');DROP TABLE textbooks;--

An Argument for SQL Injection Coverage in Database Textbooks

Cynthia Taylor
Oberlin College
ctaylor@oberlin.edu

Saheel Sakharkar
University of Illinois at Chicago
ssakha2@uic.edu

ABSTRACT

In this position paper, we look at the representation of SQL injection within undergraduate database textbooks, and argue that both discussion of security issues and security of example code must be improved. SQL injection is a common database exploit which takes advantage of programs that incorrectly incorporate user input into SQL statements. Teaching students how to write parameterized SQL statements is key to preventing this wide-spread attack. We look at the current editions of seven textbooks used at the top 50 US CS programs, and analyze their coverage of SQL injection, use of parameterized queries, and correctness of examples. We find a wide variety in the amount of coverage given to the topic, from none at all to in-depth coverage of defenses. Additionally, we find cases of SQL injectable code given as examples of how to correctly write queries in two of seven textbooks.

KEYWORDS

SQL injection, database textbooks, database security

ACM Reference Format:
Cynthia Taylor and Saheel Sakharkar. 2019. ');DROP TABLE textbooks;--: An Argument for SQL Injection Coverage in Database Textbooks. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19), February 27-March 2, 2019, Minneapolis, MN, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3287324.3287429

1 INTRODUCTION

It has frequently been argued that if we want all students to be exposed to computer security and secure coding practices, these topics must be integrated into many CS courses, not isolated in computer security classes [17, 19–21, 33, 35]. However, even though including relevant security topics in courses has been promoted since the nineties [17, 33], many courses still do not cover them.

Education research has shown that textbooks frequently guide the topics covered in a course [11, 14]. Because textbooks help define what is covered, their inclusion of security topics can determine whether or not students are exposed to security guidelines and best practices. When assessing the current state of security coverage in courses, textbooks can give us important information about how likely a topic is to be included in most classes. Furthermore, adding security concepts into textbooks could be key to getting them widely discussed.

With that in mind, we look at a single security issue that we think all undergraduates should be exposed to, SQL injection, and analyze how it is covered in the most recent editions of database textbooks used in the top fifty Computer Science departments in the United States. SQL (Structured Query Language) is a family of languages used to run queries on databases. SQL injections take advantage of SQL queries which do not safeguard against improperly formatted user input, and as a result allow malevolent users to effectively run their own SQL statements against a database, allowing them to exfiltrate user information, log in as arbitrary users, and destroy data. SQL injection attacks have stolen credit card data, hacked the websites of election boards, and revealed cheating spouses [13, 32, 38]. SQL injection vulnerabilities are among the most common database vulnerabilities and have consistently appeared at the top of vulnerability lists [23, 39], despite the vulnerability being over fifteen years old [7]. These attacks have become even more common over time, as automated tools to detect and exploit them have become readily available [7].

This common attack can easily be prevented with proper coding techniques, namely using parameterized queries. The fact that vulnerable coding techniques are still commonly used has been blamed by some industry managers on a lack of education about this attack, with a recent article quoting one manager as saying "Any serious programmer should know about SQLi [SQL injection], but there's a massive shortage of programmers, so companies hire anyone even if they don't have the right training" [7]. The same article describes another manager who blames bad educational resources, saying "he lamented about the large number of tutorials available to web developers online that, instead of providing decent advice, detail how to make systems that are vulnerable to SQLi" [7].

Given that SQL injection can be prevented with a properly educated workforce, it is important that database textbooks demonstrate how to correctly parameterize queries to protect against it, and include in-depth discussion of the vulnerability and the importance of preventing it. In this work, we look at how commonly used database textbooks address the issue. We analyzed each textbook with regard to the following questions:

(1) Is SQL injection discussed in the textbook?
(2) Are there examples demonstrating how to incorporate user input while protecting against SQL injection?
(3) Are there SQL examples that would be vulnerable to SQL injection?

We find that there is a wide variety in how different texts discuss the issue, ranging from having code examples which are vulnerable to SQL injection, having no examples which take user input, or not

Table 1: Text name, author, publication date, and number of universities using the text. All editions are the most recent for the text.

