AFRL-RI-RS-TR-2008-144
Final Technical Report
May 2008

PREVENTING SQL CODE INJECTION BY COMBINING STATIC AND RUNTIME ANALYSIS

Georgia Institute of Technology

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2008-144 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/ /s/

FRANCES A. ROSE
Work Unit Manager

WARREN H. DEBANEY, Jr.
Technical Advisor, Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MAY 08	Final	May 05 – Oct 07

4. TITLE AND SUBTITLE

PREVENTING SQL CODE INJECTION BY COMBINING STATIC AND RUNTIME ANALYSIS

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA8750-05-2-0214

5c. PROGRAM ELEMENT NUMBER
N/A

6. AUTHOR(S)

Alessandro Orso, Wenke Lee and Adam Shostack

5d. PROJECT NUMBER
DHSD

5e. TASK NUMBER
GI

5f. WORK UNIT NUMBER
OT

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

PRIME
Georgia Institute of Technology
505 10th St. NW
Atlanta GA 30332

SUB
Reflective, LLC
1444 Fairview Road
Atlanta GA 30332

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RIGA
525 Brooks Rd
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2008-144

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# WPAFB 08-2846

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Many software systems have evolved to include a Web-based component that makes them available to the public via the Internet and can expose them to a variety of Web-based attacks. One of these attacks is SQL injection, which can give attackers unrestricted access to the databases underlying Web applications and has become increasingly frequent and serious. In this project, we developed techniques and tools to detect, prevent, and report SQL injection attacks. Our techniques leverage static and dynamic analysis, are effective and efficient, and have minimal deployment requirements. Given a previously developed Web application, our tools automatically transform the application into an equivalent application that is protected from SQL injection attacks. In the project, we also developed a testbed that can be used to evaluate SQL injection detection and prevention tools. Our testbed has been used extensively both by us and by other organizations. The tools and techniques developed within the project are being disseminated through different channels and are currently being commercialized by our industrial partner.

15. SUBJECT TERMS
SQL Injection, Static and Dynamic Analysis, Vulnerability Discovery and Remediation

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Frances Rose
U	U	U	UU	67	19b. TELEPHONE NUMBER (*Include area code*) N/A

Table of Contents

List of Figures

1 Overview

This document provides a final summary of the work performed by Georgia Tech and Reflective while supported by the Department of Homeland Security and United States Air Force under Contract No. FA8750-05-2-0214. The work described was performed between May 2005 and October 2007.

1.1 Motivation for the Project

Many software systems have evolved to include a Web-based component that makes them available to the public via the Internet and can expose them to a variety of Web-based attacks. One of these attacks is SQL injection, which can give attackers unrestricted access to the databases underlying Web applications and has become increasingly frequent and serious.

In general, *SQL Injection Attack*s (*SQLIA*s) are a class of code injection attacks that take advantage of a lack of validation of user input. These attacks occur when developers combine hard-coded strings with user-provided input to create dynamic queries. Intuitively, if user input is not properly validated, attackers may be able to change the developer's intended SQL command by inserting new SQL keywords or operators through specially crafted input strings. To better illustrate, we introduce an example application that contains a simple SQL injection vulnerability and show how an attacker can leverage that vulnerability to perform an SQLIA.

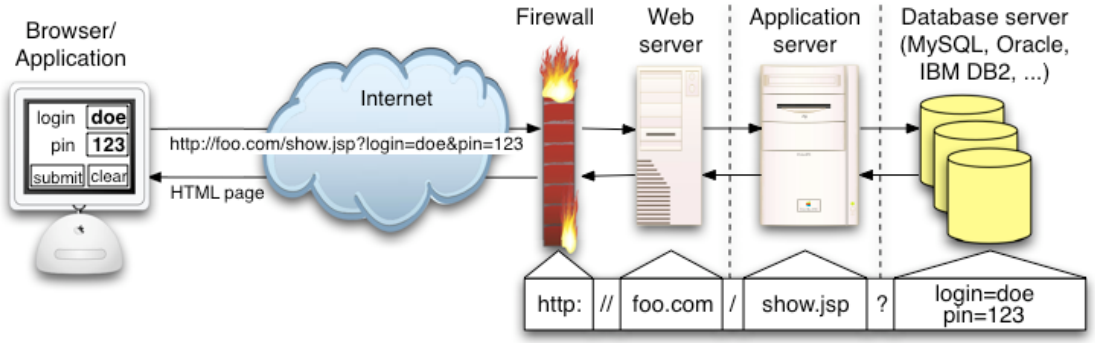


Figure 1: Example of interaction between a user and a typical Web application.

Figure 1 shows an example of a typical Web-application architecture. In the example, the user interacts with a Web form that takes a login name and pin as inputs and submits them to a Web server. The Web server passes the user-supplied credentials to a servlet (show.jsp), which is a special type of Java application that runs on a Web application server, and whose execution is triggered by the submission of a URL from a client.

The example servlet, whose code is partially shown in Figure 2, implements a typical login functionality. It uses input parameters `login` and `pin` to dynamically build an SQL query or command. The `login` and `pin` are checked against the credentials stored in the database. If they match, the corresponding user's account information is returned. Otherwise, a null set is returned by the database, and the authentication fails.

```
1. String login = getParameter("login");
2. String pin = getParameter("pin");
3. Statement stmt = connection.createStatement();
4. String query = "SELECT acct FROM users WHERE login='";
5. query += login + "' AND pin=" + pin;
6. ResultSet result = stmt.executeQuery(query);
7. if (result != null)
8.    displayAccount(result); // Show account
9. else
10.   sendAuthFailed(); // Authentication failed
```

Figure 2: Excerpt of a Java servlet implementation.

The servlet then uses the response from the database to generate HTML pages that are sent back to the user's browser by the Web server. Given the servlet code, if a user submits `login` and `pin` as "`doe`" and "`123`," the application dynamically builds the query:

```
SELECT acct FROM usersWHERElogin='doe'ANDpin=123
```

If `login` and `pin` match the corresponding entry in the database, `doe`'s account information is returned and then displayed by function `displayAccount()`. If there is no match in the database, function `sendAuthFailed()` displays an appropriate error message. An application that uses this servlet is vulnerable to SQLIAs. For example, if an attacker enters "`admin' --` " as the user name and any value as the pin (*e.g.*, "`0`"), the resulting query is:

```
SELECT acct FROM users WHERE login='admin' -- ' AND pin=0
```

In SQL, "`--`" is the comment operator, and everything after it is ignored. Therefore, when performing this query, the database simply searches for an entry where `login` is equal to `admin` and returns that database record. After the "successful" login, function `displayAccount()` reveals the `admin`'s account information to the attacker.

Note that this example represents an extremely simple kind of SQLIA, and we present it for illustrative purposes only. There is a wide variety of complex and sophisticated SQL exploits available to attacks, as we discuss in detail in [6].

1.2 Teaming Information

The project team involves two organizations: Georgia Tech and Reflective LLC. Georgia Tech's team consists of the following members:

- Alessandro Orso – Assistant professor (static/dynamic program analysis, testing).
- Wenke Lee – Associate professor (security).
- William Halfond, James Clause, and Jeremy Viegas – Graduate students.

Reflective's team is composed as follows:

- Adam Shostack – CTO (security analysis, growing customer base).
- Jonathan Amsler – VP for Technology.
- Dave Clauson – CEO.
- Engineering support staff.

1.3 Project Goals

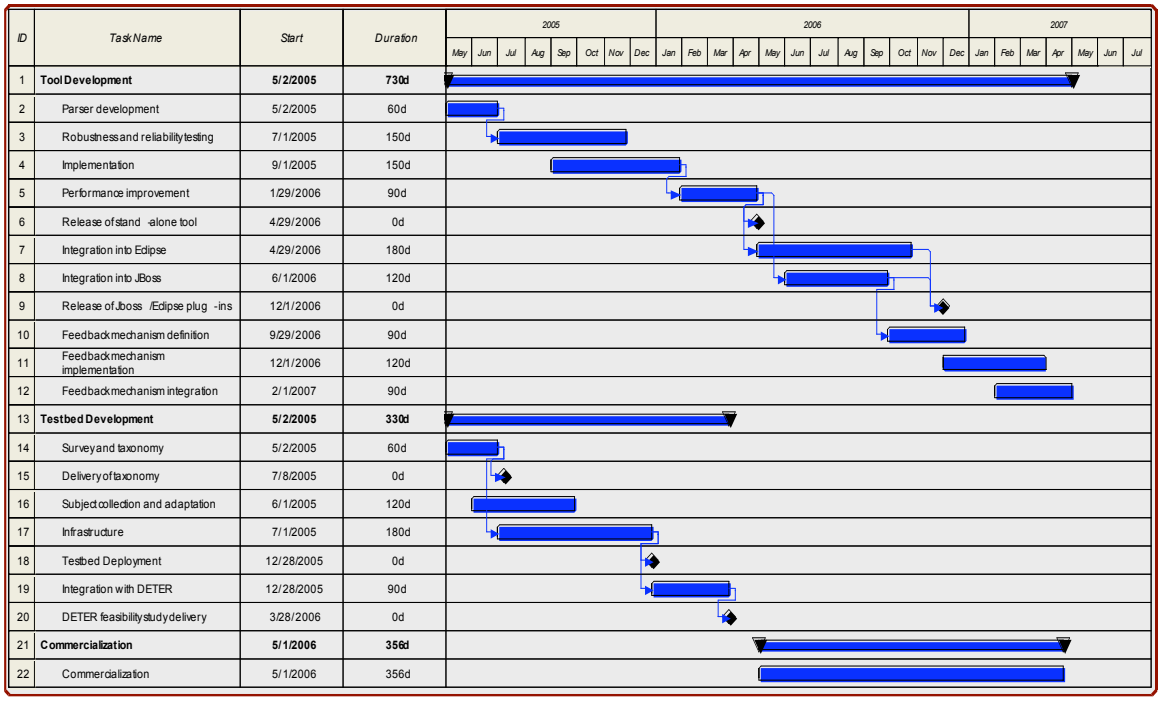
The *first goal* of the project was to develop and implement a highly automated technique against SQLIAs that is able to detect, stop, and report injection attacks before they reach the database and do any harm.

Figure 3: General overview of the proposed approach.

Figure 3 provides a general, intuitive overview of the proposed approach. Given a previously developed Web application, our tool would automatically transform the application into a semantically equivalent application that is protected from SQLIAs. The *second goal* of the project was to develop a testbed that could be used by us and by other researchers and practitioners to evaluate tools for SQL injection detection and prevention. The *third goal* of the project was to make the tools developed during the project industrial strength and to commercialize them.

Each of these three goals corresponds to a main task in the project. Figure 4 shows a Gantt diagram that includes milestones and deliverables for such tasks. (Note that the diagram represents the project as it was originally planned, over 24 month. The project was later on granted a five-month extension to help the commercialization effort.) In the project, Georgia Tech was the main responsible for Tasks/Goals 1 and 2, whereas Reflective was in charge of the commercialization of the tool, with technical support from Georgia Tech for the technology transfer.

In the rest of this document, we first discuss how and to what extent we achieved the goals of the project (Section 2) and then list the publications that were produced during the project (Section 3). These publications are attached at the end of the document to provide further details about the work performed during the project.



ID	Task Name	Start	Duration
1	**Tool Development**	**5/2/2005**	**730d**
2	Parser development	5/2/2005	60d
3	Robustness and reliability testing	7/1/2005	150d
4	Implementation	9/1/2005	150d
5	Performance improvement	1/29/2006	90d
6	Release of stand -alone tool	4/29/2006	0d
7	Integration into Edipse	4/29/2006	180d
8	Integration into JBoss	6/1/2006	120d
9	Release of Jboss /Edipse plug -ins	12/1/2006	0d
10	Feedback mechanism definition	9/29/2006	90d
11	Feedback mechanism implementation	12/1/2006	120d
12	Feedback mechanism integration	2/1/2007	90d
13	**Testbed Development**	**5/2/2005**	**330d**
14	Survey and taxonomy	5/2/2005	60d
15	Delivery of taxonomy	7/8/2005	0d
16	Subject collection and adaptation	6/1/2005	120d
17	Infrastructure	7/1/2005	180d
18	Testbed Deployment	12/28/2005	0d
19	Integration with DETER	12/28/2005	90d
20	DETER feasibility study delivery	3/28/2006	0d
21	**Commercialization**	**5/1/2006**	**356d**
22	Commercialization	5/1/2006	356d

Figure 4: Project milestones and deliverables.

2 Accomplishments

This section discusses the accomplishments for this project grouped by tasks and subtasks.

2.1 Task 1 – Tool Development

The overall goal of this task was to implement a tool to detect and prevent SQLIAs based on a technique, called Amnesia, which we defined in preliminary work for the project. The original plan was to develop the tool in three flavors: a stand-alone tool, an Eclipse plug-in, and a plug-in for the JBoss application server. In the first part of the project, we implemented Amnesia as a stand-alone tool and empirically assessed its performance on a set of real applications and attacks. Although Amnesia performed well in our evaluation, it also showed some limitations. In particular, being based on a conservative static analysis, Amnesia could generate false positives in cases where the analysis was too imprecise. To address this issue, we leveraged what we learned while defining and developing Amnesia and developed a new technique and tool called Wasp. Wasp improves on Amnesia in many ways. In particular, under some assumptions, it is fully automated, generates no false negatives, produces few (and easy to eliminate) false positives, and has minimal deployment requirements. Therefore, we modified our initial plan and, instead of implementing two additional versions of the Amnesia tool, we focused on Wasp and on its implementation. Wasp is therefore the tool that is currently being commercialized by Reflective.

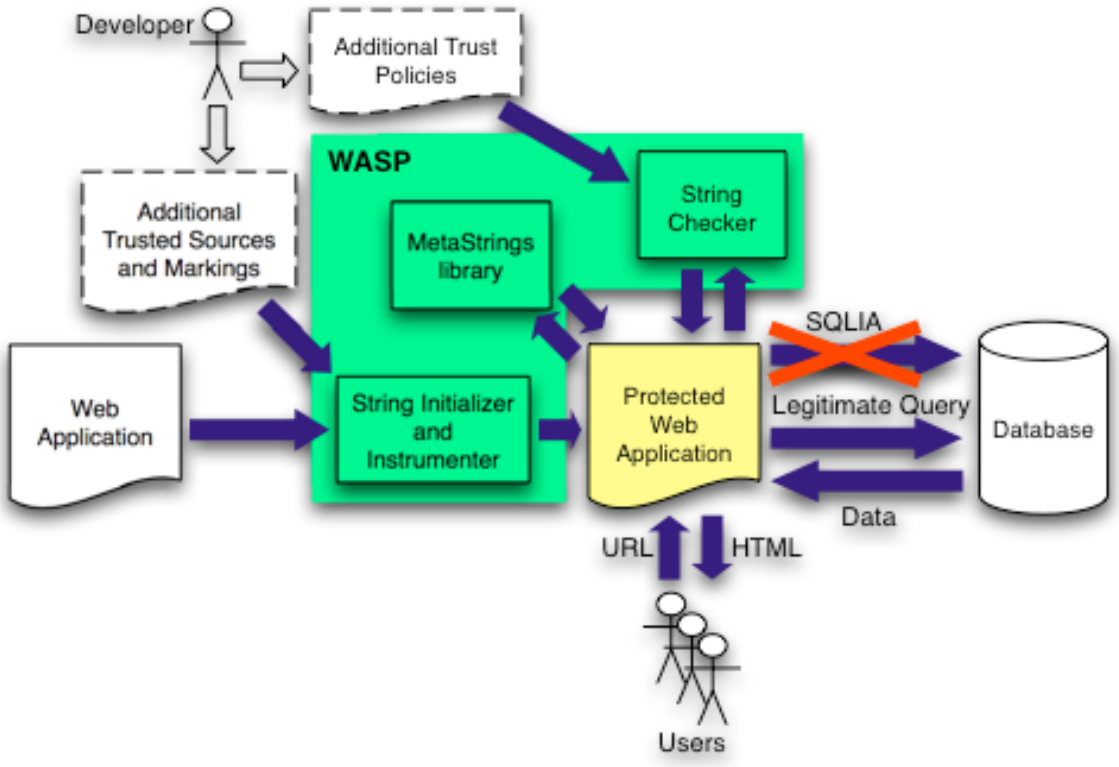


Figure 5: High-level overview of the Wasp approach and tool.

Figure 5 provides a high-level view of Wasp, which is discussed in detail in [1] and [3]. We provide an overview of the approach to make the document self-contained. Intuitively, Wasp works by identifying "trusted" strings in an application and allowing only these trusted strings to be used to create the semantically relevant parts of a SQL query, such as keywords or operators. The general mechanism that we use to implement this approach is based on dynamic tainting, which marks and tracks certain data in a program at runtime. The kind of dynamic tainting that we use gives our approach several important advantages over techniques based on other mechanisms. In particular, compared to other existing techniques based on dynamic tainting, our approach makes several conceptual and practical improvements that take advantage of the specific characteristics of SQLIAs.

The first conceptual advantage of our approach is the use of positive tainting. *Positive tainting* identifies and tracks trusted data, whereas traditional ("negative") tainting focuses on untrusted data. In the context of SQLIAs, there are several reasons why positive tainting is more effective than negative tainting. First, in Web applications, sources of trusted data can be identified more easily and accurately than untrusted data sources. Therefore, the use of positive tainting leads to increased automation. Second, the two approaches significantly differ in how they are affected by incompleteness. With negative tainting, failure to identify the complete set of untrusted data sources can result in false negatives, that is, successful and undetected attacks. With positive tainting, missing trusted data sources could result in false positives (*i.e.*, legitimate accesses can be prevented from completing). Using our approach, however, false positives are likely to be

detected during testing, before release. Our approach provides specific mechanisms for helping developers detect false positives early, identify their sources, and easily eliminate them in future runs by tagging the identified sources as trusted.

The second conceptual advantage of our approach is the use of flexible syntax-aware evaluation. Syntax-aware evaluation lets us address security problems that are derived from mixing data and code while still allowing for this mixing to occur. More precisely, it gives developers a mechanism for regulating the usage of string data based not only on its source, but also on its syntactic role in a query string. This way, developers can use a wide range of external input sources to build queries, while protecting the application from possible attacks introduced via these sources.

The practical advantages of our approach are that it imposes a low overhead on the application and has minimal deployment requirements. Efficiency is achieved by using a specialized library, called MetaStrings, that accurately and efficiently assigns and tracks trust markings at runtime. The only deployment requirements for our approach are that the Web application must be instrumented and it must be deployed with our MetaStrings library, which is done automatically. The approach does not require any customized runtime system or additional infrastructure.

Wasp performed extremely well in our empirical evaluation, which we performed on the testbed developed as part of Task 2. For each application in the testbed, we protected it with WASP, targeted it with a large set of attacks and legitimate accesses, and assessed the ability of Wasp to detect and prevent attacks without stopping legitimate accesses. Wasp was able to stop all of the attacks without generating false positives for any of the legitimate accesses. Moreover, Wasp proved to be efficient, imposing a low overhead on the Web applications.

We conclude the discussion of Task 1 with a summary of its subtasks and how we accomplished them:

- **Parser development**: *completed*. The goal of this subtask was to develop a database parser that would integrate with the tools developed within the project. We developed two different database parsers; one for generic SQL-92 based queries and a second one for a popular variant that is used by PostgreSQL databases. The implementation is modular and has been integrated with both Amnesia and Wasp.
- **Robustness and reliability testing**: *completed*. The goal of this subtask was to evaluate the implementation and development of our tools with respect to their robustness and reliability. We performed this evaluation in-house (through our own testing), in the field (based on feedback from users of the tool), and through Reflective.
- **Tool implementation**: *completed*. The goal of this subtask was to improve the implementation of Amnesia and Wasp to make them ready to be deployed as commercial applications. We developed both tools and then decided to focus mostly on Wasp, as discussed earlier in this section. The tool is currently completed and is being commercialized by Reflective.
- **Performance improvement**: *completed*. The goal of this subtask was to improve the performance of our tools. We completed this subtask by leveraging the testbed

developed as part of Task 2 and assessing the performance of our tools when used on the applications in the testbed. We identified several ways to improve Wasp's performance and implemented them in the tool.

- **Integration into Eclipse and JBoss**: *canceled*. As discussed at the beginning of this section, we redirected the effort allocated for these two tasks to the development of our second tool, Wasp, which was not initially planned.
- **Feedback mechanism definition, implementation, and integration**: *completed*. The goal of these subtasks was to develop a feedback mechanism that allowed system administrator to analyze, in case of SQLIAs, the attacks and their causes. We developed two different mechanisms and integrated them in both Amnesia and Wasp. The first one is a logging mechanism, which stores information about attacks identified by the tools together with contextual information (e.g., the source of a malicious string). The second is a visualization mechanism that allows for visualizing, in graphical and easier to consume fashion, the information logged about the attacks.

2.2 Task 2 – Testbed Development

The overall goal of this task was to develop an evaluation testbed that provides tool developers (and users) with a small network on which they can launch SQLIAs against various applications and measure the success of their detection and prevention technique along with its execution overhead. The testbed should consist of (1) a set of machines connected through a network, (2) a set of applications installed on the machines and potentially vulnerable to SQLIAs, and (3) a set of tools and utilities to automatically perform SQLIAs on the applications. Figure 6 provides a high-level view of the testbed. As shown in the figure, the testbed will let a user select an application and a set of attacks and will automatically perform the attacks and assess their outcome.

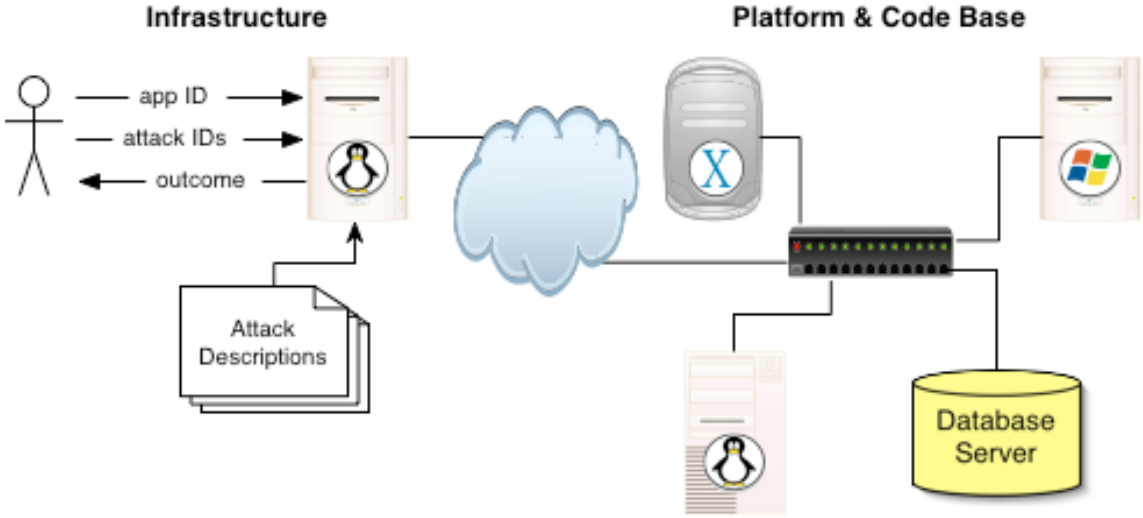


Figure 6: Overview of the testbed.

We developed the testbed and released it. So far, the testbed has been used extensively by us for our evaluation of Amnesia and Wasp. The testbed has also been used externally by other research groups to evaluate their techniques against SQL injection.

As we did for Task 1, we summarize Task 2's subtasks and how we accomplished them:

- **Survey and taxonomy**: *completed*. The goal of this subtask was to (1) provide an overview of current techniques that are being used to attack Web applications using SQL Injection and (2) compare the ability of existing protection techniques to prevent these attacks. We completed the survey, uploaded it to the Jiffy system, and published it [6].
- **Subject collection and adaptation**: *completed*. The goal of this subtask was to identify and collect applications to be used within our testbed. During the life of the project, we have collected fourteen applications that are vulnerable to SQLIAs. These applications are mainly commercial applications, but include also applications from other sources that have been used for evaluation purposes in related work. All of the applications are able to run within our testbed infrastructure and are associated with a large set of representative attacks and legitimate accesses.
- **Infrastructure development**: *completed*. The goal of this subtask was to develop the support infrastructure for the testbed. All elements of the testbed have been developed. These elements include the fourteen vulnerable applications collected as part of the previous subtask, a web application scanner and an attack generator, a testing harness for running and evaluating SQL injection countermeasures, and configuration tools to manage server loads effectively. This infrastructure provides us with the ability to evaluate the effectiveness of the tools developed within the project against real-world applications.
- **Feasibility study for Integration with DETER**: *completed*. The goal of this subtask was to investigate whether our testbed infrastructure could be integrated with DETER and to explore a possible design for such integration. We produced a document that discusses our findings and that was uploaded on the Jiffy system.

