: hackedUNIONSELECT*FROMusers
```

Vulnerable: Vulnerabilities exist in the user input from the user adding a semi colon and an additional statement to access the database.

Weak Mitigation: Check for a ‘;’ in the input and if it exists return a negative check value. If this happens do not create the query to the database.

Strong Mitigation: This was accomplished specifying the range of valid characters. The individual inputs are checked to make sure they can not submit sql commands and use special characters.

Comment Tests



```
Comment Tests:
-------------------------------------------------------
Test 1
        Username: something' OR USERNAME LIKE 'ryan%';--
        Password: password
Failed 'tautology' test
Failed 'additional statement' test
Failed 'comment' test
        Test failed, invalid credentials.
        Sanitized Username: somethingORUSERNAMELIKEryan
        Sanitized Password: password
-------------------------------------------------------
Test 2
        Username: Jigglyname'; /*
        Password: Jigglypassword!
Failed 'additional statement' test
Failed 'comment' test
        Test failed, invalid credentials.
        Sanitized Username: Jigglyname
        Sanitized Password: Jigglypassword!
-------------------------------------------------------
Test 3
        Username: Tyler'; /*
        Password: DeFreitas
Failed 'additional statement' test
Failed 'comment' test
        Test failed, invalid credentials.
        Sanitized Username: Tyler
        Sanitized Password: DeFreitas
-------------------------------------------------------
Test 4
        Username: sndychks'; /*
        Password: pwd_removed
Failed 'additional statement' test
Failed 'comment' test
        Test failed, invalid credentials.
        Sanitized Username: sndychks
        Sanitized Password: pwd_removed
-------------------------------------------------------
Test 5
        Username: admin'; --
        Password: wrongPass
Failed 'additional statement' test
Failed 'comment' test
        Test failed, invalid credentials.
        Sanitized Username: admin
        Sanitized Password: wrongPass
-------------------------------------------------------
Test 6
        Username: admin'; --
        Password: admin
Failed 'additional statement' test
Failed 'comment' test
        Test failed, invalid credentials.
        Sanitized Username: admin
        Sanitized Password: admin
```

Vulnerable: The user can place a '--' or '/*' in the input to comment out code after the query statement. This can remove things such as password verification and allow elevated access to the database.

Weak Mitigation: Check for both '--' or '/*' in the input and if either exists or both exist return a negative check value. If this happens do not create the query to the database.

Strong Mitigation: This was accomplished specifying the range of valid characters. The individual inputs are checked to make sure they can not submit sql commands and use special characters.