Text	Author	Pub. Date	Using Text
Database Management Systems, 3rd ed. [24]	Ramakrishnan	2002	16
Database System Concepts, 6th ed. [28]	Silberschatz	2010	10
Database Systems: The Complete Book, 2nd ed. [12]	Garcia-Molina	2008	10
Fundamentals of Database Systems, 7th ed. [9]	Elmasri	2015	4
Database Systems: A Practical Approach to Design, Implementation, and Management, 6th ed. [6]	Connolly	2014	2
Data Modeling Essentials, 3rd ed. [29]	Simsion	2004	1
Learning SQL, 2nd ed. [1]	Beaulieu	2009	1

discussing SQL injection at all within the text, to discussing the issue and dangers in depth and demonstrating the correct way to use variables within a SQL query. Across seven textbooks, we look at 745 SQL examples, of which only 59 incorporate user input. Of these 59, four examples include user input in ways that make them vulnerable to SQL injection. While this is only 6.8% of examples that include user input, any of these examples have the potential to make student code vulnerable to attack if the techniques are followed in deployed code. Given that SQL injection attempts against internet-deployed programs are inevitable in modern environments, this has the potential to cause real harm to users of these programs.

Even more troubling is that we find the majority of these textbooks (five out of seven) do not discuss SQL injection at all. This means that even if the book itself does not contain vulnerable examples, students who later find such examples on the internet tutorials and help sites (e.g., Stack Overflow) will be unlikely to recognize the code as dangerous.

In this position paper, we first analyze the most recent editions of popular database textbooks to quantify their coverage of SQL injection and defenses against it. Following this, we make recommendations for how instructors, textbooks authors, and the security community. We end with an argument on the importance of presenting secure code in textbooks.

2 SQL INJECTIONS AND HOW TO PREVENT THEM

There are a number of variants of SQL attacks, and a vast array of advice on how to prevent them [16, 27]. In a classic SQL injection, a malevolent actor takes advantage of improperly sanitized data input in order to run their own query against the database. The key to a successful SQL injection attack is the ability to have the database interpret user input as SQL code, rather than as data.

Consider the following dynamically constructed query, where the variables `user` and `pass` are input by the user and concatenated with the rest of the string to form a SQL query:

```
'SELECT * FROM usertable WHERE username="'
+ user + '" AND password="' + pass + '";'
```

A cooperative user will enter their username and password (say, "alice" and "rabbit"), to result in the string

```
SELECT * FROM usertable WHERE
username="alice" AND password="rabbit";
```

which will return all information on the user alice with password rabbit.

A malevolent user could enter `x" OR "1"="1` as the password. When the string is concatenated together, this will result in the following query:

```
SELECT * FROM usertable WHERE
username="alice" AND password="x" OR "1"="1";
```

By using a quote in their input string, the user causes the database to interpret OR as part of the SQL code rather than as part of the password, indicating that they should select information on the user either if the password is equal to x, or if the second part of the clause is true. The phrase `"1"="1"` will always evaluate to true, allowing a bad actor to pull details for any user. Using a similar phrase for the username will allow them to view the entire table without having to know specific usernames. Additionally, adding `;--` to the end of the input will end the SQL statement and comment out everything following on the same line. New SQL statements can be added between ending the previous statement and commenting out the rest of the line, allowing hackers to run arbitrary code against the database. While the details of this attack may vary depending on the SQL variant being used, the basic attack works across all SQL varietals.

The most effective measure for preventing SQL injection is always using parameterized queries to incorporate user input, rather than dynamically creating SQL statements via string concatenation. In the example above, a parameterized version of the query would look something like:

```
SELECT * FROM usertable WHERE username=@user
AND password=@pass;
```

At runtime, the values of @user and @pass will be programatically set to values input by the user. Using parameters allows the rest of the SQL statement to be compiled ahead of time, with only the parameters changing at run time. As a result, user input will always be treated as data and never be compiled into SQL, voiding the ability of user input to be interpreted semantically as part of SQL queries.

Using some sort of input filtering is additionally recommended to prevent obvious bad input, although the impossibility of being able to filter for every possible bad input means that input filtering alone is not enough. Additionally, there are many tools designed to

Table 2: Analysis of database texts and their SQL examples. Inputless queries refer to SQL statements that do not contain variables or user input. Parameter Queries refer to SQL statements that demonstrate using parameters for user input. Unsafe Queries refer to SQL statements which are vulnerable to SQL injection. SQLi mentions refer to in text discussions of SQL injection, and parameter mentions refere to in text discussion of how and why to use parameterized. The last two rows indicate whether or not the text contains content related to SQL injections, namely if it contains a chapter focusing on interfacing with SQL via applications or the web, and whether it contains a chapter on security.

