Michael Weaver, Caleb Miller, Mathew Paschall

CST – 235

Brandon Bass

12/24/17

**CLC 7**

GitHub Link: https://github.com/battousairurik/CST-235

**CLC Questions**

1. ProgrammableWeb.com contains several API, or application programming interfaces, many of which operate the same way. One such program is the Unofficial Pokémon Go java API, which allow you to create your own interactive player portal for the Pokémon Go game.

***Pokemon Go API Location****: https://www.programmableweb.com/api/unofficial-pokemongo-java*

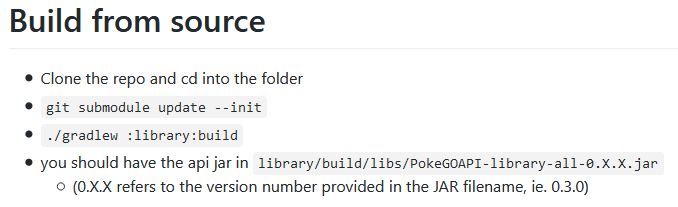
This package includes a GitHub link to the files for download and the API endpoint location for your designed application to link to. The files are located here:

***Pokémon Go GitHub Location****: https://github.com/Grover-c13/PokeGOAPI-Java*

The files include a ReadMe meant to describe how to install the folder and extract the data, but it is not detailed enough to actually work. Each of the provided files are structured to open a link to the Pokémon Go API then send or receive information over that open link. This is the way API function, so long as the data intending to be set is formatted correctly, the API will function. One such example is the CheckEvolutionExample.java which checks the current Pokémon id against the database to determine if there are any evolutions for the selected Pokémon.

**Assignment Approach**

1. For this assignment I researched what API are, then utilized the provided links on the API homepage to locate the GitHub documentation for the Pokémon Go API. The GitHub page contains instructions, albeit incomplete, on how to unpackage the PokeGOAPI-Java. The provided description is not detailed enough for someone with only a small amount of programming knowledge to unpackage and use:



**List of Classes, Methods, Variables, etc.**

1. All public example code located in the GitHub link

**Screenshots**

Pokemon Go Evolution Check



**Mathew Paschall**

For the security vulnerabilities portion of this assignment, Caleb and I decided to each post describe two vulnerabilities. The two potential security vulnerabilities for web services I chose were SQL injection attacks and automated attempted login to your web service. SQL injection attacks can happen when you have poor data validation on your web service. In order to ensure that code is not written in any login field, you should ensure that usernames and passwords will be acceptably short and contain only letters and numbers. Injection attacks happen when someone attempts to write code in a field that is normally for user input, this forces the program to behave in a way that it is not intended. A code that is vulnerable to injection attack can be seen below.

HTML login page

<form action=”servlet” method=”post”>

Username: <input type=”text” name=”username”>

Password: <input type=”text” name=”password”>

<input type=”Submit” value=”Login”>

<form>

Servlet

@Override

protected void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

String un = request.getParameter(“username”);

String pw = request.getParameter(“password”);

if(un == getDBUsername() && pw == getDBPassword()){

login();

}

else(){

request.getWriter().write(“Incorrect login.”);

}

}

In this scenario, someone could enter getDBUsername() and getDBPassword() in the login form to login without a username and password that is valid. This can be fixed with data validation by checking input for special characters and length first, then proceeding to match the information to database information. The second vulnerability is automated login attempts. This can be countered by only allowing a user to attempt to login so many times before requiring a password reset. A count variable can assist with this process. Using a java bean, a servlet, and a jsp page, you can assist the user in attempting login again as well as count the login attempts.

Servlet

public class myServlet extends HttpServlet {

int count = 0;

@Override

protected void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

Bean bean = new Bean();

count++;

bean.setCount(count);

String un = request.getParameter(“username”);

String pw = request.getParameter(“password”);

checkInput(un, pw);

if(checkInput == true && count <= 3){

if(un == getDBUsername() && pw == getDBPassword()){

login();

}

Else if(count > 3){

DisableAccountLoginAbility();

request.getWriter().write(“You have been locked out, please use password reset to log back in.”);

}

else(){

request.setAttribute("bean", bean);

RequestDispatcher r = request.getRequestDispatcher("Incorrect.jsp");

r.forward(request,response);

}

}

}

JSP file

<body>

Attempt Count: <jsp:getProperty name = "bean" property="count"/>

<br>

<form action="Login.html" method="post">

<input type="Submit" value="Try Again">

</form>

This code will ensure that the user cannot attempt to log in more than three times. If the user fails three times, they will need to perform a password reset through the system.

**Caleb Miller**

1. **Describe 3 potential security vulnerabilities of web services. Substantiate your answer with code examples and screenshots of execution.** Buffer Overflow is a denial of service, data corruptor, and malicious code executer. An attacker can craft XML data causing the XML to call upon itself repetitively therefore constantly increasing in size. This causes a memory overflow, or trigger error messages which reveal information about the application. A DOS attack can be caused by forcing a server to parse an abnormally long XML file, which in essence uses up much more resources then actually generating one, and can crash the application. Another type of attack consists of sending a block of data to an application, which is stored in a buffer of insufficient size. This block of data can then overwrite genuine data and cause a function return which gives control to the malicious code in the hacker’s data block.
2. public class Overflow
3. {
4. public static void main(String[] args)
5. {
6. int importantData =1;
7. int[] buffer = new int[10];
8. for (int i =0; i < 15; i++)
9. buffer[i] = 7;
10. System.out.println("after buffer overflow ");
11. System.out.println("Important data = "+importantData);
12. }
13. }

*The buffer in the above example is 10 but the program is trying to write 15 resulting in a buffer overflow.*

XML Injection effects command execution, data can be stolen or deleted, and schema poisoning. QL Injection is a high-risk exploit which may be performed using SOAP messages. If a server does not validate data correctly, a SOAP message can easily be used to create XML data which inserts a parameter into an SQL query and have the server execute it with the rights of the Web Service. SQL Injection is only one of the threats a server is exposed to if data is not validated. A schema file is what an XML parser uses to understand the XML’s grammar and structure, and contains essential pre-processor instructions. An attacker may damage the XML schema or replace it with a modified one which would then allow the parser to process malicious SOAP messages and specially crafted XML files to inject OS commands on the server or database.

<?xml version="1.0" encoding="ISO-8859-1"?>

<users>

<user>

<uname>joepublic</uname>

<pwd>r3g</pwd>

<uid>0<uid/>

<mail>joepublic@example1.com</mail>

</user>

<user>

<uname>janedoe</uname>

<pwd>an0n</pwd>

<uid>500<uid/>

<mail>janedoe@example2.com</mail>

</user>

</users>