Preventing SQL Injection Attacks Using AMNESIA

William G.J. Halfond and Alessandro Orso
College of Computing
Georgia Institute of Technology
{whalfond|orso}@cc.gatech.edu

ABSTRACT

AMNESIA is a tool that detects and prevents SQL injection attacks by combining static analysis and runtime monitoring. Empirical evaluation has shown that AMNESIA is both effective and efficient against SQL injection.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging—*Monitors*;

General Terms: Security, Verification

Keywords: SQL injection, static analysis, runtime monitoring

1. INTRODUCTION

Companies and organizations use Web applications to provide a broad range of services to users, such as on-line banking and shopping. Because the databases underlying Web applications often contain confidential information (e.g., customer and financial records), these applications are a frequent target for attacks. One particular type of attack, SQL injection, can give attackers a way to gain access to the databases underlying Web applications and, with that, the power to leak, modify, or even delete information that is stored on these databases. In recent years, both commercial and government institutions have been victims of SQLIAs.

SQL injection vulnerabilities are due to insufficient input validation. More precisely, *SQL Injection Attack*s (*SQLIA*s) can occur when a Web application receives user input and uses it to build a database query without adequately validating it. An attacker can take advantage of a vulnerable application by providing it with input that contains embedded malicious SQL commands that are then executed by the database. Although the vulnerabilities that lead to SQLIAs are well understood, they continue to be a significant problem because of a lack of effective techniques to detect and prevent them. Conceptually, SQLIAs could be prevented by a more rigorous application of defensive coding techniques [10]. In practice, however, these techniques have been less than effective in addressing the problem because they are susceptible to human errors and expensive to apply on large legacy code-bases.

In our demonstration, we present AMNESIA (Analysis and Monitoring for NEutralizing SQL-Injection Attacks), a tool that implements our technique for detecting and preventing SQLIAs [7, 8]. AMNESIA uses a model-based approach that is specifically designed to target SQLIAs and combines static analysis and runtime monitoring. It uses static analysis to analyze the Web-application code and automatically build a model of the legitimate queries that the application can generate. At runtime, the technique monitors all dynamically-generated queries and checks them for compliance with the statically-generated model. When the technique detects a query that violates the model, it classifies the query as an attack, prevents it from accessing the database, and logs the attack information.

ICSE'06, May 20–28, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

2. EXAMPLE OF SQL INJECTION

To illustrate how an SQLIA occurs, we introduce a simple example that we will use throughout the paper. The example is based on a servlet, `show.jsp`, for which a possible implementation is shown in Figure 1.

```
public class Show extends HttpServlet {
    ...
 1. public ResultSet getUserInfo(String login, String password) {
 2.   Connection conn = DriverManager.getConnection("MyDB");
 3.   Statement stmt = conn.createStatement();
 4.   String queryString = "";

 5.   queryString = "SELECT info FROM userTable WHERE ";
 6.   if ((! login.equals("")) && (! password.equals(""))) {
 7.     queryString += "login='" + login +
                     "' AND pass='" + password + "'";
 8.   } else {
 9.     queryString+="login='guest'";
      }
10.   ResultSet tempSet = stmt.execute(queryString);
11.   return tempSet;
    }
    ...
}
```

Figure 1: Example servlet.

Method `getUserInfo` is called with a login and password provided by the user, in string format, through a Web form. If both `login` and `password` are empty, the method submits the following query to the database:

```
SELECT info FROM users WHERE login='guest'
```

Conversely, if the user submits `login` and `password`, the method embeds the submitted credentials in the query. For instance, if a user submits `login` and `password` as "`doe`" and "`xyz`," the servlet dynamically builds the query:

```
SELECT info FROM users WHERE login='doe' AND pass='xyz'
```

A Web application that uses this servlet would be vulnerable to SQLIAs. For example, if a user enters "`' OR 1=1 --`" and "", instead of "`doe`" and "`xyz`", the resulting query is:

```
SELECT info FROM users WHERE login='' OR 1=1 --' AND pass=''
```