	Ramakrishnan	Silberschatz	Garcia-Molina	Elmasri	Connolly	Simsion	Beaulieu
Inputless Queries	53	172	60	80	51	8	262
Parameter Queries	15	10	17	11	2	0	0
Unsafe Queries	0	0	2	2	0	0	0
SQLi Mentions	0	2	0	4	0	0	0
Parameter Mentions	0	1	0	4	0	0	0
Web/App Chapter	Yes	Yes	Yes	Yes	Yes	No	No
Security Chapter	Yes	Yes	Yes	Yes	Yes	No	No

detect or prevent SQL injection [16, 27]. We treat the coverage of these tools as out of the scope of this project (and did not see them mentioned in any of the textbooks we analyzed).

3 METHODOLOGY

To determine which textbooks to analyze, we began by compiling a list of database textbooks used by the top fifty Computer Science departments in the United States [31]. We looked at the syllabus of the most recently taught undergraduate database courses at each university. We used the most recent available edition of each textbook. The textbooks are listed by popularity in Table 1.

For each text, we looked at queries regarding creating and dropping tables; insert, update, and delete statements; join queries; views and cursors; and granting and revoking privileges. We did not count queries regarding constraints, triggers, and transactions. We only looked at examples within chapters, i.e. we skipped SQL examples/statements that were presented in end-of-chapter exercises and review questions.

Our categorization of the SQL examples separated them into three categories: examples with no input, parameterized examples, and unsafe examples. Queries with no input do not require any user information, and thus are protected from SQL injection by default. Parameterized queries demonstrate incorporating user input in a way that protects from SQL injection. Unsafe queries are susceptible to SQL injection and demonstrate bad practices when incorporating user input. We did not count examples as unsafe if they were included as part of of a discussion of SQL injection, i.e. if they were illustrating what not to do. We followed the OWASP guidelines [22] when deciding if an example was vulnerable to SQL injection, namely checking if the query was formed using string concatenation to add user inputs.

We also looked at mentions of SQL injection within the text, and counted the number of times the issue was discussed, as well as the number of discussions of using parametrized queries to prevent SQL injections. These counts looked for discussion within the text, rather than query examples. Lastly, we looked at whether the textbook had chapters discussing interfacing SQL with applications or the web or chapters discussing security as a proxy for whether SQL injection was within the scope of topics covered by the text.

4 RESULTS

The results of our categorization of the SQL examples and discussion in these database texts is presented in Table 2.

We found that database texts vary widely in all aspects of SQL injection coverage, ranging from not mentioning it at all or containing unsafe SQL query examples, to having explicit discussions of SQL injection and multiple examples demonstrating the correct way to handle user input. As shown in Table 2, most of the textbooks we looked at include examples of incorporating parameters to generate safe, non-injectable SQL queries. However, most do not explicitly discuss SQL injection. We also found two database texts, Molina [12] and Elmasri [9], which include unsafe examples that are easily SQL injectable, even though Elmasri [9] discusses parameterizing SQL queries to prevent SQL injection at other locations in the text.

4.1 Discussion of SQL Injection

Five of the seven textbooks we looked at do not mention SQL injection at all. Five of these seven textbooks had chapters on both using other programming languages to access SQL databases, and on database security, making SQL injection highly relevant to their content. However, only two of these textbooks explicitly discuss it.

Silberschatz [28] mentions SQL injection where relevant, both in the section on interfacing SQL with other programming languages and in a section on application security. It advises students to use parameterized statements. However, it implies that using parameters is equivalent to using a function to add escape characters around user input. This is incorrect, as using parameters allows SQL statements to be pre-compiled, and prevents any user input from being interpreted as code, while escaping user input is not recommended as a sole defense since imperfect escape functions can easily be subverted.

