Proceedings of CONF-SPML 2027 Symposium: Intelligent Network Security and Machine Learning for Communication Systems
DOI: 10.54254/2755-2721/2027.MELB37531

Design of an Automated SQL Injection Detection Tool for Web Applications

Shuyan Hong

School of Computer Science and Technology, Huaqiao University, Xiamen, China
2793949741@qq.com

Abstract. World Wide Web (Web) applications have become an important interface for network services. However, the direct connection among user input, business logic, and database operations has also increased security risks. SQL injection vulnerabilities remain an important issue affecting Web application security. To address the low efficiency and repetitive nature of traditional manual detection, this paper designs an automated SQL injection vulnerability detection tool for Web applications. By analyzing the principles of SQL injection and common attack types, a detection process consisting of parameter extraction, payload construction, request transmission, response analysis, and vulnerability confirmation is established. The tool is tested in experimental environments involving MySQL, a Content Management System (CMS), and Damn Vulnerable Web Application (DVWA). Experimental results show that the tool can identify different types of SQL injection vulnerabilities based on characteristics such as page content, error messages, response length, and response time. It can improve vulnerability detection efficiency and the consistency of result interpretation, providing a useful reference for Web application security testing.

Keywords: SQL Injection, Web Application, Vulnerability Detection, Dynamic Testing

1. Introduction

World Wide Web (Web) applications have become an important interface for network services. However, the direct connection among user input, business logic, and database operations has also increased security risks. Structured Query Language (SQL) injection is a typical input validation vulnerability in which external parameters may enter dynamically constructed SQL statements and thereby alter the original query logic. Existing studies have investigated SQL injection detection from perspectives such as static analysis, runtime monitoring, heuristic testing, and behavioral analysis, providing a foundation for automated vulnerability discovery [1-4].

However, practical testing still faces challenges such as parameter discovery, test data selection, interpretation of response results, and blind injection detection when no obvious data is returned. This paper focuses on automated detection of SQL injection vulnerabilities in Web applications. By combining literature analysis with controlled experiments using MySQL and Damn Vulnerable Web Application (DVWA), a detection process involving parameter extraction, payload generation, request transmission, response analysis, and vulnerability confirmation is designed. Existing

© 2026 The Authors. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

DOI: 10.54254/2755-2721/2027.MELB37531

experimental data are also used to analyze the detection characteristics of UNION query injection, error-based injection, Boolean-based blind injection, and time-based blind injection. The results can provide a reference for the implementation of lightweight SQL injection detection tools.

2. SQL injection vulnerabilities and principles of automated detection

2.1. Principles of SQL injection vulnerabilities

In general, SQL injection occurs when an application directly concatenates Web request parameters supplied by users into SQL query statements for execution. When a user sends parameters such as id to the server through a Uniform Resource Locator (URL), the server receives these parameters and constructs a database query. If effective restrictions are not applied to the input parameters, user-supplied characters or keywords such as single quotation marks, and, or select may modify the original SQL statement.

Experimental results show that GET parameters are generally located in the query string of a URL, whereas POST parameters are located in the request body. An automated tool must first identify the locations of the parameters and then perform injection tests and compare the results. For GET requests, parameter names and values can be parsed directly from the URL. For POST requests, form fields must be read or HyperText Transfer Protocol (HTTP) requests must be captured to extract the parameters.

GET parameters are normally contained in the URL query string and can be obtained directly. POST parameters are located in the request body and therefore require form-field identification or HTTP request capture for extraction. The baseline detection process is the same for both types of requests, namely parameter discovery, submission of test data, and analysis of response results. For requests that require authentication, information such as Cookies must also be preserved so that test requests remain consistent with normal requests.

2.2. Types of SQL injection

Based on the materials used in the experiments, this paper mainly discusses UNION query injection, error-based injection, Boolean-based blind injection, and time-based blind injection.

UNION query injection relies on visible page output. In general, order by is first used to determine the number of columns, after which positions capable of displaying data are identified. In the CMS experiment, the query result was found to contain 15 columns, and the third and eleventh columns were identified as visible output positions.

Error-based injection uses database error information as feedback. During testing, abnormal parameters are submitted to determine whether error messages are returned, and the returned information is then analyzed to identify the database and database structure. Because the amount of information that can be displayed in an error message is limited, group_concat was used in the experiment to combine information, while substr was used to extract portions of data when the output exceeded the permitted length.

When there is no obvious data output on the page, blind injection can be used. Boolean-based blind injection determines results by comparing differences in page content or page length caused by true and false conditions. Functions such as length and substr can be used to infer the database name character by character. Time-based blind injection determines results based on differences in response time when a condition is satisfied. In the experimental records, a delay of approximately 3 seconds was used as an indicator of time-based blind injection.

