November 24, 2020

Mitigation of a Known SQL Injection Vulnerability in One of the Webpages of ChessArena

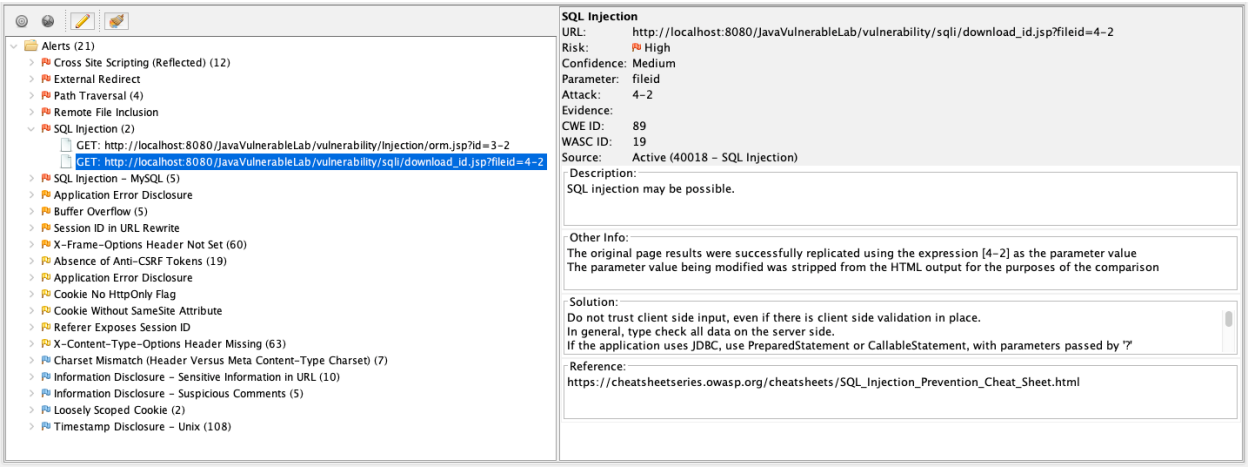
Ahmad Aryan
CSC 129

Introduction

On November 12, 2020, ChessArena's secure software team was assigned a bug report to investigate by the Head of Operations of the company Mr. Doug Lundin. The problem report was #JVL13579 and according to the description of the problem, a routine scan by our company's cyber-team had identified a possible exploit. This vulnerability has been identified in a key component of our product infrastructure and was assigned a priority of 7 (out of 10) at the time. However, since then, the vulnerability has been confirmed by the secure software team which has automatically increased its priority level to 9. Chess Arena's secure software team has received approval to investigate and mitigate this vulnerability. An in-depth analysis of the vulnerability by our team confirmed the critical nature of this vulnerability and after careful study, the secure software team has produced a mitigation approach which is discussed in this report.

Analysis of security vulnerability

The bug report that was assigned to secure software team for further investigation and mitigations pointed to a vulnerability in our system that could allow an attacker gain access to confidential information about our database system. In case attackers gained access to the information related to our system database, they could use this information to potentially conduct further attacks and gain control of company's vital systems and resources. This had the potential to jeopardize all our customers' records inflicting devastating effects on our services and the image a reputation of our company. The details of bug report which was discovered by a OWASP Zap scan and a screen shot of the scan window are included below:



http://localhost:8080/JavaVulnerableLab/vulnerability/sqli/download_id.jsp?fileid=2+AND+1%3D1+--+

Method: GET
Parameter: fileid
Attack: 2 AND 1=1 –
CWE ID: 89
WASC ID: 19

According to the Zap scan report, an SQL injection attack was possible in one of our web pages. The problem report identified this vulnerability as a GET request which means all the parameters were on

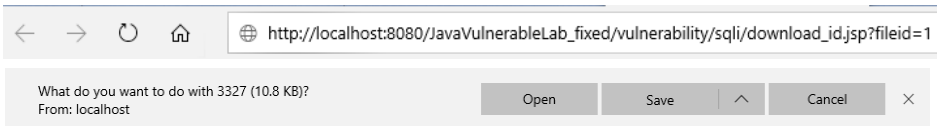
the URL rather being embedded within the HTTP request object. It should be noted that the above URL was generated by the OWASP Zap scan randomly. However, further analysis confirmed the existence of this vulnerability which existed in the *fileid* parameter of the URL.