Elmasri [9] mentions SQL injection in the text next to an example of a properly parameterized query, and devotes a section of the security chapter to it. The advice in their security chapter generally follows best practices, recommending students use parameterized statements, and recommending filtering input while also pointing out its limitations (namely, the impossibility of filtering out all possible dangerous characters).

Listing 1: Figure 9.31 transcribed from Molina [12]

```
$result = $myCon->query("INSERT INTO
Starsln VALUES($_POST['title'],
$_POST['year'], $_POST['starName'])");
```

Listing 2: Figure 10.13 transcribed from Elmasri [9]

```
0) import java.io.* ;
1) import java.sql.*
 ...
2) class printDepartmentEmps {
3)  public static void main (String args [])
        throws SQLException, IOException{
4)  try { Class.forName("oracle.jbdc.driver.
        OracleDriver")
5)  } catch (ClassNotFoundException x){
6)      System.out.println ("Driver could not
            be loaded") ;
7)  }
8)  String dbacct, passwrd, lname ;
9)  Double salary ;
10) Integer dno ;
11) dbacct = readentry("Enter database
    account:");
12) passwrd = readentry("Enter password:") ;
13) Connection conn = DriverManager.
        getConnection
14)   ("jdbc:oracle:oci8:" + dbacct + "/"
            + passwrd) ;
15) dno = readentry("Enter a Department
        Number: ");
16) String q = "select Lname, Salary from
    EMPLOYEE where Dno = " + dno.tostring() ;
17) Statement s = conn.createStatement() ;
18) ResultSet r = s.executeQuery(q) ;
19) while (r.next()) {
20)     lname = r.getString(1) ;
21)     salary = r.getDoublde(2) ;
22)     system.out.printline(lname + salary) ;
23)  } }
24) }
```

Listing 3: Figure transcribed from Section 6.1.3 of Ramakrishnan [24]

```
char c_sqlstring[] = {"DELETE FROM Sailors
WHERE rating>5"};
EXEC SQL PREPARE readytogo FROM :csqlstring;
EXEC SQL EXECUTE readytogo;
```

Listing 4: Figure 9.10 transcribed from Molina [12]. An almost identical example exists as Figure 10.4 in Elmasri [9].

```
1) void readQuery() {
2)      EXEC SQL BEGIN DECLARE SECTION;
3)          char *query;
4)      EXEC SQL END DECLARE SECTION;

5)      /*prompt user for a query, allocate
        space (e.g., use malloc) and make
        shared variable :query point
        to the first character of the query */
6)      EXEC SQL PREPARE SQLquery from :query;
7)      EXEC SQL EXECUTE SQLquery;
    }
```

Listing 5: Figure 11.6 transcribed from Elmasri [9]

```
0)  require 'DB.php';
1)  $d = DB::connect('oci8://act:pass12@
                     www.host.com/db1');
2)  if (DB::isError($d))
    { die("cannot connect - " .
      $d->getMessage()); }
...
3)  $q = $d->query("CREATE TABLE EMPLOYEE
4)                 (Emp_id INT,
5)                 Name VARCHAR(15),
6)                 Job VARCHAR(10),
7)                 Dno INT);" );
8)  if (DB::isError($q))
    { die("table creation not successful
           - " . $q->getMessage()); }
...
9)  $d->setErrorHandling(PEAR_ERROR_DIE);
...
10) $eid = $d->nextID('EMPLOYEE');
11) $q = $d->query("INSERT INTO EMPLOYEE
                   VALUES
12)                ($eid,
                    $_POST['emp_name'],
                    $_POST['emp_job'],
                    $_POST['emp_dno'])" );
...
13) $eid = $d->nextID('EMPLOYEE');
14) $q = $d->query('INSERT INTO EMPLOYEE
                  VALUES (?, ?, ?, ?)',
15) array($eid,
          $_POST['emp_name'],
          $_POST['emp_job'],
          $_POST['emp_dno']) );
```

How SQL injectable examples are framed and described is important. Listing 5 from Elmasri [9] demonstrates both the correct (lines 14-15) and incorrect (lines 11-12) way to incorporate form data into a SQL query using PHP. However, the fact that the first example should not be used is not discussed until two pages after the example in the text, and is not mentioned at all in the caption or on the page where the figure appears. This means a student who is skimming the text looking for an example to modify for their own code could simply copy the code that first appears in the example, without being aware that this is in fact an example of what they shouldn't do.