2.3 Task 3 – Commercialization

The commercialization of our first tool, Amnesia, started in May 2006. Shortly after that, we decided to switch our focus to our second tool, Wasp. As we discussed above, Wasp improved several aspects of Amnesia and was therefore a better candidate for our commercialization efforts. Due to some IP related issues between Georgia Tech and Reflective, the commercialization of Wasp started late in the lifetime of the project. For this reason, we asked for and obtained a five-month no-cost extension to the contract.

Our technology transition plan is manifold, including the distribution of the tool through the Georgia Tech Research Center (GTRC), through a website in the form of a free download for researchers, and through personal contacts. These channels have led so far to the release of the tool to twelve universities and two research labs. Although these contacts are promising from a dissemination standpoint, they are unlikely to become commercialization venues. Therefore, our main commercialization effort is being performed through Reflective. In the first phase of the commercialization, Reflective evaluated both of the tools developed within the project (Amnesia in Quarter 5 and Wasp in Quarter 6), with positive results. After deciding to focus mostly on Wasp for the remainder of the project, Reflective started to contact organizations and schedule demos of the tool. To support this effort, Reflective and Georgia Tech developed a demo of Wasp that can be run both locally and remotely, through a Web interface.

Reflective is currently discussing possible commercialization opportunities with several organizations, including Internet Security Systems, TKCC, Dept. of Military Health, NSA, SAIC, Inovis, Bank of America, Citicorp, and RSA Security. The first demos of Wasp were shown to these organizations at the end of 2007, and two of the organizations are currently evaluating Wasp in-house for possible adoption. Despite the end of the contract, Georgia Tech and Reflective have an agreement in place and will continue the commercialization of the tool.

3 Publications (refereed)

The work performed within the project led to a number of publications in premier international journals and conferences. We provide a list of these publications and attach them at the end of this document.

3.1 Journals

- [1] W. Halfond, A. Orso, and P. Manolios. WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. IEEE Transactions on Software Engineering (TSE), Vol 34, Issue 1, Jan 2008, pages 65–81.

3.2 Books and Parts of Books

- [2] W. Halfond and A. Orso. Detection and Prevention of SQL Injection Attacks. Malware Detection, Series: Advances in Information Security, Springer, Vol. 27, M. Christodorescu, S. Jha, D. Maughan, D. Song, C. Wang (Eds.), 2007, XII.

3.3 Conferences

- [3] W. Halfond, A. Orso, and P. Manolios. Using Positive Tainting and Syntax-Aware Evaluation to Protect Web Applications. Proceedings of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2006), Portland, Oregon, USA, November 2006, pages 175–185.
- [4] W. Halfond and A. Orso. Command-Form Coverage for Testing Database Applications. Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2006), Tokyo, Japan, September 2006, pages 69–78.
- [5] W. Halfond and A. Orso. Preventing SQL Injection Attacks Using AMNESIA. Proceedings of the 28th IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE 2006) – Formal Demos track. Shanghai, China, May 2006, pages 795–798.
- [6] J. Viegas, W. Halfond, and A. Orso. A Classification of SQL Injection Attacks and Prevention Techniques. Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE 2006), Washington, D.C., USA, March 2006, pages 12–23.

Appendix – Copy of Publications

WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation

William G.J. Halfond, Alessandro Orso, Member, IEEE Computer Society, and
Panagiotis Manolios, Member, IEEE Computer Society

Abstract—Many software systems have evolved to include a Web-based component that makes them available to the public via the Internet and can expose them to a variety of Web-based attacks. One of these attacks is SQL injection, which can give attackers unrestricted access to the databases that underlie Web applications, which has become increasingly frequent and serious. This paper presents a new highly automated approach for protecting Web applications against SQL injection that has both conceptual and practical advantages over most existing techniques. From a conceptual standpoint, the approach is based on the novel idea of positive tainting and on the concept of syntax-aware evaluation. From a practical standpoint, our technique is precise and efficient, has minimal deployment requirements, and incurs a negligible performance overhead in most cases. We have implemented our techniques in the Web Application SQL-injection Preventer (WASP) tool, which we used to perform an empirical evaluation on a wide range of Web applications that we subjected to a large and varied set of attacks and legitimate accesses. WASP was able to stop all of the otherwise successful attacks and did not generate any false positives.

Index Terms—Security, SQL injection, dynamic tainting, runtime monitoring.

Ç

1 Introduction

Web applications are applications that can be accessed over the Internet by using any compliant Web browser that runs on any operating system and architecture. They have become ubiquitous due to the convenience, flexibility, availability, and interoperability that they provide.

Unfortunately, Web applications are also vulnerable to a variety of new security threats. SQL Injection Attacks (SQLIAs) are one of the most significant of such threats [6]. SQLIAs have become increasingly frequent and pose very serious security risks because they can give attackers unrestricted access to the databases that underlie Web applications.

Web applications interface with databases that contain information such as customer names, preferences, credit card numbers, purchase orders, and so on. Web applications build SQL queries to access these databases based, in part, on user-provided input. The intent is that Web applications will limit the kinds of queries that can be generated to a safe subset of all possible queries, regardless of what input users provide. However, inadequate input validation can enable attackers to gain complete access to such databases. One way in which this happens is that attackers can submit input strings that contain specially encoded database commands. When the Web application builds a query by using these strings and submits the query to its underlying database, the attacker's embedded commands are executed by the database and the attack succeeds. The results of these attacks are often disastrous and can range from leaking of sensitive data (for example, customer data) to the destruction of database contents.

Researchers have proposed a wide range of alternative techniques to address SQLIAs, but many of these solutions have limitations that affect their effectiveness and practicality. For example, one common class of solutions is based on defensive coding practices, which have been less than successful for three main reasons. First, it is difficult to implement and enforces a rigorous defensive coding discipline. Second, many solutions based on defensive coding address only a subset of the possible attacks. Third, legacy software poses a particularly difficult problem because of the cost and complexity of retrofitting existing code so that it is compliant with defensive coding practices.

In this paper, we propose a new highly automated approach for dynamic detection and prevention of SQLIAs. Intuitively, our approach works by identifying "trusted" strings in an application and allowing only these trusted strings to be used to create the semantically relevant parts of a SQL query such as keywords or operators. The general mechanism that we use to implement this approach is based on dynamic tainting, which marks and tracks certain data in a program at runtime.

The kind of dynamic tainting that we use gives our approach several important advantages over techniques based on other mechanisms. Many techniques rely on complex static analyses in order to find potential vulnerabilities in the code (for example, [11], [18], [29]). These kinds of conservative static analyses can generate high rates of false positives and can have scalability issues when

. W.G.J. Halfond and A. Orso are with the College of Computing, Georgia Institute of Technology, Klaus Advanced Computing Building, 266 Ferst Drive, Atlanta, GA 30332-0765. E-mail: {whalfond, orso}@cc.gatech.edu.
. P. Manolios is with the College of Computer and Information Science, Northeastern University, 360 Huntington Avenue, Boston, MA 02115. E-mail: pete@ccs.neu.edu.

Manuscript received 24 Feb. 2007; revised 8 Aug. 2007; accepted 29 Aug. 2007; published online 24 Sept. 2007.
Recommended for acceptance by P. McDaniel and B. Nuseibeh.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-0084-0207.
Digital Object Identifier no. 10.1109/TSE.2007.70748.

applied to large complex applications. In contrast, our approach does not rely on complex static analyses and is both efficient and precise. Other techniques involve extensive human effort (for example, [5], [21], [27]). They require developers to manually rewrite parts of the Web applications, build queries using special libraries, or mark all points in the code at which malicious input could be introduced. Our approach is highly automated and, in most cases, requires minimal or no developer intervention. Last, several proposed techniques require the deployment of extensive infrastructure or involve complex configurations (for example, [2], [26], [28]). Our approach does not require additional infrastructure and can be automatically deployed.

Compared to other existing techniques based on dynamic tainting (for example, [9], [23], [24]), our approach makes several conceptual and practical improvements that take advantage of the specific characteristics of SQLIAs. The first conceptual advantage of our approach is the use of positive tainting. Positive tainting identifies and tracks trusted data, whereas traditional ("negative") tainting focuses on untrusted data. In the context of SQLIAs, there are several reasons why positive tainting is more effective than negative tainting. First, in Web applications, sources of trusted data can more easily and accurately be identified than untrusted data sources. Therefore, the use of positive tainting leads to increased automation. Second, the two approaches significantly differ in how they are affected by incompleteness. With negative tainting, failure to identify the complete set of untrusted data sources can result in false negatives, that is, successful and undetected attacks. With positive tainting, missing trusted data sources can result in false positives (that is, legitimate accesses can be prevented from completing). False positives that occur in the field would be problematic. Using our approach, however, false positives are likely to be detected during prerelease testing. Our approach provides specific mechanisms for helping developers detect false positives early, identify their sources, and easily eliminate them in future runs by tagging the identified sources as trusted.

The second conceptual advantage of our approach is the use of flexible syntax-aware evaluation. Syntax-aware evaluation lets us address security problems that are derived from mixing data and code while still allowing for this mixing to occur. More precisely, it gives developers a mechanism for regulating the usage of string data based not only on its source but also on its syntactical role in a query string. This way, developers can use a wide range of external input sources to build queries while protecting the application from possible attacks introduced via these sources.

The practical advantages of our approach are that it imposes a low overhead on the application and it has minimal deployment requirements. Efficiency is achieved by using a specialized library, called MetaStrings, that accurately and efficiently assigns and tracks trust markings at runtime. The only deployment requirements for our approach are that the Web application must be instrumented and it must be deployed with our MetaStrings library, which is done automatically. The approach does not require any customized runtime system or additional infrastructure.

In this paper, we also present the results of an extensive empirical evaluation of the effectiveness and efficiency of our technique. To perform this evaluation, we implemented our approach in a tool called Web Application SQL-injection Preventer (WASP) and evaluated WASP on a set of 10 Web applications of various types and sizes. For each application, we protected it with WASP, targeted it with a large set of attacks and legitimate accesses, and assessed the ability of our technique to detect and prevent attacks without stopping legitimate accesses. The results of the evaluation are promising. Our technique was able to stop all of the attacks without generating false positives for any of the legitimate accesses. Moreover, our technique proved to be efficient, imposing only a low overhead on the Web applications.

The main contributions of this work are listed as follows:

1. a new automated technique for preventing SQLIAs based on the novel concept of positive tainting and on flexible syntax-aware evaluation,
2. a mechanism to perform efficient dynamic tainting of Java strings which precisely propagates trust markings while strings are manipulated at runtime,
3. a tool that implements our SQLIA prevention technique for Java-based Web applications and has minimal deployment requirements, and
4. an empirical evaluation of the technique that shows its effectiveness and efficiency.

The rest of this paper is organized as follows: In Section 2, we introduce SQLIAs. Sections 3 and 4 discuss the approach and its implementation. Section 5 presents the results of our evaluation. We discuss related work in Section 6 and conclude in Section 7.

2 Motivation: Structured Query Language Injection Attacks

In this section, we first motivate our work by introducing an example of an SQLIA that we use throughout the paper to illustrate our approach and, then, we discuss the main types of SQLIAs in detail.

In general, SQLIAs are a class of code injection attacks that take advantage of the lack of validation of user input. These attacks occur when developers combine hard-coded strings with user-provided input to create dynamic queries. Intuitively, if user input is not properly validated, attackers may be able to change the developer's intended SQL command by inserting new SQL keywords or operators through specially crafted input strings. Interested readers can refer to the work of Su and Wassermann [27] for a formal definition of SQLIAs. SQLIAs leverage a wide range of mechanisms and input channels to inject malicious commands into a vulnerable application [12]. Before providing a detailed discussion of these various mechanisms, we introduce an example application that contains a simple SQL injection vulnerability and show how an attacker can leverage that vulnerability.

Fig. 1 shows an example of a typical Web application architecture. In the example, the user interacts with a Web form that takes a login name and pin as inputs and submits them to a Web server. The Web server passes the user-

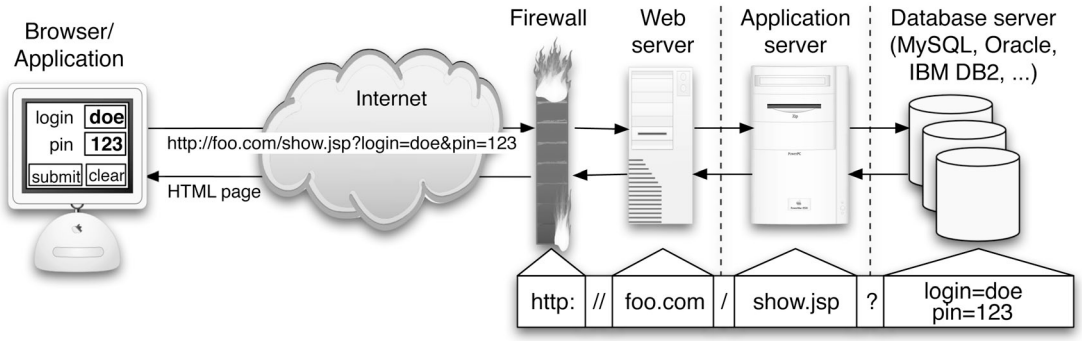


Fig. 1. Example of interaction between a user and a typical Web application.

supplied credentials to a servlet (`show.jsp`), which is a special type of Java application that runs on a Web application server and whose execution is triggered by the submission of a URL from a client.

The example servlet, whose code is partially shown in Fig. 2, implements a login functionality that we can find in a typical Web application. It uses input parameters `login` and `pin` to dynamically build an SQL query or command. (For simplicity, in the rest of this paper, we use the terms query and command interchangeably.) The `login` and `pin` are checked against the credentials stored in the database. If they match, the corresponding user's account information is returned. Otherwise, a null set is returned by the database and the authentication fails. The servlet then uses the response from the database to generate HTML pages that are sent back to the user's browser by the Web server.

For this servlet, if a user submits `login` and `pin` as "doe" and "123," the application dynamically builds the query:

```
SELECT acct FROM users WHERE login='doe' AND pin=123
```

If `login` and `pin` match the corresponding entry in the database, doe's account information is returned and then displayed by function `displayAccount()`. If there is no match in the database, function `sendAuthFailed()` displays an appropriate error message. An application that uses this servlet is vulnerable to SQLIAs. For example, if an attacker enters "`admin' --`" as the username and any value as the pin (for example, "0"), the resulting query is

```
SELECT acct FROM users WHERE login='admin' -- ' AND pin=0
```

In SQL, "– –" is the comment operator and everything after it is ignored. Therefore, when performing this query, the database simply searches for an entry where `login` is equal to `admin` and returns that database record. After the "successful" login, the function `displayAccount()` reveals the `admin`'s account information to the attacker.

```
1. String login = getParameter("login");
2. String pin = getParameter("pin");
3. Statement stmt = connection.createStatement();
4. String query = "SELECT acct FROM users WHERE login='";
5. query += login + "' AND pin=" + pin;
6. ResultSet result = stmt.executeQuery(query);
7. if (result != null)
8.    displayAccount(result); // Show account
9. else
10.   sendAuthFailed(); // Authentication failed
```

Fig. 2. Excerpt of a Java servlet implementation.

It is important to stress that this example represents an extremely simple kind of attack and we present it for illustrative purposes only. Because simple attacks of this kind are widely used in the literature as examples, they are often mistakenly viewed as the only types of SQLIAs. In reality, there is a wide variety of complex and sophisticated SQL exploits available to attackers. We next discuss the main types of such attacks.

2.1 Main Variants of Structured Query Language Injection Attacks

Over the past several years, attackers have developed a wide array of sophisticated attack techniques that can be used to exploit SQL injection vulnerabilities. These techniques go beyond the well-known SQLIA examples and take advantage of esoteric and advanced SQL constructs. Ignoring the existence of these kinds of attacks leads to the development of solutions that only partially address the SQLIA problem.

For example, developers and researchers often assume that SQLIAs are introduced only via user input that is submitted as part of a Web form. This assumption misses the fact that any external input that is used to build a query string may represent a possible channel for SQLIAs. In fact, it is common to see other external sources of input such as fields from an HTTP cookie or server variables used to build a query. Since cookie values are under the control of the user's browser and server variables are often set using values from HTTP headers, these values are actually external strings that can be manipulated by an attacker. In addition, second-order injections use advanced knowledge of vulnerable applications to introduce attacks by using otherwise properly secured input sources [1]. A developer may suitably escape, type-check, and filter input that comes from the user and assume that it is safe. Later on, when that data is used in a different context or to build a different type of query, the previously safe input may enable an injection attack.

Once attackers have identified an input source that can be used to exploit an SQLIA vulnerability, there are many different types of attack techniques that they can leverage. Depending on the type and extent of the vulnerability, the results of these attacks can include crashing the database, gathering information about the tables in the database

schema, establishing covert channels, and open-ended injection of virtually any SQL command. Here, we summarize the main techniques for performing SQLIAs. We provide additional information and examples of how these techniques work in [12].

2.1.1 Tautologies

Tautology-based attacks are among the simplest and best known types of SQLIAs. The general goal of a tautology-based attack is to inject SQL tokens that cause the query's conditional statement to always evaluate to true. Although the results of this type of attack are application specific, the most common uses are bypassing authentication pages and extracting data. In this type of injection, an attacker exploits a vulnerable input field that is used in the query's WHERE conditional. This conditional logic is evaluated as the database scans each row in the table. If the conditional represents a tautology, the database matches and returns all of the rows in the table as opposed to matching only one row, as it would normally do in the absence of injection. An example of a tautology-based SQLIA for the servlet in our example in Section 2 is the following:

```
SELECT acct FROM users WHERE login='' OR 1=1 -- ' AND pin=
```

Because the WHERE clause is always true, this query will return account information for all of the users in the database.

2.1.2 Union Queries

Although tautology-based attacks can be successful, for instance, in bypassing authentication pages, they do not give attackers much flexibility in retrieving specific information from a database. Union queries are a more sophisticated type of SQLIA that can be used by an attacker to achieve this goal in that they cause otherwise legitimate queries to return additional data. In this type of SQLIA, attackers inject a statement of the form "UNION < injected query > ." By suitably defining < injected query > , attackers can retrieve information from a specified table. The outcome of this attack is that the database returns a data set that is the union of the results of the original query with the results of the injected query. In our example, an attacker could perform a Union Query injection by injecting the text "' UNION SELECT cardNo from CreditCards whereacctNo ¼ 7032 " into the login field. The application would then produce the following query:

```
SELECT acct FROM users WHERE login='' UNION SELECT cardNo
from CreditCards where acctNo=7032 -- AND pin=
```

The original query should return the null set and the injected query returns data from the "CreditCards" table. In this case, the database returns field "cardNo" for account "7032." The database takes the results of these two queries, unites them, and returns them to the application. In many applications, the effect of this attack would be that the value for "cardNo" is displayed with the account information.

2.1.3 Piggybacked Queries

Similar to union queries, this kind of attack appends additional queries to the original query string. If the attack is successful, the database receives and executes a query string that contains multiple distinct queries. The first query is generally the original legitimate query, whereas subsequent queries are the injected malicious queries. This type of attack can be especially harmful because attackers can use it to inject virtually any type of SQL command. In our example, an attacker could inject the text "0; drop table users" into the pin input field and have the application generate the following query:

```
SELECT acct FROM users WHERE login='doe' AND pin=0; drop
table users
```

The database treats this query string as two queries separated by the query delimiter (";") and executes both. The second malicious query causes the database to drop the users table in the database, which would have the catastrophic consequence of deleting all user information. Other types of queries can be executed using this technique such as the insertion of new users into the database or the execution of stored procedures. Note that many databases do not require a special character to separate distinct queries, so simply scanning for separators is not an effective way to prevent this attack technique.

2.1.4 Malformed Queries

Union queries and piggybacked queries let attackers perform specific queries or execute specific commands on a database, but require some prior knowledge of the database schema, which is often unknown. Malformed queries allow for overcoming this problem by taking advantage of overly descriptive error messages that are generated by the database when a malformed query is rejected. When these messages are directly returned to the user of the Web application, instead of being logged for debugging by developers, attackers can make use of the debugging information to identify vulnerable parameters and infer the schema of the underlying database. Attackers exploit this situation by injecting SQL tokens or garbage input that causes the query to contain syntax errors, type mismatches, or logical errors. Considering our example, an attacker could try causing a type mismatch error by injecting the following text into the pin input field: "convertðint; ðselect top 1 name from sysobjects where xtype ¼ 'u'ÞÞ. " The resulting query generated by the Web application is the following:

```
SELECT acct FROM users WHERE login='' AND pin=convert(int,
(select top 1 name from sysobjects where xtype='u'))
```

The injected query extracts the name of the first user table xtype ¼ 'u' from the database's metadata table sysobjects. It then converts this table name to an integer. Because the name of the table is a string, the conversion is illegal and the database returns an error. For example, a SQL Server may return the following error: "Microsoft OLE DB Provider for SQL Server (0x80040E07) Error converting nvarchar value 'CreditCards' to a column of data type int." From this message, the attacker can 1) see that the database is an SQL Server and 2) discover that the name of the first user-defined table in the database is "CreditCards" (the string that caused the type conversion to occur). A similar strategy can be used to systematically extract the name and type of

each column in the given table. Using this information about the schema of the database, an attacker can create more precise attacks that specifically target certain types of information. Malformed queries are typically used as a preliminary information-gathering step for other attacks.

2.1.5 Inference

Similar to malformed queries, inference-based attacks let attackers discover information about a database schema. This type of SQLIAs creates queries that cause an application or database to behave differently based on the results of the query. This way, even if an application does not directly provide the results of the query to the attacker, it is possible to observe side effects caused by the query and deduce its results. One particular type of attack based on inference is a timing attack, which lets attackers gather information from a database by observing timing delays in the database's responses. To perform a timing attack, attackers structure their injected queries in the form of an if-then statement whose branch condition corresponds to a question about the contents of the database. The attacker then uses the WAITFOR keyword along one of the branches, which causes the database to delay its response by a specified time. By measuring the increase or decrease in the database response time, attackers can infer which branch was taken and the answer to the injected question. For our example servlet, an attacker could inject the following text into the login parameter: "legalUser' AND ASCII(SUBSTRING((select top 1 name from sysobjects),1,1)) > X WAITFOR 5 --." This injection produces the following query:

```
SELECT acct FROM users WHERE login='legalUser' and
ASCII(SUBSTRING((select top 1 name from sysobjects),1,1))
> X WAITFOR 10 -- ' AND pin=
```

In the attack, the SUBSTRING function is used to extract the first character of the database's first table's name, which is then converted into an ASCII value and compared with the value of X. If the value is greater, the attacker will be able to observe a 10 s delay in the database response. The attacker can continue this way and use a binary-search strategy to identify the value of each character in the table's name. Another well-known type of inference attack is the blind SQL injection [12].

2.1.6 Alternate Encodings

Many types of SQLIAs involve the use of special characters such as single quotes, dashes, or semicolons as part of the inputs to a Web application. Therefore, basic protection techniques against these attacks check the input for the presence of such characters and escape them or simply block inputs that contain them. Alternate encodings let attackers modify their injected strings in a way that avoids these typical signature-based and filter-based checks. Encodings such as ASCII, hexadecimal, and Unicode can be used in conjunction with other techniques to allow an attack to escape straightforward detection approaches that simply scan for certain known "bad characters." Even if developers account for alternate encodings, this technique can still be successful because alternate encodings can target different layers in the application. For example, a developer may scan for a Unicode or hexadecimal encoding of a single quote and not realize that the attacker can leverage database functions to encode the same character. An effective code-based defense against alternate encodings requires developers to be aware of all of the possible encodings that could affect a given query string as it passes through the different application layers. Because developing such a complete protection is very difficult in practice, attackers have been successful in using alternate encodings to conceal attack strings. The following example attack (from [13]) shows the level of obfuscation that can be achieved using alternate encodings. In the attack, the pin field is injected with string "0; exec(char(0x73687574646f776e))," which results in the following query:

```
SELECT acct FROM users WHERE login='' AND pin=0;
exec(char(0x73687574646f776e))
```

This attack leverages the char() function provided by some databases and uses ASCII hexadecimal encoding. The stream of numbers in the second part of the injection is the ASCII hexadecimal encoding of the attack string. This encoded string is inserted into a query by using some other type of attack profile and, when it is executed by the database, translates into the shutdown command.

2.1.7 Leveraging Stored Procedures

Another strongly advertised solution for the problem of SQLIAs is the use of stored procedures, that is, procedures that are stored in the database and can be run by the database engine. Stored procedures provide developers with an extra layer of abstraction because they can enforce businesswide database rules, independent of the logic of individual Web applications. Unfortunately, it is a common misconception that the mere use of stored procedures protects an application from SQLIAs: Similarly to any other software, the safety of stored procedures depends on the way in which they are coded and on the use of adequate defensive coding practices. Therefore, parametric stored procedures could also be vulnerable to SQLIAs, just like the rest of the code in a Web application.

