Automatic Whitelist Generation for SQL Queries Using Web Application Tests

Komei Nomura
Pepabo R&D Institute,
GMO Pepabo, Inc.
Email: komei.nomura@pepabo.com

Kenji Rikitake
Pepabo R&D Institute,
GMO Pepabo, Inc. / KRPEO
Email: kenji.rikitake@acm.org

Ryosuke Matsumoto
SAKURA Research Center,
SAKURA Internet Inc.
Email: r-matsumoto@sakura.ad.jp

***Abstract*—Stealing confidential information from a database has become a severe vulnerability issue for web applications. The attacks can be prevented by defining a whitelist of SQL queries issued by web applications and detecting queries not in list. For large-scale web applications, automated generation of the whitelist is conducted because manually defining numerous query patterns is impractical for developers. Conventional methods for automated generation are unable to detect attacks immediately because of the long time required for collecting legitimate queries. Moreover, they require application-specific implementations that reduce the versatility of the methods. As described herein, we propose a method to generate a whitelist automatically using queries issued during web application tests. Our proposed method uses the queries generated during application tests. It is independent of specific applications, which yields improved timeliness against attacks and versatility for multiple applications.**

1. Introduction

Attacks exploiting web application vulnerability are occurring continually. Confidential information of services such as personal information is leaked because of the attacks [1], [2]. The attacks are caused by executing illegal queries to the database, which are not unexpected by application developers. To prevent this outcome, illegal queries must be detected before they are executed in the database.

Applying detection methods used for network attack detection [3], [4] is a common practice for detecting illegal database queries. Two categories of methods are common for detecting illegal queries: using a blacklist and a whitelist [5]. In a blacklist method, developers define the known illegal patterns in the blacklist. The matched queries are detected. The blacklist method is useful in other web applications as common illegal patterns because the blacklist defines specific strings of SQL used for attacks. However, defining illegal patterns requires broad and detailed knowledge about the attacks. A lack of developer's knowledge causes incompleteness in the definition of the list. Even if all the illegal patterns were definable, unexpected illegal queries that are not defined in the blacklist would not be detectable [6].

When using a whitelist method, the whitelist defines the queries issued by the web application. Queries not on the whitelist are detected. The whitelist method can detect a completely unexpected illegal query that is issued. However, the developers must define the whitelist for each web application because the queries issued mutually differ. We have chosen to pursue the whitelist method because detecting illegal queries to the greatest extent possible is fundamentally important to satisfy the goal of our research, which is to protect confidential information in a web application database.

A related work [7] proposes a non-automatic generation of a whitelist of queries issued by a web application, conducted by the developers. This non-automatic generation method, however, imposes an impractical burden on developers. The number of queries issued by the large-scale and complex web application is enormous. Developers must grasp all queries issued by the web application. In addition, as the queries issued by web application change with updating of the web application, the developer must update the whitelist according to those changes. Another related work [8], [9] proposes a method of generating the whitelist of queries issued by the web application automatically to reduce the burden on the developer. We note two shortcomings associated with these methods: (1) the illegal queries cannot be detected using a whitelist immediately after the start-up of the web application because generation of the whitelist requires queries generated by the user input; and (2) each web application requires a separate method because of differences of implementation.

This study examines an automatic whitelist generation method using queries issued during the development testing phase, where developers maintain the test code following changes of the target web application. The method reduces developer burdens for implementing countermeasures against illegal queries. Furthermore, the proposed method solve the shortcommings of conventional methods. The proposed method can detect illegal queries from the start-up of the web application by incorporating a generated whitelist at the testing phase. A database proxy placed in front of a database collects the queries necessary to generate the whitelist. The collected queries are converted into the query structures replacing the literals inside the queries with the placeholders. The structures are registered in the

whitelist. Collecting queries with the database proxy enable whitelist generation independent of the web application implementation. Because the whitelist is generated by queries during testing phase, the queries detected using the proposed method are queries that were not issued during testing. The detected queries include illegal queries.

The remainder of the paper is organized as explained below. First, we summarize issues related to generating a whitelist of queries issued by the web application in section 2. We describe the development process with the whitelist. The detection characteristics and the architecture of the proposed method are discussed in section 3. In section 4, we evaluate the detection accuracy of the whitelist generated using the proposed method. Then we discuss the effectiveness of the proposed method in section 5. Section 6 concludes the paper.

