CQL/SQL Injection and its prevention

What is CQL Injection?

As per OWASP (on SQL Injection - rewritten for CQL Injection):

"A CQL injection attack consists of insertion or "injection" of a CQL query via the input data from the client to the application. A successful CQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database, recover the content of a given file present on the DB file system and in some cases issue commands to the operating system. CQL injection attacks are a type of injection attack, in which CQL commands are injected into data-plane input in order to effect the execution of predefined CQL commands."

In simple terms, modifying a Query's behavior by injecting/inserting unintended String into the query. That is, you intend the query to do something, but by injecting something into the query, the actual query execution does something else. Let's see a quick example in the next section.

A Simple demonstration with the much similar SQL Injcetion

Let's assume we have a simple banking web application. To access this bank app contents, one need to login to it. The bank app follows a simple form-based authentication, wherein the user fills a form with his username and password.

The form data is posted to the server on form submission. On the Server side, we validate the submitted username and password against out DB records. The SQL query we use is as follows(assume we are executing the following query using java.sql.Statement - General Statement).

```
//Validates the user login. Returns true if username and pwd is valid. Else
returns false
public boolean login(String userName, String pwd) {
  Connection conn = getConnection();
 Statement stmt = connection.createStatement();
 return stmt.execute("SELECT * FROM user WHERE uname = '" + userName + "'
AND pwd = '" + pwd + "'");
}
```

As you can see from the above code snippet, we simply check if the username, password combination exists in our DB.

Now let's look into some use case:

1) A normal, valid user tries to login

- The user enters username: "ram" and password: "ram@123" in the login form and submits it.
- The login form is submitted, and the form data reaches the server
- The login(..) java method is invoked. The following SQL statement is executed.

```
SELECT * FROM user WHERE uname = 'ram' AND pwd = 'ram@123'
```

- The above SQL returns a Resultset as there is an entry in the DB matching this uname and pwd and the method simply returns true.

2) Now, a hacker tries to login to the bank web app without having valid credentials.

- The hacker enters username: "' OR '1'='1' --" and password: "idonotcare" in the login form and submits it.
- The login form is submitted, and the form data reaches the server

- The login(..) java method is invoked. The following SQL statement is executed.

```
SELECT * FROM user WHERE uname = '' OR '1'='1' --' AND pwd =
'idonotcare'
```

- The above SQL also returns a Resultset(the set of all the user records in the user table) and the method returns true.
- More on the above SQL query: The username field is set to empty string and an 'OR' clause is added. The right-hand side of the 'OR' clause is always true and by adding '- -' we comment out everything after it.

As you can see, the hacker got logged in to the bank app without even having a valid username and password. The case is even worse if the app shows any user information after any successfully login. The hacker can manipulate the query to not just login, but also retrieve other users' information and even the DB admin details.

Since most of the CQL follows the same syntax as SQL, all the above exploits can be used on CQL as well. And, the above case is probably the simplest SQL injection, and there are lot more powerful injections that affects the queries. For more info: OWASP SQL Injection

How CQL/SQL can be prevented

The biggest vulnerability in the code that lets easily exploit the application using CQL/SQL injection is String manipulation for Query creation. Dynamic creation of the Query string is not just performance wise inefficient, but also safety wise.

The following points help us prevent CQL/SQL injection:

1. Use of PreparedStatement(both in CQL and SQL driver worlds) instead of General Statement for read/write operations. Since PreparedStatement binds fields with values, manipulating the Query by injection will only modify the value for that field and not others. Hence Query injection can be prevented.
2. **Never to use String manipulation for Query string creation (The root of all evil in Query Injections)**
3. If you are using SQL - Stored procedures can prevent SQL Injection (not to follow as this is ugly and messy)
4. In Cassandra datastax driver, we can create Queries through the QueryBuilder API. This is also a safe approach(currently what is being using in RnR system).

Cassandra Datastax Drivers

There are two datastax APIs that are considered safe against CQL Injection:

1. com.datastax.driver.core.PreparedStatement
2. com.datastax.driver.core.querybuilder.QueryBuilder

Even though, both the above approaches are safe, the first one is better performance wise as well, since the prepared queries are precompiled and stored on the server side. Only the values are bound during every query execution. Thus, it's very efficient for repeat queries.

Other Resources

1. https://www.owasp.org/index.php/**SQL_Injection**
2. **https://www.owasp.org/**
3. https://www.owasp.org/index.php/Top_10_2013-Top_10
4. http://planetcassandra.org/getting-started-with-apache-cassandra-and-java-part/
5. http://planetcassandra.org/getting-started-with-apache-cassandra-and-java-part-2/