The following example demonstrates how a (parametric) stored procedure can be exploited via an SQLIA. In this scenario, assume that the query string constructed by our example servlet has been replaced by a call to the following stored procedure:

```
CREATE PROCEDURE DBO.isAuthenticated
    @userName varchar2, @pin int
AS
    EXEC("SELECT acct FROM users WHERE login='"
    + @userName + "' and pin= " +@pin);
GO
```

To perform an SQLIA that exploits this stored procedure, the attacker can simply inject the text "' ; SHUTDOWN; --" into the userName field. This injection causes the stored procedure to generate the following query, which would result in the database being shut down:

```
SELECT acct FROM users WHERE login=' ';SHUTDOWN; -- AND pin=
```

3 Our Approach

Our approach against SQLIAs is based on dynamic tainting, which has previously been used to address security problems related to input validation. Traditional dynamic tainting approaches mark certain untrusted data (typically user input) as tainted, track the flow of tainted data at runtime, and prevent this data from being used in potentially harmful ways. Our approach makes several conceptual and practical improvements over traditional dynamic tainting approaches by taking advantage of the characteristics of SQLIAs and Web applications. First, unlike existing dynamic tainting techniques, our approach is based on the novel concept of positive tainting, that is, the identification and marking of trusted, instead of untrusted, data. Second, our approach performs accurate and efficient taint propagation by precisely tracking trust markings at the character level. Third, it performs syntax-aware evaluation of query strings before they are sent to the database and blocks all queries whose nonliteral parts (that is, SQL keywords and operators) contain one or more characters without trust markings. Finally, our approach has minimal deployment requirements, which makes it both practical and portable. The following sections discuss these key features of our approach in detail.

3.1 Positive Tainting

Positive tainting differs from traditional tainting (hereafter, negative tainting) because it is based on the identification, marking, and tracking of trusted, rather than untrusted, data. This conceptual difference has significant implications for the effectiveness of our approach in that it helps address problems caused by incompleteness in the identification of relevant data to be marked. Incompleteness, which is one of the major challenges when implementing a security technique based on dynamic tainting, has very different consequences in negative and positive tainting. In the case of negative tainting, incompleteness leads to trusting data that should not be trusted and, ultimately, to false negatives. Incompleteness may thus leave the application vulnerable to attacks and can be very difficult to detect, even after attacks actually occur, because they may go completely unnoticed. With positive tainting, incompleteness may lead to false positives, but it would never result in an SQLIA escaping detection. Moreover, as explained in the following, the false positives generated by our approach, if any, are likely to be detected and easily eliminated early during prerelease testing. Positive tainting uses a white-list, rather than a black-list, policy and follows the general principle of fail-safe defaults, as outlined by Saltzer and Schroeder [25]: In case of incompleteness, positive tainting fails in a way that maintains the security of the system. Fig. 3 shows a graphical depiction of this fundamental difference between negative and positive tainting.

In the context of preventing SQLIAs, the conceptual advantages of positive tainting are especially significant. The way in which Web applications create SQL commands makes the identification of all untrusted data especially problematic and, most importantly, the identification of most trusted data relatively straightforward. Web applications are deployed in many different configurations and interface with

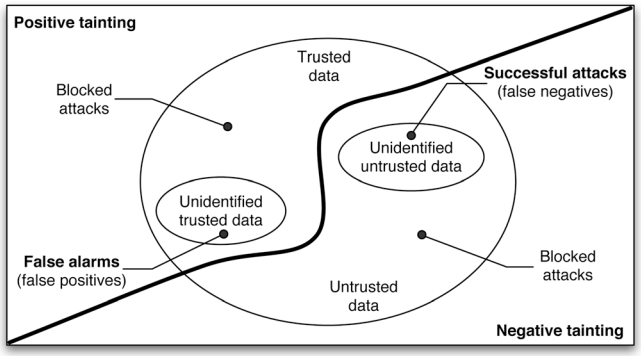


Fig. 3. Identification of trusted and untrusted data.

a wide range of external systems. Therefore, there are often many potential external untrusted sources of input to be considered for these applications and enumerating all of them is inherently difficult and error prone. For example, developers initially assumed that only direct user input needed to be marked as tainted. Subsequent exploits demonstrated that additional input sources such as browser cookies and uploaded files also needed to be considered. However, accounting for these additional input sources did not completely solve the problem either. Attackers soon realized the possibility of leveraging local server variables and the database itself as injection sources [1]. In general, it is difficult to guarantee that all potentially harmful data sources have been considered and even a single unidentified source could leave the application vulnerable to attacks.

The situation is different for positive tainting because identifying trusted data in a Web application is often straightforward and always less error prone. In fact, in most cases, strings hard-coded in the application by developers represent the complete set of trusted data for a Web application.[1] This is because it is common practice for developers to build SQL commands by combining hard-coded strings that contain SQL keywords or operators with user-provided numeric or string literals. For Web applications developed this way, our approach accurately and automatically identifies all SQLIAs and generates no false positives. Our basic approach, as explained in the following sections, automatically marks as trusted all hard-coded strings in the code and then ensures that all SQL keywords and operators are built using trusted data.

In some cases, this basic approach is not enough because developers can also use external query fragments—partial SQL commands that come from external input sources—to build queries. Because these string fragments are not hard-coded in the application, they would not be part of the initial set of trusted data identified by our approach and the approach would generate false positives when the string fragments are used in a query. To account for these cases, our technique provides developers with a mechanism for specifying sources of external data that should be trusted. The data sources can be of various types such as files, network connections, and server variables. Our approach

1. Without loss of generality, we assume that developers are trustworthy. An attack encoded in the application by a developer would not be an SQLIA but a form of backdoor, which is not the problem addressed in this work.

uses this information to mark data that comes from these additional sources as trusted.

In a typical scenario, we expect developers to specify most of the trusted sources before testing and deployment. However, some of these sources might be overlooked until after a false positive is reported, in which case, developers would add the omitted items to the list of trusted sources. In this process, the set of trusted data sources monotonically grows and eventually converges to a complete set that produces no false positives. It is important to note that false positives that occur after deployment would be due to the use of external data sources that have never been used during in-house testing. In other words, false positives are likely to occur only for totally untested parts of applications. Therefore, even when developers fail to completely identify additional sources of trusted data beforehand, we expect these sources to be identified during normal testing and the set of trusted data to quickly converge to the complete set.

It is also worth noting that none of the subjects that we collected and examined so far required us to specify additional trusted data sources. All of these subjects used only hard-coded strings to build query strings.

3.2 Accurate and Efficient Taint Propagation

Taint propagation consists of tracking taint markings associated with the data while the data is used and manipulated at runtime. When tainting is used for security-related applications, it is especially important for the propagation to be accurate. Inaccurate propagation can undermine the effectiveness of a technique by associating incorrect markings to data, which would cause the data to be mishandled. In our approach, we provide a mechanism to accurately mark and propagate taint information by 1) tracking taint markings at the "right" level of granularity and 2) precisely accounting for the effect of functions that operate on the tainted data.

Character-level tainting. We track taint information at the character level rather than at the string level. We do this because, for building SQL queries, strings are constantly broken into substrings, manipulated, and combined. By associating taint information to single characters, our approach can precisely model the effect of these string operations. Another alternative would be to trace taint data at the bit level, which would allow us to account for situations where string data are manipulated as character values using bitwise operators. However, operating at the bit level would make the approach considerably more expensive and complex to implement and deploy. Most importantly, our experience with Web applications shows that working at a finer level of granularity than a character would not yield any benefit in terms of effectiveness. Strings are typically manipulated using methods provided by string library classes and we have not encountered any case of query strings that are manipulated at the bit level.

Accounting for string manipulations. To accurately maintain character-level taint information, we must identify all relevant string operations and account for their effect on the taint markings (that is, we must enforce complete mediation of all string operations). Our approach achieves this goal by taking advantage of the encapsulation offered by object-oriented languages, in particular by Java, in which all string manipulations are performed using a small set of classes and methods. Our approach extends all such classes and methods by adding functionality to update taint markings based on the methods' semantics.

We discuss the language-specific details of our implementation of the taint markings and their propagation in Section 4.

3.3 Syntax-Aware Evaluation

Aside from ensuring that taint markings are correctly created and maintained during execution, our approach must be able to use the taint markings to distinguish legitimate from malicious queries. Simply forbidding the use of untrusted data in SQL commands is not a viable solution because it would flag any query that contains user input as an SQLIA, leading to many false positives. To address this shortcoming, researchers have introduced the concept of declassification, which permits the use of tainted input as long as it has been processed by a sanitizing function. (A sanitizing function is typically a filter that performs operations such as regular expression matching or substring replacement.) The idea of declassification is based on the assumption that sanitizing functions are able to eliminate or neutralize harmful parts of the input and make the data safe. However, in practice, there is no guarantee that the checks performed by a sanitizing function are adequate. Tainting approaches based on declassification could therefore generate false negatives if they mark as trusted supposedly sanitized data that is actually still harmful. Moreover, these approaches may also generate false positives in cases where unsanitized but perfectly legal input is used within a query.

Syntax-aware evaluation does not rely on any (potentially unsafe) assumptions about the effectiveness of sanitizing functions used by developers. It also allows for the use of untrusted input data in a SQL query as long as the use of such data does not cause an SQLIA. The key feature of syntax-aware evaluation is that it considers the context in which trusted and untrusted data is used to make sure that all parts of a query other than string or numeric literals (for example, SQL keywords and operators) consist only of trusted characters. As long as untrusted data is confined to literals, we are guaranteed that no SQLIA can be performed. Conversely, if this property is not satisfied (for example, if a SQL operator contains characters that are not marked as trusted), we can assume that the operator has been injected by an attacker and identify the query as an attack.

Our technique performs syntax-aware evaluation of a query string immediately before the string is sent to the database to be executed. To evaluate the query string, the technique first uses a SQL parser to break the string into a sequence of tokens that correspond to SQL keywords, operators, and literals. The technique then iterates through the tokens and checks whether tokens (that is, substrings) other than literals contain only trusted data. If all such tokens pass this check, the query is considered safe and is allowed to execute. If an attack is detected, a developer-specified action can be invoked. As discussed in Section 3.1, this approach can also handle cases where developers use external query fragments to build SQL commands. In these cases, developers would specify which external data

sources must be trusted, and our technique would mark and treat data that comes from these sources accordingly.

This default approach, which 1) considers only two kinds of data (trusted and untrusted) and 2) allows only trusted data to form SQL keywords and operators, is adequate for most Web applications. For example, it can handle applications where parts of a query are stored in external files or database records that were created by the developers. Nevertheless, to provide greater flexibility and support a wide range of development practices, our technique also allows developers to associate custom trust markings to different data sources and provide custom trust policies that specify the legal ways in which data with certain trust markings can be used. Trust policies are functions that take as input a sequence of SQL tokens and perform some type of check based on the trust markings associated with the tokens.

BUGZILLA (http://www.bugzilla.org) is an example of a Web application for which developers might wish to specify a custom trust marking and policy. In BUGZILLA, parts of queries used within the application are retrieved from a database when needed. Of particular concern to developers in this scenario is the potential for second-order injection attacks [1] (that is, attacks that inject into a database malicious strings that result in an SQLIA only when they are later retrieved and used to build SQL queries). In the case of BUGZILLA, the only subqueries that should originate in the database are specific predicates that form a query's WHERE clause. Using our technique, developers could first create a custom trust marking and associate it with the database's data source. Then, they could define a custom trust policy that specifies that data with such a custom trust marking is legal only if it matches a specific pattern such as ðidjseverityÞ ¼ nw þ ðð ANDjORÞ ðidjseverityÞ ¼ nwþ?.

When applied to subqueries that originate in the database, this policy would allow them to be used only to build conditional clauses that involve the id or severity fields and whose parts are connected using the AND or OR keywords.

3.4 Minimal Deployment Requirements

Most existing approaches based on dynamic tainting require the use of customized runtime systems and/or impose a considerable overhead on the protected applications (see Section 6). In contrast, our approach has minimal deployment requirements and is efficient, which makes it practical for use in real settings. Our technique does not necessitate a customized runtime system. It requires only minor localized instrumentation of the application to 1) enable the use of our string library and 2) insert the calls that perform syntax-aware evaluation of a query before it is sent to the database. The protected application is then deployed as a normal Web application except that the deployment must include our string library. Both instrumentation and deployment are fully automated. We discuss the deployment requirements and the overhead of the approach in greater detail in Sections 4.5 and 5.3.

4 Our Implementation: WASP

To evaluate our approach, we developed a prototype tool called WASP (Web Application SQL-injection Preventer), which is written in Java and implements our technique for Java-based Web applications. We target Java because it is one of the most commonly used languages for Web applications. (We discuss the applicability of the approach in other contexts in Section 4.1.)

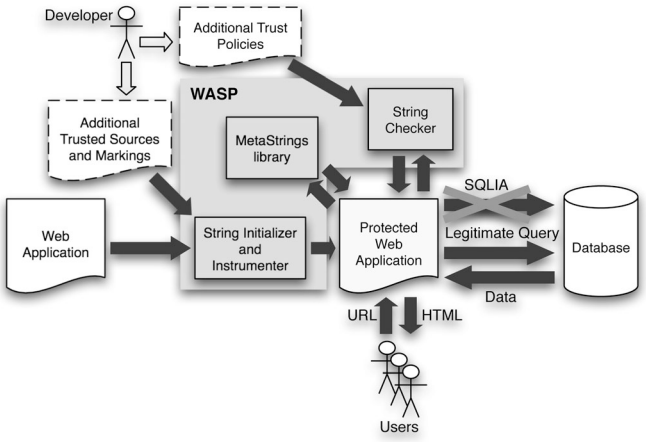


Fig. 4. High-level overview of the approach and tool.

Fig. 4 shows the high-level architecture of WASP. As this figure shows, WASP consists of a library (MetaStrings) and two core modules (STRING INITIALIZER AND INSTRUMENTER and STRING CHECKER). The MetaStrings library provides functionality for assigning trust markings to strings and precisely propagating the markings at runtime. Module STRING INITIALIZER AND INSTRUMENTER instruments Web applications to enable the use of the MetaStrings library and adds calls to the STRING CHECKER module. Module STRING CHECKER performs syntax-aware evaluation of query strings right before the strings are sent to the database.

In the next sections, we discuss WASP's modules in more detail. We use the sample code introduced in Section 2 to provide examples of various implementation aspects.

4.1 The MetaStrings Library

MetaStrings is our library of classes that mimic and extend the behavior of Java's standard string classes (that is, Character, String, StringBuilder, and StringBuffer).[2] For each string class C, MetaStrings provides a "meta" version of the class MetaC, which has the same functionality as C, but allows for associating metadata with each character in a string and tracking the metadata as the string is manipulated at runtime.

The MetaStrings library takes advantage of the object-oriented features of the Java language to provide complete mediation of string operations that could affect string values and their associated trust markings. Encapsulation and information hiding guarantee that the internal representation of a string class is accessed only through the class's interface. Polymorphism and dynamic binding let us add functionality to a string class by 1) creating a subclass that overrides relevant methods of the original class and 2) replacing instantiations of the original class with instantiations of the subclass. In our implementation, we leverage the object-oriented features of Java and the approach

2. For simplicity, hereafter we use the term string to refer to all string-related classes and objects in Java.

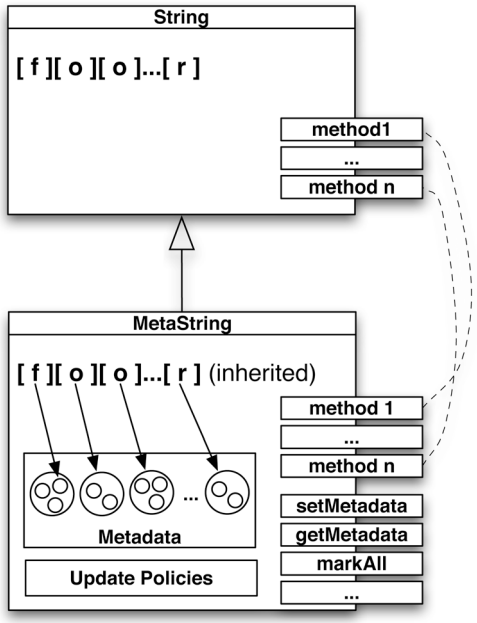


Fig. 5. An intuitive view of a MetaStrings library class.

should be easily applicable to applications built using other object-oriented languages such as .NET. Although the use of object-oriented features allows our current implementation to be elegant and minimally intrusive, we expect the approach to be portable, with suitable engineering, to non-object-oriented languages. For example, in C, the approach could be implemented by identifying and instrumenting calls to functions and operations that manipulate strings or characters. In general, our approach should be portable to all contexts where 1) string-creation and string-manipulation operations can be identified and 2) a character-level taint initialization and propagation mechanism can be implemented (either through instrumentation or by modifying the runtime system).

To illustrate our MetaStrings library with an example, Fig. 5 shows an intuitive view of the MetaStrings class that corresponds to Java's `String` class. As this figure shows, `MetaString` extends class `String`, has the same internal representation, and provides the same methods. `MetaString` also contains additional data structures for storing metadata and associating the metadata with characters in the string. Each method of class `MetaString` overrides the corresponding method in `String`, providing the same functionality as the original method but also updating the metadata based on the method's semantics. For example, a call to method `substring(2, 4)` on an object `str` of class `MetaString` would return a new `MetaString` that contains the second and third characters of `str` and the corresponding metadata. In addition to the overridden methods, MetaStrings classes also provide methods for setting and querying the metadata associated with a string's characters.

The use of MetaStrings has the following benefits:

1. It allows for associating trust markings at the granularity level of single characters.
2. It accurately maintains and propagates trust markings.
3. It is completely defined at the application level and thus does not require a customized runtime system.
4. Its usage requires only minimal and automatically performed changes in the application's bytecode.
5. It imposes a low execution overhead on Web applications, as shown in Section 5.3.

The main limitations of the current implementation of the MetaStrings library are related to the handling of primitive types, native methods, and reflection. MetaStrings cannot currently assign trust markings to primitive types, so it cannot mark `char` values. Because we do not instrument native methods, if a string class is passed as an argument to a native method, the trust marking associated with the string might not be correct after the call. In the case of hard-coded strings created through reflection (by invoking a string constructor by name), our instrumenter for MetaStrings would not recognize the constructors and would not change these instantiations to instantiations of the corresponding metaclasses. However, the MetaStrings library can handle most other uses of reflection, such as invocation of string methods by name.

In practice, these limitations are of limited relevance because they represent programming practices that are not normally used to build SQL commands (for example, representing strings by using primitive `char` values). Moreover, during the instrumentation of a Web application, we identify and report these potentially problematic situations to the developers.

4.2 Initialization of Trusted Strings

To implement positive tainting, WASP must be able to identify and mark trusted strings. There are three categories of strings that WASP must consider: hard-coded strings, strings implicitly created by Java, and strings originating in external sources. In the following sections, we explain how strings from each category are identified and marked.

4.2.1 Hard-Coded Strings

The identification of hard-coded strings in an application's bytecode is fairly straightforward. In Java, hard-coded strings are represented using `String` objects that are automatically created by the Java Virtual Machine (JVM) when string literals are loaded onto the stack. (The JVM is a stack-based interpreter.) Therefore, to identify hard-coded strings, WASP simply scans the bytecode and identifies all load instructions whose operand is a string constant. WASP then instruments the code by adding, after each of these load instructions, code that creates an instance of a `MetaString` class by using the hard-coded string as an initialization parameter. Finally, because hard-coded strings are completely trusted, WASP adds to the code a call to the method of the newly created `MetaString` object that marks all characters as trusted. At runtime, polymorphism and dynamic binding allow this instance of the MetaString object to be used in any place where the original `String` object would have been used.

Fig. 6 shows an example of this bytecode transformation. The Java code at the top of the figure corresponds to line 4 of our servlet example (see Fig. 2), which creates one of the hard-coded strings in the servlet. Underneath, we show the original bytecode (left column) and the modified bytecode (right column). The modified bytecode contains additional instructions that 1) load a new `MetaString` object on the

Source Code: `4. String query = "SELECT acct FROM users WHERE login='";`

Original Bytecode	Modified Bytecode

`24. ldc "SELECT acct FROM users WHERE login='"`	`24a. new MetaString`
	`24b. dup`
	`24c. ldc "SELECT acct FROM users WHERE login='"`
	`24e. invokespecial MetaString.<init>:(LString)V`
	`24d. iconst_1`
	`24e. invokevirtual MetaString.markAll:(I)V`

Fig. 6. Instrumentation for hard-coded strings.

Source Code: `5. query += login + "' AND pin=" + pin;`

Original Bytecode	Modified Bytecode

`28. new StringBuilder`
`31. dup`
`32. aload 4`
`34. invokestatic String.valueOf:(Object)LString;`
`37. invokespecial StringBuilder.<init>:(LString;)V`
`40. aload_1`
`41. invokevirtual StringBuilder.`
`append:(LString;)LStringBuilder;`
`44. ldc "' AND pin="`
`46. invokevirtual StringBuilder.`
`append:(LString;)LStringBuilder;`
`49. aload_2`
`50. invokevirtual StringBuilder.`
`append:(LString;)LStringBuilder;`
`53. invokevirtual StringBuilder.toString:()LString;`
	`28. new MetaStringBuilder`
	`31. dup`
	`32. aload 4`
	`34. invokestatic String.valueOf:(LObject)LString;`
	`37. invokespecial MetaStringBuilder.<init>:(LString;)V`
	`40. aload_1`
	`41. invokevirtual StringBuilder.append:(LString;)LStringBuilder;`
	`44a. new MetaString`
	`44b. dup`
	`44c. ldc "' AND pin="`
	`44e. invokespecial MetaString.<init>:(LString)V`
	`44d. iconst_1`
	`44e. invokevirtual MetaString.markAll:(I)V`
	`46. invokevirtual StringBuilder.append:(LString;)LStringBuilder;`
	`49. aload_2`
	`50. invokevirtual StringBuilder.append:(LString;)LStringBuilder;`
	`53. invokevirtual StringBuilder.toString:()LString;`

Fig. 7. Instrumentation for implicitly created strings.

stack, 2) call the `MetaString` constructor by using the previous string as a parameter, and 3) call the method `markAll`, which assigns the given trust marking to all characters in the string.

4.2.2 Implicitly Created Strings

In Java programs, the creation of some string objects is implicitly added to the bytecode by the compiler. For example, Java compilers typically translate the string concatenation operator ("+") into a sequence of calls to the append method of a newly created `StringBuilder` object. WASP must replace these string objects with their corresponding MetaStrings objects so that they can maintain and propagate the trust markings of the strings on which they operate. To do this, WASP scans the bytecode for instructions that create new instances of the string classes used to perform string manipulation and modifies each such instruction so that it creates an instance of the corresponding MetaStrings class instead. In this situation, WASP does not associate any trust markings with the newly created MetaStrings objects. These objects are not trusted per se and they become marked only if the actual values assigned to them during execution are marked.

Fig. 7 shows the instrumentation added by WASP for implicitly created strings. The Java source code corresponds to line 5 in our example servlet. The `StringBuilder` object at offset 28 in the original bytecode is added by the Java compiler when translating the string concatenation operator ("+"). WASP replaces the instantiation at offset 28 with the instantiation of a `MetaStringBuilder` class and then changes the subsequent invocation of the constructor at offset 37 so that it matches the newly instantiated class. Because `MetaStringBuilder` extends `StringBuilder`, the subsequent calls to the append method invoke the correct method in the `MetaStringBuilder` class.

4.2.3 Strings from External Sources

To use query fragments that come from external (trusted) sources, developers must list these sources in a configuration file that WASP processes before instrumenting the application. The specified sources can be of different types such as files (specified by name), network connections (specified by host and port), and databases (specified by database name, table, field, or combination thereof). For each source, developers can either specify a custom trust marking or use the default trust marking (the same used for hard-coded strings). WASP uses the information in the configuration file to instrument the external trusted sources according to their type.

To illustrate this process, we describe the instrumentation that WASP performs for trusted strings that come from a file. In the configuration file, the developer specifies the name of the file (for example, `foo.txt`) as a trusted source of strings. Based on this information, WASP scans the bytecode for all instantiations of new file objects (that is, `File`, `FileInputStream`, and `FileReader`) and adds instrumentation that checks the name of the file being accessed. At runtime, if the name of the file matches the name(s) specified by the developer (`foo.txt` in this case), the file object is added to an internal list of currently trusted file objects. WASP also instruments all calls to methods of file stream objects that return strings such as the `Buffered`

Reader's readLine method. At runtime, the added code checks to see whether the object on which the method is called is in the list of currently trusted file objects. If so, it marks the generated strings with the trust marking specified by the developer for the corresponding source.

We use a similar strategy to mark network connections. In this case, instead of matching filenames at runtime, we match hostnames and ports. The interaction with databases is more complicated and requires WASP to not only match the initiating connection but also trace tables and fields through instantiations of the Statement and ResultSet objects created when querying the database.

Instrumentation optimization. Our current instrumentation approach is conservative and may generate unnecessary instrumentation. We could reduce the amount of instrumentation inserted in the code by leveraging static information about the program. For example, data-flow analysis could identify strings that are not involved with the construction of query strings and therefore do not need to be instrumented. A static analysis could also identify cases where the filename associated with a file object is never one of the developer-specified trusted filenames and avoid instrumenting that object and subsequent operations on it. Analogous optimizations could be implemented for other external sources. We did not incorporate any of these optimizations in the current tool because the overhead imposed by our current (conservative) implementation was insignificant in most of the cases.