2. Issues of whitelist generation

A method exists for manual creation of a whitelist of queries issued by a web application, which is conducted by the developer [7]. A query structure that replaces literals of the query with placeholders is registered in the whitelist because they are issued by a web application change depending on the user's input [10]. Therefore, the developer specifies all processes of issuing a query from the web application source code and registers the query structure in the whitelist.

Generating a whitelist without using an automatic method imposes a burden on the developer. Because web applications become larger and more complicated as development progress, the number of queries issued by a web application increases. Consequently, the number of queries that the developer must grasp becomes enormous, making whitelist generation difficult. To maintain detection accuracy of the whitelist, a developer must update the whitelist according to changes of queries issued by the web application. A developer frequently updates the whitelist because queries issued by a web application is changed by updating the web application frequently [11].

Furthermore, when object-relational mapping (ORM) [12] is used to implement the web application, the developer is less aware of the queries issued by the application. Actually, ORM provides a function to associate records in a database and objects in object-oriented languages. Furthermore, ORM allows the developer to handle records in the database as objects. Developers using ORM rarely write an SQL statement in this case. For example, when using Ruby on Rails [13] as the web application framework for implementing a web application, ActiveRecord [14] is used as the ORM. A record in the database is associated with a Ruby class object by ActiveRecord. The developer can describe the process of issuing an SQL query with Ruby code. Table 1 shows correspondence between Ruby code and SQL query when the class object `User` is associated with the records of the table `users` in the database. When using ORM for implementation of the web application, it is difficult for the developer to understand the issued queries because they can extract data from the database using an object. Generating a whitelist without an automated method is unsuitable for development of the web application using ORM because understanding the SQL queries increases the developer workload. These facts demonstrate that generating whitelist without a burden on the developer is required.

TABLE 1. EXAMPLE OF SQL QUERY ISSUED BY RUBY CODE

Ruby code	`User.find(1)`
SQL query	`SELECT * FROM users WHERE (users.id = 1) LIMIT 1`

2.1. Generating a whitelist using issued queries

A method that generates a whitelist by collecting queries issued by the web application when it is running has been proposed [8], [15]. The collected queries are converted into the query structure and are registered in the whitelist. That method can generate the whitelist automatically when the web application is running. This method has a learning phase for collecting queries and generating the whitelist. It also has a detection phase for detection using the whitelist. Illegal queries cannot be detected during the learning phase because the whitelist has not been generated yet. Therefore, the method cannot immediately detect illegal queries after the web application is running.

For a web application having a high update frequency, a learning phase is frequently required, so that a period during which detection cannot be performed each time occurs. Generating a whitelist should be done before running the web application because generating a whitelist after running it causes a period during which an illegal query cannot be detected. That method can create a whitelist from query logs of web applications collected in the past. However, generating a whitelist by query logs cannot update the whitelist correctly according to changes of queries when the web application is updated because the query log includes queries that came to be newly issued and queries that ceased to be issued after the web application was updated.

2.2. Generating a whitelist using static analysis

A method that generates a whitelist by analyzing the process of query issuance in the web application source code has been proposed [9]. This method can detect illegal queries immediately after the web application is running because it generates a whitelist before it runs. Because the analysis process of the source code depends on the language or web application framework used for the implementation, it is necessary to implement it for each application. Various implementation languages such as PHP, Ruby, and web application frameworks are used to develop web applications. Consequently, some web applications are developed with different implementation. In this case, a method that depends on the web application implementation is unsuitable because it requires implementation for each. Some method that is independent of the web application implementation is required to resolve this issue.

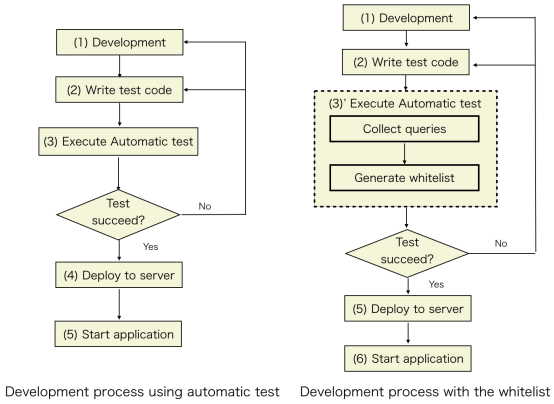

Figure 1. Development process using automatic testing and development process with whitelist generation.

