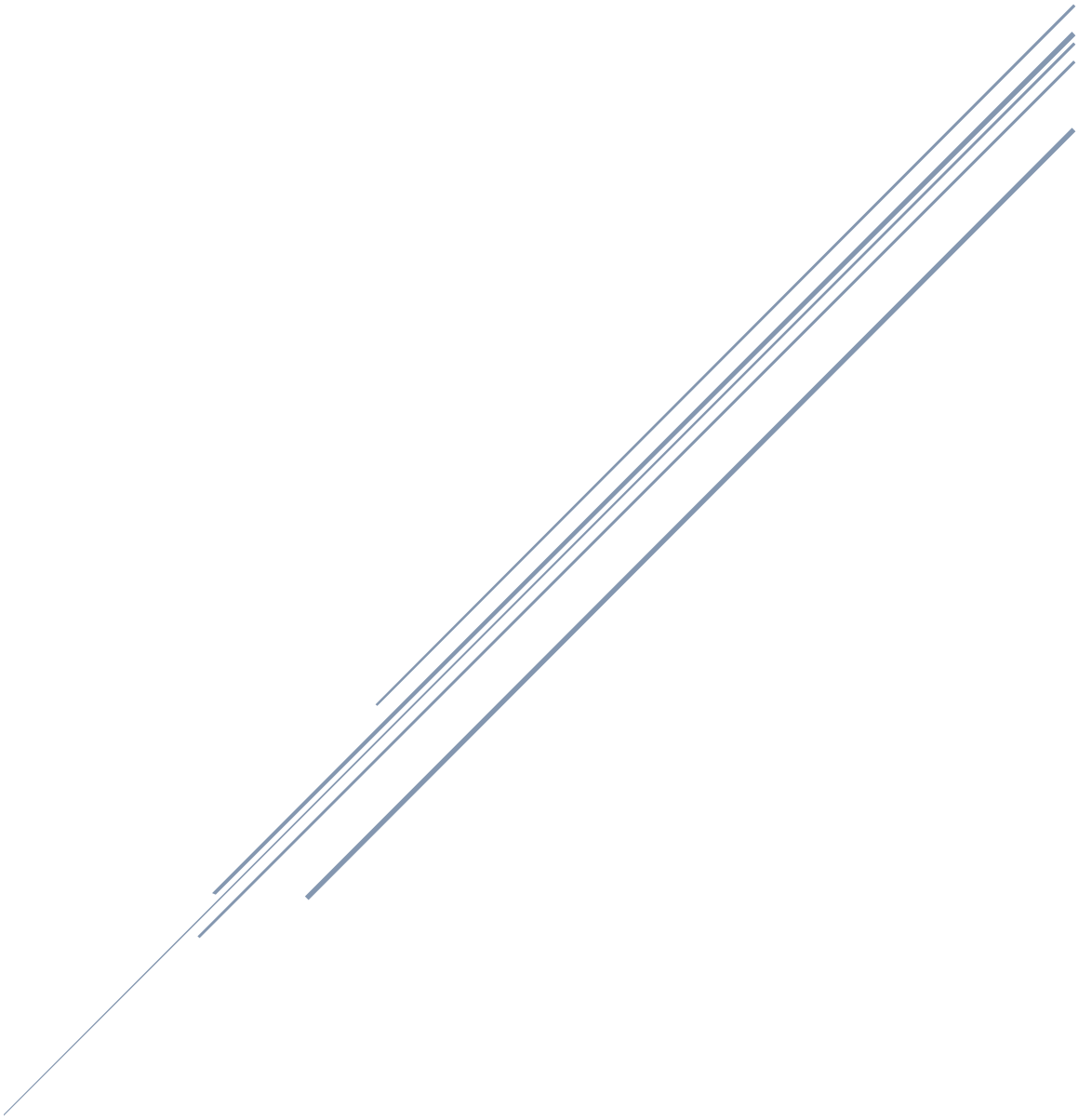
PATRICK WALSH

Graded Assignment: Homework 2



University of Maryland Global Campus

SDEV 325 – Detecting Software Vulnerabilities

Executive Summary

The first example demonstrates CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). This examples shows a piece of software that is vulnerable to an SQL injection, and the code provided carries out this attack. The example also shows how the attack is mitigated.