4.3 Handling False Positives

As discussed in Section 3, sources of trusted data that are not specified by the developers beforehand would cause WASP to generate false positives. To assist the developers in identifying data sources that they initially overlooked, WASP provides a special mode of operation, called the "learning mode," that would typically be used during in-house testing. When in the learning mode, WASP adds an additional unique taint marking to each string in the application. Each marking consists of an ID that maps to the fully qualified class name, method signature, and bytecode offset of the instruction that instantiated the corresponding string.

If WASP detects an SQLIA while in the learning mode, it uses the markings associated with the untrusted SQL keywords and operators in the query to report the instantiation point of the corresponding string(s). If the SQLIA is a false positive, knowing the position in the code of the offending string(s) helps developers in correcting omissions in the set of trusted inputs.

4.4 Syntax-Aware Evaluation

The STRING CHECKER module performs syntax-aware evaluation of query strings and is invoked right before the strings are sent to the database. To add calls to the STRING CHECKER module, WASP first identifies all of the *database interaction points*, that is, points in the application where query strings are issued to an underlying database. In Java, all calls to the database are performed via specific methods and classes in the JDBC library (http://java.sun.com/products/jdbc/). Therefore, these points can be conservatively identified through a simple matching of method signatures. After identifying the database interaction points, WASP inserts a call to the syntax-aware evaluation function MetaChecker immediately before each interaction point. MetaChecker takes as a parameter the MetaStrings object that contains the query about to be executed.

When invoked, MetaChecker processes the SQL string about to be sent to the database, as discussed in Section 3.3. First, it tokenizes the string by using a SQL parser. Ideally, WASP would use a database parser that recognizes the exact same dialect of SQL that is used by the database. This would guarantee that WASP interprets the query in the same way as the database and would prevent attacks based on alternate encodings [1] (see Section 2.1.6). Our current implementation includes parsers for SQL-92 (ANSI) and PostgreSQL and allows for adding other parsers in a modular fashion. After tokenizing the query string, MetaChecker enforces the default trust policy by iterating through the tokens that correspond to keywords and operators and examining their trust markings. If any of these tokens contains characters that are not marked as trusted, an attack is identified. When MetaChecker identifies an attack, it can execute any developer-specified action. In our evaluation, we configured WASP so that it blocked the malicious query from executing and logged the attempted attack.

If developers specify additional trust policies, MetaChecker invokes the corresponding checking function(s) to ensure that the query complies with them. In our current implementation, trust policies are developer-defined functions that take the list of SQL tokens as input, check them based on their trust markings, and return a true or false value, depending on the outcome of the check. Trust policies can implement functionality that ranges from simple pattern matching to sophisticated checks that use externally supplied contextual information. If all custom trust policies return a positive outcome, WASP allows the query to be executed on the database. Otherwise, it identifies the query as an SQLIA.

We illustrate how the default policy for syntax-aware evaluation works by using our example servlet and the legitimate and malicious query examples from Section 2. For the servlet, there are no external sources of strings or additional trust policies, so WASP only marks the hard-coded strings as trusted and only the default trust policy is applied. Fig. 8 shows the sequence of tokens in the legitimate query as they would be parsed by MetaChecker. In this figure, SQL keywords and operators are surrounded by boxes. The figure also shows the trust markings associated with the strings, where an underlined character is a character with full trust markings. Because the default trust policy is that all keyword and operator tokens must have originated in trusted strings, MetaChecker simply checks whether all of these tokens are comprised of trusted characters. The query in Fig. 8 conforms to the trust policy and is thus allowed to execute on the database.

Consider the malicious query, where the attacker submits "admin' – –" as the login and "0" as the pin. Fig. 9 shows the sequence of tokens for the resulting query, together with the trust markings. Recall that "– –" is the SQL comment operator, so everything after this is identified by the parser as a literal. In this case, the MetaChecker would find that the last two tokens, ['] and [ ], contain untrusted characters. It would therefore identify the query as an SQLIA.

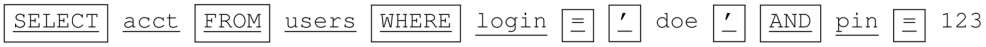


Fig. 8. Example query 1 after parsing by the runtime monitor.

Fig. 9. Example query 2 after parsing by the runtime monitor.

4.5 Deployment Requirements

Using WASP to protect a Web application requires the developer to run an instrumented version of the application. There are two general implementation strategies that we can follow for the instrumentation: offline and online. Offline instrumentation statically instruments the application and deploys the instrumented version of the application. Online instrumentation deploys an unmodified application and instruments the code at load time (that is, when classes are loaded by the JVM). This latter option allows for a great deal of flexibility and can be implemented by leveraging the new instrumentation package introduced in Java 5 (http://java.sun.com/j2se/1.5.0/). Unfortunately, the current implementation of the Java 5 instrumentation package is still incomplete and does not yet provide some key features needed by WASP. In particular, it does not allow for clearing the `final` flag in the string library classes, which prevents the MetaStrings library from extending them. Because of this limitation, for now, we have chosen to rely on offline instrumentation and insert into the Java library a version of the string classes in which the `final` flag has been cleared.

Overall, the deployment requirements for our approach are fairly lightweight. The modification of the Java library is performed only once, in a fully automated way, and takes just a few seconds. (Moreover, this modification is a temporary workaround for the current limitations of Java's instrumentation package.) No modification of the JVM is required. The instrumentation of a Web application is also automatically performed. Given the original application, WASP creates a deployment archive that contains the instrumented application, the MetaStrings library, and the string checker module. At this point, the archive can be deployed like any other Web application. WASP can therefore be easily and transparently incorporated into an existing build process.

5 Empirical Evaluation

In our evaluation, we assessed the effectiveness and efficiency of our approach. To do this, we used WASP to protect several real vulnerable Web applications while subjecting them to a large number of attacks and legitimate accesses and investigated three research questions:

- RQ1. What percentage of attacks can WASP detect and prevent that would otherwise go undetected and reach the database?
- RQ2. What percentage of legitimate accesses are incorrectly identified by WASP as attacks?
- RQ3. What is the runtime overhead imposed by WASP on the Web applications that it protects?

The first two questions deal with the effectiveness of the technique: RQ1 investigates the false-negative rate of the technique and RQ2 investigates the false-positive rate. RQ3 deals with the efficiency of the proposed technique. The next sections discuss our experiment setup, protocol, and results.

5.1 Experiment Setup

The framework that we use for our experiments consists of a set of vulnerable Web applications, a large set of test inputs that contain both legitimate accesses and SQLIAs, and monitoring and logging tools. We developed the initial framework in our previous work [11] and it has since been used both by us and by other researchers [10], [27]. In this study, we have expanded the framework by 1) including additional open source Web applications with known vulnerabilities, 2) generating legitimate and malicious inputs for these new applications, and 3) expanding the set of inputs for the existing applications. In the next two sections, we discuss the Web applications and the set of inputs used in our experiments in more detail.

5.1.1 Software Subjects

Our set of software subjects consists of 10 Web applications that are known to be vulnerable to SQLIAs. Five of the applications are commercial applications that we obtained from GotoCode (http://www.gotocode.com/): Employee Directory, Bookstore, Events, Classifieds, and Portal. Two applications, OfficeTalk and Checkers, are student-developed applications that have been used in a related work [8]. Two other applications, Daffodil and Filelister, are open source applications that have been identified in the Open Source Vulnerability Database (http://osvdb.org/, entries 22879 and 21416) as containing one or more SQL injection vulnerabilities. The last subject, WebGoat, is a purposely insecure Web application that was developed by the Open Web Application Security Project (http://www.owasp.org/) to demonstrate common Web application vulnerabilities. Among these 10 subjects, the first seven applications contain a wide range of vulnerabilities, whereas the last three contain specific and known SQLIA vulnerabilities.

Table 1 provides summary information about each of the subjects in our evaluation. It shows, for each subject, its size (LOC), number of database interaction points (DBIs), number of vulnerable servlets (Vuln Servlets), and total number of servlets (Total Servlets). We considered all of the servlets in the first seven subjects that accepted user input to be potentially vulnerable because we had no initial information about their vulnerabilities. For the remaining three applications, we considered as vulnerable only those servlets with specific and known vulnerabilities.

5.1.2 Malicious and Legitimate Inputs

For each of the Web applications considered, we created two sets of inputs: LEGIT, which consists of legitimate inputs for the application, and ATTACK, which consists of SQLIAs.

TABLE 1
Subject Programs for the Empirical Study

Subject	LOC	DBIs	Servlets	
			Vuln	Total
Checkers	5,415	5	33	33
Office Talk	4,670	40	38	38
Employee Directory	5,529	23	8	11
Bookstore	19,402	71	25	28
Events	7,164	31	11	13
Classifieds	10,702	34	16	19
Portal	16,089	67	25	28
Daffodil	18,706	156	1	69
Filelister	8,671	10	1	10
WebGoat	20,725	184	4	27

TABLE 2
Results of Testing for False Negatives (RQ1)

Subject	Total # Attacks	Successful Attacks	
		Original Web Apps	WASP Protected Web Apps
Checkers	4,431	922	0
Office Talk	5,888	499	0
Empl. Dir.	6,398	2,066	0
Bookstore	6,154	1,999	0
Events	6,207	2,141	0
Classifieds	5,968	1,973	0
Portal	6,403	3,016	0
Daffodil	19	19	0
Filelister	96	80	0
WebGoat	96	88	0

To create the ATTACK sets, we employed a Master's level student with experience in developing commercial penetration testing tools. The student first assembled a list of actual SQLIAs by surveying different sources: exploits developed by professional penetration-testing teams to take advantage of SQL-injection vulnerabilities, online vulnerability reports such as US-CERT (http://www.us-cert.gov/) and CERT/CC Advisories (http://www.cert.org/advisories/), and information extracted from several security-related mailing lists. The resulting set of attack strings contained 24 unique attacks. All types of attacks reported in the literature [12] were represented in this set, except for multiphase attacks such as second-order injections. Since multiphase attacks require human intervention and interpretation, we omitted them to keep our testbed fully automated. These attack strings were then used to build inputs for all of the vulnerable servlets in each application. The resulting ATTACK sets contained a broad range of potential SQLIAs.

The LEGIT sets were created in a similar fashion. However, instead of using attack strings to generate sets of inputs, the student used legitimate values. To create "interesting" legitimate values, we asked the student to generate input strings that, although legal, would stress and possibly break naive SQLIA detection techniques (for example, techniques based on simple identification of keywords or special characters in the input). For instance, the legitimates values contained SQL keywords (for example, "SELECT" and "DROP"), query fragments (for example, "or 1¼1"), and properly escaped SQL operators (for example, the single quote ' ' and the percent sign n%). These values were used to build inputs for the vulnerable servlets that looked "suspicious" without actually resulting in an SQLIA.

5.2 Experimental Protocol

RQ1 addresses the issue of false negatives. To investigate this question, we 1) ran the inputs in the ATTACK sets against our subject applications and 2) tracked the result of each attack to check whether it was detected and prevented by WASP. The results of this evaluation are shown in Table 2. The second column reports the total number of attacks in the application's ATTACK set. The next two columns show the number of attacks that were successful against the original unprotected Web application and the number of attacks that were successful on the application protected using WASP. The reason that some attacks were not successful on the unprotected applications is twofold. First, not all of the 24 attack strings represented viable attacks against all vulnerable servlets. Second, many of the applications performed some type of input validation that could catch and prevent a subset of the attempted attacks.

RQ2 deals with false positives. To address this question, we ran all of the test inputs in each application's LEGIT set against the application. As before, we tracked the result of each of these legitimate accesses to see if WASP reported it as an attack, which would be a false positive. The results for this evaluation are summarized in Table 3. The table shows the number of legitimate accesses that WASP allows to execute (# Legitimate Accesses) and the number of accesses blocked by WASP (False Positives).

To address RQ3, we measured the overhead incurred by applications that were protected using WASP. To do this, we measured and compared the times needed to run the LEGIT set against a protected version and an unprotected version of each application. We used only the LEGIT set for this part of the study because our current implementation of WASP terminates the execution when it detects an attack, which would have made the total execution time for the WASP-protected version faster than the time for the normal version. To reduce problems with the precision of the timing measurements, we measured the total time that it

TABLE 3
Results of Testing for False Positives (RQ2)

Subject	# Legitimate Accesses	False Positives
Checkers	1,359	0
Office Talk	424	0
Empl. Dir.	1,244	0
Bookstore	3,239	0
Events	1,324	0
Classifieds	2042	0
Portal	3,435	0
Daffodil	19	0
Filelister	40	0
WebGoat	40	0

TABLE 4
Overhead Measurements for the Macro Benchmarks (RQ3)

Subject	*# Inputs*	*Avg Time Uninst (ms)*	*Avg Ovhd (ms)*	*% Ovhd*
Checkers	1,359	122	5	5%
Office Talk	424	56	1	2%
Empl. Dir.	658	63	3	5%
Bookstore	607	70	4	6%
Events	900	70	1	1%
Classifieds	574	70	3	5%
Portal	1,080	83	16	19%
Daffodil	19	90	6	6%
Filelister	40	172	1	1%
WebGoat	40	940	40	5%

took to run the entire LEGIT set against an application, instead of the single times for each input in the set, and divided this time by the number of accesses to get an average value. In addition, to account for possible external factors beyond our control, such as network traffic or other OS activities, we repeated these measurements 100 times for each application and averaged the results. All measurements were performed on two machines that act as client and server. The client was a 2.4 GHz Pentium 4 with 1 Gbyte memory, running GNU/Linux 2.4. The server was a 3.0 GHz dual-processor Pentium D with 2 Gbyte memory, running GNU/Linux 2.6.

In addition to measuring the overhead for these macro benchmarks, we also measured the overhead imposed by individual MetaStrings methods on a set of micro benchmarks. To do this, we first identified the methods most commonly used in the subject Web applications, seven methods overall. We then measured, for each method m, the runtime of 1) a driver that performed 10,000 calls to the original m and 2) a driver that performed the same number of calls to the MetaStrings version of m. In addition, in this case, we performed the measurements 100 times and averaged the results. This second set of measurements was also performed on a 3.0 GHz dual-processor Pentium D with 2 Gbyte memory, running GNU/Linux 2.6.

Table 4 shows the results of the timing measurements for the macro benchmarks. For each subject, the table reports the number of inputs in the LEGIT set (# Inputs), the average time per access for the uninstrumented Web application (Avg Time Uninst), the average time overhead per access for the instrumented version (Avg Overhead), and the average time overhead as a percentage (% Overhead). In the table, all absolute times are expressed in milliseconds. Fig. 10 provides another view of the timing measurements by using a bar chart. In this figure, the total servlet access times for the instrumented and uninstrumented versions are shown side by side. As the figure shows, the difference between the two bars for a subject, which represents the WASP overhead, is small in both relative and absolute terms.

Table 5 reports the results of the timing measurements for the micro benchmarks. For each of the seven methods considered, the table shows the average runtime, in milliseconds, for 10,000 executions of the original method and of its MetaStrings version. Note that the measured overhead is due to either the creation and initialization of a new `Set` object for each character (for the default constructors) or the copying of the trust markings from one object to another (for the parameterized constructors and for the methods append and `concat`). Although the measured overhead is considerable in relative terms, it is mostly negligible in absolute terms. In the worst case, for method `StringBuilder.Append(StringBuilder)`, the MetaStrings version of the method takes 71 ms more than its original version for 10,000 executions.

5.3 Discussion of the Experimental Results

The results of our evaluation show that, overall, WASP is an effective technique for preventing SQLIAs. In our

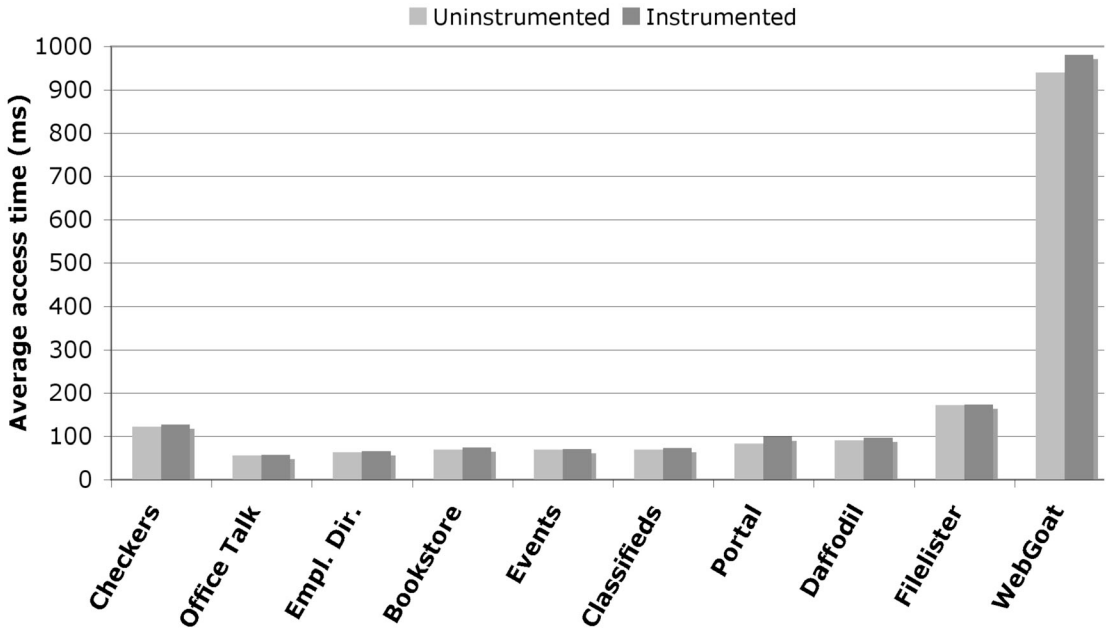


Fig. 10. Runtime overhead imposed by WASP's instrumentation on the subject Web applications.

TABLE 5
Overhead Measurements for the Micro Benchmarks (RQ3)

Method *Method*	*Original method*	*MetaStrings version*
String.Concat(String)	24	65
String.New()	4	9
String.New(String)	3	22
StringBuilder.Append(StringBuilder)	1	72
StringBuilder.Append(String)	2	52
StringBuilder.New()	2	21
StringBuilder.New(String)	2	16

evaluation, WASP was able to correctly identify all SQLIAs while generating no false positives. In total, WASP stopped 12,826 viable SQLIAs without preventing any of the 13,166 legitimate accesses from executing.

The overhead imposed by WASP was also relatively low. For the 10 applications, the average overhead was about 8 ms (5.5 percent). For most Web applications, this cost is low enough that it would be dominated by the cost of the network and database accesses. Furthermore, we believe that, by using some of the optimizations discussed in Section 4.2.3, it would be possible to lower this number even further, if deemed necessary, after performing more experimentation. Portal, the application that incurred the highest overhead, is an example of an application that would benefit enormously from these optimizations. Portal generates a large number of string-based lookup tables. Although these strings are not used to build queries, WASP associates trust markings with them and propagates these markings at runtime. In this specific case, a simple dependency analysis would be able to determine that these markings are unnecessary and avoid the overhead associated with these operations.

Like all empirical studies, our evaluation has limitations that may affect the external and internal validity of its results. The primary threat to the external validity of the results is that the attacks and applications used in our studies may not be representative of real-world applications and attacks. To mitigate this issue, we have included in our set of subjects Web applications that come from a number of different sources and were developed using different approaches (for example, the five GotoCode applications are developed using an approach that is based on automated code generation). In addition, our set of attacks was independently developed by a Master's level student who had considerable experience with SQLIAs and penetration testing but was not familiar with our technique. Finally, the attack strings used by the student as a basis for the generation of the attacks were based on real-world SQLIAs.

For this study, threats to internal validity mainly concern errors in our implementation or in our measurement tools that could affect outcomes. To control these threats, we validated the implementations and tools on small-scale examples and performed a considerable amount of spot checking for some of the individual results.

6 Related Work

The use of dynamic tainting to prevent SQLIAs has been investigated by several researchers. The two approaches most similar to ours are those by Nguyen-Tuong et al. [23] and Pietraszek and Berghe [24]. Similarly, we track taint information at the character level and use a syntax-aware evaluation to examine tainted input. However, our approach differs from theirs in several important aspects. First, our approach is based on the novel concept of positive tainting, which is an inherently safer way of identifying trusted data (see Section 3.1). Second, we improve on the idea of syntax-aware evaluation by 1) using a database parser to interpret the query string before it is executed, thereby ensuring that our approach can handle attacks based on alternate encodings, and 2) providing a flexible mechanism that allows different trust policies to be associated with different input sources. Finally, a practical advantage of our approach is that it has more lightweight deployment requirements. Their approaches require the use of a customized PHP runtime interpreter, which adversely affects the portability of the approaches.

Other dynamic tainting approaches more loosely related to our approach are those by Haldar et al. [9] and Martin et al. [20]. Although they also propose dynamic tainting approaches for Java-based applications, their techniques significantly differ from ours. First, they track taint information at the level of granularity of strings, which introduces imprecision in modeling string operations. Second, they use declassification rules, instead of syntax-aware evaluation, to assess whether a query string contains an attack. Declassification rules assume that sanitizing functions are always effective, which is an unsafe assumption and may leave the application vulnerable to attacks. In many cases, attack strings can pass through sanitizing functions and may still be harmful. Another dynamic tainting approach, proposed by Newsome and Song [22], focuses on tainting at a level that is too low to be used for detecting SQLIAs and has a very high execution overhead. Xu et al. [31] propose a generalized tainting mechanism that can address a wide range of input-validation-related attacks, targets C programs, and works by instrumenting the code at the source level. Their approach can be considered a framework for performing dynamic taint analysis on C programs. As such, it could be leveraged to implement a version of our approach for C-based Web applications.

Researchers also proposed dynamic techniques against SQLIAs that do not rely on tainting. These techniques include Intrusion Detection Systems (IDSs) and automated penetration testing tools. Scott and Sharp propose Security Gateway [26], which uses developer-provided rules to filter Web traffic, identify attacks, and apply preventive transformations to potentially malicious inputs. The success of this approach depends on the ability of developers to write accurate and meaningful filtering rules. Similarly, Valeur et al. [28] developed an IDS that uses machine learning to distinguish legitimate and malicious queries. Their approach, like most learning-based techniques, is limited by the quality of the IDS training set. Machine learning was also used in WAVES [14], an automated penetration testing tool that probes Web sites for vulnerability to SQLIAs. Like all testing tools, WAVES cannot provide any guarantees of completeness. SQLrand [2] appends a random token to SQL keywords and operators in the application code. A proxy

server then checks to make sure that all keywords and operators contain this token before sending the query to the database. Because the SQL keywords and operators injected by an attacker would not contain this token, they would be easily recognized as attacks. The drawbacks of this approach are that the secret token could be guessed, thus making the approach ineffective, and that the approach requires the deployment of a special proxy server.

Model-based approaches against SQLIAs include AMNESIA [11], SQL-Check [27], and SQLGuard [3]. AMNESIA, previously developed by two of the authors, combines static analysis and runtime monitoring to detect SQLIAs. The approach uses static analysis to build models of the different types of queries that an application can generate and dynamic analysis to intercept and check the query strings generated at runtime against the model. Queries that do not match the model are identified as SQLIAs. A problem with this approach is that it is dependent on the precision and efficiency of its underlying static analysis, which may not scale to large applications. Our new technique takes a purely dynamic approach to preventing SQLIAs, thereby eliminating scalability and precision problems. SQLCheck [27] identifies SQLIAs by using an augmented grammar and distinguishing untrusted inputs from the rest of the strings by means of a marking mechanism. The main weakness of this approach is that it requires the manual intervention of the developer to identify and annotate untrusted sources of input, which introduces incompleteness problems and may lead to false negatives. Our use of positive tainting eliminates this problem while providing similar guarantees in terms of effectiveness. SQLGuard [3] is an approach similar to SQLCheck. The main difference is that SQLGuard builds its models on the fly by requiring developers to call a special function and to pass to the function the query string before user input is added.

Other approaches against SQLIAs rely purely on static analysis [15], [16], [17], [18], [30]. These approaches scan the application and leverage information flow analysis or heuristics to detect code that could be vulnerable to SQLIAs. Because of the inherently imprecise nature of the static analysis that they use, these techniques can generate false positives. Moreover, since they rely on declassification rules to transform untrusted input into safe input, they can also generate false negatives. Wassermann and Su propose a technique [29] that combines static analysis and automated reasoning to detect whether an application can generate queries that contain tautologies. This technique is limited, by definition, in the types of SQLIAs that it can detect.