In the web page where the problem existed, there are two links which allow the users to download two pdf files that we have placed on the website. Both links appear to work as expected. However, we confirmed that a SQL injection attack was possible using the vulnerable URL. By conducting a SQL injection using the vulnerable URL, an attacker could get information about the version and name of our database. Our team was able to gain access to such information using a simulated attack, the details of which are as follows:

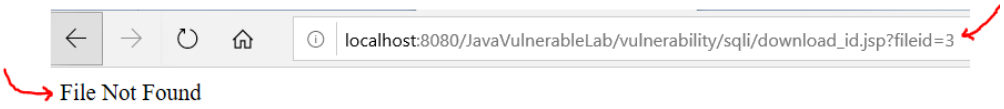
The vulnerability existed in the download_id.jsp file. The HTML generated by that file would allow the download of two PDFs on the webpage. As it is expected, when we enter the following URL, we will get a save dialogue to download the provided PDF of the webpage.

http://localhost:8080/JavaVulnerableLab/vulnerability/sqli/download_id.jsp?fileid=1

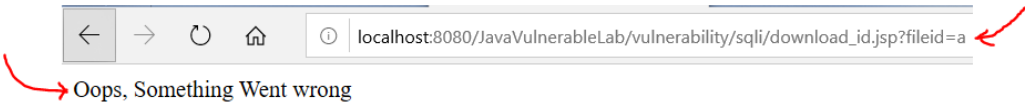
If we change the *fileid* parameter from 1 to 2, we get another saved dialog which corresponds to the second PDF. This means that the *fileid* parameter is set to be true.



However, if we change the *fileid* parameter to 3 or any other number, we get *file not found* message which means *fileid* parameter is false.



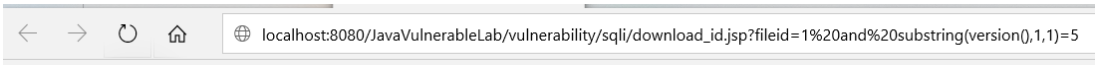
Changing the fileid parameter to a string, instead of a number produces an error message.



Our team conducted a SQL injection by setting the *fileid* parameter to be true while attempting to get the database version by inputting the following URL.

http://localhost:8080/JavaVulnerableLab/vulnerability/sqli/download_id.jsp?fileid=1%20and%20substring(version(),1,1)=5

Since *fileid* = 1 *and substring(version()), 1,1) = 5* is true we determined that the first number in our database version is 5. Furthermore, since changing the *substring(version()), 3,1) = 7* gave us a download dialogue, we determined the second number of the database version to be 7. Continuing with the attack, we determined all the numbers in our system's database version to be 5.7.32.



Using a similar attack approach, we determined the character length of our system's database name.

https://localhost:8080/JavaVulnerableLab/vulnerability/sqli/download_id.jsp?fileid=1%20and%20char_length(database())=3

We got a save dialogue after entering *fileid* = 1 and *char_length(database()) = 3*. This gave us information about the length of characters in database name which is 3. Armed with information about the character length of the database name, we took the attack to new level, and decided to find out the full name of the database. To do that, we injected the following code into the URL:

https://localhost:8080/JavaVulnerableLab/vulnerability/sqli/download_id.jsp?fileid=1%20and%20substring(database(),1,1)='a'

This code gave us a true output: *fileid* = 1 and substring(database(), 1, 1) = 'a'. Therefore, we determined the first character of the name of the database to be 'a'. Similar attack approach gave us the second and third characters of the database. At the conclusion of the attack, we were able to find the following information about our system's database:
Database version 5.7.32 and Database name: abc

As it is evident from our investigation, by exploiting the vulnerability of our system, an attacker could get information about the version and name of our database. Once the attackers were armed with this knowledge, they could search for the known vulnerabilities of the database version and conduct further, and potentially more destructive attacks that might have compromise the security and integrity of our system and services. Therefore, the secure software took immediate action to fix the vulnerability without hesitation. Our team came up with two possible fixes to this security flaw which are presented in the following section.

Mitigation of the security vulnerability

ChessArena's secure software team traced the problem back to the *download_id.jsp* file and we devised the fixes to the code in DEV environment. After investigating the code, we narrowed down the problem to line 24 of the code as it is presented in this screen shot:

As it is seen in the code above, SQL attacks were possible due to a problem with *fileid* parameter. Here a numeric value was treated as string which allows for concatenation. To address this problem, the string value should be first converted to an integer and then passed as value to the *fileid* parameter. There are a few ways to fix this problem. Attempting to pars the value as integer and using a prepared statement. We discuss both in turn.