The database interprets everything after the WHERE token as a conditional statement, and the inclusion of the "`OR 1=1`" clause turns this conditional into a tautology. (The characters "`--`" mark the beginning of a comment, so everything after them is ignored.) As a result, the database would re-

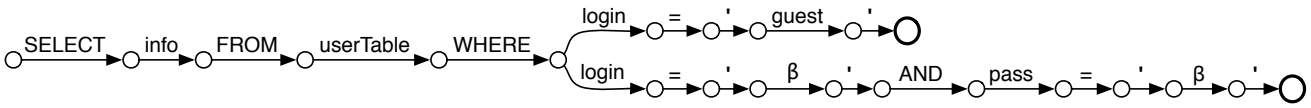


Figure 2: SQL-query model for the servlet in Figure 1.

turn information about all users. Introducing a tautology is only one of the many possible ways to perform SQLIAs, and variations can have a wide range of effects, including modification and destruction of database tables. We provide a thorough survey of SQLIAs in [9].

3. THE AMNESIA TOOL

In this section we summarize our technique, implemented in the AMNESIA tool, and then discuss the main characteristics of the tool implementation. A detailed description of the approach is provided in [7].

3.1 Underlying Technique

Our technique uses a combination of static analysis and runtime monitoring to detect and prevent SQLIAs. It consists of four main steps.

Identify hotspots: Scan the application code to identify *hotspots*—points in the application code that issue SQL queries to the underlying database.

Build SQL-query models: For each hotspot, build a model that represents all of the possible SQL queries that may be generated at that hotspot. An *SQL-query model* is a non-deterministic finite-state automaton in which the transition labels consist of SQL tokens, delimiters, and placeholders for string values.

Instrument Application: At each hotspot in the application, add calls to the runtime monitor.

Runtime monitoring: At runtime, check the dynamically-generated queries against the SQL-query model and reject and report queries that violate the model.

3.1.1 Identify Hotspots

This step performs a simple scanning of the application code to identify hotspots. For the example servlet in Figure 1, the set of hotspots would contain a single element, the statement at line 10.

3.1.2 Build SQL-Query Models

To build the SQL-query model for each hotspot, we first compute all of the possible values for the hotspot's query string. To do this, we leverage the Java String Analysis (JSA) library developed by Christensen, Møller, and Schwartzbach [3]. The JSA library produces a non-deterministic finite automaton (NDFA) that expresses, at the character level, all the possible values the considered string can assume. The string analysis is conservative, so the NDFA for a string is an overestimate of all the possible values of the string.

To produce the final SQL-query model, we perform an analysis of the NDFA and transform it into a model in which all of the transitions represent semantically meaningful tokens in the SQL language. This operation creates an NDFA in which all of the transitions are annotated with SQL keywords, operators, or literal values. (This step is configurable to recognize different dialects of SQL.) In our model, we mark transitions that correspond to externally defined strings with the symbol β.

To illustrate, Figure 2 shows the SQL-query model for the hotspot in the example provided in Section 2. The model reflects the two different query strings that can be generated by the code depending on the branch followed after the `if` statement at line 6 (Figure 1). In the model, β marks the position of the user-supplied inputs in the query string.

3.1.3 Instrument Application

In this step, we instrument the application code with calls to a monitor that checks the queries at runtime. For each hotspot, we insert a call to the monitor before the call to the database. The monitor is invoked with two parameters: the query string that is about to be submitted and a unique identifier for the hotspot. The monitor uses the identifier to retrieve the SQL-query model for that hotspot.

Figure 3 shows how the example application would be instrumented by our technique. The hotspot, originally at line 10 in Figure 1, is now guarded by a call to the monitor at line 10a.

```
...
10a. if (monitor.accepts (<hotspot ID>, queryString))
     {
10b.    ResultSet tempSet = stmt.execute(queryString);
11.     return tempSet;
     }
...
```

Figure 3: Example hotspot after instrumentation.