4.2 Demonstration of Parameterized Queries

Textbooks did much better with demonstrating parameterized queries, with five of the seven textbooks we looked at including examples of parameterized queries. This included every text that discussed interfacing with SQL from another programming language. This is very promising, as it shows that students are being shown the correct way to write queries.

The texts that did not cover them may consider parameterized queries to be out of the scope of what is covered in the text. For example, Simsion [29] focused on data modeling and only contained 8 SQL queries in total. Likewise, Beaulieu [1], while covering SQL queries in great detail with 262 query examples, did not discuss any techniques for using user input in queries, or writing queries to interface with the web or other programming languages. However, both these texts were being used in a database course at the same institution, where they were the only required textbooks, which means that students in that course had no text reference for parameterized queries.

4.3 SQL injectable examples

We found SQL injectable examples in two of the textbooks we looked at, as well as examples which introduced SQL injectable coding techniques while not being exploitable themselves.

The example shown in Listing 1 uses the PHP $_POST variable, which obtains user input from form elements. This means that any user input will be directly concatenated with the SQL statement, with no alteration. This example is clearly vulnerable to SQL injection, yet is presented in the text as the correct way to incorporate user input into SQL queries within a website. Students who follow this example in their code will be creating websites which are trivially hackable, and actively dangerous to their users.

Some of the examples, such as Listing 2, were not strictly SQL injectable but demonstrated a coding style that was. Listing 2 is not itself SQL injectable because `dno`, the value it reads in line 15 and concatenates into a SQL query in line 16, is read as a integer. However, this example introduces the student to the idea of creating SQL statements via string concatenation, and includes concatenating strings taken directly from the user in line 14. The idea that this might be dangerous is not mentioned anywhere in the text surrounding the example, although SQL injection is mentioned elsewhere in the text. Ramakrishnan [24] also discusses constructing dynamic SQL statements and while the example it gives, shown in Listing 3, is not SQL injectable (in fact, it does not involve any variables), it demonstrates a technique which enables SQL injection without any caution to the student about why one might want to be careful when using it.

Listing 4 is an example which appears in Molina [12], although an almost identical example appears within Elmasri [9]. This example is trivially SQL injectable, but at least makes clear to the student that the user will be able to run arbitrary code against the database (although the surrounding text does not explain why this might be a bad idea).

5 DISCUSSION

In this section, we discuss the ramifications of this work and offer recommendations for both instructors, textbook authors, and the security community.

5.1 Recommendations for Instructors

As SQL injection is one of the most exploited and most common software vulnerabilities, we urge instructors make it a priority in textbook selection, and to select textbooks carefully. Textbooks should always show parameterized statements in examples incorporating user input, and should explicitly discuss the danger and popularity of SQL injection attacks.

Our finding vulnerable code examples in multiple textbooks means that instructors must use caution when selecting a databases textbook. Choosing a textbook with these examples may do real harm in leading students to write vulnerable code. While instructors can cover the correct way to do this in class, separate from the textbook, students may continue to use the text as a reference after the course is over (and after their recollection of covered topics has faded). In this case, a textbook which demonstrates vulnerable SQL queries may actually be dangerous, as students copying such examples can introduce vulnerabilities in their own code. If an instructor must use one of these texts for some reason, we recommend they specifically point out the unsafe examples to students, and encourage the students to correct or cross them out within the text.

While this work considers currently popular database texts, new books and new editions will be released. When selecting a textbook, we recommend instructors carefully read chapters that cover interfacing with SQL through other languages or via the web, as this was where we found vulnerable examples, and where SQL injection should be mentioned. We also caution instructors against assuming that this problem will be solved in new textbook editions, as the most recent textbook we looked at ([9]) contained multiple SQL injectible code examples.

Lastly, regardless of the text used, we urge instructors to spend significant class time educating students on the very real danger of SQL injection attacks, and how they can defend their code against it.

5.2 Recommendations for Authors

Although we looked only at database texts, and only at one specific vulnerability, we suspect that textbooks in other subjects demonstrate insecure examples as well. (For example, a colleague recently shared an example from an introductory C text which was vulnerable to buffer overflows.) While in most cases making a mistake or not including information in a textbook may be merely regrettable, in the case of security this can be actively dangerous. For many security vulnerabilities, the best defense we have is educating programmers. The ramifications of not educating them in these cases is real harm to the users of their applications, in the form of stolen credit cards and identifiers, personal information being revealed, and more.