DOI: 10.54254/2755-2721/2027.MELB37531

2.3. Principles of automated detection

Automated detection requires converting manual observation into repeatable rules. Existing studies have shown that error information, application output, and request behavior can all serve as evidence for dynamic detection [5, 6]. Detection evidence can mainly be divided into content-based evidence, error-based evidence, and time-based evidence.

Content-based evidence is mainly used for UNION query injection and Boolean-based blind injection. It includes changes in page length, changes in result areas, and stable differences generated under true and false conditions.

Error-based evidence is mainly used for error-based injection. Database error messages can be analyzed to determine whether a parameter has entered an SQL execution path.

Time-based evidence is mainly used for time-based blind injection. The response time of a normal request is compared with that of a test request, and the time difference is used as the basis for judgment.

3. Design of the automated detection tool

3.1. Overall design logic

The overall process of the tool is "parameter parsing – payload construction – request transmission – response analysis – vulnerability confirmation."

First, the Web page or HTTP request to be tested is provided as input. The parameter parsing module identifies GET and POST parameters and creates test tasks. Next, test payloads are generated according to parameter locations and detection types, after which requests are sent. The response analysis module extracts features such as status codes, page content, response length, error messages, and response time. Finally, the tool determines whether an SQL injection risk exists.

The tool adopts a modular design. Different types of payloads are managed according to predefined rules, and detection results are stored in a unified format consisting of the parameter, test type, response characteristics, and judgment basis. To reduce false positives, a baseline response from a normal request is obtained before testing and compared with the test response. If a significant difference is found, a verification request is sent. In this way, the detection process is divided into two stages, discovery and confirmation, preventing a single abnormal response from directly becoming the final conclusion.

3.2. Functional design

3.2.1. Automatic parameter extraction

Automatic parameter extraction serves as the entry point of the tool. For GET requests, the URL query string can be parsed directly. For POST requests, form fields in the request body are read.

Existing experiments involve URL parameters such as cid, listid, and newsid in news.php, as well as POST form parameters. Therefore, the tool must distinguish between request methods and store the corresponding parameter locations. After extraction, test tasks are created to provide input for subsequent payload generation and request transmission.

DOI: 10.54254/2755-2721/2027.MELB37531

3.2.2. Payload construction

The payload construction module selects test cases according to the detection type. UNION query injection is used to verify the number of columns and visible output positions. Error-based testing is used to observe database errors. Boolean-based blind injection constructs true and false conditions, while time-based blind injection constructs observable response delays.

Different databases vary in syntax and error information. Therefore, payloads should be classified and managed according to both the detection type and database characteristics, rather than assuming that a single set of experimental data can cover all situations.

3.2.3. Response analysis

Response analysis is the core of the tool. For UNION query injection, the main focus is whether new data appears on the page. For error-based injection, database error messages are analyzed. For Boolean-based blind injection, page content and response length under true and false conditions are compared. For time-based blind injection, response time is recorded.

The analysis results should be compared with the baseline response of a normal request rather than being determined solely by a single HTTP status code.

3.2.4. Vulnerability confirmation

The vulnerability confirmation module first determines whether a parameter presents an SQL injection risk rather than directly retrieving large amounts of database information. After a vulnerability is confirmed, structural information such as the database type and database name may be recorded for result verification.

In existing experiments, UNION query injection and error-based injection were able to gradually confirm the database structure, while blind injection relied on differences in content or response time for confirmation.

4. Experiments using DVWA and MySQL

4.1. Experimental environment

The experiments were conducted based on existing MySQL manual injection experiments, CMS injection experiments, and Web penetration testing reports. UNION query injection, error-based injection, Boolean-based blind injection, and time-based blind injection were tested in a MySQL vulnerable test environment, a CMS article management system, and DVWA. Page output, database errors, response length, and response time were observed.

4.2. Test results and data analysis

UNION query injection and error-based injection were mainly used to verify the content analysis and error analysis modules. In the CMS experiment, order by was used to determine that the query contained 15 columns, and the third and eleventh columns were identified as visible output positions. Another MySQL experiment used UNION queries to confirm information about the database, tables, and fields. This process shows that the tool can use the number of columns, visible output positions, and database errors as intermediate detection evidence.

DOI: 10.54254/2755-2721/2027.MELB37531

Figure 1 shows a section of an error-based injection test extracted from the experimental report. The test request entered the page through a URL parameter, and the server returned database error information. Such information can be used as evidence of abnormal SQL execution, but it cannot independently prove all conditions of SQL injection. Therefore, the automated tool should combine this information with normal requests and other test results for confirmation.