3. Proposed method

For this study, we propose a method for the automatic generation of a whitelist using web application testing to reduce the whitelist generation burden on developers. We assume a development process by which the operation test of the web application is managed as test code and by which the developer prepares the test code according to a change of the web application. In this development process, the proposed method collects queries issued during testing and generates a whitelist from them.

3.1. Development process with the whitelist

The proposed method incorporates a process of whitelist generation at the testing phase of the development process using the automatic test. To show the whitelist generation flow using the proposed method, we describe development process using automatic test and development process with whitelist generation using Figure1.

We explain the development process using the automatic test portrayed in Figure 1. In (1), the developer develops new functions or modifies the existing functions of a web application. In (2), the developer writes test cases, which are operation procedures used when testing web applications and an expected operation result occurs in the test code. In (3), the developer executes all tests using the test code and checks whether the web application operates as specified. If the tests fail, then the behavior of the developed function does not operate as specified or the test code is incorrect. In this case, the developer identifies the cause of the failure of the tests and modifies the source code of the web application or the test code. If the tests succeed, then the developers assume that the developed function operates as specified. In this case, the source code of the new web application is placed on the server in (4) and the web application is started in (5).

We explain the development process with the whitelist of Figure 1. The process of generating whitelist is incorporated into the execution phase of the automatic test as shown in (3)'. The whitelist is generated using queries issued from the web application during automatic testing. The source code of the new web application and the whitelist is placed on the server after successful testing.

Because the queries issued by the web application that are changed by functions are added or modified, it is necessary to generate a whitelist correspondingly. In the development process using the automatic test, the queries issued during testing change because the test code is prepared according to the change of the web application. The proposed method maintains the consistency of the whitelist and the queries issued by the web application using the queries issued during testing. Illegal queries can be detected immediately after the web application start-up because the whitelist is generated entirely at the testing phase. This characteristic cannot be realized using the method of starting the generation of the whitelist during running of the web application. Our proposed method requires no addition of new development tasks. The effect on the development process when introducing the proposed method is small.

Let us describe the differences in the burdens of writing the test code and manually creating the whitelist by developers. The developer writes test cases that assume the behavior of the web application in the test code. The developer must understand the behavior of the web application to derive the test cases. The developer registers the query structures of the query issued by the web application in the whitelist. Consequently, the developer must understand the processes of issuing the queries in detail. Developers must also understand the behavior of ORM when ORM is used for web application development. Therefore, it is unlikely that the developer understands details of the process issuing a query in this case. Writing the test code is likely to demand less than creating the whitelist manually.

3.2. Detection characteristics

In this section, we describe the detection characteristics of the whitelist generated using the proposed method. False positive and false negative are used as indicators of detection. False positive occurs when a query issued by a web application receiving a user input assumed by a developer is determined as illegal. False negative occurs when a query issued by a web application vulnerability attacks cannot be determined as illegal. The proposed method generates a whitelist using the query issued during testing. Therefore, the whitelist detects queries that were not issued during testing. The detected queries include untested non-illegal queries and illegal queries. Non-illegal queries that have not been tested are queries issued by user input while the web application is running. However, because of a lack of test cases, they are queries that were not issued at the time of testing. These queries cause false positives. Illegal queries are those issued by attacking web application vulnerabilities. The whitelist might also contain illegal queries issued only during the testing phase. These queries cause false negatives. Examples of such queries include queries for registering test

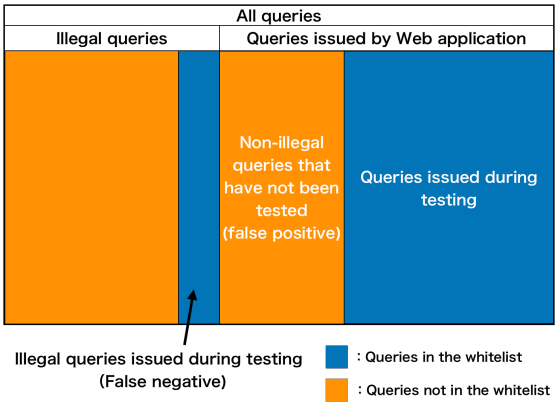

Figure 2. Relationship of queries.

TABLE 2. EXAMPLE OF QUERY THAT IS NOT DETECTED USING THE PROPOSED METHOD

Legitimate query	`SELECT * FROM users LIMIT 30`
Abnormal query (anomaly)	`SELECT * FROM users LIMIT 1000`

data in a database and queries for deleting test data. Figure 2 shows the relationships of the queries above.