Finally, researchers have investigated ways to statically eliminate vulnerabilities from the code of a Web application. Defensive coding best practices [13] have been proposed as a possible approach, but they have limited effectiveness because they rely almost exclusively on the ability and training of developers. Moreover, there are many well-known ways to evade some defensive-coding practices, including "pseudoremedies" such as stored procedures and prepared statements (for example, [1], [13], [19]). Researchers have also developed special libraries that can be used to safely create SQL queries [5], [21]. These approaches, although highly effective, require developers to learn new APIs, can be very expensive to apply on legacy code, and sometimes limit the expressiveness of SQL. Finally, JDBC-Checker [7], [8] is a static analysis tool that detects potential type mismatches in dynamically generated queries. Although it was not intended to prevent SQLIAs, JDBC-Checker can be effective against SQLIAs that leverage vulnerabilities due to type mismatches, but will not be able to prevent other kinds of SQLIAs.

7 Conclusion

This paper presented a novel highly automated approach for protecting Web applications from SQLIAs. Our approach consists of 1) identifying trusted data sources and marking data coming from these sources as trusted, 2) using dynamic tainting to track trusted data at runtime, and 3) allowing only trusted data to form the semantically relevant parts of queries such as SQL keywords and operators. Unlike previous approaches based on dynamic tainting, our technique is based on positive tainting, which explicitly identifies trusted (rather than untrusted) data in a program. This way, we eliminate the problem of false negatives that may result from the incomplete identification of all untrusted data sources. False positives, although possible in some cases, can typically be easily eliminated during testing. Our approach also provides practical advantages over the many existing techniques whose application requires customized and complex runtime environments: It is defined at the application level, requires no modification of the runtime system, and imposes a low execution overhead.

We have evaluated our approach by developing a prototype tool WASP and using the tool to protect 10 applications when subjected to a large and varied set of attacks and legitimate accesses. WASP successfully and efficiently stopped over 12,000 attacks without generating any false positives. Both our tool and the experimental infrastructure are available to other researchers.

We have two immediate goals for future work. First, we will extend our experimental results by using WASP to protect actually deployed Web applications. Our first target will be a set of Web applications that run at Georgia Tech. This will allow us to assess the effectiveness of WASP in real settings and also to collect a valuable set of real legal accesses and, possibly, attacks. Second, we will implement the approach for binary applications. We have already started developing the infrastructure to perform tainting at the binary level and developed a proof-of-concept prototype [4].

Acknowledgments

This work was supported by US National Science Foundation Awards CCF-0438871 and CCF-0541080 to Georgia Tech and by the US Department of Homeland Security and US Air Force under Contract FA8750-05-2-0214. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the US Air Force. The anonymous reviewers provided useful feedback that helped improve the quality of this paper.

References

[1] C. Anley, "Advanced SQL Injection In SQL Server Applications," white paper, Next Generation Security Software, 2002.

[2] S.W. Boyd and A.D. Keromytis, "SQLrand: Preventing SQL Injection Attacks," Proc. Second Int'l Conf. Applied Cryptography and Network Security, pp. 292-302, June 2004.

[3] G.T. Buehrer, B.W. Weide, and P.A.G. Sivilotti, "Using Parse Tree Validation to Prevent SQL Injection Attacks," Proc. Fifth Int'l Workshop Software Eng. and Middleware, pp. 106-113, Sept. 2005.

[4] J. Clause, W. Li, and A. Orso, "Dytan: A Generic Dynamic Taint Analysis Framework," Proc. Int'l Symp. Software Testing and Analysis, pp. 196-206, July 2007.

[5] W.R. Cook and S. Rai, "Safe Query Objects: Statically Typed Objects as Remotely Executable Queries," Proc. 27th Int'l Conf. Software Eng., pp. 97-106, May 2005.

[6] "Top Ten Most Critical Web Application Vulnerabilities," OWASP Foundation, http://www.owasp.org/documentation/topten.html, 2005.

[7] C. Gould, Z. Su, and P. Devanbu, "JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications," Proc. 26th Int'l Conf. Software Eng., formal demos, pp. 697-698, May 2004.

[8] C. Gould, Z. Su, and P. Devanbu, "Static Checking of Dynamically Generated Queries in Database Applications," Proc. 26th Int'l Conf. Software Eng., pp. 645-654, May 2004.

[9] V. Haldar, D. Chandra, and M. Franz, "Dynamic Taint Propagation for Java," Proc. 21st Ann. Computer Security Applications Conf., pp. 303-311, Dec. 2005.

[10] W. Halfond, A. Orso, and P. Manolios, "Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks," Proc. ACM SIGSOFT Symp. Foundations of Software Eng., pp. 175-185, Nov. 2006.

[11] W.G. Halfond and A. Orso, "AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks," Proc. 20th IEEE and ACM Int'l Conf. Automated Software Eng., pp. 174-183, Nov. 2005.

[12] W.G. Halfond, J. Viegas, and A. Orso, "A Classification of SQL-Injection Attacks and Countermeasures," Proc. IEEE Int'l Symp. Secure Software Eng., Mar. 2006.

[13] M. Howard and D. LeBlanc, Writing Secure Code, second ed. Microsoft Press, 2003.

[14] Y. Huang, S. Huang, T. Lin, and C. Tsai, "Web Application Security Assessment by Fault Injection and Behavior Monitoring," Proc. 12th Int'l Conf. World Wide Web, pp. 148-159, May 2003.

[15] 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," Proc. 13th Int'l Conf. World Wide Web, pp. 40-52, May 2004.

[16] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities," Proc. IEEE Symp. Security and Privacy, May 2006.

[17] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise Alias Analysis for Static Detection of Web Application Vulnerabilities," Proc. Workshop Programming Languages and Analysis for Security, pp. 27-36, June 2006.

[18] V.B. Livshits and M.S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis," Proc. 14th Usenix Security Symp., Aug. 2005.

[19] O. Maor and A. Shulman, "SQL Injection Signatures Evasion," white paper, Imperva, http://www.imperva.com/application_defense_center/white_papers/sql_injection_signatures_evasion.html, Apr. 2004.

[20] M. Martin, B. Livshits, and M.S. Lam, "Finding Application Errors and Security Flaws Using PQL: A Program Query Language," Proc. 20th Ann. ACM SIGPLAN Conf. Object Oriented Programming Systems Languages and Applications, pp. 365-383, Oct. 2005.

[21] R. McClure and I. Krüger, "SQL DOM: Compile Time Checking of Dynamic SQL Statements," Proc. 27th Int'l Conf. Software Eng., pp. 88-96, May 2005.

[22] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," Proc. 12th Ann. Network and Distributed System Security Symp., Feb. 2005.

[23] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically Hardening Web Applications Using Precise Tainting Information," Proc. 20th IFIP Int'l Information Security Conf., May 2005.

[24] T. Pietraszek and C.V. Berghe, "Defending against Injection Attacks through Context-Sensitive String Evaluation," Proc. Eighth Int'l Symp. Recent Advances in Intrusion Detection, Sept. 2005.

[25] J. Saltzer and M. Schroeder, "The Protection of Information in Computer Systems," Proc. Fourth ACM Symp. Operating System Principles, Oct. 1973.

[26] D. Scott and R. Sharp, "Abstracting Application-Level Web Security," Proc. 11th Int'l Conf. World Wide Web, pp. 396-407, May 2002.

[27] Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in Web Applications.," Proc. 33rd Ann. Symp. Principles of Programming Languages, pp. 372-382, Jan. 2006.

[28] F. Valeur, D. Mutz, and G. Vigna, "A Learning-Based Approach to the Detection of SQL Attacks," Proc. Conf. Detection of Intrusions and Malware and Vulnerability Assessment, July 2005.

[29] G. Wassermann and Z. Su, "An Analysis Framework for Security in Web Applications," Proc. FSE Workshop Specification and Verification of Component-Based Systems, pp. 70-78, Oct. 2004.

[30] Y. Xie and A. Aiken, "Static Detection of Security Vulnerabilities in Scripting Languages," Proc. 15th Usenix Security Symp., Aug. 2006.

[31] W. Xu, S. Bhatkar, and R. Sekar, "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks," Proc. 15th Usenix Security Symp., Aug. 2006.

William G.J. Halfond received the BS degree in computer science from the University of Virginia, Charlottesville, and the MS degree in computer science from the Georgia Institute of Technology, Atlanta, in May 2004. He is currently working toward the PhD degree in computer science in the College of Computing at the Georgia Institute of Technology. His research work is in software engineering and security, with emphasis on techniques that can be applied to improve the security and survivability of large-scale computer systems.

Alessandro Orso received the MS degree in electrical engineering and the PhD degree in computer science from the Politecnico di Milano, Italy, in 1995 and 1999, respectively. He was a visiting researcher in the Department of Electrical Engineering and Computer Science at the University of Illinois at Chicago in 1999. Since March 2000, he has been with the College of Computing at the Georgia Institute of Technology, first as a research faculty member and then as an assistant professor. His area of research is software engineering, with emphasis on software testing and program analysis. His research interests include the development of techniques and tools for improving software reliability, security, and trustworthiness and the validation of such techniques on real systems. He is a member of the IEEE Computer Society.

Panagiotis Manolios received the BS and MA degrees in computer science from Brooklyn College, New York, in 1991 and 1992, respectively, and the PhD degree in computer science from the University of Texas at Austin in 2001. He joined the College of Computing at the Georgia Institute of Technology in 2001. He became an adjunct assistant professor in the School of Electrical and Computer Engineering at the Georgia Institute of Technology in 2003. He is currently an associate professor at Northeastern University. His main research interest is mechanized formal verification and validation. His other areas of interest include programming languages, distributed computing, logic, software engineering, algorithms, computer architecture, aerospace, and pedagogy. He is a member of the IEEE Computer Society.

Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks

William G.J. Halfond, Alessandro Orso, and Panagiotis Manolios
College of Computing – Georgia Institute of Technology
{whalfond, orso, manolios}@cc.gatech.edu

ABSTRACT

SQL injection attacks pose a serious threat to the security of Web applications because they can give attackers unrestricted access to databases that contain sensitive information. In this paper, we propose a new, highly automated approach for protecting existing Web applications against SQL injection. Our approach has both conceptual and practical advantages over most existing techniques. From the conceptual standpoint, the approach is based on the novel idea of positive tainting and the concept of syntax-aware evaluation. From the practical standpoint, our technique is at the same time precise and efficient and has minimal deployment requirements. The paper also describes WASP, a tool that implements our technique, and a set of studies performed to evaluate our approach. In the studies, we used our tool to protect several Web applications and then subjected them to a large and varied set of attacks and legitimate accesses. The evaluation was a complete success: WASP successfully and efficiently stopped all of the attacks without generating any false positives.

Categories and Subject Descriptors: D.2.0 [Software Engineering]: General—*Protection mechanisms*;

General Terms: Security

Keywords: SQL injection, dynamic tainting, runtime monitoring

1. INTRODUCTION

SQL injection attacks (SQLIAs) are one of the major security threats for Web applications [5]. Successful SQLIAs can give attackers access to and even control of the databases that underly Web applications, which may contain sensitive or confidential information. Despite the potential severity of SQLIAs, many Web applications remain vulnerable to such attacks.

In general, SQL injection vulnerabilities are caused by inadequate input validation within an application. Attackers take advantage of these vulnerabilities by submitting input strings that contain specially-encoded database commands to the application. When the application builds a query using these strings and submits the query to its underlying database, the attacker's embedded commands are executed by the database, and the attack succeeds. Although this general mechanism is well understood, straightforward solutions based on defensive coding practices have been less than successful for several reasons. First, it is difficult to implement and enforce a rigorous defensive coding discipline. Second, many solutions based on defensive coding address only a subset of the possible attacks. Finally, defensive coding is problematic in the case of legacy software because of the cost and complexity of retrofitting existing code. Researchers have proposed a wide range of alternative techniques to address SQLIAs, but many of these solutions have limitations that affect their effectiveness and practicality.

In this paper we propose a new, highly automated approach for dynamic detection and prevention of SQLIAs. Intuitively, our approach works by identifying "trusted" strings in an application and allowing only these trusted strings to be used to create certain parts of an SQL query, such as keywords or operators. The general mechanism that we use to implement this approach is based on dynamic tainting, which marks and tracks certain data in a program at runtime.

The kind of dynamic tainting we use gives our approach several important advantages over techniques based on different mechanisms. Many techniques rely on complex static analyses in order to find potential vulnerabilities in code (*e.g.*, [9, 15, 26]). These kinds of conservative static analyses can generate high rates of false positives or may have scalability issues when applied to large, complex applications. Our approach does not rely on complex static analyses and is very efficient and precise. Other techniques involve extensive human effort (*e.g.*, [4, 18, 24]). They require developers to manually rewrite parts of their applications, build queries using special libraries, or mark all points in the code at which malicious input could be introduced. In contrast, our approach is highly automated and in most cases requires minimal or no developer intervention. Lastly, several proposed techniques require the deployment of extensive infrastructure or involve complex configurations (*e.g.*, [2, 23, 25]). Our approach does not require additional infrastructure and can be deployed automatically.

Compared to other existing techniques based on dynamic tainting (*e.g.*, [8, 20, 21]), our approach makes several conceptual and practical improvements that take advantage of the specific characteristics of SQLIAs. The *first conceptual advantage* of our approach is the use of positive tainting. Positive tainting identifies and tracks trusted data, whereas traditional ("negative") tainting focuses on untrusted data. In the context of SQLIAs, there are several reasons why positive tainting is more effective than negative tainting. First, in Web applications, trusted data sources can be more easily and accurately identified than untrusted data sources; therefore, the use of positive tainting leads to increased automation.

SIGSOFT'06/FSE-14, November 5–11, 2006, Portland, Oregon, USA.

Second, the two approaches differ significantly in how they are affected by incompleteness. With negative tainting, failure to identify the complete set of untrusted data sources would result in false negatives, that is, successful undetected attacks. With positive tainting, conversely, missing trusted data sources would result in false positives, which are undesirable, but whose presence can be detected immediately and easily corrected. In fact, we expect that most false positives would be detected during pre-release testing. The *second conceptual advantage* of our approach is the use of flexible syntax-aware evaluation, which gives developers a mechanism to regulate the usage of string data based not only on its source, but also on its syntactical role in a query string. In this way, developers can use a wide range of external input sources to build queries, while protecting the application from possible attacks introduced via these sources.

The practical advantages of our approach are that it imposes a low overhead on the application and has minimal deployment requirements. Efficiency is achieved by using a specialized library, called MetaStrings, that accurately and efficiently assigns and tracks trust markings at runtime. The only deployment requirements for our approach are that the Web application must be instrumented and deployed with our MetaStrings library, which is done automatically. The approach does not require any customized runtime system or additional infrastructure.

In this paper, we also present the results of an extensive empirical evaluation of the effectiveness and efficiency of our technique. To perform this evaluation, we implemented our approach in a tool called WASP (Web Application SQL-injection Preventer) and evaluated WASP on a set of seven Web applications of various types and sizes. For each application, we protected it with WASP, targeted it with a large set of attacks and legitimate accesses, and assessed the ability of our technique to detect and prevent attacks without stopping legitimate accesses. The results of the evaluation are promising; our technique was able to stop all of the attacks without generating false positives for any of the legitimate accesses. Moreover, our technique proved to be efficient, imposing only a low overhead on the Web applications.

The main contributions of this work are:

- A new, automated technique for preventing SQLIAs based on the novel concept of positive tainting and on flexible syntax-aware evaluation.
- A mechanism to perform efficient dynamic tainting of Java strings that precisely propagates trust markings while strings are manipulated at runtime.
- A tool that implements our SQLIA prevention technique for Java-based Web applications and has minimal deployment requirements.
- An empirical evaluation of the technique that shows its effectiveness and efficiency.

The rest of this paper is organized as follows. In Section 2, we introduce SQLIAs with an example that is used throughout the paper. Sections 3 and 4 discuss the approach and its implementation. Section 5 presents the results of our evaluation. We discuss related work in Section 6 and conclude in Section 7.

2. SQL INJECTION ATTACKS

Intuitively, an SQL Injection Attack (SQLIA) occurs when an attacker changes the developer's intended structure of an SQL command by inserting new SQL keywords or operators. (Su and Wassermann provide a formal definition of SQLIAs in [24].) SQLIAs leverage a wide range of mechanisms and input channels to inject

```
1. String login = getParameter("login");
2. String pin = getParameter("pin");
3. Statement stmt = connection.createStatement();
4. String query = "SELECT acct FROM users WHERE login='";
5. query += login + "' AND pin=" + pin;
6. ResultSet result = stmt.executeQuery(query);
7. if (result != null)
8.    displayAccount(result); // Show account
9. else
10.   sendAuthFailed(); // Authentication failed
```

Figure 1: Excerpt of a Java servlet implementation.

malicious commands into a vulnerable application [10]. In this section we introduce an example application that contains an SQL injection vulnerability and show how an attacker can leverage the vulnerability to perform an SQLIA. Note that the example represents an extremely simple kind of attack, and we present it for illustrative purposes only. Interested readers may refer to References [1] and [10] for further examples of the different types of SQLIAs.

The code excerpt in Figure 1 represents the implementation of login functionality that we can find in a typical Web application. This type of login function would commonly be part of a Java *servlet*, a type of Java application that runs on a Web application server, and whose execution is triggered by the submission of a URL from a user of the Web application. The servlet in the example uses the input parameters `login` and `pin` to dynamically build an SQL query or command.[1] The `login` and `pin` are checked against the credentials stored in the database. If they match, the corresponding user's account information is returned. Otherwise, a null set is returned by the database and the authentication fails. The servlet then uses the response from the database to generate HTML pages that are sent back to the user's browser by the the Web server.

Given the servlet code, if a user submits `login` and `pin` as "`doe`" and "`123`," the application dynamically builds the query:

```
SELECT acct FROM users WHERE login='doe' AND pin=123
```

If `login` and `pin` match the corresponding entry in the database, `doe`'s account information is returned and then displayed by function `displayAccount()`. If there is no match in the database, function `sendAuthFailed()` displays an appropriate error message. An application that uses this servlet is vulnerable to SQLIAs. For example, if an attacker enters "`admin' --`" as the user name and any value as the pin (*e.g.*, "0"), the resulting query is:

```
SELECT acct FROM users WHERE login='admin' -- ' AND pin=0
```

In SQL, "`--`" is the comment operator, and everything after it is ignored. Therefore, when performing this query, the database simply searches for an entry where `login` is equal to `admin` and returns that database record. After the "successful" login, the function `displayAccount()` would therefore reveal the `admin`'s account information to the attacker.

3. OUR APPROACH

Our approach is based on dynamic tainting, which has been widely used to address security problems related to input validation. Traditional dynamic tainting approaches mark certain untrusted data (typically, user input) as tainted, track the flow of tainted data at runtime, and prevent this data from being used in potentially harmful ways. Our approach makes several conceptual and practical improvements over traditional dynamic-tainting approaches by taking advantage of the characteristics of SQLIAs. First, unlike any existing dynamic tainting techniques that we are aware of, our ap-

[1] For simplicity, in the rest of this paper we use the terms query and command interchangeably.

proach is based on the novel concept of *positive tainting*—the identification and marking of trusted instead of untrusted data. Second, our approach performs *accurate taint propagation* by precisely tracking trust markings at the character level. Third, it performs *syntax-aware evaluation* of query strings before they are sent to the database and blocks all queries whose non-literal parts (*i.e.*, SQL keywords and operators) contain one or more characters without trust markings. Finally, our approach has *minimal deployment requirements*, which makes it both practical and portable. The following sections discuss the key features of our approach in detail.

3.1 Positive Tainting

Positive tainting differs from traditional tainting (hereafter, *negative tainting*) because it is based on the identification, marking, and tracking of trusted, rather than untrusted, data. This conceptual difference has significant implications for the effectiveness of our approach, in that it helps address problems caused by incompleteness in the identification of relevant data to be marked. Incompleteness, which is one of the major challenges when implementing a security technique based on dynamic tainting, has very different consequences in negative and positive tainting. In the case of negative tainting, incompleteness leads to trusting data that should not be trusted and, ultimately, to false negatives. Incompleteness may thus leave the application vulnerable to attacks and can be very difficult to detect even after attacks occur. With positive tainting, incompleteness may lead to false positives, but never results in an SQLIA escaping detection. Moreover, as explained below, the false positives generated by our approach are likely to be detected and easily eliminated early, during pre-release testing. Positive tainting follows the general principle of *fail-safe defaults* as outlined by Saltzer and Schroeder in [22]: in case of incompleteness, positive tainting fails in a way that maintains the security of the system.

In the context of preventing SQLIAs, these conceptual advantages of positive tainting are especially significant. The way in which Web applications create SQL commands makes the identification of all untrusted data especially problematic and, most importantly, the identification of all trusted data relatively straightforward. Web applications are deployed in many different configurations and interface with a wide range of external systems. Therefore, there are often many potential external untrusted sources of input to be considered for these applications, and enumerating all of them is inherently difficult and error-prone. For example, developers initially assumed that only direct user input needed to be marked as tainted. Subsequent exploits demonstrated that additional input sources, such as browser cookies and uploaded files, also needed to be considered. However, accounting for these additional input sources did not completely solve the problem either. Attackers soon realized the possibility of leveraging local server variables and the database itself as injection sources [1]. In general, it is difficult to guarantee that all potentially harmful data sources have been considered, and even a single unidentified source could leave the application vulnerable to attacks.

The situation is different for positive tainting because identifying *trusted* data in a Web application is often straightforward, and always less error prone. In fact, in most cases, strings hard-coded in the application by developers represent the complete set of trusted data for a Web application.[2] The reason for this is that it is common practice for developers to build SQL commands by combining hard-coded strings that contain SQL keywords or operators with user-provided numeric or string literals. For Web applications developed in this way, which includes the applications used in our empirical evaluation, our approach *accurately* and *automatically* identifies all SQLIAs and generates no false positives; our basic approach, as explained in the following sections, automatically marks as trusted all hard-coded strings in the code and then ensures that all SQL keywords and operators are built using trusted data.

In some cases, this basic approach is not enough because developers can also use *external query fragments*—partial SQL commands coming from external input sources—to build queries. Because these string fragments are not hard-coded in the application, they would not be part of the initial set of trusted data identified by our approach, and the approach would generate false-positives when the string fragments are used in a query. To account for these cases, our technique provides developers with a mechanism to specify additional sources of external data that should be trusted. The data sources can be of various types, such as files, network connections, and server variables. Our approach uses this information to mark data coming from these additional sources as trusted.

In a typical scenario, we expect developers to specify most of the trusted sources beforehand. However, some of these sources might be overlooked until after a false positive is reported, in which case developers would add the omitted data source to the list of trusted sources. In this process, the set of trusted data sources grows monotonically and eventually converges to a complete set that produces no false positives. It is important to note that false positives that occur after deployment would be due to the use of external data sources that have never been used during in-house testing. In other words, false positives are likely to occur only for totally untested parts of the application. Therefore, even when developers fail to completely identify and mark additional sources of trusted input beforehand, we expect these sources to be identified during normal testing of the application, and the set of trusted data to quickly converge to the complete set.

3.2 Accurate Taint Propagation

Taint propagation consists of tracking taint markings associated with the data while the data is used and manipulated at runtime. When tainting is used for security-related applications, it is especially important for the propagation to be accurate. Inaccurate propagation can undermine the effectiveness of a technique by associating incorrect markings to data, which would cause the data to be mishandled. In our approach, we provide a mechanism to accurately mark and propagate taint information by (1) tracking taint markings at a low level of granularity and (2) precisely accounting for the effect of functions that operate on the tainted data.

Character-level tainting. We track taint information at the character level rather than at the string level. We do this because, for building SQL queries, strings are constantly broken into substrings, manipulated, and combined. By associating taint information to single characters, our approach can precisely model the effect of these string operations.

Accounting for string manipulations. To accurately maintain character-level taint information, we must identify all relevant string operations and account for their effect on the taint markings (*i.e.*, we must enforce complete mediation of all string operations). Our approach achieves this goal by taking advantage of the encapsulation offered by object-oriented languages, and in particular by Java, in which all string manipulations are performed using a small set of classes and methods. Our approach extends all such classes and methods by adding functionality to update taint markings based on the methods' semantics.

[2] We assume that developers are trustworthy. An attack encoded by a developer would not be an SQLIA but a form of back-door attack, which is not the problem addressed in this paper.

We discuss the language specific details of our implementation of the taint markings and their propagation in Section 4.

3.3 Syntax-Aware Evaluation

Besides ensuring that taint markings are correctly created and maintained during execution, our approach must be able to use the taint markings to distinguish legitimate from malicious queries. An approach that simply forbids the use of untrusted data in SQL commands is not a viable solution because it would flag any query that contains user input as an SQLIA, leading to many false positives. To address this shortcoming, researchers have introduced the concept of *declassification*, which permits the use of tainted input as long as it has been processed by a sanitizing function. (A sanitizing function is typically a filter that performs operations such as regular expression matching or sub-string replacement.) The idea of declassification is based on the assumption that sanitizing functions are able to eliminate or neutralize harmful parts of the input and make the data safe. However, in practice, there is no guarantee that the checks performed by a sanitizing function are adequate. Tainting approaches based on declassification could therefore generate false negatives if they mark as trusted supposedly-sanitized data that is in fact still harmful. Moreover, these approaches may also generate false positives in cases where unsanitized, but perfectly legal input is used within a query.

Syntax-aware evaluation does not depend on any (potentially unsafe) assumptions about the effectiveness of sanitizing functions used by developers. It also allows for the use of untrusted input data in an SQL query as long as the use of such data does not cause an SQLIA. The key feature of syntax-aware evaluation is that it considers the context in which trusted and untrusted data is used to make sure that all parts of a query other than string or numeric literals (*e.g.*, SQL keywords and operators) consist only of trusted characters. As long as untrusted data is confined to literals, we are guaranteed that no SQLIA can be performed. Conversely, if this property is not satisfied (*e.g.*, if an SQL operator contains characters not marked as trusted), we can assume that the operator has been injected by an attacker and block the query.

Our technique performs syntax-aware evaluation of a query string immediately before the string is sent to the database to be executed. To evaluate the query string, the technique first uses an SQL parser to break the string into a sequence of tokens that correspond to SQL keywords, operators, and literals. The technique then iterates through the tokens and checks whether tokens (*i.e.*, substrings) other than literals contain only trusted data. If all of the tokens pass this check, the query is considered safe and allowed to execute. As discussed in Section 3.1, this approach can also handle cases where developers use external query fragments to build SQL commands. In these cases, developers would specify which external data sources must be trusted, and our technique would mark and treat data coming from these sources accordingly.

This default approach, which (1) considers only two kinds of data (trusted and untrusted) and (2) allows only trusted data to form SQL keywords and operators, is adequate for most Web applications. For example, it can handle applications where parts of a query are stored in external files or database records that were created by the developers. Nevertheless, to provide greater flexibility and support a wide range of development practices, our technique also allows developers to associate custom trust markings to different data sources and provide custom trust policies that specify the legal ways in which data with certain trust markings can be used. *Trust policies* are functions that take as input a sequence of SQL tokens and perform some type of check based on the trust markings associated with the tokens.

BUGZILLA (http://www.bugzilla.org) is an example of a Web application for which developers might wish to specify a custom trust marking and policy. In BUGZILLA, parts of queries used within the application are retrieved from a database when needed. Of particular concern to developers, in this scenario, is the potential for *second-order injection* attacks [1] (*i.e.*, attacks that inject into a database malicious strings that result in an SQLIA only when they are later retrieved and used to build SQL queries). In the case of BUGZILLA, the only sub-queries that should originate from the database are specific predicates that form a query's WHERE clause. Using our technique, developers could first create a custom trust marking and associate it with the database's data source. Then, they could define a custom trust policy that specifies that data with such custom trust marking are legal only if they match a specific pattern, such as the following:

```
(id|severity)='\w+' ((AND|OR) (id|severity)='\w+')*
```

When applied to sub-queries originating from the database, this policy would allow them to be used only to build conditional clauses that involve the `id` or `severity` fields and whose parts are connected using the AND or OR keywords.

3.4 Minimal Deployment Requirements

Most existing approaches based on dynamic tainting require the use of customized runtime systems and/or impose a considerable overhead on the protected applications (see Section 6). On the contrary, our approach has minimal deployment requirements and is efficient, which makes it practical for usage in real settings. The use of our technique does not necessitate a customized runtime system. It requires only minor, localized instrumentation of the application to (1) enable the usage of our modified string library and (2) insert the calls that perform syntax-aware evaluation of a query before the query is sent to the database. The protected application is then deployed as any normal Web application, except that the deployment must include our string library. Both instrumentation and deployment are fully automated. We discuss deployment requirements and overhead of the approach in greater detail in Sections 4.5 and 5.3.

4. IMPLEMENTATION

To evaluate our approach, we developed a prototype tool called WASP (Web Application SQL Injection Preventer) that is written in Java and implements our technique for Java-based Web applications. We chose to target Java because it is a commonly-used language for developing Web applications. Moreover, we already have a significant amount of analysis and experimental infrastructure for Java applications. We expect our approach to be applicable to other languages as well.

Figure 2 shows the high-level architecture of WASP. As the figure shows, WASP consists of a library (MetaStrings) and two core modules (STRING INITIALIZER AND INSTRUMENTER and STRING CHECKER). The MetaStrings library provides functionality for assigning trust markings to strings and precisely propagating the markings at runtime. Module STRING INITIALIZER AND INSTRUMENTER instruments Web applications to enable the use of the MetaStrings library and add calls to the STRING CHECKER module. Module STRING CHECKER performs syntax-aware evaluation of query strings right before the strings are sent to the database.

In the next sections, we discuss WASP's modules in more detail. We use the sample code introduced in Section 2 to provide illustrative examples of various implementation aspects.

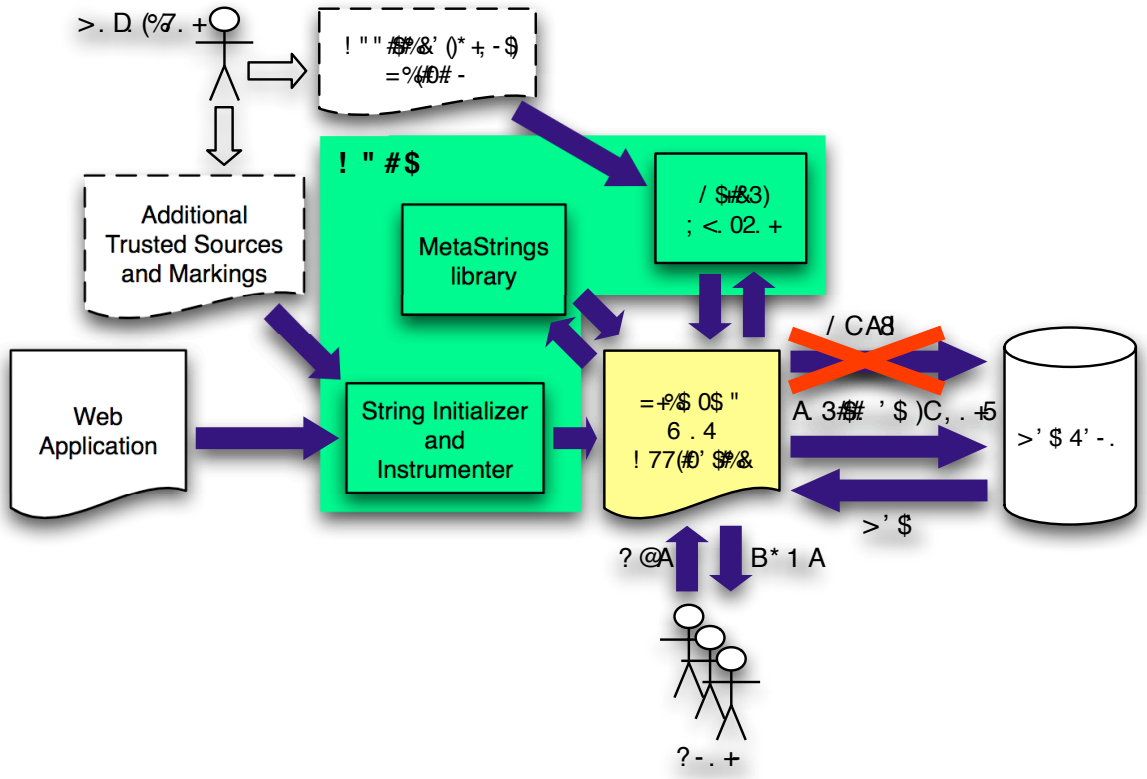


Figure 2: High-level overview of the approach and tool.

4.1 The MetaStrings Library

MetaStrings is our library of classes that mimic and extend the behavior of Java's standard string classes (*i.e.*, `Character`, `String`, `StringBuilder`, and `StringBuffer`).[3] For each string class C, MetaStrings provides a "meta" version of the class, `MetaC`, that has the same functionality as C, but allows for associating metadata with each character in a string and tracking the metadata as the string is manipulated at runtime.

The MetaStrings library takes advantage of the object-oriented features of the Java language to provide complete mediation of string operations that could affect string values and their associated trust markings. Encapsulation and information hiding guarantee that the internal representation of a string class is accessed only through the class's interface. Polymorphism and dynamic binding let us add functionality to a string class by (1) creating a subclass that overrides all methods of the original class and (2) replacing instantiations of the original class with instantiations of the subclass.

As an example, Figure 3 shows an intuitive view of the MetaStrings class that corresponds to Java's `String` class. As the figure shows, `MetaString` extends class `String`, has the same internal representation, and provides the same methods. `MetaString` also contains additional data structures for storing metadata and associating the metadata with characters in the string. Each method of class `MetaString` overrides the corresponding method in `String`, providing the same functionality as the original method, but also updating the metadata based on the method's semantics. For example, a call to method `substring(2,4)` on an object `str` of class `MetaString` would return a new `MetaString` that contains the second and third characters of `str` and the corresponding metadata. In addition to the overridden methods, MetaStrings classes also provide methods for setting and querying the metadata associated with a string's characters.

The use of MetaStrings has the following benefits: (1) it allows for associating trust markings at the granularity level of single characters; (2) it accurately maintains and propagates trust markings; (3) it is defined completely at the application level and therefore does not require a customized runtime system; (4) its usage requires only minimal and automatically performed changes to the application's bytecode; and (5) it imposes a low execution overhead on the Web application (See Section 5.3).

The main limitations of the current implementation of the MetaStrings library are related to the handling of primitive types, native methods, and reflection. MetaStrings cannot currently assign trust markings to primitive types, so it cannot mark `char` values. Because we do not instrument native methods, if a string class is passed as an argument to a native method, the trust marking associated with the string might not be correct after the call. In the case of hard-coded strings created through reflection (by invoking a string constructor by name), our instrumenter for MetaStrings would not recognize the constructors and would not change these instantiations to instantiations of the corresponding meta classes. However, the MetaStrings library can handle most other uses of reflection, such as invocation of string methods by name.

In practice, these limitations are of limited relevance because they represent programming practices that are not normally used to build SQL commands (e.g., representing strings using primitive `char` values). Moreover, during instrumentation of a Web application, we identify and report these potentially problematic situations to the developers.

4.2 Initialization of Trusted Strings

To implement positive tainting, WASP must be able to identify and mark trusted strings. There are three categories of strings that

[3] For simplicity, hereafter we use the term string to refer to all string-related classes and objects in Java.

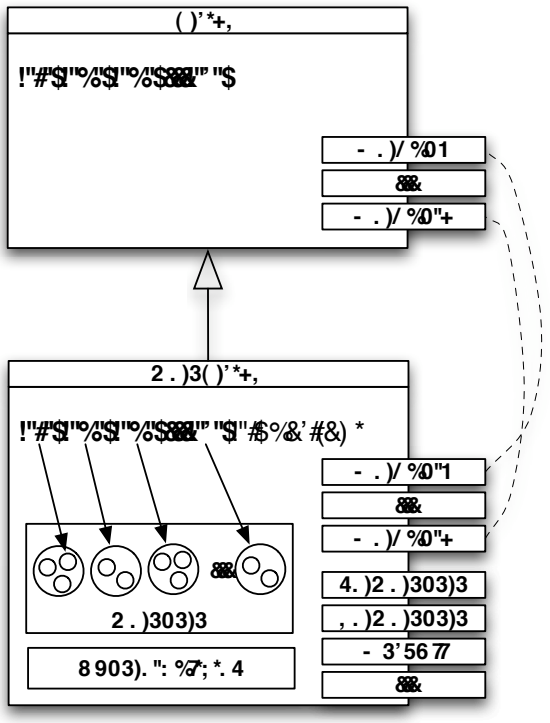


Figure 3: Intuitive view of a MetaStrings library class.

WASP must consider: hard-coded strings, strings implicitly created by Java, and strings originating from external sources. In the following sections, we explain how strings from each category are identified and marked.

Hard-Coded Strings. The identification of hard-coded strings in an application's bytecode is a fairly straightforward process. In Java, hard-coded strings are represented using `String` objects that are created automatically by the Java Virtual Machine (JVM) when string literals are loaded onto the stack. (The JVM is a stack-based interpreter.) Therefore, to identify hard-coded strings, WASP simply scans the bytecode and identifies all load instructions whose operand is a string constant. WASP then instruments the code by adding, after each of these load instructions, code that creates an instance of a `MetaString` class using the hard-coded string as an initialization parameter. Finally, because hard-coded strings are completely trusted, WASP adds to the code a call to the method of the newly created `MetaString` object that marks all characters as trusted. At runtime, polymorphism and dynamic binding allow this instance of the MetaString object to be used in any place where the original `String` object would have been used.

Figure 4 shows an example of this bytecode transformation. The Java code at the top of the figure corresponds to line 4 of our servlet example (see Figure 1), which creates one of the hard-coded strings in the servlet. Underneath, we show the original bytecode (left), and the modified bytecode (right). The modified bytecode contains additional instructions that (1) load a new `MetaString` object on the stack, (2) call the `MetaString` constructor using the previous string as a parameter, and (3) call the method `markAll`, which assigns the given trust marking to all characters in the string.

Implicitly-Created Strings. In Java programs, the creation of some string objects is implicitly added to the bytecode by the compiler. For example, Java compilers typically translate the string concatenation operator ("+") into a sequence of calls to the `append` method of a newly-created `StringBuilder` object. WASP must replace these string objects with their corresponding MetaStrings objects so that they can maintain and propagate the trust markings of the strings on which they operate. To do this, WASP scans the bytecode for instructions that create new instances of the string classes used to perform string manipulation and modifies each such instruction so that it creates an instance of the corresponding MetaStrings class instead. In this case, WASP does not associate any trust markings with the newly-created MetaStrings objects. These objects are not trusted per se, and they become marked only if the actual values assigned to them during execution are marked.

Figure 5 shows the instrumentation added by WASP for implicitly-created strings. The Java source code corresponds to line 5 in our example servlet. The `StringBuilder` object at offset 28 in the original bytecode is added by the Java compiler when translating the string concatenation operator ("+"). WASP replaces the instantiation at offset 28 with the instantiation of a `MetaStringBuilder` class and then changes the subsequent invocation of the constructor at offset 37 so that it matches the newly instantiated class. Because `MetaStringBuilder` extends `StringBuilder`, the subsequent calls to the append method invoke the correct method in the `MetaStringBuilder` class.

Strings from External Sources. To use query fragments coming from external (trusted) sources, developers must list these sources in a configuration file that WASP processes before instrumenting the application. The specified sources can be of different types, such as files (specified by name), network connections (specified by host and port), and databases (specified by database name, table, field, or combination thereof). For each source, developers can either specify a custom trust marking or use the default trust marking (the same used for hard-coded strings). WASP uses the information in the configuration file to instrument the external trusted sources according to their type.

To illustrate this process, we describe the instrumentation that WASP performs for trusted strings coming from a file. In the configuration file, the developer specifies the name of the file (*e.g.*, `foo.txt`) as a trusted source of strings. Based on this information, WASP scans the bytecode for all instantiations of new file objects (*i.e.*, `File`, `FileInputStream`, `FileReader`) and adds instrumentation that checks the name of the file being accessed. At runtime, if the name of the file matches the name(s) specified by the developer (`foo.txt` in this case), the file object is added to an internal list of currently trusted file objects. WASP also instruments all calls to methods of file-stream objects that return strings, such as `BufferedReader`'s `readLine` method. At runtime, the added code checks to see whether the object on which the method is called is in the list of currently trusted file objects. If so, it marks the generated strings with the trust marking specified by the developer for the corresponding source.

We use a similar strategy to mark network connections. In this case, instead of matching file names at runtime, we match hostnames and ports. The interaction with databases is more complicated and requires WASP not only to match the initiating connection, but also to trace tables and fields through instantiations of the `Statement` and `ResultSet` objects created when querying the database.

Instrumentation Optimization. Our current instrumentation approach is conservative and may generate unneeded instrumentation. We could limit the amount of instrumentation inserted in the code by leveraging static information about the program. For example, data-flow analysis could identify strings that are not involved

Source Code: 4. `String query = "SELECT acct FROM users WHERE login='";`

Original Bytecode	Modified Bytecode
24. ldc "SELECT acct FROM users WHERE login='"	24a. new MetaString 24b. dup 24c. ldc "SELECT acct FROM users WHERE login='" 24e. invokespecial MetaString.<init>:(LString)V 24d. iconst_1 24e. invokevirtual MetaString.markAll:(I)V

Figure 4: Instrumentation for hard-coded strings.

Source Code: 5. `query += login + "' AND pin=" + pin;`

Original Bytecode	Modified Bytecode
28. new StringBuilder 31. dup 32. aload 4 34. invokestatic String.valueOf:(Object)LString; 37. invokespecial StringBuilder.<init>:(LString;)V 40. aload_1 41. invokevirtual StringBuilder. append:(LString;)LStringBuilder; 44. ldc "' AND pin=" 46. invokevirtual StringBuilder. append:(LString;)LStringBuilder; 49. aload_2 50. invokevirtual StringBuilder. append:(LString;)LStringBuilder; 53. invokevirtual StringBuilder.toString:()LString;	28. new MetaStringBuilder 31. dup 32. aload 4 34. invokestatic String.valueOf:(LObject)LString; 37. invokespecial MetaStringBuilder.<init>:(LString;)V 40. aload_1 41. invokevirtual StringBuilder.append:(LString;)LStringBuilder; 44a. new MetaString 44b. dup 44c. ldc "' AND pin=" 44e. invokespecial MetaString.<init>:(LString)V 44d. iconst_1 44e. invokevirtual MetaString.markAll:(I)V 46. invokevirtual StringBuilder.append:(LString;)LStringBuilder; 49. aload_2 50. invokevirtual StringBuilder.append:(LString;)LStringBuilder; 53. invokevirtual StringBuilder.toString:()LString;

Figure 5: Instrumentation for implicitly-created strings.

with the construction of query strings and thus do not need to be instrumented. Another example involves cases where static analysis could determine that the filename associated with a file object is never one of the developer-specified trusted filenames, that object would not need to be instrumented. Analogous optimizations could be implemented for other external sources. We did not incorporate any of these optimizations in the current tool because we were mostly interested in having an initial prototype to assess our technique. However, we are planning to implement them in future work to further reduce runtime overhead.

4.3 Handling False Positives

As discussed in Section 3, sources of trusted data that are not specified by the developers beforehand would cause WASP to generate false positives. To assist the developers in identifying data sources that they initially overlooked, WASP provides a special mode of operation, called "learning mode", that would typically be used during in-house testing. When in learning mode, WASP adds an additional unique taint marking to *each* string in the application. Each marking consists of an ID that maps to the fully qualified class name, method signature, and bytecode offset of the instruction that instantiated the corresponding string.

If WASP detects an SQLIA while in learning mode, it uses the markings associated with the untrusted SQL keywords and operators in the query to report the instantiation point of the corresponding string(s). If the SQLIA is actually a false positive, knowing the position in the code of the offending string(s) would help developers correct omissions in the set of trusted inputs.

4.4 Syntax-Aware Evaluation

The STRING CHECKER module performs syntax-aware evaluation of query strings and is invoked right before the strings are sent to the database. To add calls to the STRING CHECKER module, WASP first identifies all of the *database interaction points*: points in the application where query strings are issued to an underlying database. In Java, all calls to the database are performed via specific methods and classes in the JDBC library (`http://java.sun.com/products/jdbc/`). Therefore, these points can be identified through a simple matching of method signatures. After identifying the database interaction points, WASP inserts a call to the syntax-aware evaluation function, `MetaChecker`, immediately before each interaction point. `MetaChecker` takes the MetaStrings object that contains the query about to be executed as a parameter.

When invoked, `MetaChecker` processes the SQL string about to be sent to the database as discussed in Section 3.3. First, it tokenizes the string using an SQL parser. Ideally, WASP would use a database parser that recognizes the exact same dialect of SQL that is used by the database. This would guarantee that WASP interprets the query in the same way as the database and would prevent attacks based on alternate encodings [1]—attacks that obfuscate keywords and operators to elude signature-based checks. Our current implementation includes parsers for SQL-92 (ANSI) and PostgreSQL. After tokenizing the query string, `MetaChecker` enforces the default trust policy by iterating through the tokens that correspond to keywords and operators and examining their trust markings. If any of these tokens contains characters that are not marked as trusted, the query is blocked and reported.

If developers specified additional trust policies, `MetaChecker` invokes the corresponding checking function(s) to ensure that the query complies with them. In our current implementation, trust policies are developer-defined functions that take the list of SQL tokens as input, perform some type of check on them based on their trust markings, and return a `true` or `false` value depending on the outcome of the check. Trust policies can implement functionality that ranges from simple pattern matching to sophisticated checks that use externally-supplied contextual information. If all custom trust policies return a positive outcome, WASP allows the query to be executed on the database. Otherwise, it classifies the query as an SQLIA, blocks it, and reports it.

Figure 6: Example query 1 after parsing by runtime monitor.

Figure 7: Example query 2 after parsing by runtime monitor.

We illustrate how the default policy for syntax-aware evaluation works using our example servlet and the legitimate and malicious query examples from Section 2. For the servlet there are no external sources of strings or additional trust policies, so WASP only marks the hard-coded strings as trusted, and only the default trust policy is applied. Figure 6 shows the sequence of tokens in the legitimate query as they would be parsed by MetaChecker. In the figure, SQL keywords and operators are surrounded by boxes. The figure also shows the trust markings associated with the strings, where an underlined character is a character with full trust markings. Because the default trust policy is that all keyword and operator tokens must have originated from trusted strings, MetaChecker simply checks whether all these tokens are comprised of trusted characters. The query in Figure 6 conforms to the trust policy and is thus allowed to execute on the database.

Consider the malicious query, where the attacker submits "admin' --" as the login and "0" as the pin. Figure 7 shows the sequence of tokens for the resulting query together with the trust markings. Recall that -- is the SQL comment operator, so everything after this is identified by the parser as a literal. In this case, the MetaChecker would find that the last two tokens, ' and -- contain untrusted characters. It would therefore classify the query as an SQLIA and prevent it from executing.

4.5 Deployment Requirements

Using WASP to protect a Web application requires the developer to run an instrumented version of the application. There are two general implementation strategies that we can follow for the instrumentation: off-line or on-line. Off-line instrumentation instruments the application statically and deploys the instrumented version of the application. On-line instrumentation deploys an unmodified application and instruments the code at load time (*i.e.*, when classes are loaded by the JVM). This latter option allows for a great deal of flexibility and can be implemented by leveraging the new instrumentation package introduced in Java 5 (http://java.sun.com/j2se/1.5.0/).

Unfortunately, the current implementation of the Java 5 instrumentation package is still incomplete and does not yet provide some key features needed by WASP. In particular, it does not allow for clearing the final flag in the string library classes, which prevents the MetaStrings library from extending them. Because of this limitation, for now we have chosen to rely on off-line instrumentation and to splice into the Java library a version of the string classes in which the final flag has been cleared.

Overall, the deployment requirements for our approach are fairly lightweight. The modification of the Java library is performed only once, in a fully automated way, and takes just a few seconds. No modification of the Java Virtual Machine is required. The instrumentation of a Web application is also performed automatically. Given the original application, WASP creates a deployment archive that contains the instrumented application, the MetaStrings library, and the string checker module. At this point, the archive can be deployed like any other Web application. WASP can therefore be easily and transparently incorporated into an existing build process.

Table 1: Subject programs for the empirical study.

Subject	*LOC*	*DBIs*	*Servlets*	*Params*
Checkers	5,421	5	18 (61)	44 (44)
Office Talk	4,543	40	7 (64)	13 (14)
Employee Directory	5,658	23	7 (10)	25 (34)
Bookstore	16,959	71	8 (28)	36 (42)
Events	7,242	31	7 (13)	36 (46)
Classifieds	10,949	34	6 (14)	18 (26)
Portal	16,453	67	3 (28)	39 (46)

5. EVALUATION

The goal of our empirical evaluation is to assess the effectiveness and efficiency of the approach presented in this paper when applied to a testbed of Web applications. In the evaluation, we used our implementation of WASP and investigated the following three research questions:

RQ1: What percentage of attacks can WASP detect and prevent that would otherwise go undetected and reach the database?

RQ2: What percentage of legitimate accesses does WASP identify as SQLIAs and prevent from executing on the database?

RQ3: How much runtime overhead does WASP impose?

The first two questions deal with the *effectiveness* of the technique: RQ1 addresses the false negative rate of the technique, and RQ2 addresses the false positive rate. RQ3 deals with the *efficiency* of the proposed technique. The following sections discuss our experiment setup, protocol, and results.

5.1 Experiment Setup

Our experiments are based on an evaluation framework that we developed and has been used by us and other researchers in previous work [9, 24]. The framework provides a testbed that consists of several Web applications, a logging infrastructure, and a large set of test inputs containing both legitimate accesses and SQLIAs. In the next two sections we summarize the relevant details of the framework.

5.1.1 Subjects

Our set of subjects consists of seven Web applications that accept user input via Web forms and use it to build queries to an underlying database. Five of the seven applications are commercial applications that we obtained from GotoCode (http://www.gotocode.com/): Employee Directory, Bookstore, Events, Classifieds, and Portal. The other two, Checkers and OfficeTalk, are applications developed by students that have been used in previous related studies [7].

For each subject, Table 1 provides the size in terms of lines of code (*LOC*) and the number of database interaction points (*DBIs*). To be able to perform our studies in an automated fashion and collect a larger number of data points, we considered only those servlets that can be accessed directly, without complex interactions with the application. Therefore, we did not include in the evaluation servlets that require the presence of specific session data (*i.e.*, cookies containing specific information) to be accessed. Column *Servlets* reports, for each application, the number of servlets considered and, in parentheses, the total number of servlets. Column *Params* reports the number of injectable parameters in the accessible servlets, with the total number of parameters in parentheses. Non-injectable parameters are state parameters whose purpose is to maintain state, and which are not used to build queries.

5.1.2 Test Input Generation

For each application in the testbed, there are two sets of inputs: *LEGIT*, which consists of legitimate inputs for the application, and

ATTACK, which consists of SQLIAs. The inputs were generated independently by a Master's level student with experience in developing commercial penetration testing tools for Web applications. Test inputs were not generated for non-accessible servlets and for state parameters.

To create the ATTACK set, the student first built a set of potential attack strings by surveying different sources: exploits developed by professional penetration-testing teams to take advantage of SQL-injection vulnerabilities; online vulnerability reports, such as US-CERT (http://www.us-cert.gov/) and CERT/CC Advisories (http://www.cert.org/advisories/); and information extracted from several security-related mailing lists. The resulting set of attack strings contained 30 unique attacks that had been used against applications similar to the ones in the testbed. All types of attacks reported in the literature [10] were represented in this set except for multi-phase attacks such as overly-descriptive error messages and second-order injections. Since multi-phase attacks require human intervention and interpretation, we omitted them to keep our testbed fully automated. The student then generated a complete set of inputs for each servlet's injectable parameters using values from the set of initial attack strings and legitimate values. The resulting ATTACK set contained a broad range of potential SQLIAs.

The LEGIT set was created in a similar fashion. However, instead of using attack strings to generate sets of parameters, the student used legitimate values. To create "interesting" legitimate values, we asked the student to create inputs that would stress and possibly break naïve SQLIA detection techniques (*e.g.*, techniques based on simple identification of keywords or special characters in the input). The result was a set of legitimate inputs that contained SQL keywords, operators, and troublesome characters, such as single quotes and comment operators.

5.2 Experiment Protocol

To address the first two research questions, we ran the ATTACK and LEGIT input sets against the testbed applications and assessed WASP's effectiveness in stopping attacks without blocking legitimate accesses. *For* **RQ1**, we ran all of the inputs in the ATTACK set and tracked the result of each attack. The results for **RQ1** are summarized in Table 2. The second column reports the total number of attacks in the LEGIT set for each application. The next two columns report the number of attacks that were successful on the original web applications and on the web applications protected by WASP. (Many of the applications performed input validation of some sort and were able to block a subset of the attacks.) *For* **RQ2**, we ran all of the inputs in the LEGIT set and checked how many of these legitimate accesses WASP allowed to execute. The results for this second study are summarized in Table 3. The table shows the number of legitimate accesses WASP allowed to execute (*# Legitimate Accesses*) and the number of accesses blocked by WASP (*False Positives*).

To address **RQ3**, we computed the overhead imposed by WASP on the subjects. To do this, we measured the times required to run all of the inputs in the LEGIT set against instrumented and uninstrumented versions of each application and compared these two times. To avoid problems of imprecision in the timing measurements, we measured the time required to run the entire LEGIT set and then divided it by the number of test inputs to get a per-access average time. Also, to account for possible external factors beyond our control, such as network traffic, we repeated these measurements 100 times for each application and averaged the results. The study was performed on two machines, a client and a server. The client was a Pentium 4, 2.4Ghz, with 1GB memory, running GNU/Linux 2.4. The server was a dual-processor Pentium D, 3.0Ghz, with 2GB of memory, running GNU/Linux 2.6.

Table 2: Results for effectiveness in SQLIAs prevention (RQ1).

Subject	*Total # Attacks*	*Successful Attacks*	
		Original Web Apps	WASP *Protected Web Apps*
Checkers	4,431	922	0
Office Talk	5,888	499	0
Empl. Dir.	6,398	2,066	0
Bookstore	6,154	1,999	0
Events	6,207	2,141	0
Classifieds	5,968	1,973	0
Portal	6,403	3,016	0

Table 3: Results for false positives (RQ2).

Subject	*# Legitimate Accesses*	*False Positives*
Checkers	1,359	0
Office Talk	424	0
Empl. Dir.	658	0
Bookstore	607	0
Events	900	0
Classifieds	574	0
Portal	1,080	0

Table 4 shows the results of this study. For each subject, the table reports the number of inputs in the LEGIT set (*# Inputs*); the average time per database access (*Avg Access Time*); the average time overhead per access (*Avg Overhead*); and the average time overhead as a percentage (*% Overhead*). In the table, all absolute times are expressed in milliseconds.

5.3 Discussion of Results

Overall, the results of our studies indicate that WASP is an effective technique for preventing SQLIAs. In our evaluation, WASP was able to correctly identify all SQLIAs without generating any false positives. In total, WASP stopped 12,616 viable SQLIAs and correctly allowed 5,602 legitimate accesses to the applications.

In most cases, the runtime average imposed by WASP was very low. For the seven applications, the average overhead was 5ms (6%). For most Web applications, this cost is low enough that it would be dominated by the cost of the network and database accesses. One application, Portal, incurred an overhead considerably higher than the other applications (but still negligible in absolute terms). We determined that the higher overhead was due to the fact that Portal generates a very large number of string-based lookup tables. Although these strings are not used to build queries, WASP associates trust markings to them and propagates these markings at runtime. The optimizations discussed in Section 4.2 would eliminate this issue and reduce the overhead considerably.

The main threat to the external validity of our results is that the set of applications and attacks considered in the studies may not be representative of real world applications and attacks. However, all but two of the considered applications are commercial applications, and all have been used in other related studies. Also, to generate our set of attacks, we employed the services of a Master's level student who had experience with SQLIAs, penetration testing, and Web scanners, but was not familiar with our technique. Finally, the attack strings used by the student as a basis for the generation of the attacks were based on real-world SQLIAs.

Table 4: Results for overhead measurements (RQ3).

Subject	*# Inputs*	*Avg Access Time (ms)*	*Avg* Overhead (ms)	*% Overhead*
Checkers	1,359	122	5	5%
Office Talk	424	56	1	2%
Empl. Dir.	658	63	3	5%
Bookstore	607	70	4	6%
Events	900	70	1	1%
Classifieds	574	70	3	5%
Portal	1,080	83	16	19%

6. RELATED WORK

The use of dynamic tainting to prevent SQLIAs has been investigated by several researchers. The two approaches most similar to ours are those by Nguyen-Tuong and colleagues [20] and Pietraszek and Berghe [21]. Similar to them, we track taint information at the character level and use a syntax-aware evaluation to examine tainted input. However, our approach differs from theirs in several important aspects. First, our approach is based on the novel concept of positive tainting, which is an inherently safer way of identifying trusted data (see Section 3.1). Second, we improve on the idea of syntax-aware evaluation by (1) using a database parser to interpret the query string before it is executed, thereby ensuring that our approach can handle attacks based on alternate encodings, and (2) providing a flexible mechanism that allows different trust policies to be associated with different input sources. Finally, a practical advantage of our approach is that it has more lightweight deployment requirements. Their approaches require the use of a customized PHP runtime interpreter, which adversely affects the portability of the approaches.

Other dynamic tainting approaches more loosely related to our approach are those by Haldar, Chandra, and Franz [8] and Martin, Livshits, and Lam [17]. Although they also propose dynamic tainting approaches for Java-based applications, their techniques differ significantly from ours. First, they track taint information at the level of granularity of strings, which introduces imprecision in modeling string operations. Second, they use declassification rules, instead of syntax-aware evaluation, to assess whether a query string contains an attack. Declassification rules assume that sanitizing functions are always effective, which is an unsafe assumption and may leave the application vulnerable to attacks—in many cases, attack strings can pass through sanitizing functions and still be harmful. Another dynamic tainting approach, proposed by Newsome and Song [19], focuses on tainting at a level that is too low to be used for detecting SQLIAs and has a very high execution overhead.

Researchers also proposed dynamic techniques against SQLIAs that do not rely on tainting. These techniques include Intrusion Detection Systems (IDS) and automated penetration testing tools. Scott and Sharp propose Security Gateway [23], which uses developer-provided rules to filter Web traffic, identify attacks, and apply preventive transformations to potentially malicious inputs. The success of this approach depends on the ability of developers to write accurate and meaningful filtering rules. Similarly, Valeur and colleagues [25] developed an IDS that uses machine learning to distinguish legitimate and malicious queries. Their approach, like most learning-based techniques, is limited by the quality of the IDS training set. Machine learning was also used in WAVES [12], an automated penetration testing tool that probes websites for vulnerability to SQLIAs. Like all testing tools, WAVES cannot provide any guarantees of completeness. SQLrand [2] appends a random token to SQL keywords and operators in the application code. A proxy server then checks to make sure that all keywords and operators contain this token before sending the query to the database. Because the SQL keywords and operators injected by an attacker would not contain this token, they would be easily recognized as attacks. The drawbacks of this approach are that the secret token could be guessed, so making the approach ineffective, and that the approach requires the deployment of a special proxy server.

Model-based approaches against SQLIAs include AMNESIA [9], SQL-Check [24], and SQLGuard [3]. AMNESIA, previously developed by two of the authors, combines static analysis and runtime monitoring to detect SQLIAs. The approach uses static analysis to build models of the different types of queries an application can generate and dynamic analysis to intercept and check the query strings generated at runtime against the model. Non-conforming queries are identified as SQLIAs. Problems with this approach are that it is dependent on the precision and efficiency of its underlying static analysis, which may not scale to large applications. Our new technique takes a purely dynamic approach to preventing SQLIAs, thereby eliminating scalability and precision problems. In [24], Su and Wassermann present a formal definition of SQLIAs and propose a sound and complete (under certain assumptions) algorithm that can identify all SQLIAs by using an augmented grammar and by distinguishing untrusted inputs from the rest of the strings by means of a marking mechanism. The main weakness of this approach is that it requires the manual intervention of the developer to identify and annotate untrusted sources of input, which introduces incompleteness problems and may lead to false negatives. Our use of positive tainting eliminates this problem while providing similar guarantees in terms of effectiveness. SQLGuard [3] is an approach similar to SQLCheck. The main difference is that SQLGuard builds its models on the fly by requiring developers to call a special function and to pass to the function the query string before user input is added.

Other approaches against SQLIAs rely purely on static analysis [13, 14, 15, 27]. These approaches scan the application and leverage information flow analysis or heuristics to detect code that could be vulnerable to SQLIAs. Because of the inherently imprecise nature of the static analysis they use, these techniques can generate false positives. Moreover, since they rely on declassification rules to transform untrusted input into safe input, they can also generate false negatives. Wassermann and Su propose a technique [26] that combines static analysis and automated reasoning to detect whether an application can generate queries that contain tautologies. This technique is limited, by definition, in the types of SQLIAs that it can detect.

Finally, researchers have also focused on ways to directly improve the code of an application and eliminate vulnerabilities. Defensive coding best practices [11] have been proposed as a way to eliminate SQL injection vulnerabilities. These coding practices have limited effectiveness because they mostly rely on the ability and training of the developer. Moreover, there are many well-known ways to evade certain types of defensive-coding practices, including "pseudo-remedies" such as stored procedures and prepared statements (*e.g.*, [1, 16, 11]). Researchers have also developed special libraries that can be used to safely create SQL queries [4, 18]. These approaches, although highly effective, require developers to learn new APIs for developing queries, are very expensive to apply on legacy code, and sometimes limit the expressiveness of SQL. Finally, JDBC-Checker [6, 7] is a static analysis tool that detects potential type mismatches in dynamically generated queries. Although it was not intended to prevent SQLIAs, JDBC-Checker can be effective against SQLIAs that leverage vulnerabilities due to type-mismatches, but will not be able to prevent other kinds of SQLIAs.

7. CONCLUSION

We presented a novel, highly automated approach for detecting and preventing SQL injection attacks in Web applications. Our basic approach consists of (1) identifying trusted data sources and marking data coming from these sources as trusted, (2) using dynamic tainting to track trusted data at runtime, and (3) allowing only trusted data to become SQL keywords or operators in query strings. Unlike previous approaches based on dynamic tainting, our technique is based on positive tainting, which explicitly identifies trusted (rather than untrusted) data in the program. In this way, we eliminate the problem of false negatives that may result from the incomplete identification of all untrusted data sources. False positives, while possible in some cases, can typically be easily eliminated during testing. Our approach also provides practical advantages over the many existing techniques whose application requires customized and complex runtime environments. The approach is defined at the application level, requires no modification of the runtime system, and imposes a low execution overhead.

We have evaluated our approach by developing a prototype tool, WASP, and using the tool to protect several applications when subjected to a large and varied set of attacks and legitimate accesses. WASP successfully and efficiently stopped over 12,000 attacks without generating any false positives. Both our tool and experimental infrastructure are available to other researchers.

We have three immediate goals for future work. The first goal is to further improve the efficiency of the technique. To this end, we will use static analysis to reduce the amount of instrumentation required by the approach. The second goal is to implement the approach for binary applications, by leveraging a binary instrumentation framework and defining a version of the MetaStrings library that works at the binary level. Finally, we plan to evaluate our technique in a completely realistic context, by protecting one of the Web applications running at Georgia Tech with WASP and assessing the effectiveness of WASP in stopping real attacks directed at the application while allowing legitimate accesses.

Acknowledgments

This work was supported by NSF awards CCR-0306372 and CCF-0438871 to Georgia Tech and by the Department of Homeland Security and US Air Force under Contract No. FA8750-05-2-0214. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the US Air Force.

8. REFERENCES

[1] C. Anley. Advanced SQL Injection In SQL Server Applications. White paper, Next Generation Security Software Ltd., 2002.

[2] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Proc. of the 2nd Applied Cryptography and Network Security Conf. (ACNS '04)*, pages 292–302, Jun. 2004.

[3] 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 Intl. Workshop on Software Engineering and Middleware (SEM '05)*, pages 106–113, Sep. 2005.

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

[5] T. O. Foundation. Top ten most critical web application vulnerabilities, 2005. http://www.owasp.org/documentation/topten.html.

[6] C. Gould, Z. Su, and P. Devanbu. JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications. In *Proc. of the 26th Intl. Conference on Software Engineering (ICSE 04) – Formal Demos*, pages 697–698, May 2004.

[7] C. Gould, Z. Su, and P. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In *Proc. of the 26th Intl. Conference on Software Engineering (ICSE 04)*, pages 645–654, May 2004.

[8] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *Proc. of the 21st Annual Computer Security Applications Conference*, pages 303–311, Dec. 2005.

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

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

[11] M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, Redmond, Washington, Second Edition, 2003.

[12] Y. Huang, S. Huang, T. Lin, and C. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *Proc. of the 12th Intl. World Wide Web Conference (WWW 03)*, pages 148–159, May 2003.

[13] 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 Intl. World Wide Web Conference (WWW 04)*, pages 40–52, May 2004.

[14] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *2006 IEEE Symposium on Security and Privacy*, May 2006.

[15] V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Usenix Security Symposium*, Aug. 2005.

[16] O. Maor and A. Shulman. SQL Injection Signatures Evasion. White paper, Imperva, Apr. 2004. http://www.imperva.com/application_defense_center/white_papers/sql_injection_signatures_evasion.html.

[17] M. Martin, B. Livshits, and M. S. Lam. Finding Application Errors and Security Flaws Using PQL: a Program Query Language. In *OOPSLA '05: Proc. of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 365–383, Oct. 2005.

[18] R. McClure and I. Krüger. SQL DOM: Compile Time Checking of Dynamic SQL Statements. In *Proc. of the 27th Intl. Conference on Software Engineering (ICSE 05)*, pages 88–96, May 2005.

[19] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. of the 12th Annual Network and Distributed System Security Symposium (NDSS 05)*, Feb. 2005.

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

[21] T. Pietraszek and C. V. Berghe. Defending Against Injection Attacks through Context-Sensitive String Evaluation. In *Proc. of Recent Advances in Intrusion Detection (RAID2005)*, Sep. 2005.

[22] J. Saltzer and M. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE*, Sep 1975.

[23] D. Scott and R. Sharp. Abstracting Application-level Web Security. In *Proc. of the 11th Intl. Conference on the World Wide Web (WWW 2002)*, pages 396–407, May 2002.

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

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

[26] 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.

[27] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the 15th USENIX Security Symposium*, July 2006.

Command-Form Coverage for Testing Database Applications

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

Abstract

The testing of database applications poses new challenges for software engineers. In particular, it is difficult to thoroughly test the interactions between an application and its underlying database, which typically occur through dynamically-generated database commands. Because traditional code-based coverage criteria focus only on the application code, they are often inadequate in exercising these commands. To address this problem, we introduce a new test adequacy criterion that is based on coverage of the database commands generated by an application and specifically focuses on the application-database interactions. We describe the criterion, an analysis that computes the corresponding testing requirements, and an efficient technique for measuring coverage of these requirements. We also present a tool that implements our approach and a preliminary study that shows the approach's potential usefulness and feasibility.

1 Introduction

Database applications are an important component of many software systems in areas such as banking, online shopping, and health care. Because they often handle critical data, it is especially important that these applications function correctly. However, database applications have peculiar characteristics that can hinder the effectiveness of traditional testing approaches. One of these characteristics is the way interactions occur between the application and its underlying database(s). Most database applications dynamically generate commands in the database language (usually, SQL—Structured Query Language), pass these commands to the database for execution, and process the results returned by the database. Traditional code-based coverage criteria, such as statement or branch coverage, do not specifically target these generated commands. Therefore, even though they can reveal faults in the database application's code, they are often unable to reveal faults in the database commands generated by the application. Several researchers have proposed alternative criteria specifically targeted at database applications (e.g., [13, 17, 22]), but none of these approaches focuses on the coverage of dynamically-generated database commands.

To address this problem, we define a new test adequacy criterion that is specifically targeted at the interactions between an application and its database. Our criterion is based on coverage of all of the possible database command forms that the application under test can generate. Intuitively, command forms are database commands with placeholders for parts that will be supplied at runtime (e.g., through user input). To compute the set of command forms for an application, we defined a technique that builds on two previously-developed analyses [6, 12]. The technique takes as input the code of the application under test and produces a conservative approximation of the possible command forms that the application can generate. The command forms are represented as a Deterministic Finite Automaton (DFA) in which each complete path identifies a unique command form. To efficiently collect and compute coverage information, we leverage a technique for efficient path profiling by Ball and Larus [1] and apply it to the DFAs generated by our technique.

We implemented our approach in a prototype tool called DITTO (Database Interaction Testing TOol). DITTO lets developers assess the adequacy of an existing test suite with respect to application-database interactions. DITTO can also help testers generate test cases by providing feedback about which database command forms have not been exercised.

To evaluate our approach, we performed two preliminary studies on a real database application using DITTO. The first study is a proof-of-concept study that shows that our approach can be used to compute testing requirements and collect coverage information. In the second study, we assess the potential usefulness of our coverage criterion as compared to a more traditional structural coverage criterion.
The contributions of this paper are:

- A new coverage criterion for database applications that focuses on adequately exercising the interactions between an application and its underlying database.
- An efficient approach for (1) computing testing requirements, (2) instrumenting an application and collecting coverage information, (3) analyzing the coverage information and providing feedback to testers.

- The development of a tool, DITTO, that implements our approach.
- A preliminary study that shows the potential usefulness and feasibility of the criterion.

2 Background and Terminology

A *database application* is typically a multi-tiered application that interacts with one or more databases during execution. The top tier (*UI tier*) provides the user interface, the middle tier (*application tier*) implements the application's logic, and the bottom tier (*database tier*) is the database. At runtime, the application interacts with the database by generating commands in the database language and using an API to issue the commands to the database. The database executes the commands and returns the results to the application.

Because of their characteristics, database applications can be considered meta-programs that generate object programs to be executed on the database. In this case, the meta-language is the language used in the application tier—typically, one or more general purpose programming languages such as Java, C, Perl, or PHP—and the object language is usually the Structured Query Language (SQL). The meta-program creates database commands (i.e., the object program) by combining hard-coded strings that contain SQL keywords, operators, and separators with literals that can originate from the user or other sources of input. In most applications, the creation of a database command spans several statements and often involves multiple procedures. We refer to the parts of a database command that cannot be determined statically (e.g., substrings that correspond to user input) as the *indeterminate parts* of the command.

Within the meta-program, there are statements that perform API calls to issue commands to the database. Using the terminology introduced by Kapfhammer and Soffa [13], we call these statements *database interaction points*. Depending on the structure of the application and user input, a specific database interaction point can issue different types of database commands. To characterize the commands that can be generated at a database interaction point, we use the concept of database command form. A *database command form* (or simply *command form*) is an equivalence class that groups database commands that differ only in the possible value of their indeterminate parts. Intuitively, one can think of a command form as a template command string in which the parts of the database command that are statically defined by the application are specified, and the indeterminate parts are marked by a placeholder. In Section 4.1 we provide a concrete example of a database command form.

```
   public ResultSet
   searchBooks(String searchString, int searchType,
               boolean showRating, boolean groupByRating,
               boolean groupByISBN) {

 1. String[] searchFields = {"tiitle", "author", "isbn"};
 2. String queryStr= "SELECT title, author, description";
 3. if (showRating) {
 4.     queryStr += ", avg(rating) ";
    }
 5. queryStr += "FROM books WHERE ";
 6. if (searchType==2) {
 7.     queryStr += searchFields[searchType] + " = " +
                    searchString;
    }
 8. else {
 9.     queryStr += searchFields[searchType] + " = '" +
                    searchString + "' ";
    }
10. if (groupByRating) {
11.     queryStr += "GROUP BY rating ";
    }
12. else if (groupByISBN) {
13.     queryStr += " GROUP BY isbn ";
    }
14. return database.executeQuery(queryStr);
  }
```

Figure 1. Excerpt of database application.

3 Motivating Example

Traditional code-based coverage criteria focus on discovering errors in the application code and can result in very limited coverage of the SQL commands that an application can generate. To illustrate this limitation, Figure 1 shows a possible snippet of code from a database application. Method searchBooks has one database interaction point (line 14) and takes five inputs: a search string (searchString), an integer representing the search type (searchType), and a set of parameters for the search (showRating, groupByRating, and groupByISBN). The last four inputs determine how the hard-coded strings in the code will be combined to produce the final command. The value of the first parameter, searchString, is directly embedded in the database command.

This code compiles correctly, but it contains four faults that manifest themselves in the object language. Certain paths through the code generate illegal SQL commands that cause database errors and, ultimately, application failures.

1. At line 1, field "title" is misspelled as "tiitle." Because "tiitle" is not a legal column name in the table, it will cause an error if it is appended to the query at line 9.
2. If both of the appends at line 7 and line 11 are executed, there will be no space delimiter between the value of searchString and the "GROUP BY" clause.
3. In SQL, grouping functions such as avg() require a corresponding "GROUP BY" clause in the query. If showRating is true, but groupByRating and groupByISBN are not, this rule will be violated.
4. If the append at line 4 is not performed, there will be no space delimiter between "description" and the "FROM" clause.

These faults manifest themselves in the generated object-program and not in the application code. Therefore, a traditional code-based adequacy criterion that requires the coverage of the application code would only detect such faults by chance. To illustrate, consider the following three test cases:

```
searchBooks("0123456789", 2, false, false, true)
searchBooks("Any Author", 1, false, false, false)
searchBooks("Any Author", 1, true, true, false)
```

These test cases achieve 100% branch (and statement) coverage of the example code, but reveal only one of the four faults in the code—the fourth one. Even using a stronger criterion, such as the all-paths criterion, could fail to expose all of the faults. A test suite could exercise all paths in the example code, but if zero is never used as a search type, the first fault would not be exposed. In the next section, we explain how our approach can provide the tester with a more effective criterion for testing interactions between applications and their underlying databases by focusing on the object program instead of the meta-program.

4 A Novel Approach for Testing Database Applications

Whereas traditional code-based adequacy criteria focus on the database application code, our approach focuses on testing the interactions between applications and underlying databases. In this sense, our approach complements existing testing criteria and ensures that database applications are more thoroughly tested. In this section, we discuss the four components of our approach: (1) a new coverage criterion for database applications, (2) a technique for computing testing requirements for the criterion, (3) a technique for efficiently collecting coverage data, and (4) a technique for analyzing and reporting coverage information.

4.1 Testing Requirements

The set of testing requirements for our criterion consists of all of the command forms for all of the database interaction points in the application under test. Because our goal is to exercise the interactions between an application and its underlying database, command forms represent a model of the database application at the right level of abstraction—they model all of the possible commands that the application can generate and execute on the database. Therefore, the number of command forms exercised by a test suite is likely to be a good indicator of the thoroughness of the testing of the interactions between the application and its database.

