SOQL Injection Prevention

There are a number of techniques you can use to prevent SOQL injection, but how you use them depends on what you're trying to accomplish with your query. We'll cover these techniques in this unit.

- Static queries with bind variables
- String.escapeSingleQuotes()
- Type casting
- Replacing characters
- Whitelisting

1. Static Bind Variables.

Example :

```
    public PageReference sampleQuery(String paramVal){
        String queryString = ' % ' + paramVal + ' % ';
        List<Contact> queryRes = [Select Id,Name From Contact Where Name like :queryString ];
        return null;
    }
```

2.Escape Single Quotes.

Examples :

```
   public PageReference sampleQuery(String paramVal){
        String queryString = ' Select Id,Name From Contact Where name like ' + '\'
            String.escapeSingleQuotes(paramVal)\')';

        List<Contact> queryRes = database.Query(queryString);
        return null;
    }
```

3. TypeCasting.

Example :

```
   public PageReference sampleQuery(String paramVal){
        String queryString = ' Select Id,Name From Contact Where name like ' + '\'
            String.valueOf(paramVal)\')';

        List<Contact> queryRes = database.execute(queryString);
        return null;
    }
```

SOQL Defense Techniques:

There are some techniques we can apply to validate our SOQL query to prevent the SOQL injection. Some of them are listed below:

1. Static queries with bind variables
2. Type casting
3. Replacing characters
4. Whitelisting

Let's see how these techniques are handled to prevent the vulnerability.

Static queries with bind variables

Using static queries in our application is the most recommended solution to prevent the SOQL Injection. Consider the following query which uses the user input (the var variable) directly in a SOQL query opens the application up to SOQL injection.

```
1. String query = 'select id from contact where firstname =\''+var+'\'';

2. queryResult = Database.execute(query);
```

We can modify the query as below so that the user input is taken as a variable and not as an executable statement in the query.

```
1. queryResult = [select id from contact where firstname =: var];
```

For this query, even though the user gives the value as test' LIMIT 1, it looks for any first names that are "test' LIMIT 1" in the database and the query is efficient now.

Type Casting

In our queries, if we are sure about what data type we are going to refer, then we can type cast it; so that we can eliminate the fake input. For example, if you are expecting an input type such as Boolean or integer, you can cast the user input value to the expected type.

```
1. String query = 'Select Name, Address from Contact where isActive = ' + Boolean(input);
```

Even though we get an erroneous input, the type casting will throw error.

Whitelisting

If we want to support dynamically selecting fields from the object we need to query, and also if we know what fields needs are expected to be selected, then we can check that the user input is one of those field names.

```
Set<String> fields = new Set<String>();
fields.add('myField1');
fields.add('myField2');
fields.add('myField3');
if(!(fields.contains(inputField)){
 .. Throw error ..
}
```

We can throw error if the user input is coming in for unexpected fields.

Replacing Characters

When Type casting / whitelisting doesn't prevent the adequate vulnerability against prevention, blacklisting otherwise called as Replacing Characters will be handy.

Consider our user input is like this,

```
String query = 'select id from user where isActive= '+var;
```

And the SOQL injection input as

```
true AND ReceivesAdminInfoEmails=true
```

thus, the resulting query will lead to retrieving unintended data.

So, the fix can be removing all white spaces to make the erroneous input invalid.

```
String query = 'select id from user where isActive='+var.replaceAll('[^\w]','');
```