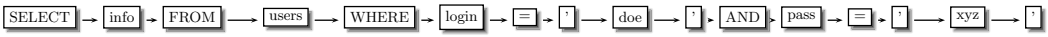
3.1.4 Runtime Monitoring

At runtime, the application executes normally until it reaches a hotspot. At this point, the query string is sent to the runtime monitor. The monitor parses the query string into a sequence of tokens according to the specific SQL dialect considered. Figure 4 shows how the last two queries discussed in Section 2 would be parsed during runtime monitoring.

After parsing the query, the runtime monitor checks whether the query violates the hotspot's SQL-query model. To do this, the runtime monitor checks whether the model accepts the sequence of tokens in the query string. When matching the query string against the SQL-query model, a token that corresponds to a numeric or string constant (including the empty string, ε) can match either an identical literal value or a β label. If the model does not accept the sequence of tokens, the monitor identifies the query as an SQLIA.

To illustrate runtime monitoring, consider again the queries from Section 2, shown in Figure 4. The tokens in query (a) specify a set of transitions that terminate in an accepting state. Therefore, query (a) is executed on the database. Conversely, query (b) contains extra tokens that prevent it from reaching an accepting state and is recognized as an SQLIA.

(a) SELECT info FROM users WHERE login='doe' AND pass='xyz'



(b) SELECT info FROM users WHERE login='' OR 1=1 -- 'AND pass=''

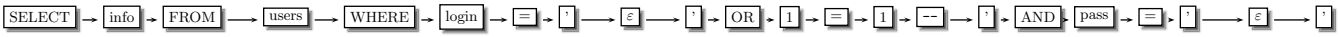


Figure 4: Example of parsed runtime queries.

3.2 Implementation

In our demonstration, we show an implementation of our technique, AMNESIA, that works for Java-based Web applications. The technique is fully automated, requiring only the Web application as input, and requires no extra runtime environment support beyond deploying the application with the AMNESIA library. We developed the tool in Java and its implementation consists of three modules:

Analysis module. This module implements Steps 1 and 2 of our technique. It inputs a Java Web application and outputs a list of hotspots and a SQL-query model for each hotspot. For the implementation of this module, we leveraged the Java String Analysis library [3]. The analysis module is able to analyze Java Servlets and JSP pages.

Instrumentation module. This module implements Step 3 of our technique. It inputs a Java Web application and a list of hotspots and instruments each hotspot with a call to the runtime monitor. We implemented this module using InsECTJ, a generic instrumentation and monitoring framework for Java [19].

Runtime-monitoring module. This module implements Step 4 of our technique. The module takes as input a query string and the ID of the hotspot that generated the query, retrieves the SQL-query model for that hotspot, and checks the query against the model.

Figure 5 shows a high-level overview of AMNESIA. In the static phase, the Instrumentation Module and the Analysis Module take as input a Web application and produce (1) an instrumented version of the application and (2) an SQL-query model for each hotspot in the application. In the dynamic phase, the Runtime-Monitoring Module checks the dynamic queries while users interact with the Web application. If a query is identified as an attack, it is blocked and reported.

To report an attack, AMNESIA throws an exception and encodes information about the attack in the exception. If developers want to access the information at runtime, they can leverage the exception-handling mechanism of the language and integrate their handling code into the application. Having this attack information available at runtime allows developers to react to an attack right after it is detected and develop an appropriate customized response. Currently, the information reported by AMNESIA includes the time of the attack, the location of the hotspot that was exploited, the attempted-attack query, and the part of the query that was not matched against the model.

3.3 Assumptions and Limitations

Our tool makes one primary assumption regarding the applications it targets—that queries are created by manipulating strings in the application. In other words, AMNESIA assumes that the developer creates queries by combining hard-coded strings and variables using operations such as concatenation, appending, and insertion. Although this assumption precludes the use of AMNESIA on some applications (e.g., applications that externalize all query-related strings in files), it is not an overly restrictive assumption. Moreover, it is an implementation-related assumption that can be eliminated with suitable engineering.