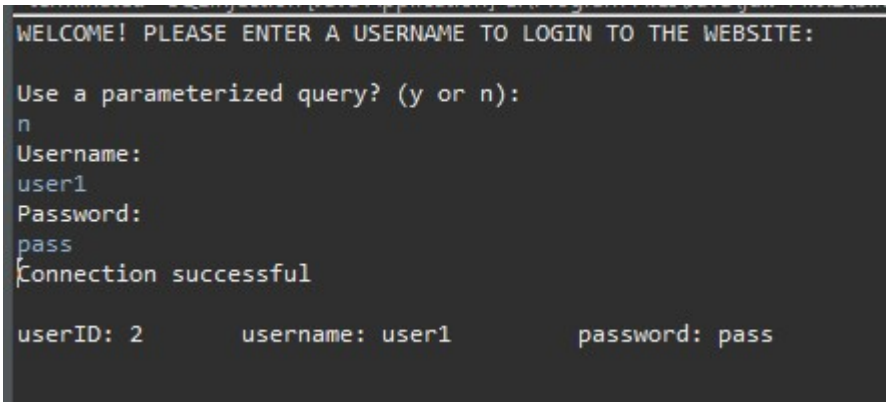
The second example demonstrates CWE-601: URL Redirection to Untrusted Site ('Open Redirect'). The example shows a web application to contains two different Open Redirects where the user is taken to a malicious site (in the first Open Redirect) and to an unexpected site (in the second Open Redirect). The example also shows how a user can check if the site they are on is likely to be safe.

Example 1 – CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Overview

This example demonstrates how to execute an SQL injection attack using a Java application running from my local machine accessing an Oracle SQL database hosted in the AWS cloud. The example also shows how to mitigate the attack through the use of a parameterized query which removes single quotes (‘) which an attacker can use to inject malicious SQL statements inside the query.

The Java application is a command line-style application that lets you login to an Oracle SQL database by entering a username and password. The username and password are checked against the database Users table and if the username and password are a match, the application returns information about the user, including their userID, username, and password. See example below:

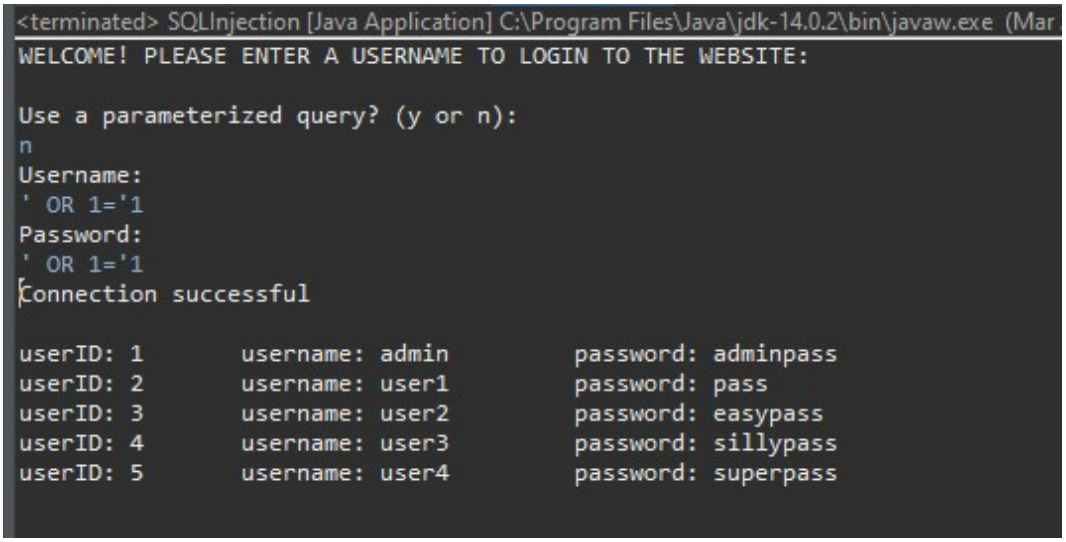


In the above example, the application also asked the user if they wanted to use a parameterized query. If they user had chosen yes (‘y’) then the application would have submitted a parameterized query.