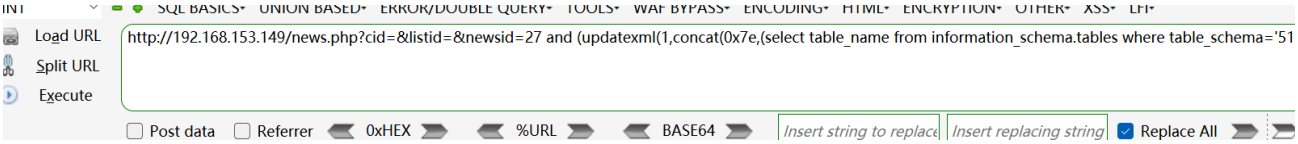

Figure 1. Error response in the MySQL experimental environment

Blind injection can be divided into Boolean-based blind injection and time-based blind injection. Boolean-based blind injection analyzes differences in Web pages produced under true and false conditions, while time-based blind injection determines whether a character or condition is correct based on response duration.

Figure 2 shows response information for character guessing from one blind injection experiment. In this experiment, the candidate characters were q, r, s, and t. Their HTTP response status codes were all 200, but the response lengths were different. This demonstrates that status codes alone cannot be used to accurately identify blind SQL injection.

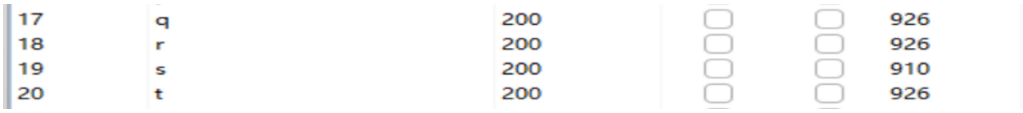

Figure 2. Response length records in blind injection testing

4.3. Comparative analysis

Testing can be conducted from three perspectives: GET requests, POST requests, and different security levels. GET testing is mainly used to verify URL parameter extraction. POST testing is mainly used to verify request-body parameters, Cookie persistence, and request construction. Testing under different security levels is used to observe whether input filtering affects the effectiveness of payloads.

These three types of tests correspond respectively to parameter processing, session maintenance, and the adaptability of detection rules.

The four detection methods discussed above UNION query injection provides higher detection efficiency when visible output is available. Error-based injection depends on error information. Boolean-based blind injection depends on differences in page content. Time-based blind injection does not require direct output but is more strongly affected by network latency (see Table 1).

Table 1. Comparison of SQL injection detection methods

Detection Method	Observable Evidence	Value for Automated Detection	Main Limitation
UNION Query Injection	Changes in page content or results	Can provide direct confirmation	Requires visible output

DOI: 10.54254/2755-2721/2027.MELB37531

Table 1. (continued)

Error-based Injection	Database error information	Error evidence is relatively clear	Depends on error-message output
Boolean-based Blind Injection	Database error information	Error evidence is relatively clear	Depends on error-message output
Boolean-based Blind Injection	Abnormal page content or response length	Can detect vulnerabilities without visible data output	Requires a relatively large number of requests
Time-based Blind Injection	Response-time delay	Does not require direct data output	Easily affected by network latency

5. Conclusion

This paper focuses on the design of an automated SQL injection vulnerability detection tool for Web applications and establishes a complete detection process from parameter extraction to vulnerability confirmation. Experimental results show that UNION query injection, error-based injection, Boolean-based blind injection, and time-based blind injection exhibit different response characteristics, which can be used as evidence for automated judgment. The study also shows that parameter discovery, response comparison, and secondary verification play important roles in reducing false positives. However, this study is still mainly based on controlled experimental environments. Quantitative testing on large-scale Web applications has not yet been conducted, and false positive and false negative rates have not been systematically analyzed. Future work can further improve automatic parameter discovery, payload selection, response normalization, and experimental datasets. Additional testing involving different databases and input-filtering rules can also be conducted to further evaluate the stability and applicability of the tool.

References

[1] Halfond, W. G. J., Viegas, J., & Orso, A. (2006). A classification of SQL-injection attacks and countermeasures. *Proceedings of the International Symposium on Secure Software Engineering*.

[2] Halfond, W. G. J., & Orso, A. (2005). AMNESIA: Analysis and monitoring for NEutralizing SQL-injection attacks. *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 174-183. DOI: 10.1145/1101908.1101935.

[3] Jovanovic, N., Krügel, C., & Kirda, E. (2006). Pixy: A static analysis tool for detecting web application vulnerabilities. *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 258-263. DOI: 10.1109/SP.2006.29.

[4] Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D. T., & Kuo, S.-Y. (2004). Securing web application code by static analysis and runtime protection. *Proceedings of the 13th International Conference on World Wide Web*, 40-52. DOI: 10.1145/988672.988679.

[5] Ciampa, A., Visaggio, C. A., & Di Penta, M. (2010). A heuristic-based approach for detecting SQL-injection vulnerabilities in web applications. *Proceedings of the ICSE Workshop on Software Engineering for Secure Systems*. DOI: 10.1145/1809100.1809107.

[6] Xiao, Z., Zhou, Z., Yang, W., & Deng, C. (2017). An approach for SQL injection detection based on behavior and response analysis. *Proceedings of the 2017 IEEE 9th International Conference on Communication Software and Networks*, 1437-1442. DOI: 10.1109/ICCSN.2017.8230346.