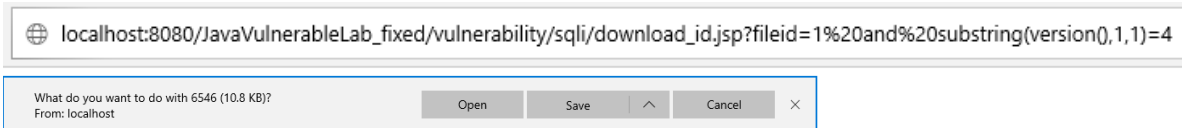
In the first method, if the values can be converted to integers, we parse the value into integer in the following code. On top of preventing the SQL injection attacks, we also introduced a line which would alert us if the program identifies a SQL injection, while preventing the attack.

```
 8 <%@page import="java.sql.ResultSet"%>
 9 <%@ page import="java.util.*,java.io.*"%>
10 <%@ page import="org.cysecurity.cspf.jvl.model.DBConnect"%>
11
12
13 <%@page import="java.io.File"%>
14 <%
15   String path = request.getContextPath();
16   try
17   {
18       String fileid=request.getParameter("fileid");
19     // Previous line represents a SQL Injection vulnerability because the parameter is not sanitized
20     // Following code mitigates the SQL Injection vulnerability
21     String value = fileid.trim();
22     NumberFormat formatter = NumberFormat.getNumberInstance();
23     ParsePosition pos = new ParsePosition(0);
24     Number safeFileID = formatter.parse(value, pos);
25     out.print("Parameter=" + safeFileID);
26     if (pos.getIndex() != value.length() || pos.getErrorIndex() != -1) {
27     // throw new RuntimeException("my error description");
28     out.print("Possible SQL Injection attack in progress!!!!");
29     }
30
31         if (safeFileID != null)
```

The second approach entails using a prepared statement. The effect of prepared statement is that it prevents an SQL injection attack by preventing concatenation of the input into the parameter itself. Another benefit of prepared statements is that it increases the performance. This is particularly useful if the code is complex and lengthy. As it can be seen in line 30, a prepared statement using *setString( )* method is created. This ensures that only validated data is passed into the parameter which eliminates the SQL injection attack.

```
 6
 7 <%@page import="java.sql.ResultSetMetaData"%>
 8 <%@page import="java.sql.ResultSet"%>
 9 <%@ page import="java.util.*,java.io.*"%>
10 <%@ page import="org.cysecurity.cspf.jvl.model.DBConnect"%>
11
12
13 <%@page import="java.io.File"%>
14 <%@ page import="java.sql.*" %>
15 <%@ page import="static java.sql.DriverManager.*" %>
16 <%
17   String path = request.getContextPath();
18   try
19   {
20     String fileid=request.getParameter("fileid");
21
22     if(fileid!=null && !fileid.equals(""))
23    {
24      Connection con=new DBConnect().connect(getServletContext().getRealPath("/WEB-INF/config.properties"));
25
26              String query = ("select * from FilesList where fileid=?");
27              PreparedStatement stmt = con.prepareStatement(query);
28              String value = fileid.trim();
29
30              stmt.setString(1,fileid);
31              ResultSet rs = stmt.executeQuery();
```

After we formulated the fixes above, per company's procedures, the codes were tested in PROD environment to ensure that they fix the vulnerability in real world situation. The results were completely satisfactory. Below is a screen shot of the behavior of the web page after the fix has been applied. As it can be seen, an SQL injection is no longer possible since, even if attackers input a wrong integer (here 4 instead of correct integer 5) and get a false boolean value, they would still get the save dialogue. In other words, the save dialogue will appear no matter a true boolean or false boolean is fed into the *fileid* parameter. Therefore, the attackers will have no way of getting information about the database out of the webpage.



Conclusion

ChessArena's secure software team had been given full authority to fix the vulnerability in whatever way deemed necessary. As discussed before, we came up with two approaches, the first being parsing the integer method and the second, by using prepared statement. Both approaches prevent the SQL injection attack; however, we decided to modify the code and fix the problem using the first method which entailed parsing integers to *fileid* parameter. The reason that we chose this method over the prepared statement method is that on top of stopping the injection attack, it gives us some information about possible attack, if the program detects such activity. The security vulnerability has since been fixed and this has been confirmed by the Test and Production teams.