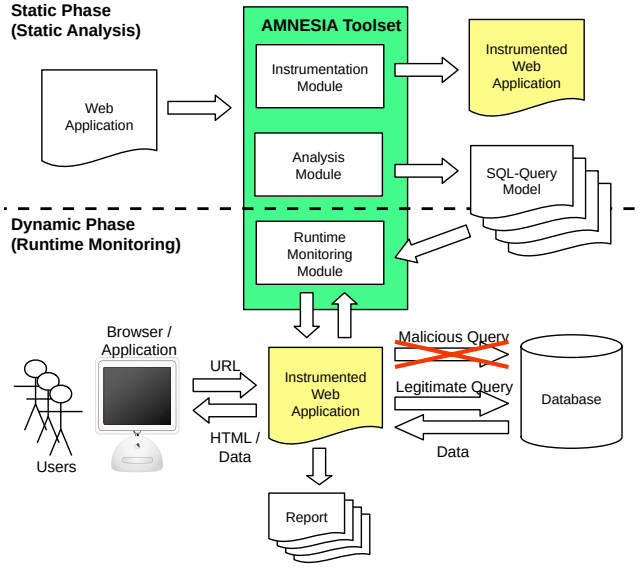


Figure 5: High-level overview of AMNESIA.

In certain situations our technique can generate false positives and false negatives. False positives can occur when the string analysis is not precise enough. For example, if the analysis cannot determine that a hard-coded string in the application is a keyword, it could assume that it is an input-related value and erroneously place a β in the SQL query model. At runtime, the original keyword would not match the placeholder for the variable, and AMNESIA would flag the corresponding query as an SQLIA. False negatives can occur when the constructed SQL query model contains spurious queries and the attacker is able to generate an injection attack that matches one of the spurious queries.

To assess the practical implications of these limitations, we conducted an extensive empirical evaluation of our technique. The evaluation used AMNESIA to protect seven applications while the applications where subjected to thousands of attacks and legal accesses. AMNESIA's performance in the evaluation was excellent: it did not generate any false positives or negatives [7].

4. RELATED WORK

To address the problem of SQLIAs, researchers have proposed a wide range of techniques. Two recent techniques [2, 20] use an approach similar to ours, in that they also build models of legitimate queries and enforce conformance with the models at runtime. Other techniques include intrusion detection [21], black-box testing [11], static code checkers [5, 12, 13, 22], Web proxy filters [18], new query-development paradigms [4, 15], instruction set randomization [1], and taint-based approaches [6, 14, 16, 17].

While effective, these approaches have limitations that affect their ability to provide general detection and prevention capabilities against SQLIAs [9]. Furthermore, some of these approaches are difficult to deploy. Static analysis techniques, such as [5, 22], address only a subset of the problem. Other solutions require developers to learn and use new APIs [4, 15], modify their application source code [2, 20], deploy their applications using customized runtime environments [1, 15, 16, 18], or accept limitations on the completeness and precision of the technique [11, 21]. Techniques based solely on static analysis, such as [12, 13], do not achieve the same levels of precision as dynamic techniques. Finally, defensive coding [10], while offering an effective solution to SQLIAs, has shown to be difficult to apply effectively in practice.

5. SUMMARY

In this paper, we have presented AMNESIA, a fully automated tool for protecting Web applications against SQLIAs. Our tool uses static analysis to build a model of the legitimate queries an application can generate and monitors the application at runtime to ensure that all generated queries match the statically-generated model. In [7], we have presented an extensive evaluation that uses commercial applications and real-world SQLIAs to evaluate the effectiveness of AMNESIA. The results of this evaluation show that AMNESIA can be very effective and efficient in detecting and preventing SQLIAs.

