CS453 - Automated Software Testing

Project Report

for

DVWA attacker

Yunju Park
Truc Anh Nguyen Phan
Diba Vosta

June 18, 2019

Contents

1 Introduction

The purpose of this report is to present the project that was carried out for the CS453 Automated Software Testing course. The aim of the project was to develop or implement an automated software testing tool which solves a problem that exists within software development and testing. This report will cover the problem that we aim to solve and discuss why it is a problem in the first place. Following that, the automated testing tool that was developed will be described in detail. The report ends with a section that covers the evaluation of the tool and how well it works.

2 Problem

The problem that was chosen for this project was about making software developers aware of if there are any vulnerable text input fields in their web applications that could be susceptible to SQL injections. By making a developer aware of any vulnerable input fields, the intention was that the developer should be able to use the output from the testing tool in order to re-write their code to become more safe as to avoid SQL injections.

2.1 SQL Injections

SQL is the abbreviation of Structured Query Language and is used when handling a connection to a database and retrieving, altering, deleting or adding any information to the database. A so-called SQL injection is when someone with bad intentions try to access data from the database. The main entry point for SQL injections is a text field on a web application (or other application) where the user can enter anything that they wish. To simplify this, let's consider the following example.

2.1.1 Example

Imagine you have written a web application for a library system. To access the library, you have to have an account that you log onto using a username and password. To keep track of this information together with which user has borrowed which books and other relevant information, you enter the data into a database that you access via SQL queries. On your web application, you've added a text search field so that the user can search for authors or titles of books. The information that the user enters into the text input field is then processed to look through the tables in the database after a book title that matches it or an author's name that matches it. In other words, the input from the user is used in the query to be able to retrieve the relevant information.

To illustrate this, the process in the backend of the web application would look something like this:

```
user_input = get_request_string("Search")

query = "SELECT title, author, year, availability FROM Author \
        JOIN Books ON Author.ID = Book.authorID \
        WHERE Author.name = " + user_input + " OR Book.name = " + user_input
```

The above example shows how the input from the user is directly concatenated with the query string that will be run. This is an example of how an input field should not be written as it is very vulnerable to harmful inputs. If an attacker wanted to, they could enter a string such as ”Harry Potter OR 1=1”, which would give them an entry to the database. The reason behind this is that the query will always check the OR statement, which in this case is 1=1, which will always be true. When the database then processes this query, it will output all the data that is stored in both the authors table and the books table. If an attacker can access this data, it means that they could also alter their input in order to get access to more sensitive data. This sensitive data could include data such as usernames and passwords of all registered library users.

2.2 Most Common SQL Injection Techniques

As mentioned in the example above, one of the common injection techniques is to write an ”OR 1=1” clause in the input field to get access to the database. However, this is not always as easy as just writing ”OR 1=1” and be done with it. In most cases, the attacker would have to try a few different variation of this before getting a correct statement that is able to run (if there is no mean of prevention or the prevention method is done improperly). For example, the query that were to be run would look like this:

```
user_input = get_request_string("Search")
query = "SELECT * FROM Book WHERE Book.name = '" + user_input+ "'"
```

In this case, the attacker would have to pay extra attention to the single quotes that are used in order to produce a syntactically correct query in the end. For an attacker to beat this vulnerable input field, they would have to use the following input to make the query vulnerable:

```
user_input = "Harry Potter' OR '1'='1"
```

The resulting query to be run would look like this:

```
query = "SELECT * FROM Book WHERE Book.name = 'Harry Potter' OR '1'='1'"
```

As the above query is syntactically correct, the attacker would be able to retrieve data that notifies him of the successful intrusion to the database. With this, the attacker could then use other queries to learn more about the contents of the database and apply that to retrieve the sensitive data.

2.3 Available SQL Injection Prevention Methods

Since most developers are aware of the possibility of SQL injections, there are several ways in which to protect text input fields from being vulnerable. As will be described in the next section of this report, the tool that was built was analysing a web application that was written in PHP. For that reason, the report will cover the prevention methods available in PHP. There are two methods that are used for protecting input fields against SQL injections when writing code in PHP. These methods are described in the two following subsections.

2.3.1 Built-in Function

In PHP, there is a built-in function called *mysqli_real_escape_string*() that takes an input string as an argument and outputs a ”safe” version of the same string. What this function does is that it prepends a backslash in front of all so-called special characters[1]. These special characters include but are not limited to single quotes, double quotes and new-line symbols. When appending a backslash before symbols as these, the attacker will not be able to penetrate the database because the potential single quotes that they use (as described in section 2.2) will not be processed as ”SQL single quotes”, but will be processed as a string instead, which makes it impossible to access the database.

However, just because a developer uses this method of protection it does not mean that their databases are completely safe. The reason behind this is that there are certain weaknesses when it comes to the use of *mysqli_real_escape_string*() that the developer has to know about in order to be able to protect their web application in a proper way. When using *mysqli_real_escape_string*(), the developer has to make sure that they add single quotes in their string query between the user's input [2], like this:

```
$param = mysql_real_escape_string($_GET['Search year']);
$query = "SELECT * FROM Book WHERE Book.year = '" + $param + "'"
```

When using it like this, the resulting query that is run would look like this:

```
"SELECT * FROM Book WHERE Book.name = '2015'"
```

However, when the developer omits the surrounding single quotes, the following query would be run:

```
"SELECT * FROM Book WHERE Book.name = 2015"
```

When this is the resulting query that is run, it is possible to inject is since it will treat the injection as regular SQL, like in the following example.

```
"SELECT * FROM Book WHERE Book.name = 2015 OR 1=1"
```

So for a developer to be on the safe side, it is recommended for them to use parameterized queries instead.

2.3.2 Parameterized Queries

A parameterized query is a means of pre-compiling a SQL statement so that all you need to supply are the parameters that need to be inserted into the statement for it to be executed. It's commonly used as a means of preventing SQL injection attacks.

3 Solving the Problem

3.1 Finding a Vulnerable Website

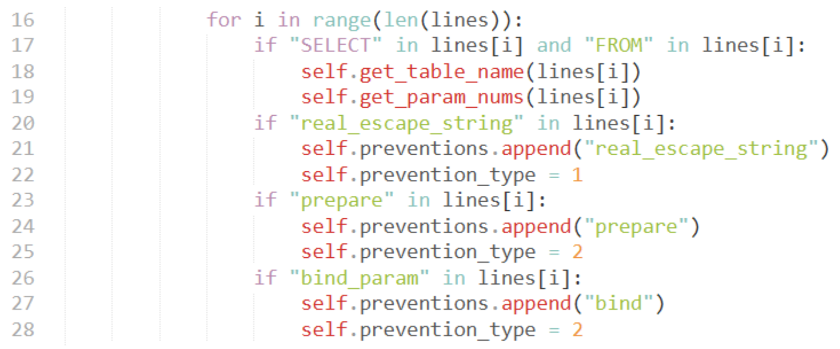
The website we found vulnerable is Damn Vulnerable Web Application (DVWA). The main goal of this PHP/MySQL web application is to help security education in a legal environment. That's why we could develop our tool based on this application. This website has different levels of difficulty to hack. 'Low', 'Medium' and 'Impossible' level corresponds to each prevention type that we pre-defined. The prevention type would be Type 0 for the low level, Type 1 for the medium level, and Type 2 for the impossible level.

Also, we modified the source code of this application little. One reason was that we should remove and update functions which were deprecated in PHP 7. Also, we modified Medium-level source code slightly to make the input field as text field. It was selection input which needs additional application to be hacked before modifying.

3.2 Description of Tool

3.2.1 PHP file parser

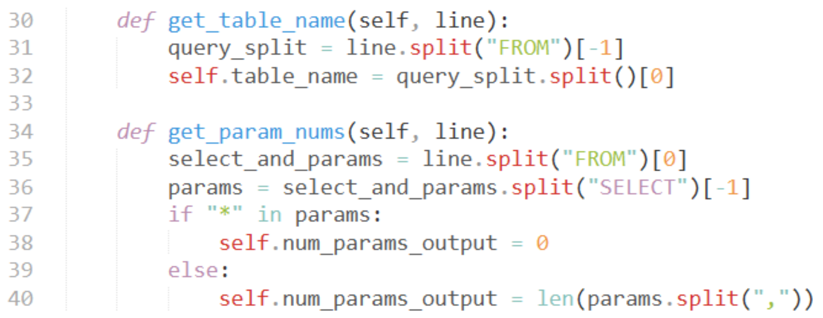
When running the tool, first, the user is required to set up the necessary environments. These steps are described in the repository link in section 5. When running the necessary file, the user is required to input the path to the PHP source file which contains the query with actual connection to the database. The PHP file parser 'fileParser.php' located under tool/ determines potential prevention type by reading each line of the PHP source file and finding "*real_escape_string*", "*prepare*" or "*bind_param*". This process is implemented from line 20 to line 28 in figure 1.



```
16                  for i in range(len(lines)):
17                      if "SELECT" in lines[i] and "FROM" in lines[i]:
18                          self.get_table_name(lines[i])
19                          self.get_param_nums(lines[i])
20                      if "real_escape_string" in lines[i]:
21                          self.preventions.append("real_escape_string")
22                          self.prevention_type = 1
23                      if "prepare" in lines[i]:
24                          self.preventions.append("prepare")
25                          self.prevention_type = 2
26                      if "bind_param" in lines[i]:
27                          self.preventions.append("bind")
28                          self.prevention_type = 2
```

Figure 1: Potential prevention type

Also, extracting information from SQL query like table name and the number of parameters is implemented. When reading each line, we try to find SQL query in line 17 in the figure. If we succeed to find, *get_table_name*() and *get_param_nums*() functions extract the information from that line, as described in figure 2. These variables will be later used during the actual injection to build the injection string.



```
30      def get_table_name(self, line):
31          query_split = line.split("FROM")[-1]
32          self.table_name = query_split.split()[0]
33
34      def get_param_nums(self, line):
35          select_and_params = line.split("FROM")[0]
36          params = select_and_params.split("SELECT")[-1]
37          if "*" in params:
38              self.num_params_output = 0
39          else:
40              self.num_params_output = len(params.split(","))
```

Figure 2: Table name and param nums

3.2.2 Injection simulator with Selenium

In order to inject the website, the tool use Selenium to automate the injection process on the Chrome web browser. The Selenium webdriver accepts commands written in Python, and send them to a browser, making it possible to automate the testing process instead of manually typing in injections in text fields.

The commands send to the browser are as follow:

1. Go to the injection site: http://localhost/dvwa/vulnerabilities/sqli/
2. Login to the DVWA website using predefined username and password (correspondingly "admin" and "password")
3. Based on the user input which contains a path to the PHP source file, the level of difficulty will be determined
4. Go to the injection site with text input field with the chosen difficulty level
5. Inject the set of injection strings and gather results on number of successful injections
6. When done injecting, close browser and display results

3.2.3 Feedback to developer

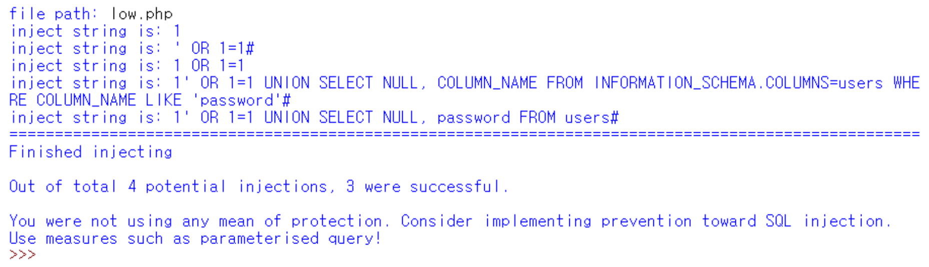
When injection process is finished, the terminal will display the injection results. It contains the number of successful injections in relation to the total number of injections. Based on the prevention type, the developer also gets different types of feedback. For example, for Type 1 error, the user would get the following feedback: *You use mysql_real_escape_string() as a prevention, but it is not being used in a proper way.*

4 Evaluation of Tool

4.1 Does it work well?

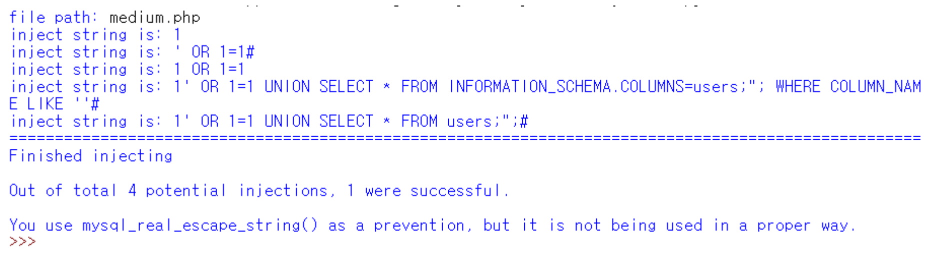
When we ran tests automatically through Selenium, we were able to get expected outputs. Here, expected output means the number of successful injections depending on prevention type. For example, for Type 0, the user should get "3 out of 4 were successful". Here 3 is the number of all possible injection that we defined for Type 0 succeeded. For improper use of Type 1, all injections for Type 1 should succeed, while they should not for proper use of type1. For Type 2, none of these should fail.

We could get these expected outputs, which means that our tool works well. For example, after executing the tool for the DVWA with low, medium, impossible difficulty by typing file name 'low.php','medium.php' and 'impossible.php' as console input(here, for convenience to debug, we positioned these files in the same folder with python files), user gets messages like figure 3,4 and 5.
Plus, one of the benefits which our tool brought is that based on the result we could give different feedback like "You were not using any mean of protection."



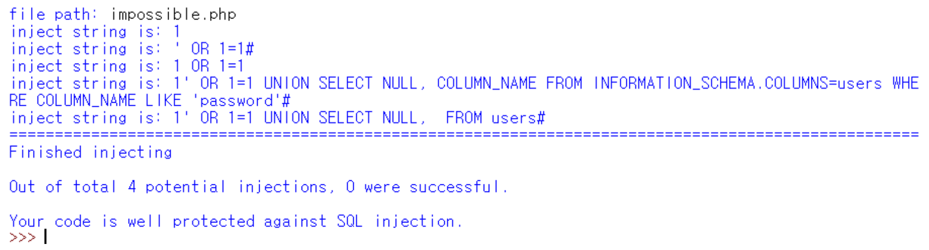
```
file path: low.php
inject string is: 1
inject string is: ' OR 1=1#
inject string is: 1 OR 1=1
inject string is: 1' OR 1=1 UNION SELECT NULL, COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS=users WHE
RE COLUMN_NAME LIKE 'password'#
inject string is: 1' OR 1=1 UNION SELECT NULL, password FROM users#
============================================================================================
Finished injecting

Out of total 4 potential injections, 3 were successful.

You were not using any mean of protection. Consider implementing prevention toward SQL injection.
Use measures such as parameterised query!
>>>
```

Figure 3: after executing for 'low.php'



```
file path: medium.php
inject string is: 1
inject string is: ' OR 1=1#
inject string is: 1 OR 1=1
inject string is: 1' OR 1=1 UNION SELECT * FROM INFORMATION_SCHEMA.COLUMNS=users;"; WHERE COLUMN_NAM
E LIKE ''#
inject string is: 1' OR 1=1 UNION SELECT * FROM users;";#
============================================================================================
Finished injecting

Out of total 4 potential injections, 1 were successful.

You use mysql_real_escape_string() as a prevention, but it is not being used in a proper way.
>>>
```

Figure 4: after executing for 'medium.php'

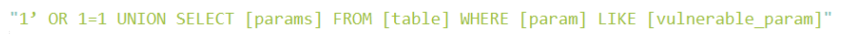


```
file path: impossible.php
inject string is: 1
inject string is: ' OR 1=1#
inject string is: 1 OR 1=1
inject string is: 1' OR 1=1 UNION SELECT NULL, COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS=users WHE
RE COLUMN_NAME LIKE 'password'#
inject string is: 1' OR 1=1 UNION SELECT NULL,  FROM users#
============================================================================================
Finished injecting

Out of total 4 potential injections, 0 were successful.

Your code is well protected against SQL injection.
>>> |
```

Figure 5: after executing for 'impossible.php'

4.2 Future work

Our tool works, but only for the DVWA for now. In the future, we should generalize some processes to make the tool available to handle general vulnerable website. First, we should remove possibly unnecessary log-in process or modify possibly different log-in process. Second, we should remove changing-level-of-difficulty process which is unnecessary because the level of difficulty exists only in DVWA for educational purpose. Third, we should generalize finding-input-field process. For now we assumed there is only one prevention type in one file when implementing the file parser. We will be able to fix this by adding iterations. Lastly, now we pre-defines several further possible injections to check further hacking to crack important data from database like this:



```
"1' OR 1=1 UNION SELECT [params] FROM [table] WHERE [param] LIKE [vulnerable_param]"
```

But this should contain more. For example, in the code, we did not yet predefine this for improper Type 1.

5 Github Repository

Github repository contains the code and installation guide:
https://github.com/taphan/ast-project

Link to demo: https://youtu.be/PM99MOUfpk4

References

[1] Php: mysqli::real_escape_string - manual. https://php.net/manual/en/mysqli.real-escape-string.php, 2019. Accessed 2019-06-15.

[2] mysql_real_escape_string sql injection - correct usage and attacks. http://www.sqlinjection.net/advanced/php/mysql-real-escape-string/, 2019. Accessed 2019-06-15.

[3] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, volume 1, pages 13–15. IEEE, 2006.

[4] Adam Kieyzun, Philip J Guo, Karthick Jayaraman, and Michael D Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *Proceedings of the 31st International Conference on Software Engineering*, pages 199–209. IEEE Computer Society, 2009.