Analysis of the Vulnerability

This application, when not using a parameterized query, is vulnerable to an SQL injection attack. To carry this attack out, the attacker inserts extra SQL statements into the query, specifically an OR statement

that always returns true by checking if 1=1. Since the query always returns true, the application ends up returning all the results in the database. See below:



In this SQL injection attack, the user was able to retrieve the usernames and passwords of all users in the database. This obviously poses a serious security flaw in the application.

Mitigation

In order to fix this security flaw, the application makes use of a parameterized query. This is an example of both queries in the Java code:

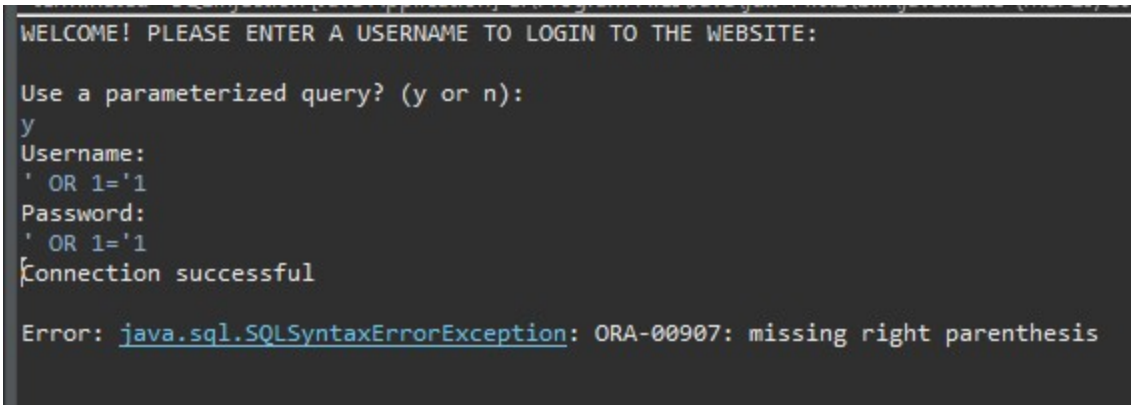
Parameterized query

```
"SELECT * FROM Users WHERE username = REPLACE('" + login_user  + "', '''', '''''')
AND passwords = REPLACE('" + login_pass + "', '''', '''''')";
```

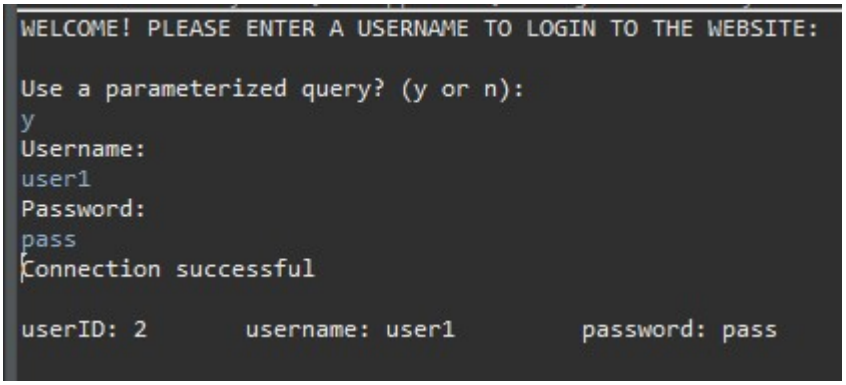
Non-parameterized query

```
"SELECT * FROM Users WHERE username = '" + login_user + "' AND passwords = '" +
login_pass + "'";
```

The parameterized query uses the SQL function REPLACE() to eliminate single quotes. This prevents a malicious user from injecting unwanted SQL statements into the query. See example below:



As can be seen, the parameterized query prevented any unwanted results from being returned when a user attempts an SQL injection. If, however, the user uses a parameterized query and enters normal username and password information, the query will run as expected:



Example 2 – CWE-601: URL Redirection to Untrusted Site ('Open Redirect')

Overview

This example demonstrates a URL redirect, also known as an Open Redirect. In this type of attack, a malicious actor tricks a user into clicking on a malicious link that redirects to a dangerous website, like a phishing site. This can happen when an unsuspecting user gets a URL link in an email or text and the user clicks on the link. The URL is often made to look like a legitimate website but will often take the user to a fraudulent copy of that website, or will download malware before redirecting the user back to a legitimate site.