Adding test cases and improving test coverage expand the area of queries issued during testing in Figure 2. Therefore, false positives are reduced. However, some concern arises that management will be complicated by increasing test cases. A method of registering non-illegal queries that have not been tested in whitelist without adding test cases must be considered.

The query structure in which literals of the query are replaced with placeholders is registered in the whitelist. For example, "`SELECT * FROM users LIMIT 10`" replaces the literal "`10`" with the placeholder "?" and registers it in the whitelist as "`SELECT * FROM users LIMIT ?`". The proposed method detects queries with the query structures that are not in the whitelist. Therefore, the attacks that inject strings containing SQL statements into literals and which change the query structure such as SQL injection attacks [16] can be detected using the proposed method. However, because the proposed method does not set thresholds of numerical literals, queries that have illegal numerical literal cannot be detected. Table 2 presents an example of queries that are not detected using the proposed method.

The query presented in Table 2 has the same query structure, but the numerical literal differs. Addressing this case requires consideration of a method of detecting an anomaly of the literal of the query.

3.3. Architecture of the proposed method

In the proposed method, a database proxy that is placed in front of the database collects queries issued during testing and detects illegal queries while the web application is running. Whitelist generation flow during testing and detection

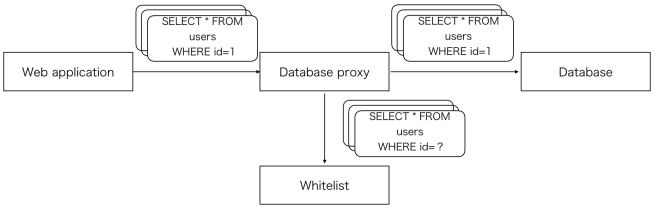

Figure 3. Architecture of the proposed method.

during running of the web application are explained using Figure 3.

First, a test is executed. Queries are issued from the web application to the database proxy. The database proxy passes the received query to the database and records the relayed query during this time. After all tests are completed, the database proxy replaces the literals of the recorded queries with the placeholder and registers them in the whitelist. Collecting queries using the database proxy realized whitelist generation independent of the web application implementation.

We also explain detection during running of the web application using Figure 3. First, the web application receives input from the user and issues a query. Next, a database proxy receives a query issued from the web application, converts the received query into the query structure, and checks whether it is on the whitelist. If the query is on the whitelist, then it passes the query to the database and executes it. If the query is not on the whitelist, then the developer is notified of the detected query.

Detection of an illegal query has the process to refer to the whitelist. If the referral process of the whitelist is slow, then the concern arises that the query stagnates at the database proxy and that the response time to the user increases. Therefore, the computational complexity of the referral process of the whitelist is fundamentally important. When searching all query structures in the whitelist to ascertain whether a query is illegal, or not, the computational complexity is $O(n)$ for the number of entries n in the whitelist. To avoid this, the whitelist is created in the proposed method, as a hash table for which the query structure is a key. The computational complexity of the referral process of the whitelist is $O(1)$ using the hash table. Consequently, the proposed method can refer to the whitelist with constant computational complexity irrespective of the number of entries in the whitelist.

4. Experiment

To verify the effectiveness of the proposed method, we evaluated the detection accuracy of the whitelist created using the proposed method. We described in section 3.2 that the whitelist generated by the proposed method might have false positives and false negatives. In this section, we present the experiment results and the evaluation. The false positive means that a query issued by a web application receiving a user input assumed by a developer is determined

TABLE 3. IMPLEMENTED METHODS

HTTP	URL	operation
GET	articles	Show all articles
GET	articles?title=""	Show articles searched by title
POST	articles	Create articles
GET	articles/:id	Show articles
PATCH	articles/:id	Update articles
DELETE	articles/:id	Delete articles

as illegal. The false negative means that a query issued by web application vulnerability attacks cannot be determined as illegal.

We constructed the experimental environment and confirmed false positives and false negatives by experimentation. We used ProxySQL [17] as a database proxy. Actually, ProxySQL has a function to collect a received query and convert it into a query structure and save it. We used this function to generate a whitelist. We created the title and content column in the article table using MySQL supported by ProxySQL. We implemented methods for the web application to control the article table using Ruby on Rails. Then we described the tests for all methods. Table 3 presents the implemented methods. We implemented SQL injection vulnerability into the GET method of the operation "Show articles searched by title" in Table 3.