Acknowledgments

This work was partially supported by DHS contract FA8750-05-2-0214 and NSF awards CCR-0205422 and CCR-0209322 to Georgia Tech.

6. REFERENCES

[1] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In *Proc. of the 2nd Applied Cryptography and Network Security Conf. (ACNS 2004)*, pages 292–302, Jun. 2004.

[2] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using Parse Tree Validation to Prevent SQL Injection Attacks. In *Proc. of the 5th Intern. Workshop on Software Engineering and Middleware (SEM 2005)*, pages 106–113, Sep. 2005.

[3] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th Intern. Static Analysis Symposium (SAS 2003)*, pages 1–18, Jun. 2003.

[4] W. R. Cook and S. Rai. Safe Query Objects: Statically Typed Objects as Remotely Executable Queries. In *Proc. of the 27th Intern. Conf. on Software Engineering (ICSE 2005)*, pages 97–106, May 2005.

[5] C. Gould, Z. Su, and P. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In *Proc. of the 26th Intern. Conf. on Software Engineering (ICSE 2004)*, pages 645–654, May 2004.

[6] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. In *Proc. of the 21st Annual Computer Security Applications Conf. (ACSAC 2005)*, pages 303–311, Dec. 2005.

[7] W. G. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proc. of the IEEE and ACM Intern. Conf. on Automated Software Engineering (ASE 2005)*, pages 174–183, Nov. 2005.

[8] W. G. Halfond and A. Orso. Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks. In *Proc. of the Third Intern. ICSE Workshop on Dynamic Analysis (WODA 2005)*, pages 22–28, May 2005.

[9] W. G. Halfond, J. Viegas, and A. Orso. A Classification of SQL-Injection Attacks and Countermeasures. In *Proc. of the Intern. Symposium on Secure Software Engineering (ISSSE 2006)*, Mar. 2006.

[10] M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, Redmond, Washington, 2^{nd} edition, 2003.

[11] Y. Huang, S. Huang, T. Lin, and C. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *Proc. of the 12th Intern. World Wide Web Conf. (WWW 2003)*, pages 148–159, May 2003.

[12] Y. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S. Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proc. of the 13th Intern. World Wide Web Conf. (WWW 2004)*, pages 40–52, May 2004.

[13] V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Usenix Security Symposium*, Aug. 2005.

[14] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proc. of the 20th annual ACM SIGPLAN Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA 2005)*, pages 365–383, Oct. 2005.

[15] R. McClure and I. Krüger. SQL DOM: Compile Time Checking of Dynamic SQL Statements. In *Proc. of the 27th Intern. Conf. on Software Engineering (ICSE 2005)*, pages 88–96, May 2005.

[16] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting Information. In *Twentieth IFIP Intern. Information Security Conf. (SEC 2005)*, May 2005.

[17] T. Pietraszek and C. V. Berghe. Defending Against Injection Attacks through Context-Sensitive String Evaluation. In *Proc. of Recent Advances in Intrusion Detection (RAID 2005)*, Sep. 2005.

[18] D. Scott and R. Sharp. Abstracting Application-level Web Security. In *Proc. of the 11^{th} Intern. Conf. on the World Wide Web (WWW 2002)*, pages 396–407, May 2002.

[19] A. Seesing and A. Orso. InsECTJ: A Generic Instrumentation Framework for Collecting Dynamic Information within Eclipse. In *Proc. of the eclipse Technology eXchange (eTX) Workshop at OOPSLA 2005*, pages 49–53, Oct. 2005.

[20] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *The 33rd Annual Symposium on Principles of Programming Languages (POPL 2006)*, pages 372–382, Jan. 2006.

[21] F. Valeur, D. Mutz, and G. Vigna. A Learning-Based Approach to the Detection of SQL Attacks. In *Proc. of the Conf. on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA 2005)*, Jul. 2005.

[22] G. Wassermann and Z. Su. An Analysis Framework for Security in Web Applications. In *Proc. of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2004)*, pages 70–78, Oct. 2004.