OWASP WebGoat Lab (conclusion)

Thomas Martin

March 16, 2018

1 Objectives

This week we will finish up our work on the attacks demonstrated in the OWASP WebGoat web application. Last week, the challenges were tackled in groups. This week, the same groups will work on different sets of challenges. You will then work on the final set of challenges individually.

2 Previous Group Challenges

We will first revisit the challenges from last week. You are now to each attempt the two sets of challenges that the other groups attempted. I will be assigning an "ambassador" from each group to help the other groups. As a reminder, the division of the challenges was:

1. Access Control Flaws and Authentication Flaws
2. Code Quality, Concurrency, Improper Error Handling, and Session Management Flaws
3. Cross-Site Scripting (XSS)

3 New Challenges - Injection Flaws

For the second half of this lab, we will look at the various challenges based on Injection Flaws:

3.1 Command Injection

Sometimes programmers will take user input and use it as a parameter in a system command. This task shows why this can be a very bad idea. What the page is supposed to do is take the file name chosen by the user from the drop-down menu and display it with the command:
`cat "<filename>"`

Both Windows and Linux have the feature where a single sequence can cause two commands to be executed in series. In Windows, the commands are separated by "&" (e.g. "`dir & ipconfig`". In Linux, the commands are separated by ";" (e.g. "`ls ; ifconfig`")[1]. A normal user would be limited by the options in the drop-down menu, none of which involve any command injection. However, using a proxy such as OWASP ZAP, the parameter can be modified to any desired value. The tricky thing to manage in this task is the quote marks. If any of the commands have bad syntax, none of the execution will work.

3.2 Numerical SQL Injection

In this example, we see the SQL statement that is used on the page (this is very unusual, but helps understand the attack). You choose an option from the drop down menu and a corresponding number is placed in the SQL `SELECT` statement. By modifying the parameter sent using ZAP, anything can be added to the statement. SQL statements allow typical Boolean operators: AND, OR, NOT. For the purposes of this task, you want the `SELECT` statement to pick every row, not just one. So you need to add something that makes the `WHERE` clause always true. This comes in handy in many SQL Injection attacks.

3.3 Log Spoofing

This attack is an attempt to add an extra line to a log file. It is true that an attacker is usually more interested in removing entries from log files. But it could be the case that the addition of an entry to the log could trigger some action that the attacker hopes to manipulate.

Entering any random username/password combination into the provided fields will generate a single line entry in the log. You need to work out how to manipulate the input to generate an extra line, that would be believable as the log of a successful login. Hint: a URL encoder/decoder will help you in this challenge, such as the one found here: `https://meyerweb.com/eric/tools/dencoder/`.

3.4 XPATH Injection

This lesson has authentication details stored in an .xml file, and uses XPATH to query if the user-provided username and password are correct. While a knowledge of XML and XPATH is useful, it is not strictly necessary for this challenge. This can be approached in mostly the same way as an SQL Injection. Two things to bear in mind: This injection will involve strings, so you will need to be careful with quote marks (most code in WebGoat uses single quote marks `'`). Also, XPATH handles logical expressions slightly differently to SQL: you will have to manipulate things so that both queries (Is the username correct? Is the password correct?) always evaluate to true.

[1] Note that these work different to the pipe: |

3.5 Lab: SQL Injection Stage 1

The previous tasks have generally been about returning all records instead of a specific one. In this case, there is a specific user we wish to impersonate (Neville). You need to rethink how you implement your SQL Injection to get the desired effect.

Stages 2 and 4 of this Lab requires the developer version, and so we will be skipping them.

3.6 Lab: SQL Injection Stage 3

This task requires some out-of-the-box thinking. We can log in as Larry (password:larry), and we want to see the profile of the boss Neville. The output of this page will only give a single profile. If we just specify the employee ID of Neville, it will be rejected because we, logged in as Larry, are not authorized. If we use a standard SQL Injection, then all rows in the table will match, but only the first one will be displayed. We need to adjust our injection to make sure the entry we want is the one that will be displayed.

3.7 String SQL Injection

If you have successfully managed all the previous Injection Flaw challenges, then this one will easy.

3.8 Database Backdoors

Just like with the Linux command prompt, the ";" symbol is used to multiple commands executed in a single line. For this task, you will also need the `UPDATE` syntax, which is explained here:

`http://www.w3schools.com/sql/sql_update.asp`

(Stage 2 can be ignored as it cannot actually be executed).

3.9 Blind Numeric SQL Injection

Much of the previous SQL Injection challenges have been made easy by the fact that they often give additional information back to the browser when odd things happen (e.g. multiple rows may be presented when the correct use of the page should only have given a single row). This challenge is more realistic in that no matter what manipulation you do, you will only ever get a single bit of information. The task is to use that efficiently to get the desired information (a PIN) efficiently, i.e. without having to test every possible value.

The page you are presented with asks for an account number. On clicking "Go!", the response will tell you whether or not that is the a valid account number. It does this by querying for that account number in the `user_data` table. What we are looking for is the value of the pin field for a particular credit card number. The first problem is that the credit card details are saved in a different table (`pins`). Fortunately, SQL allows you to have sub-queries

within a query. If you put the sub-query in brackets, and it returns a numerical value, you can use it in the calculations in the injected query. The other problem is that you must form your injected query to respond either True or False in a useful way. You could try testing all possible PIN values, but every for a 4-digit pin that would take too long.

Note: This challenge is quite complex and rely on good understanding of SQL syntax. It is recommended that you consult the hints if you are not making progress.

3.10 Blind String SQL Injection

The principle of this task is the same as the previous one, but it is complicated by the fact that the field we want is a string, not an integer. An SQL functions that is useful in this task are `SUBSTRING()` which will allow you to access portions of a string of any length (including just a single character). It takes three arguments: the string, the start of the substring, and the desired substring length.

3.11 Denial of Service from Multiple Logins

This is another SQL injection problem.

4 Beyond the Scope of this course

Some of the tasks not included in the ones we have covered are beyond the scope of this course. Feel free to browse them in your own time.

5 Extension Task - The Challenge

The final test in WebGoat is listed at the very end of the left column ("Challenge" - "The Challenge"). There are no hints and no solution provided. To succeed, you will need to apply all the lessons learned in the previous challenges.

6 Summary

These exercises demonstrate many of the OWASP Top 10 Risks in Web Applications. As we have seen, some are relatively simple to exploit and some are very subtle. It is very important to understand these vulnerabilities as the use of web technologies continues to grow.