In the case where textbook authors are not security experts, collaboration with security researchers when writing textbooks could be beneficial. A "security editor" could check code examples, and recommend key security concerns to cover in the text, and where they should be covered.

In order to design secure applications, we must incorporate the idea of security from the very beginning of the design process, rather than attempting to add it on at the end. Likewise, if we want students to go on to write secure code, we must center the idea of security within their education, rather than isolating it to a separate chapter at the end of a textbook, or expecting it to be covered in a separate security course. It is unfortunate that we must expect that any application that is made public will be attacked. However, if we want users to be safe, we must emphasize that expectation within our topic coverage.

5.3 Implications for the Security Community

Since textbooks shape what is covered in class [11, 14], getting security topics into textbooks could have a dramatic impact on getting security integrated into courses. Our survey showed that three textbooks were used in 81% of the courses we looked at. Getting more security coverage in these textbooks would positively impact a huge number of students. Security educators should consider collaborating with textbook authors to ensure both secure code examples and integrated coverage of security topics. SQL injection is just one of many attacks where the solution is programmer education, and all signs indicate that we have a great deal of work to do within education before these problems are actually solved.

6 RELATED WORK

Prior work has looked at analyzing examples presented in computer science textbooks; our approach followed a model similar to Börstler et al. [2], where examples from commonly used introductory programming textbooks were reviewed for their quality. Within the context of databases, both Said et al. [25] and Conklin and Heinrich [5] compare the topics covered in various database textbooks, Said [25] being more focused on database security topics whereas Conklin [5] is more focused on database topics in general. However, neither focused specifically on SQL injection, as we do in this work.

Previous work has looked at how to incorporate security into database courses, frequently citing a current lack of coverage on the subject. Yang [34], Li et al. [18], Guimaraes et al. [15], and Stalvey et al. [30] all offer guidelines, coursework and labs on database security, which include SQL injection among other topics. A great deal of work has been done creating hands-on labs which teach SQL injection, to be used either in database or security courses. Yuan et al. [37] evaluates several different lab frameworks for teaching students about SQL injection, including [3, 4, 8, 36]. Other lab frameworks which include SQL injection are presented in [10, 18, 26]. Given the wide variety of lab work covering this topic, SQL injection is clearly considered an important topic by the security community.

7 CONCLUSION

The majority of textbooks we looked at did not discuss one of the most widespread and dangerous security problems today, SQL injection, an attack for which the most effective defense is programmer education. Additionally, multiple textbooks included code examples that would make programs vulnerable to this attack. Given the pervasiveness of SQL injection on the modern web, not educating students about it is irresponsible and dangerous, and supplying vulnerable code is actively harmful.

To stop SQL injection vulnerabilities in the wild, we need to educate students on how to defend against them. We need database textbooks not only to demonstrate parameterized queries, but to explain to students why it is important to use them. If we wish to secure the web in the future, both instructors and textbook authors need to emphasize the importance of protecting against this attack.

Given how easy and widely known the SQL injection attack is, it is at this point a fact of life that any code put on the internet will have SQL injection attempts against it. Students need to know that this is not an academic or abstract possibility, but something that will happen to their code. Most of the texts we looked at showed secure examples, but did not mention SQL injection or explain why using parameters was important. Without this knowledge, students who later see vulnerable code examples will not know that they show a coding style that will introduce dangerous vulnerabilities into their code.

It may be very tempting to view things like web programming or SQL injection as tangential to the core concepts of databases, and perhaps out of scope for textbooks. However, is very likely that in any databases course, some students will go on to write internet-facing code. Given the lack of dedicated web programming courses, this may be the only place students are exposed to creating SQL queries before they write applications that are widely used and publicly available via the web. Students who are not taught how to protect their queries from SQL injection in their databases course may end up learning about it only after their applications are hacked.

In our investigation, we saw that the same two textbooks are used by over half of the top fifty computer science programs in the US. A single textbook may be read by tens of thousands of students who eventually work in the tech industry. How that textbook deals with security issues may have a profound effect on the future of application security.

8 ACKNOWLEDGEMENTS

Thanks to Stephen Checkoway for many useful conversations, and to the anonymous reviewers for their helpful comments.

REFERENCES

[1] Beaulieu, A. *Learning SQL: Master SQL Fundamentals, 2nd Edition.* O'Reilly Media, 2009.

[2] Börstler, J., Nordström, M., and Paterson, J. H. On the quality of examples in introductory Java textbooks. *ACM Transactions on Computing Education (TOCE) 11*, 1 (2011), 3.

[3] Chen, L., Tao, L., Li, X., and Lin, C. A tool for teaching web application security. In *Proceedings of the 14th Colloquium for Information Systems Security Education* (2010), pp. 17–24.

[4] Chu, B., Stranathan, W., Cody, J., Peterson, J., Wenner, A., and Yu, H. Teaching secure software development with vulnerability assessment. In *Proceedings of the 13 Colloquium for Information Systems Security Education (CISSE 2009). Seattle, Washington* (2009).

[5] Conklin, M., and Heinrichs, L. In search of the right database text. *Journal of Computing Sciences in Colleges 21*, 2 (2005), 305–312.

[6] Connolly, T., and Begg, C. *Database Systems: A Practical Approach to Design, Implementation, and Management, 6th Edition.* Pearson, 2014.

[7] Cox, J. *The History of SQL Injection, the Hack That Will Never Go Away*, 2015. https://motherboard.vice.com/en_us/article/aekzez/the-history-of-sql-injection-the-hack-that-will-never-go-away, Accessed on 07/14/2017.

[8] Du, W., and Wang, R. Seed: A suite of instructional laboratories for computer security education. *Journal on Educational Resources in Computing (JERIC) 8*, 1 (2008), 3.

[9] Elmasri, R., and Navathe, S. B. *Fundamentals of Database Systems, 7th Edition.* Pearson, 2015.

[10] Ernits, M., Tammekänd, J., and Maennel, O. i-tee: A fully automated cyber defense competition for students. In *ACM SIGCOMM Computer Communication Review* (2015), vol. 45, ACM, pp. 113–114.

[11] Freeman, D. J., and Porter, A. C. Do textbooks dictate the content of mathematics instruction in elementary schools? *American educational research journal 26*, 3 (1989), 403–421.

[12] Garcia-Molina, H., Ullman, J. D., and Widom, J. *Database Systems: The Complete Book, 2nd Edition.* Pearson, 2008.

[13] Greenberg, A. *Hack Brief: As FBI Warns Election Sites Got Hacked, All Eyes Are on Russia*, 2016. https://www.wired.com/2016/08/hack-brief-fbi-warns-election-sites-got-hacked-eyes-russia//, Accessed on 08/10/2017.

[14] Grossman, P., and Thompson, C. Learning from curriculum materials: Scaffolds for new teachers? *Teaching and teacher education 24*, 8 (2008), 2014–2026.

[15] Guimaraes, M., Murray, M., and Austin, R. Incorporating database security courseware into a database security class. In *Proceedings of the 4th annual conference on Information security curriculum development* (2007), ACM, p. 5.

[16] Halfond, W. G., Viegas, J., and Orso, A. A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering* (2006), vol. 1, IEEE, pp. 13–15.

[17] Irvine, C. E., Chin, S.-K., and Frincke, D. Integrating security into the curriculum. *IEEE Computer 31*, 12 (Dec. 1998), 25–30. Online at http://cisr.nps.edu/downloads/papers/98paper_integrate.pdf.

[18] Li, L., Qian, K., Chen, Q., Hasan, R., and Shao, G. Developing hands-on labware for emerging database security. In *Proceedings of the 17th Annual Conference on Information Technology Education* (2016), ACM, pp. 60–64.

[19] Meghanathan, N., Kim, H., and Moore, L. A. Incorporation of aspects of systems security and software security in senior capstone projects. In *Proceedings of ITNG 2012* (Apr. 2012), S. Latifi, Ed., IEEE Computer Society, pp. 319–324. Online at http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6209193.

[20] Mullins, P., Wolfe, J., Fry, M., Wynters, E., Calhoun, W., Montante, R., and Oblitey, W. Panel on integrating security concepts into existing computer courses. In *Proceedings of SIGCSE 2002* (Feb. 2002), S. Grissom and D. Knox, Eds., ACM Press, pp. 365–66. Online at http://dl.acm.org/citation.cfm?id=563480.

[21] Null, L. Integrating security across the computer science curriculum. *Journal of Computing Sciences in Colleges 19*, 5 (May 2004), 170–178. Online at http://dl.acm.org/citation.cfm?id=1060104.

[22] OWASP. *Reviewing Code for SQL Injection*, 2010. https://www.owasp.org/index.php/Reviewing_Code_for_SQL_Injection, Accessed on 11/17/2016.

[23] OWASP. *OWASP Top Ten Project*, 2017. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project#tab=OWASP_Top_10_for_2017_Release_Candidate, Accessed on 7/14/2017.

[24] Ramakrishnan, R., and Gehrke, J. *Database Management Systems, 3rd Edition.* McGraw-Hill, 2002.

[25] Said, H. E., Guimaraes, M. A., Maamar, Z., and Jololian, L. Database and database application security. In *ACM SIGCSE Bulletin* (2009), vol. 41, ACM, pp. 90–93.

[26] Schweitzer, D., and Boleng, J. Designing web labs for teaching security concepts. *Journal of Computing Sciences in Colleges 25*, 2 (2009), 39–45.

[27] Shegokar, A. M., and Manjaramkar, A. K. A survey on SQL injection attack, detection and prevention techniques. *Int. J. Comput. Sci. Inf. Technol 5*, 2 (2014), 2553–2555.

[28] Silberschatz, A., Korth, H., and Sudarshan, S. *Database System Concepts, 6th Edition.* McGraw-Hill, 2010.

[29] Simsion, G., and Witt, G. *Data Modeling Essentials, 3rd Edition.* Morgan Kaufmann, 2004.

[30] Stalvey, R. H., Farkas, C., and Eastman, C. First use: introducing information security in high school oracle academy courses. In *Information Reuse and Integration (IRI), 2012 IEEE 13th International Conference on* (2012), IEEE, pp. 653–658.

[31] Stanger, M., and Martin, E. *The 50 Best Computer-Science and Engineering Schools in America*, 2015. http://www.businessinsider.com/best-computer-science-engineering-schools-in-america-2015-7, Accessed on 10/05/2016.

[32] Trainer, K. *Between 35,000 and 40,000 credit cards exposed to hackers after coding errors led to SQL Injection*, 2016. https://www.foregenix.com/blog/credit-cards-exposed-to-hackers-poor-coding-sql-injection.

[33] White, G., and Nordstrom, G. Security across the curriculum: Using computer security to teach computer science principles. In *Proceeding of NISSC 1996* (Oct. 1996), pp. 483–88. Online at http://csrc.nist.gov/nissc/1996/papers/NISSC96/paper003/sec_cur.pdf.

[34] Yang, L. Teaching database security and auditing. In *ACM SIGCSE Bulletin* (2009), vol. 41, ACM, pp. 241–245.

[35] Yang, T. A. Computer security and impact on computer science education. *Journal of Computing Sciences in Colleges 16*, 4 (2001), 233–246. Online at http://dl.acm.org/citation.cfm?id=378722.

[36] Yuan, X., Hernandez, J., Waddell, I., Chu, B., and Yu, H. Hands-on laboratory exercises for teaching software security. In *Proceedings of the 16th Colloquium for Information Systems Security Education* (2012).

[37] Yuan, X., Williams, I., Kim, T. H., Xu, J., Yu, H., and Kim, J. H. Evaluating hands-on labs for teaching sql injection: a comparative study. *Journal of Computing Sciences in Colleges 32*, 4 (2017), 33–39.

[38] Zetter, K. *Answers to Your Burning Questions on the Ashley Madison Hack*, 2015. https://www.wired.com/2015/08/ashley-madison-hack-everything-you-need-to-know-your-questions-explained/, Accessed on 08/10/2017.

[39] Zetter, K. *Hacker Lexicon: SQL Injections, an Everyday Hacker's Favorite Attack*, 2016. https://www.wired.com/2016/05/hacker-lexicon-sql-injections-everyday-hackers-favorite-attack/, Accessed on 01/17/2017.