A whitelist with 23 entries was generated using the query structures recorded in ProxySQL after running the web application tests. We obtained 17 legitimate queries. The legitimate queries were generated manually from the browser by sending HTTP requests to the web application implementing the method shown in Table 3. A false positive rate is derived by counting queries that are not on the whitelist in the legitimate queries. We obtained 265 illegal queries. Illegal queries were generated by sending HTTP requests that have SQL injection attack using sqlmap [18] for the method that shows articles searched by title. The false negative rate is derived by counting queries that are on the whitelist in the illegal queries. We measured the test coverage of web application using SimpleCov [19] because the queries registered in the whitelist change depend on it. SimpleCov counts the lines of the source code executed during testing and calculates the test coverage.

As a result of the experiment when test coverage is 68.97%, the false positive rate was 17.65%; the false negative rate was 0%. Results show that the proposed method causes false positives, but it does not cause any false negative.

Let us describe false positives in the experiment. False positives occur although the tests are described for all methods of web applications. Queries detected as false positives were 3, which is 17.65% of the legitimate queries. Figure 4 shows the queries detected as false positive.

The web application implemented by Ruby on Rails does not issue queries shown in Figure 4 because the article data before the update on the database were the same as the article data generated after executing the query of UPDATE. The method of issuing UPDATE query is included

- `UPDATE 'articles' SET 'content' = ?, 'updated_at' = ? WHERE 'articles'.'id' = ?`
- `UPDATE 'articles' SET 'title' = ?, 'updated_at' = ? WHERE 'articles'.'id' = ?`
- `UPDATE 'articles' SET 'title' = ?, 'content' = ?, 'updated_at' = ? WHERE 'articles'.'id' = ?`

Figure 4. Queries detected as false positive.

- `ROLLBACK`
- `SELECT COUNT(*) FROM 'articles'`
- `RELEASE SAVEPOINT active_record_1`
- `SAVEPOINT active_record_1`
- `SET FOREIGN_KEY_CHECKS = ?`
- `SELECT 'articles'.* FROM 'articles' ORDER BY 'articles'.'id' DESC LIMIT ?`
- `DELETE FROM 'articles'; INSERT INTO 'articles' ('id', 'title', 'content', 'created_at', 'updated_at') VALUES (?, ?, ?, ?, ?), (?, ?, ?, ?, ?), (?, ?, ?, ?, ?);`
- `SELECT @@FOREIGN_KEY_CHECKS`
- `SELECT @@max_allowed_packet`

Figure 5. Illegal queries issued during testing.

in the calculation of test coverage because it is executed during testing. The difference of behavior by Ruby on Rails suggests that some queries are not issued during testing depending on the test case, even if the test coverage is 100%. Complementing queries that were not issued during testing because of a lack of test cases is necessary to reduce the false positives.

Let us describe false negatives in the experiment. No false negative occurred in the experiment. Nevertheless, the possibility of false negative exists because the whitelist includes illegal queries issued only during testing. We investigated the rate of illegal queries contained in the whitelist and its query structure. The rate of illegal queries contained in the whitelist was 39.13%. Figure 5 shows only the query structures issued during testing.

The proposed method cannot detect queries with the query structure shown in Figure 5. The SQL injection attack injects arbitrary SQL string into the query issued by the web application. It changes the execution result of the query. A query issued by a SQL injection attack is a query issued by a web application plus specific SQL strings used for the attack. The possibility that these queries match the query structures shown in the Figure 5 cannot be denied. However, the possibility of such a case occurring is regarded as low because false negatives did not occur in the experiment.

5. Discussion

The proposed method can use an automatic test to generate a whitelist during the development process. The developer can obtain a whitelist in the development process without changing it. The queries issued during testing include queries issued by changes of the web application because the

test code is prepared according to their changes. Therefore, the whitelist can be updated as the web application updates. The proposed method can detect illegal queries immediately after a web application is running because it creates a whitelist at the testing phase before the web application is running. These facts show that the proposed method can take countermeasures against illegal queries while reducing the burden on developers by whitelist creation.

The proposed method can create a whitelist independent of the implementation of the web application because the database proxy collects queries and creates it. Therefore, the proposed method is suitable for creating a whitelist for some web applications with different implementations.