In this example I use a Python web application built on the microweb framework Flask. The web application takes the user to the main page of the site written in HTML. The site is simple, with a welcome message and two URL hyperlinks to click on. See below:

Analysis of the Vulnerability

This site contains two different Open Redirects. The first Open Redirect is on the main page. When the user clicks on the www.google.com link, it redirects the user to the webpage I created that is a fake version of Google:

Welcome to my Website

Visit the fake Google:

www.google.com

Visit the real Google:

www.google.com

Links takes user to this webpage:

Mwahahahah! You are at the FAKE Google

Now please login so that I can STEAL your login credentials..... ;)

Or, go to the real Google site below: (totally NOT another redirect...)

www.google.com

If the fake was really good and looked just like the real Google (or any other website), the user could be tricked into logging in or downloading a file, in which case the fake website would capture their login credentials, install malware through other conduct other attacks like a Cross-Site Request Forgery (CSRF) attack. In the below code snippet of HTML, you can see the link that the user is take to, even though what is presented is www.google.com:

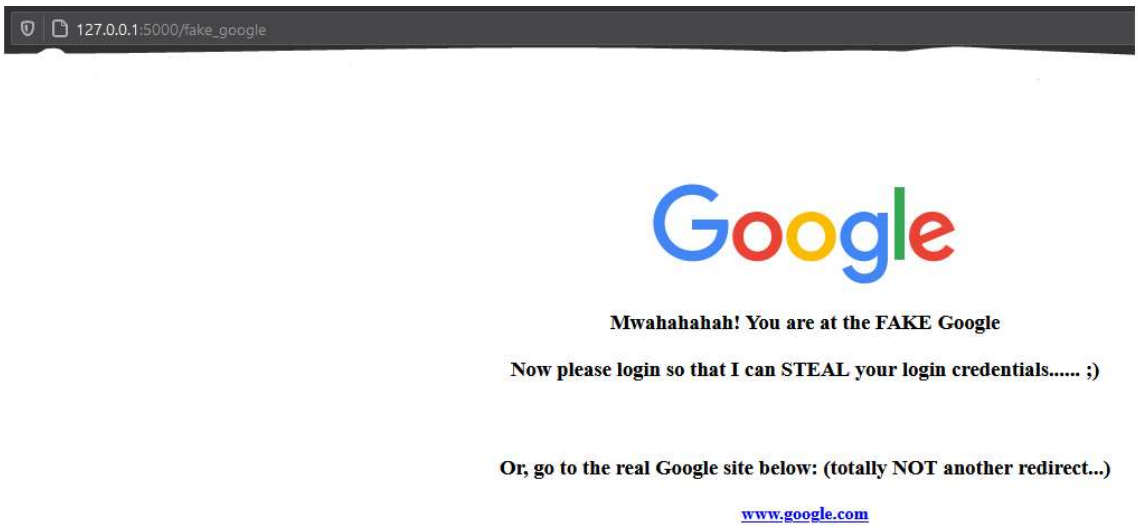
```
<h2 class="main">Visit the fake Google:</h2>
<a class="main" href="fake_google"><h4>www.google.com</h4></a>
```

A second Open Redirect is found on the same page. The user sees www.google.com, and if they hover their mouse over the URL, the link is displayed below:

At first glance this may seem like the correct URL for Google, since the URL begins with the google.com domain. However, upon closer inspection we see a redirect to cwe.org, a website which provides additional information on software security, including Open Redirects!

Mitigation

Open Redirects can be stopped through various mitigation techniques, including automated black box tools which look for malicious links. One of the easiest and most basic way a user can watch out for this attack is to carefully view links before clicking on them to ensure that they lead to the intended destination. Another mitigation technique is to check the security certificate to ensure that the website is legitimate. For example, on the fake Google page that I created, we can see that there is no security certificate:



The real Google has a trusted security certificate: