A Learning Platform for SQL Injection

Nada Basit
University of Virginia
basit@virginia.edu

Abdeltawab Hendawi
University of Virginia
hendawi@virginia.edu

Joseph Chen
University of Virginia
jmc2fz@virginia.edu

Alexander Sun
University of Virginia
ahs9vt@virginia.edu

ABSTRACT

We present a web application system where users can learn about and practice SQL injection attacks. Our system is designed for students in a university level database or computer security class, and is aimed towards students familiar with SQL but with little experience in web security. Our platform currently contains 12 levels, each of which demonstrates a SQL vulnerability that the user must exploit. For each level, we explain the goal of the challenge, and also provide detailed solutions. Our system provides advantages over other methods of teaching SQL injection because it is hands-on, the challenges provide a greater scope of vulnerability coverage, and is easily extensible, allowing instructors to add their own SQL injection problems for their students.

CCS CONCEPTS

• **Security and privacy → Database and storage security**; • **Applied computing → Interactive learning environments**;

KEYWORDS

SQL injection; database security; learning tool; education

ACM Reference Format:
Nada Basit, Abdeltawab Hendawi, Joseph Chen, and Alexander Sun. 2019. A Learning Platform for SQL Injection. In *SIGCSE '19: 50th ACM Technical Symposium on Computer Science Education, February 27–March 2, 2019, Minneapolis, MN, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3287324.3287490

1 INTRODUCTION

In today's technology-driven society, website applications have a predominant role in everyday life. People use websites for online shopping, banking, communicating with friends, and much more. Many websites use databases in their backend to store user information. Because valuable information such as passwords, credit card numbers, or social security numbers are stored in those databases, they become common targets for malicious hackers.

One common attack against SQL databases is known as SQL injection. SQL injection is a web vulnerability where malicious users input carefully crafted SQL statements into a website form. In order to execute the SQL query, websites often concatenate untrusted user input with code to be executed. Successful SQL injections have the potential to read sensitive data from a database or execute administrator actions [14]. According to the Open Web Application Security Project (OWASP), SQL injection is the most critical web vulnerability [11]. Similarly, according to MITRE's Common Weakness Enumeration, the most dangerous programming error is SQL injection [6]. SQL injection attacks have been used in real world websites on numerous occasions. In as early as 2002, over 200,000 credit card numbers were leaked from Guess.com using a SQL injection attack [1][10]. The year after, over 500,000 credit card numbers were leaked from PetCo.com using another SQL injection attack [7]. In 2013, the hacker group "RedHack" found a SQL injection vulnerability in the Istanbul Administration Site. They were able to erase people's debts to water, gas, electricity, and Internet companies [12]. They also tweeted the injection so that others could also exploit the vulnerability. Therefore, it is important that web developers understand the risk that SQL injection poses.

We believe there are two main reasons why SQL injection remains a problem today. One is that some web developers are either unaware or unconcerned about SQL injection, which causes the websites they create to be vulnerable. The second is that even if security is considered, hackers are constantly looking for new attacks to leverage. Even though there have been many countermeasures [2][3][4][8], each method begets another method of attack, and new vulnerabilities are continually being found. Although some of these vulnerabilities are reported, others go unreported and the public has no knowledge they exist. These factors make it difficult to keep our websites secure.

In this paper, we address the problem of SQL injection by describing a SQL injection learning platform we created. The platform is a web application that is designed to be used in a classroom setting. It demonstrates the vulnerabilities related to SQL present in web applications by allowing users to craft their own injections to hack into intentionally vulnerable databases. Our goal is that users will understand the importance and consequences of SQL injection through attempting injections themselves. The platform aims to address the lack of awareness of SQL injection by teaching and providing examples of SQL injection, and attempts to address newer vulnerabilities by being easily extensible. The platform is centered around a series of levels, which get progressively more difficult and build off the previous ones. Each level presents a different scenario, along with a vulnerable website form. The user

SIGCSE'19, February 27–March 2, 2019, Minneapolis, MN, USA

ACM ISBN 978-1-4503-5890-3/19/2…$15.00
https://doi.org/10.1145/3287324.3287490

must understand the scenario and then craft an appropriate SQL injection query which will exploit the vulnerability. Users create their own exploits because we believe a hands-on approach is more beneficial when conveying the importance of computer security.

Among the work that have tried to teach SQL injection, our platform differs in that it is user-friendly towards people who are new to SQL injection and provides extensibility to educators who want to add new levels. Some sites aim to be a practice tool with harder challenges without much guidance towards beginners. In contrast, our site aims to be a simple and effective teaching tool that starts users with basic challenges and progresses through harder problems.

We review related work in the area of teaching SQL injection in Section II. We then discuss an overview of our system, including the contents and architecture of our system, in Section III. In Section IV, we explain the extensibility of our system and show how a professor would add a new challenge to the site. In Section V, we explore each level, explaining the challenge, goal, and solution for each one. Finally, we present reflections on our tool in Section VI.

2 RELATED WORK

Uskov's paper [15] gives an overview of "Software and Web Applications Security" courses that are based on hands-on teaching of security topics, one of which is SQL injection. The course framework first gives an overview of SQL injection, explaining the vulnerability and under what circumstances it occurs. Next, a demonstration of the attack is shown, along with an example of a tautology SQL injection. The framework then shows how SQL injection can be prevented and defended against. Finally, the author provides some hands-on SQL injection assignments for students to try.

The authors in [9] present a survey of SQL injection attacks along with methods of prevention. They also discuss online practice tools that aid security professionals in testing their skills. These tools are similar to the platform that we have developed in that they are websites intentionally vulnerable to SQL injection and have the user practice exploiting them. These tools include DVMA (Damn Vulnerable Web App), Web Security Dojo, and OWASP. The authors go into the details of the pros and cons of these tools, for example, they mention that DVMA leaves out hints and explanations of exercises, how OWASP is completely Java-oriented, ignoring PHP, and how Web Security Dojo is not user-friendly for beginners to SQL injection. We took these considerations into account when designing our platform.

The authors in [13] created a way to teach database security through mobile labs. Their goal was to broaden mobile security education, and they built a mobile application for Android that consisted of several labs on computer security. The modules not only taught SQL injection but also focused on ethical issues and explained how to fix the vulnerability.

Chen and Tao [5] developed a teaching tool called SWEET, which uses virtualization so students can learn and practice web security in a safe environment. They illustrate most web security threats through the use of an external application called OWASP WebGoat. WebGoat contains a series of web challenges that tries to simulate real life situations such as stealing credit card numbers and logging into websites as admin. It includes SQL injection challenges that are friendly to new learners and can be adopted separately for different courses.

Besides the aforementioned related work, much of the work that has been done in developing database security curriculums are based on lectures, and not hands-on activities. However, besides published papers and pedagogical material, there exist several sites that offer SQL injection challenges. We reviewed a number of these sites, including SQLZoo, Overthewire, pentesterlab, the SEED project, the NICE challenge, and Hackme. Overall, we believe our platform offers two main improvements. First, we designed the site with extensibility in mind from the beginning. We attempted to make the process of adding new levels as simple as possible so that instructors can include their own problems for their students. The second is that our site is more user-friendly towards students who are new to SQL injection. Some websites are behind a paywall, or require users to first set up a virtual environment. In contrast, our challenges are free to access and are hosted online. Many of the existing sites are aimed at people who are already familiar with SQL injection and want to practice, consequently, their challenges can be unclear or even discouraging to beginners with little experience in security. Our platform starts with basic injection challenges, eventually building up to the more advanced ones. Along the way, we provide step-by-step explanations. Overall, the difference is that our platform aims to be a teaching tool rather than a practice tool.

3 SYSTEM OVERVIEW

Our system is a web application running on a LAMP (Linux, Apache, MariaDB, PHP) architecture. The application introduces the user to SQL injection and challenges users to write their own injections to solve increasingly difficult SQL injection problems. In addition to the SQL injection challenges, there is a wiki that guides the users as they progress through the challenges. The wiki focuses on conveying the important ideas and methods in SQL injection attacks, such as intuition for analyzing the situation and how to utilize SQL commands and syntax to craft exploits to retrieve the desired information.

Our platform has 12 challenges, each of which introduces a different type of SQL injection or vulnerability. Each of the challenges is associated with a different database, which contain the tables necessary to run the challenge. For each level, the website will execute a provided query that has been combined with the user input, which is an injection. When running the query, the SQL engine will fetch data from the corresponding database. The 12 databases are isolated from each other and are each associated with a MariaDB user that only has `SELECT` permissions for the corresponding database.

The process of submitting an injection is displayed in Figure 1. We describe the steps in more detail.

User Input For 11 of the levels, users are presented with a website that has a form with an empty query box. Users put their injection in the box and submit the form. One level does not have an input box, but rather asks the user to perform a SQL injection through a GET request. Each level has a provided query, which is what the user input will be inserted into. The provided query is given for the first few levels but is removed later to provide a

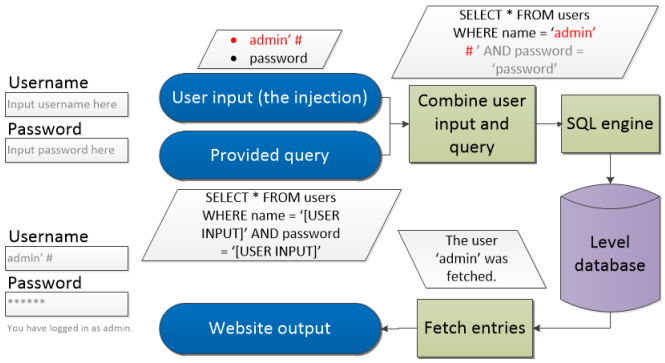

Figure 1: Process of submitting an injection, from user input to website output, with a username and password interface.

challenge. In Figure 1, the website has a login functionality where the provided query is

```
SELECT * FROM users WHERE name = [USER INPUT] AND
        password = [USER INPUT]
```

For the user input, the user should input an injection that allows them to log in as a specific user, in this example, the injection `admin'#` allows the user to log in as admin.

Combine user input and query All inputs through the input box are inserted into the SQL statement, just as in an actual web application, and then are processed and executed. The specifics of the combiner vary per level, but most work by concatenating sections of the provided query and user input together to get the entire injection. All levels that involve SQL injection do not sanitize queries, nor do they use prepared statements. The levels are designed to mimic insecure web pages that might not have any sanitization or protection.

SQL engine, Database The SQL statement, containing the user's injection, is executed. The engine goes to the database and searches through the rows of the `users` table.

Fetch entries The system goes through the rows of the table and retrieves the entries that matched the SQL query. In this example, the row of the table with username "admin" will be fetched.

Website Output The requested data is displayed on the website. The output format varies depending on the level. In some levels, a table of all the fetched data is displayed on the page, in other cases, only one row can be displayed at once. In more advanced levels, a restricted amount of information is returned to the user, such as "Your query has executed" without displaying the retrieved data. In the example above, the website will display "You have logged in as admin".

4 EXTENSIBILITY

A key feature of our system is that it was designed with extensibility in mind. SQL injection attacks are constantly evolving, and we would like our application to be able to adapt. In the future, when new vulnerabilities have been discovered, it will not be enough to just know how to solve the 12 challenges available on the current platform. In this section, we explain the process of adding another level to our site. Instructors that want to create a challenge that demonstrates a new vulnerability can do so by following these steps.

(1) First obtain the source code for the platform and get a copy of the site running for their own personal or classroom use. The platform should work on most UNIX based systems. We envision that every professor who uses our platform and wishes to extend the challenges will eventually have their own version of the site, with the 12 provided challenges and then followed by their own.
(2) Create a database called "level13", with tables, columns, and entries as necessary. Also, create a user called "level13" which has `SELECT` privileges for the level13 database. No other privileges should be granted unless they are needed for the challenge.
(3) There is a "levels/" folder where the necessary files for each level are stored. A new folder called "level13" should be created. In this folder needs to be a PHP file that contains the code to run the challenge.
(4) Finally, after the setup for the new level is done, a few minor changes must be made so that students can navigate to that challenge. On the webpage for each level, there is a button that links to the next, so a link should be created from level 12 to level 13. A level 13 button should also be added on the main page.

5 LEVEL OVERVIEW

In this section, we explain the objective of each challenge and the type of attack each level introduces. Each level follows the same overall format and is displayed in the screenshots below. In the white box, a description and goal of the level is provided. Next, there is an input box where users should input their injection. Some levels display the provided query concatenated with the user input to help students with their injections. After submitting the form, the results are displayed back to the user.

(1) **Display all rows of a table**

In the first level, users are given a form with an input box asking for a username. After submitting a username, all rows in the table `users_level1` that matched the given username will be retrieved and displayed. The challenge of this level is to use SQL injection to display all rows of this table, even though the user does not know all the usernames. The provided query is

```
SELECT username FROM users_level1 WHERE username =
                '[USER INPUT]'
```

The provided query contains a `WHERE` clause that checks whether the username in the table matched the inputted username. If the user inputs "Bob", all rows in the table with username "Bob" will be returned. To fetch every row in the table, we need to make the `WHERE` clause always true no matter the username. A well-known method is to use `' OR ''='`. The query with the injection is

```
SELECT username FROM users_level1 WHERE username = ''
                OR ''=''
```

where the text in red is the user input. The `WHERE` clause now checks if the username is the empty string, or if the empty string is equal to the empty string. As the second comparison will always be true, all rows of the tables will be fetched. Figure 2 displays the challenge, showing the general layout of each website and also the output from inputting the injection.

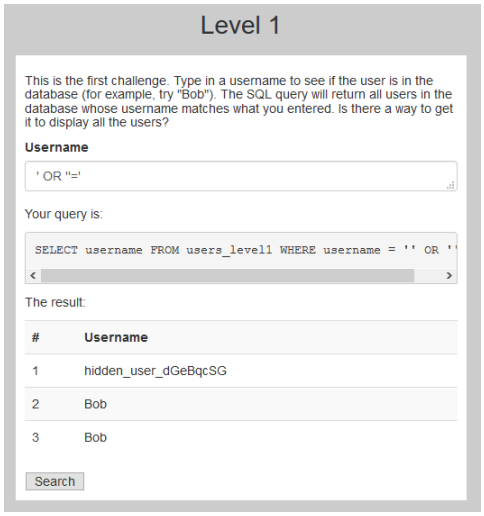

Figure 2: Level 1 page with injection query and output of usernames.

(2) **Display other tables in the database**

Level 2 asks the user to fetch the names of other tables in the database. The provided query is

```
SELECT username FROM users_level2 WHERE username =
                '[USER INPUT]'
```

Students can fetch data from the `TABLE_NAME` column of the `TABLES` table of the `INFORMATION_SCHEMA` database using a query such as

```
SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
```

To solve the challenge, the quotation mark should be closed and a query that selects table names should be appended using the `UNION` operator. Finally, the # symbol can be used to comment out the rest of the query to ignore the closing ' character. The injection is shown in the SQL statement below in red.

```
SELECT username FROM users_level2 WHERE username = ''
          UNION SELECT TABLE_NAME FROM
          INFORMATION_SCHEMA.TABLES #'
```

(3) **Find the column names of a table**

Level 3 asks for the column names of a hidden table in the database. This requires users to first list the tables, using the level 2 technique and then finding the columns in the other table. The provided query for visualization is given as:

```
SELECT username FROM users_level3 WHERE username =
                '[USER INPUT]'
```

The first step is to use the injection from level 2 to list the table names in the database. After finding a suspicious table, students can find the column names of the table from the `COLUMNS` table in `INFORMATION_SCHEMA`. The injection is shown in the SQL statement below in red.

```
SELECT username FROM users_level3 WHERE username = ''
          UNION SELECT COLUMN_NAME FROM
          INFORMATION_SCHEMA.COLUMNS WHERE
          TABLE_NAME='hidden_table_NSXOkukMOh'; #
```

(4) **Retrieve information about the database**

All SQL databases have a database name, DBMS version, and current user associated with it. The name of the database or current user can help with creating an injection and knowing the DBMS version helps with identifying known vulnerabilities. The query:

```
SELECT username FROM users_level4 WHERE username =
                '[USER INPUT]'
```

The database name, username, and DBMS version can be retrieved using the `DATABASE()`, `CURRENT_USER()`, and `VERSION()` functions, respectively. Users can find these functions either through online search or the SQL manual. The query below shows the injection to accomplish this.

```
SELECT username FROM users_level4 WHERE username = ''
          UNION SELECT DATABASE() UNION SELECT CURRENT_USER()
          UNION SELECT VERSION() #'
```

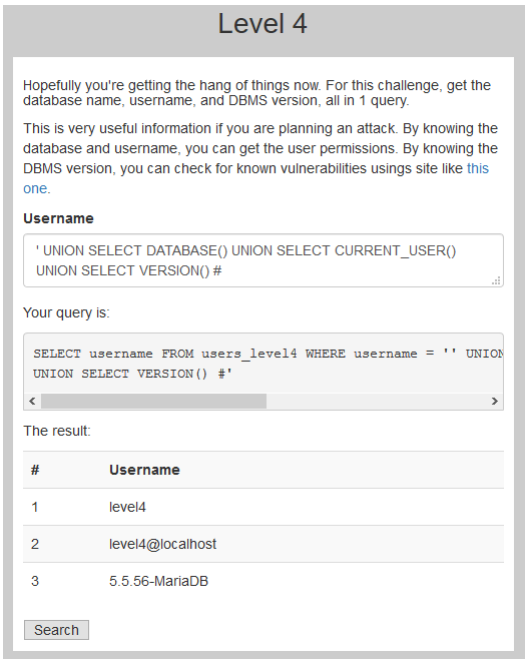

Figure 3: Level 4 page with injection query and output.

(5) **SQL injection through a GET request**

The form for this level does not have an input box. Rather, the user should input their injection through the GET request in the URL. The provided query is the same as in Level 1, so the level 1 SQL injection query can be reused. The current URL ends with

```
?username=Bob
```

which means that the website will send a GET request with the username equal to Bob. Students should replace Bob with the injection and must also learn about URL encoding.

(6) **Log in as admin**

This level challenges a student to log in as admin through a login form with username and password fields. The provided query is

```
SELECT username FROM users_level6 WHERE username =
  '[USER INPUT]' AND password = '[USER INPUT] '
```

This level is slightly different from the previous ones in that there are two inputs, one for the username and the other for the password. The query compares the given username and password to the entries in the table, and if they match, then the user will be logged in. The username is given as "admin", but the password is unknown. However, using SQL injection, the username check can be closed with a quote and the password check can be bypassed using #. The query is

```
SELECT username FROM users_level6 WHERE username =
    'admin' # ' AND password = '[USER INPUT] '
```

The result is that everything after the # is ignored and the SQL executor will only compare the usernames. This bypasses the need to know the password.

(7) **Retrieve admin's password**

Another vulnerability in login forms is that a malicious attacker can retrieve user passwords. The provided query is

```
SELECT username FROM users_level7 WHERE username =
  '[USER INPUT]' AND password = '[USER INPUT] '
```

The form will display whatever is selected from the query onto the website. Currently, the query selects usernames from the `users_level7` table. The solution to getting the password is making the form return the password instead of the username. From the provided query, students can deduce that there is a column named password that exists in the same table. Therefore, the username selection query can be closed and a new query that selects the admin's password can be appended.

(8) **Read a file on the server**

Hackers can use SQL injection to read files located on an improperly configured server and obtain sensitive information. This level demonstrates how file read is done through SQL. The query is:

```
SELECT username FROM users_level8 WHERE username =
                  '[USER INPUT] '
```

SQL has the built-in function `load_file()`, which will return the contents of a file. The query with the injection is

```
SELECT username FROM users_level8 WHERE username = ''
                  UNION SELECT
load_file('/var/lib/mysql-files/hidden_file.txt')#'
```

The filepath was given in the level description. For security reasons, the server was set up so that only files in that folder could be read from.

(9) **SQL Truncation attack**

Although not a injection, this challenge introduces a SQL vulnerability related to how MariaDB truncates entries in a table. When a user inputs a value into a database, sometimes MariaDB will truncate the input if it is larger than the maximum column width. To simulate this, a table was created where the maximum length of a username is 20 characters.

The site allows two actions, to register and to log in. The goal of the challenge is to log in as the admin user. The PHP source code was given for this level. When registering a user, the inputted username is first checked against the existing entries in the database, and this line of code is executed.

```
$username = trim(substr($_POST["username"], 0, 20))
```

Given a username input, the code takes the substring of the first 20 characters and then removes the whitespace from the substring. This means a 21 character input such as "admin a" will be truncated to "admin ", and then trimmed to just "admin". The check if the username is already in the database occurs before the trim, therefore, an account can be registered with that username and a new password, which will not be caught because "admin a" is not equal to "admin". After this user is inserted into the database, students can log in with "admin" and the password they used.

(10) **Boolean SQL injection**

This level again challenges the student to fetch the admin password. However, in level 7 the website displayed the result of the query, which allowed us to directly print the password out on the page. However, this level only shows if the login was successful, without printing any information from the database onto the page.

One strategy is to brute force the admin password one letter at a time using the LIKE operator. For example, using this SQL injection:

```
admin' AND password LIKE 'a%'#
```

can determine if the password begins with the letter a. If the website displays that the login was successful, then the password starts with a. Otherwise, we try the next letter, b. Once the first character is found, which happens to be q, a second letter can be tested and so on until the full password is found. This type of attack is known as boolean SQL injection.

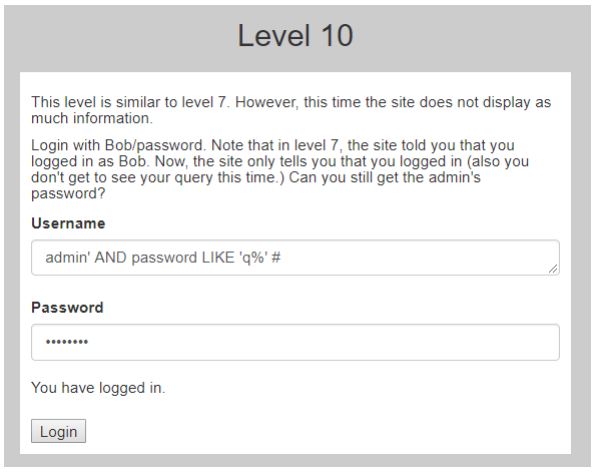

Figure 4: Level 10 page with the injection that shows the passwords starts with the letter q.

(11) **Retrieve table name and then admin password**

Level 11 is an extension of two previous levels, challenging the user to first find the table name and then the admin password. An additional challenge is that each query can only return one row of

a table. Students need to use additional SQL commands to solve this challenge.

For this level, users first need to know the table name, which can be found using the injection from level 2. An additional challenge in this level is that only one row can be returned from the table at once. Students need to use additional SQL commands to solve this, such as the `OFFSET` operator or `GROUP_CONCAT`. After finding the table name, students can use the Level 7 query to finish the challenge.

(12) **Retrieve the admin password using a timing attack**

Level 12 is an important level which introduces side-channel attacks, namely a timing attack. Level 12 challenges the student to try to get the admin password when the form only displays that the query has finished executing. Because it only displays that the query has executed, previous attacks do not tell enough information to retrieve the password. The level 7 approach cannot be used because the admin's password is not displayed. The level 10 approach cannot be used because the server does not display if the query was successful, only that it has finished.

The solution for this level is the same as the solution for level 10, but with the addition of a timing attack. The query will delay the output by 5 seconds if the injection matched a row in the table, and immediately return the output otherwise. Using the MariaDB function `CASE`, a query can be built.

For example, a query can be built to check the letter `a`. The query will first check if the admin's password starts with `a`, just like in level 10. If it does, it will pause for 5 seconds, if the password does not start with `a`, it will not pause. From the user's perspective, if there is a 5 second pause, then a character of the password was found. This process is continued by checking `b`, then `c`, etc, just like in Level 10.

We have provided descriptions and explanations of the 12 levels we implemented for our SQL injection practice tool. We hope that they provide an idea of the type of injection and attack strategies we are aiming to teach. Future administrators can extend the website to include challenges that should build off the existing 12.

6 REFLECTIONS

The system was used during a Spring 2018 database systems course which had 117 students. The students were all 3rd or 4th year computer science majors. Students formed small groups of two or three and worked through the challenges on the platform together. They were asked to solve and review at least 4 of the challenges during class and finish for homework if necessary.

After the students completed the activity, they were asked to respond to a short survey as groups. Students were asked about which challenges they found most and least interesting or useful. They were also asked about their overall opinion of the site and for any other comments. Subjectively, the feedback was overall positive, with many students saying they enjoyed the platform and the challenges. Students liked the hands-on aspect of the tool and stated that certain aspects of our platform, such as the ability to see the injection within the query, were helpful to learn SQL injection. Collectively, they said that our platform was a nice way to learn about SQL injection. After reading through the student feedback, we believe our platform offers a practical way to teach SQL injection to students.

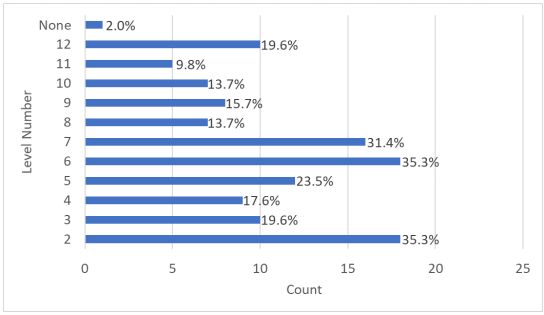

Figure 5: Levels students found most useful.

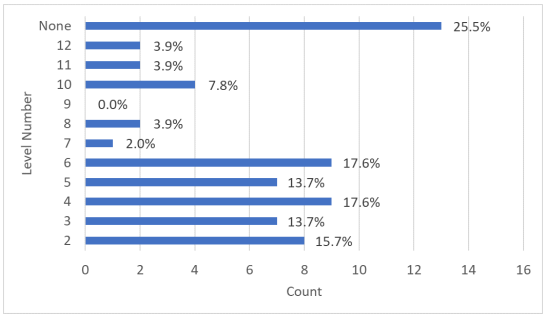

Figure 6: Levels students found least interesting.

Figures 5 and 6 show the student responses as to which levels they found most and least interesting or useful. Level 1 was not included in this survey because the professor demonstrated that challenge for the class. According to the survey, the most useful levels were levels 2 and 6, with 35.3% of the groups finding them useful. The least interesting were levels 4 and 6, with 17.6% of the groups finding them the least interesting.

We also read through the student comments and noted their reasons for not liking a level. The most common reason was that the level was too simple, and therefore not interesting. Some other faults were that it was unclear what to do, or too similar to a previous level. We noticed that most of the responses for the least interesting level were the first six levels. This is reasonable because we intended the introductory challenges to be simple.

In the future, we would like to integrate our platform into the course rather than having it be a separate activity. We will also take student feedback into consideration and improve the levels that students found unhelpful or unclear. Problems can be changed to have a different direction to make the challenge unique, and more context will be added so students find the challenges to be clear. More difficult SQL injection challenges can be included for advanced students. Finally, we would like our platform to be publicly available to other institutions so that instructors can add their own problems.

REFERENCES

[1] Aarafat Aldhaqm, Shukor Razak, Siti Othman, Abdulalem Aldolah, and Md Ngadi. 2016. Conceptual Investigation Process Model for Managing Database Forensic Investigation Knowledge. 12 (02 2016), 386–394.

[2] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. 2007. CANDID: Preventing Sql Injection Attacks Using Dynamic Candidate Evaluations. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, New York, NY, USA, 12–24. https://doi.org/10.1145/1315245.1315249

[3] Martin Bravenboer, Eelco Dolstra, and Eelco Visser. 2007. Preventing Injection Attacks with Syntax Embeddings. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE '07)*. ACM, New York, NY, USA, 3–12. https://doi.org/10.1145/1289971.1289975

[4] Gregory Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti. 2005. Using Parse Tree Validation to Prevent SQL Injection Attacks. In *Proceedings of the 5th International Workshop on Software Engineering and Middleware (SEM '05)*. ACM, New York, NY, USA, 106–113. https://doi.org/10.1145/1108473.1108496

[5] L. Chen and L. Tao. 2011. Teaching Web Security Using Portable Virtual Labs. In *2011 IEEE 11th International Conference on Advanced Learning Technologies*. 491–495. https://doi.org/10.1109/ICALT.2011.153

[6] Steve Christey. [n. d.]. CWE -2011 CWE/SANS Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25/.

[7] J Clarke. 2009. *SQL Injection Attacks and Defense*. 1–473 pages.

[8] William G. J. Halfond and Alessandro Orso. 2006. Preventing SQL Injection Attacks Using AMNESIA. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, New York, NY, USA, 795–798. https://doi.org/10.1145/1134285.1134416

[9] Diallo Abdoulaye Kindy and Al-Sakib Khan Pathan. 2013. A Detailed Survey on Various Aspects of SQL Injection: Vulnerabilities, Innovative Attacks, and Remedies. *IJCNIS* 5 (2013).

[10] Odunayo Esther Oduntan and Temitope Sunday Aluko. 2014. Securing Web Application from Structured Query Language Injection Attacks : A Four-Tier Approach.

[11] OWASP. [n. d.]. OWASP Top 10 - 2017: The Ten Most Critical Web Application Security Risks.

[12] Burak Polat, Cemile Tokgöz Bakıroğlu, and Mira Elif Demirhan Sayın. 2013. Hactivism in Turkey: The Case of Redhack. 4 (10 2013). https://doi.org/10.5901/mjss.2013.v4n9p628

[13] Kai Qian, Dan Chia-Tien Lo, Hossain Shahriar, Lei Li, Fan Wu, and Prabir Bhattacharya. 2017. Learning database security with hands-on mobile labs. In *2017 IEEE Frontiers in Education Conference, FIE 2017, Indianapolis, IN, USA, October 18-21, 2017*. IEEE Computer Society, 1–6. https://doi.org/10.1109/FIE.2017.8190716

[14] A. K. Sood and R. J. Enbody. 2013. Targeted Cyberattacks: A Superset of Advanced Persistent Threats. *IEEE Security Privacy* 11, 1 (Jan 2013), 54–61. https://doi.org/10.1109/MSP.2012.90

[15] A. V. Uskov. 2013. Hands-On Teaching of Software and Web Applications Security. In *2013 3rd Interdisciplinary Engineering Design Education Conference*. 71–78. https://doi.org/10.1109/IEDEC.2013.6526763