We confirmed from the experiment described in section 4 that the proposed method can detect illegal queries issued by an SQL injection attack. We confirmed that false positives occurred because the queries issued by methods might change depending on test cases, even if tests are described for all methods. Therefore, when the proposed method is applied to a larger-scale web application, the number of false positive queries is regarded as increasing because of the increase of the number of issued queries. To resolve this difficulty, one must consider a method of supplementing queries that can not be registered in the whitelist among the queries issued by the web application. Additionally, the whitelist includes queries issued only during the test. Consequently, the proposed method is unable to detect cases in which queries with the same query structure are issued as attacks. To prevent this eventuality, it is necessary to remove queries issued only during the test from the whitelist.

6. Conclusion and future work

As described in this paper, we proposed a method for automatically generating a whitelist using queries issued during testing to reduce the burden on developer given by whitelist creation, while particularly addressing the testing of the development process. The proposed method can detect illegal queries immediately from the start-up of the web application because the whitelist was generated at the testing phase of the development process. The generation was automated. It effectively reduced the burden on the developers. Furthermore, the proposed method did not depend on implementation of the web application because the database proxy collected the queries necessary for whitelist generation irrespective of the web application. Results show that the proposed method is suitable for creating a whitelist for a set of web applications with multiple implementations.

Our proposed method can be used effectively as a countermeasure against SQL injection attacks. We confirmed the following observations.

- The proposed method can detect illegal queries issued by SQL injection attack without false negatives.
- False positives can occur even if the tests described all possible methods of the web application because the query issued by the method of the web application changes depending on the test case details.

Our future work shall include the following:

- to reduce false positives, supplementing queries not registered in the whitelist that were actually issued by web applications;
- removal of the illegal queries issued only during the testing phase from the whitelist, thereby eliminating the possibility of false negatives;
- verifying detection accuracy in large-scale web applications; and
- measuring the overhead of illegal query detection to the response time of the database from queries of the web applications.

References

[1] IT Security Center, Information-technology promotion agency (IPA), *How to Secure your Website Fifth edition*, Apr 2011.

[2] J. Rahul and S. Pankaj, "A survey on web application vulnerabilities (sqlia, xss) exploitation and security engine for sql injection," in *2012 International Conference on Communication Systems and Network Technologies*, May 2012, pp. 453–458.

[3] S. Axelsson, "Intrusion detection systems: A survey and taxonomy," in *Technical Report 99-15*, Mar 2000.

[4] F. S. Rietta, "Application layer intrusion detection for sql injection," in *Proceedings of the 44th annual Southeast regional conference*, Mar 2006, pp. 531–536.

[5] V. Luong, "Intrusion detection and prevention system: Sql-injection attacks," in *Master's Projects. 16*, 2010.

[6] M. Ofer and S. Amichai, *SQL injection signatures evasion*. Imperva, Inc. White paper, Apr 2004.

[7] K. Kemalis and T. Tzouramanis, "Sql-ids: a specification-based approach for sql-injection detection," in *Proceedings of the 2008 ACM symposium on Applied computing*, 2008, pp. 2153–2158.

[8] F. Valeur, D. Mutz, and G. Vigna, "A learning-based approach to the detection of sql attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2005, pp. 123–140.

[9] W. G. Halfond and A. Orso, "Amnesia: analysis and monitoring for neutralizing sql-injection attacks," in *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*, Nov 2005, pp. 174–183.

[10] D. Kar and S. Panigrahi, "Prevention of sql injection attack using query transformation and hashing," in *Advance Computing Conference (IACC), 2013 IEEE Third International*, May 2013, pp. 1317–1323.

[11] J. Mehdi, "Some trends in web application development," in *2007 Future of Software Engineering*, 2007, pp. 199–213.

[12] S. W. Ambler, *Mapping objects to relational databases*, 2000.

[13] M. Bächle and P. Kirchberg, "Ruby on rails," *IEEE software*, vol. 24, no. 6, pp. 105–108, 2007.

[14] M. Fowler, *Patterns of enterprise application architecture*, 2002.

[15] F. José, V. Marco, and M. Henrique, "Detecting malicious sql," in *International Conference on Trust, Privacy and Security in Digital Business*, 2007, pp. 259–268.

[16] W. G. Halfond and A. Orso, "A classification of sql injection attacks and countermeasures," in *Proceedings of the IEEE International Symposium on Secure Software Engineering, Vol.1*, 2006, pp. 13–15.

[17] "Proxysql," http://www.proxysql.com/.

[18] "sqlmap," http://sqlmap.org/.

[19] "Simplecov," https://github.com/colszowka/simplecov.