For our example code in Figure 1, the set of testing requirements consists of the command forms that can be executed at line 14, the only database interaction point. By looking at the different paths in the code, we can see that it can generate eighteen distinct command forms. For the sake of space, we only list one of them as an example:

```
SELECT title, author, description, avg(rating)
FROM books WHERE author = ' ' GROUP BY rating
```

We use symbol as a placeholder for the indeterminate part of the command (in this simple case, the part corresponding to the value of `searchString`). All other parts of the database command, which can be determined statically, are specified in the command form.

4.2 Computing Command Forms

The main challenge when generating command forms is the accurate identification of the possible SQL commands that could be issued at a given database interaction point. Because these commands are generated at runtime and often inter-procedurally, this task requires the application of sophisticated program-analysis techniques. We perform this task in three steps.

In the *first* step, we leverage the Java String Analysis (JSA) developed by Christensen, Møller, and Schwartzbach [6]. Given a program P, a string variable[1] *str*, and a program point s, JSA analyzes P and computes a Non-deterministic Finite Automaton (NFA) that encodes, at the character level, all of the possible values that *str* can assume at s. JSA builds the NFA in a conservative way, by taking into account all string operations on *str* along program paths leading to s. We apply JSA to the command string variable used at each database interaction point and obtain an NFA for each string.

In the *second* step, we refine the NFAs by using a technique from our previous work [12]. This technique parses the character-level NFAs and produces corresponding SQL-level models by aggregating characters that correspond to SQL keywords and operators. Therefore, an SQL-level model is an NFA in which transitions correspond to SQL tokens (keywords, operators, and delimiters) and input placeholders, instead of single characters or character ranges (as in the original JSA models).

In the *third* step, we compute the set of command forms from the SQL-level models. We first determinize and then minimize the SQL-level models to obtain what we call an *SQL command form model*. By construction, the set of command forms for a specific database interaction point is exactly the set of all accepting paths in the command form model. To keep the number of requirements finite and avoid the need to enumerate all of the possible command forms, we adapt the efficient path profiling approach proposed by Ball and Larus [1]. Using this approach, we (1) transform any cyclic models into directed acyclic graphs and (2) assign integer *edge values* to a subset of the transitions in

[1] We use the term string to refer to all of the Java string-related classes, such as StringBuilder, StringBuffer, Character, and String.

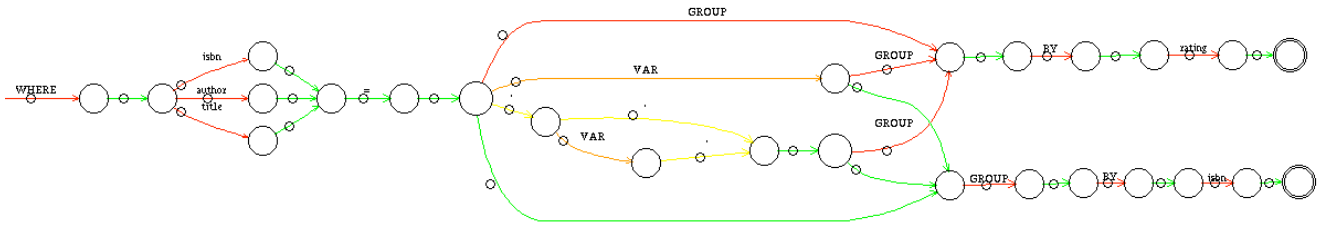


Figure 2. Excerpt of the command form model for the code in Figure 1.

the models, such that the sum of the edge values along each path is unique and the encoding is minimal. Since each command form corresponds to a unique path in the command form model, the unique integer associated with a path can be used as the ID for the corresponding command form. Moreover, because the path encoding is minimal, the largest path ID gives the total number of requirements for a database interaction point. This allows us to calculate the total number of testing requirements and assign unique IDs to requirements without having to enumerate all of the command forms.

As an example, Figure 2 shows an excerpt of the command form model for the database interaction point of the code in Figure 1. The command form shown in Section 4.1 corresponds to a specific path in this command form model.

The size of the command form model can, in the worst case, be quadratic with respect to the size of the program [6]. However, this worst case corresponds to a program that, at every statement, modifies the command string and has a branch. As Tables 1 and 2 show, the models tend to be linear with respect to the size of the application.

4.3 Coverage Collection

To measure the adequacy of a test suite with respect to our coverage criterion, we monitor the execution of the application and determine which command forms are exercised. We consider a command form associated with a database interaction point to be covered if, during execution, an SQL command that corresponds to the command form is issued at that point. An SQL command *corresponds* to a command form if they differ only in the value of the command form's indeterminate part For example, the query:

```
SELECT title, author, description, avg(rating)
FROM books WHERE author = 'Edward Bunker' GROUP
BY rating
```

would match the command form

```
SELECT title, author, description, avg(rating)
FROM books WHERE author = ' ' GROUP BY rating
```

because the former can be obtained from the latter by replacing the placeholder with the string "Edward Bunker."

We collect coverage information by inserting a call to a monitor immediately before each database interaction point. At runtime, the monitor is invoked with two parameters: the string that contains the actual SQL command about to be executed and a unique identifier for the interaction point. First, the monitor parses the command string into a sequence of SQL tokens. Second, using the interaction point's identifier, it retrieves the corresponding SQL command form model. To find which command form corresponds to the command string, the monitor traverses the model by matching SQL tokens and transition labels until it reaches an accepting state. (Label can match any number of tokens.) At the end of the traversal, the path followed corresponds to the command form covered by the command string, and the ID of the command form is given by the sum of the edge values associated with the transitions in the traversed path. At this point, the monitor adds to the set of coverage data a pair consisting of the ID of the covered command form and the ID of the database interaction point.

4.4 Coverage Analysis and Reporting

Given a set of coverage data, the database command form coverage measure can be expressed as:

$$coverage = \frac{number\ of\ command\ forms\ covered}{total\ number\ of\ command\ forms}$$

The number of command forms covered is simply the number of unique entries in the coverage data. The total number of command forms is given, as discussed in Section 4.2, by the sum of each database interaction point's maximum command form ID. All command form IDs that do not appear in the coverage data correspond to command forms that were not covered during testing. Given an ID, we can easily reconstruct the string representation of the corresponding command form and show it to the testers. To do this, we use the same approach used to reconstruct paths from path IDs in Ball and Larus's profiling approach [1].

5 The DITTO Tool

To automate the use of our testing approach and enable experimentation, we designed and implemented a prototype tool called DITTO (Database Interaction Testing TOol). DITTO is implemented in Java, provides fully automated support for all aspects of our approach, and can guide the developer in testing database applications written in Java. Figure 3 provides a high-level view of DITTO's architecture. As the figure shows, DITTO has three main operating modes and consists of several modules.

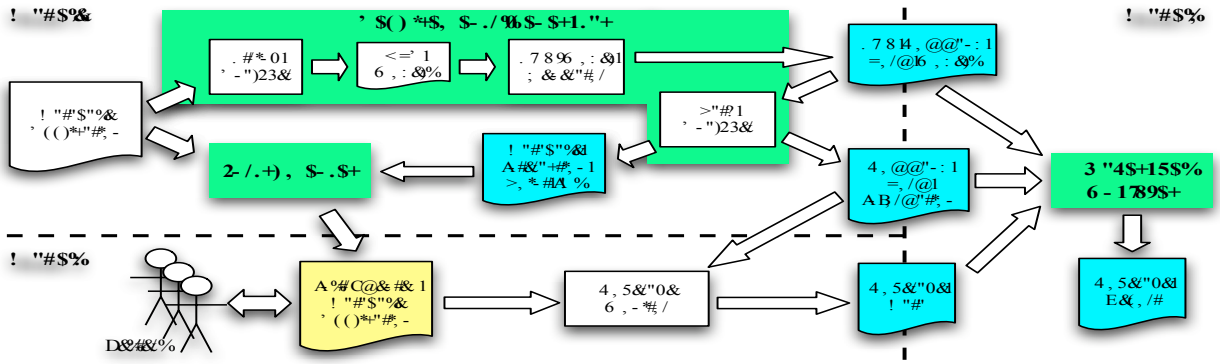


Figure 3. High-level overview of DITTO.

We expect that in a typical usage scenario DITTO would be used iteratively to support the testing process. Testers would create a set of test cases for their application or use a previously-developed test suite. Then they would use DITTO to instrument the application (Mode 1), run their test cases against the application (Mode 2), and get a coverage report (Mode 3). If testers are not satisfied with the level of coverage achieved, DITTO can provide detailed feedback about which command forms were not covered. The feedback can include both a visual display of the command form models, marked with coverage information, and a textual list of uncovered command forms. Testers can use this information to guide the development of new test cases. At this point, DITTO would be used again in Modes 2 and 3 to assess whether the additional test cases helped improved coverage. As in traditional testing, this process could continue until the testers are either satisfied with the coverage results or run out of resources.

5.1 Mode 1: Instrumentation

In Mode 1 DITTO generates the command form models and instruments the code for collecting coverage data.

To generate the command form models, DITTO statically analyzes the database application under test, as discussed in Section 4.2. For each database interaction point, the *String Analyzer* uses the JSA library [6] to produce an NFA model of the SQL command string used at that point. The *SQL-Model Generator* uses a modified version of our AMNESIA tool [12] to process the NFA models and generate the corresponding SQL command form models. Finally, the *Path Analyzer* takes as input the SQL command form models, annotates them with the edge values for the path encoding, and generates some command form information used for bookkeeping.

To produce coverage data at runtime, the *Instrumenter* modifies the code as described in Section 4.3. The *Instrumenter* inserts a call to the *Coverage Monitor* immediately before each database interaction point. The call to the monitor provides as parameters (1) the string variable that contains the SQL command about to be executed and (2) the unique identifier for the database interaction point. The instrumentation is performed using bytecode rewriting and leverages the Byte Code Engineering Library (BCEL – http://jakarta.apache.org/bcel/). For our example application, the *Instrumenter* would modify the database interaction point at line 14 (Figure 1) as follows:

```
...
monitor.log(<interaction point ID>,queryStr);
return database.executeQuery(queryStr);
...
```

5.2 Mode 2: Execution

In Mode 2 DITTO collects coverage data and records it for later analysis. The instrumented database application executes normally until it reaches a database interaction point. At this point, the string that is about to be submitted as an SQL command is sent to the *Coverage Monitor* together with the interaction point's ID. The monitor traverses the command form model for that interaction point, as described in Section 4.3, and logs the pair consisting of the ID of the covered command form and the ID of the database interaction point.

5.3 Mode 3: Analysis and Reporting

In Mode 3 DITTO computes the command form coverage measure and provides feedback to the testers. The *Coverage Analyzer* uses the coverage data collected in Mode 2 and calculates the coverage as described in Section 4.4. The test adequacy score alone does not give testers any information about which parts of the code were insufficiently exercised. To provide more detailed feedback, DITTO also allows testers to visually examine the command form models and see which paths were not covered by their tests. This information is visualized by coloring and annotating covered paths in the models. The testers can also list the command forms that were not covered in the model in textual format. Both of these feedback mechanisms provide testers with an intuitive way to understand coverage results and can guide further test-case development.

6 Current Limitations

Stored procedures. A common development practice is to encapsulate sequences of SQL commands and save them in the database as stored procedures. Developers can then issue a SQL command that invokes the stored procedure, possibly with some input parameters. Our approach models calls to stored procedures just like any other command issued to the database, but does not consider the SQL commands within a stored procedure (as they are stored in the database and not explicitly generated by the application). In other words, our current approach treats stored procedures as atomic instructions. If needed, the coverage criterion could be expanded to include the contents of stored procedures.

External fragments. In some applications, developers input constant strings from external sources, such as files, and use these strings to build SQL commands. These external strings are typically SQL command fragments that contain SQL keywords and operators (in contrast with user input, which typically consists of string or numeric literals). This situation does not cause any conceptual problem for our approach, as an indeterminate part of a command form can match the tokens that correspond to external fragments. From a practical standpoint, however, simply considering external fragments as indeterminate parts may decrease the effectiveness of the criterion. (For an extreme example, consider the case in which all SQL commands are simply read from external files.) This limitation is mostly implementation related: we could extend our technique so that developers can specify which external fragments are used in their application, and the technique would account for these fragments when building the SQL command form model. We have not implemented this solution yet because none of the applications that we have examined so far uses external fragments.

Infeasibility. Infeasibility is one of the main problems for structural coverage criteria. Computing structural coverage requirements for an application typically involves some form of static analysis of the application's code. In general, because determining the reachability of a statement for a given program is an undecidable problem [20], static analysis tends to generate spurious requirements that cannot be satisfied. The presence of unsatisfiable requirements in a criterion makes it impossible to reach 100% coverage for that criterion and limits its usefulness. Infeasibility can affect the command form criterion by causing the presence of spurious command forms that do not correspond to any command that could be generated by the application. Intuitively, this problem should occur primarily because the string analysis may add to the model strings that are generated along infeasible paths. Therefore, we expect the infeasibility problem to affect us to a similar extent in which it affects path-based coverage criteria. As discussed in Section 9, we plan to investigate infeasibility issues for our criterion through empirical evaluation.

Servlet	*LOC*	*# Methods*
Header	130	9
AdvSearch	253	13
Default	693	26
CategoriesGrid	309	18
CardTypesGrid	270	17
OrdersRecord	463	20
MembersInfo	488	21
CardTypesRecord	368	18
Footer	129	9
Login	290	14
EditorialCatGrid	310	18
EditorialsGrid	325	18
ShoppingCartRecord	412	19
Registration	515	20
CategoriesRecord	368	18
EditorialsRecord	441	19
Books	534	22
EditorialCatRecord	365	18
MembersRecord	618	22
BookMaint	514	21
MyInfo	649	19
BookDetail	921	25
AdminBooks	609	22
OrdersGrid	602	20
ShoppingCart	705	21
AdminMenu	429	11
MembersGrid	578	20

Table 1. Summary information about Bookstore's servlets.

Analysis limitations. Our approach relies on the ability of the underlying string analysis to build the initial NFA models for the database interaction points. Imprecision (i.e., over-approximation) in the string analysis could limit the effectiveness of our criterion. For example, a worst case scenario in which the analysis generates an automaton that accepts any strings would result in command-form models that are covered by any test case that reaches the corresponding database interaction point. Note that manual inspections showed that imprecision was not an issue for any of the models that we generated in our evaluation.

7 Evaluation

In our evaluation, we performed two studies. The first one is a proof of concept study in which we used DITTO on a real database application to assess whether it was able to successfully generate test requirements and measure coverage. The second study explores the effectiveness of traditional coverage criteria in generating test suites that are adequate with respect to command-form coverage.

Servlet	*# DIP*	*# Edges*	*# States*	*% Cov.*
Header	0	0	0	N/A
AdvSearch	1	16	17	N/A
Default	1	406	407	13%
CategoriesGrid	1	48	49	N/A
CardTypesGrid	1	48	49	N/A
OrdersRecord	2	371	259	<1%
MembersInfo	2	201	172	7%
CardTypesRecord	2	191	143	2%
Footer	1	1	2	N/A
Login	1	67	58	4%
EditorialCatGrid	1	48	49	N/A
EditorialsGrid	1	157	158	N/A
ShoppingCartRecord	2	296	210	N/A
Registration	3	478	309	<1%
CategoriesRecord	2	191	143	5%
EditorialsRecord	2	527	345	<2%
Books	1	7928	6267	N/A
EditorialCatRecord	2	191	143	2%
MembersRecord	3	1282	772	<1%
BookMaint	2	1163	681	<1%
MyInfo	2	588	354	N/A
BookDetail	4	1211	854	1%
AdminBooks	1	344	258	N/A
OrdersGrid	1	326	265	N/A
ShoppingCart	2	154	140	N/A
AdminMenu	1	1	2	N/A
MembersGrid	1	235	207	N/A

Table 2. Information on the SQL command form models for Bookstore.

For both studies, we used a database application called Bookstore (available at http://www.gotocode.com). Bookstore implements an online bookstore and uses Java servlets to implement the UI and application tiers. Table 1 shows summary information about each of the servlets in the application. For each servlet (*Servlet*), the table shows its size (*LOC*) and its number of methods (*# Methods*).

7.1 Study 1

The first study provides a proof of concept evaluation of DITTO by showing that it can work on a specific application. To achieve this goal, we used DITTO on Bookstore to generate testing requirements and measure command form coverage for a set of test cases. DITTO successfully computed the test requirements for each of the database interaction points and instrumented all of the servlets. The entire process of extracting the models took less than five minutes on a Pentium III machine with 1GB of memory running the GNU/Linux Operating System. We then deployed the instrumented servlets and ran a previously developed test suite against them.

Table 2 summarizes the results of the study. For each servlet, the table shows the number of database interaction points it contained (*#DIP*), the total number of states and transitions in the models (*#States* and *#Edges*), and the percentage of command-form coverage achieved during testing (*%Cov.*). Some of the servlets were not exercised by the test suite, and their coverage measure is reported as "N/A."

The test suite that we used in this study was developed in previous work [12] (and also used in related work [16]) to target specific security issues. It was not developed to achieve coverage, and we did not try to improve it because the goal of this study was not to test the subject application, but to demonstrate a successful use of DITTO. Even under these premises, the results provide some initial evidence that command-form coverage cannot be trivially achieved, and that specialized test cases may be needed to suitably exercise the interactions between applications and their underlying databases.

7.2 Study 2

The second study addresses the research question: *Does command-form coverage provide for a more thorough testing of database applications than alternative traditional approaches?* For this study, we selected branch coverage as the representative traditional criterion because it is widely used. A typical way to address this question would be to (1) create a number of branch-adequate test suites, (2) create the same number of adequate test suites for the command-form coverage criterion, (3) run both sets of test suites on several versions of an application with seeded faults, and (4) compare the fault-detection capability of both sets of test suites.

However, there is a significant technical challenge that complicates this type of evaluation: the lack of an effective way to automatically seed different types of SQL-related errors. Whereas there are mutant generation tools that can be used to seed traditional faults in programs, there are no such tools for SQL-related faults. Seeding the errors by hand or building an ad-hoc tool are less than ideal options because they would introduce problems of bias. Alternatively, collecting real database-command related faults from open-source projects would be a good solution, but may involve an extensive search and still result in too few data points to draw significant conclusions.

Due to these issues, we decided to use an indirect and approximated method to compare the effectiveness of our criterion with the effectiveness of the traditional branch-coverage criterion. The method that we use is to compute an upper bound to the number of command forms that could be exercised by a branch-adequate test suite and compare this number to the total number of command forms for the application. A higher total number of command forms would be an indication that branch coverage (and possibly other traditional testing criteria) may not adequately test interactions between the application and the database, and that command-form coverage may be needed. For instance, consider the example code in Figure 1. As discussed in Sec-

tion 3, we could achieve 100% branch coverage of that code with just three test cases, each of which would exercise only one command form. Because there are eighteen possible command forms, it is clear that the considered branch-adequate test suite would not thoroughly exercise the SQL commands generated by the application.

The total number of command forms for an application is computed by DITTO, as discussed in Section 4.2. To calculate an upper bound to the number of command forms that would be executed in a servlet by a branch-adequate test suite, we use the cyclomatic complexity of the servlet. The cyclomatic complexity is an upper bound to the minimal number of test cases needed to achieve 100% branch coverage of a program [19]. An analysis of the servlets used in the study revealed that no test case can execute a database interaction point more than once (i.e., no database interaction point is in the body of a loop). Therefore, if we conservatively assume that each test case exercises a different command form, we can use the cyclomatic complexity as an upper bound to the number of possible command forms that a minimal, branch-adequate test suite would execute. In practice, however, such an assumption could vastly overestimate the number of command forms exercised by a branch-adequate test suite because many paths in the code do not actually generate a database command. To obtain a better estimate, we must compute the cyclomatic complexity on only the subset of the servlet code that is involved with creating, modifying, and executing database commands. We thus generate an executable backward slice [18] for each command string variable at each database interaction point using JABA[2] and compute the cyclomatic complexity only for the subset of the servlet in the slice. Because the JABA-based slicer that we use is still a prototype and requires a considerable amount of human intervention, in the study we consider only a subset of the Bookstore servlets.

The results of our analysis are shown in Table 3. For each servlet considered, we report the number of database interaction points (*#DIP*), the number of command forms (*# Command Forms*), and the cyclomatic complexity of the servlet's slice. As the data shows, the number of command forms is considerably higher than the cyclomatic complexity in several cases, and the average number of command forms per database interaction point (253) is almost five times the average cyclomatic complexity (57). Because the numbers we used in the study are estimates, and we only considered a small number of servlets, we cannot draw any definitive conclusion from the study. Nevertheless, this preliminary study indicates that command-form coverage may result in a more thorough testing of database interactions than traditional coverage criteria. These results encourage further research and a more extensive empirical evaluation.

[2] http://www.cc.gatech.edu/aristotle/Tools/jaba.html

Servlet	*# DIP*	*# Command Forms*	*Cyclomatic Complexity*
MyInfo	1	6	136
BookDetail	4	1583	150
AdminBooks	1	617	31
OrdersGrid	1	394	26
ShoppingCart	2	20	28
AdminMenu	1	1	6
MembersGrid	1	162	21

Table 3. Results of the evaluation.

8 Related Work

The problem of ensuring the correctness of database applications has been approached in several different ways. The approaches most closely related to ours are those that also propose new test adequacy criteria for database applications. Within this group, there are two types of criteria, those that focus on data-flow and those that focus on the structure of the SQL commands sent to the database. Suárez-Cabal and Tuya [17] propose a structural coverage criterion that requires the coverage of all of the conditions in a SQL command's "FROM," "WHERE," and "JOIN" clauses. This criterion is analogous to the multiple condition coverage criterion [5], but applied to SQL clauses instead of code predicates. This work differs from ours in that it focuses on SQL commands that are completely statically defined and only considers coverage of a subset of the SQL language, namely, conditions in queries' clauses. In contrast, our technique considers coverage of all types of SQL commands, including dynamically-constructed ones. Also similar to our criterion are the criteria proposed by Willmor and Embury [22]. In particular, they propose the *all database operations* criterion, which requires the execution of all of a program's database interaction points. Our proposed criterion subsumes this criterion because it requires not only the execution of each database interaction point, but also the coverage of all of the command forms that can be generated at that point.

Kapfhammer and Soffa [13] propose a set of data-flow test adequacy criteria based on definition-use coverage of database entities. These entities can be defined at different levels of granularity that include the database, relation, attribute, record, and attribute-value level. These criteria parallel conventional data-flow testing criteria [10], but are defined and evaluated based on database entities instead of program variables. Willmor and Embury [22] refine these criteria and expand them to accommodate database transactions. Both approaches differ from ours in that they focus on covering all of the definitions and uses of database entities instead of the different command forms. The data-flow criteria do not subsume command form coverage because at a database interaction point it is possible to have several command forms that exercise the same set of database en-

tities. In this case, satisfying the data-flow criteria would not satisfy command form coverage. The data flow criteria are complementary to ours; they target faults related to the definition and use of database entities, whereas our approach targets errors in the database commands generated by a database application.

The work of Chan and Cheung [2] is similar to ours in that it aims to thoroughly test database applications by taking into account the generated SQL commands. Their approach translates SQL commands into Relational Algebra Expressions (RAE), converts the RAE into the meta-language of the application, and replaces the SQL command in the application with the generated code. After the transformation, they use standard white-box testing criteria to test, albeit indirectly, the SQL commands. To illustrate their approach, consider the case in which the developer issues a SQL JOIN command. They would first convert the SQL command into equivalent statements in the meta-language. In this case, the JOIN would be translated into two nested for loops (the JOIN command is similar to a cross product between two database tables). Testers would then create test cases to properly exercise the additional for loops in the code. (Using our approach, the JOIN command would simply be counted as an additional command form to be covered.) Chan and Cheung's approach enforces a thorough testing of database applications, but it has several limitations when compared to our approach. First, and most importantly, the translation of the SQL commands into RAE requires that the SQL commands be statically defined as constant strings. This is a fundamental limitation because it precludes the usage of the technique on the many database applications that build command strings by appending different substrings along non-trivial control-flow paths. Our approach does not have this problem because the static analysis can typically account for all possible commands, including dynamically-constructed ones. Another limitation is that the RAE is less expressive than SQL, so certain SQL commands cannot be translated and will not be adequately tested. We are not affected by this issue because we measure coverage directly on the database command forms.

Another proposed approach is to perform static verification of the possible SQL commands. Christensen, Møller, and Schwartzbach introduce the Java String Analysis (JSA) [6] and use it to extract non-deterministic finite automata that represent the potential SQL commands that could be generated at a given database interaction point. They then intersect the automata with a regular language approximation of SQL to determine if the commands are syntactically correct. Gould, Su, and Devanbu propose JDBC Checker [11], which builds on JSA and adds type analysis to statically verify that dynamically generated commands are type-safe. This type of verification is powerful, but does not necessarily eliminate the need for testing. First, it is not always possible to check SQL commands statically (e.g., in cases where the application allows keywords or operators to be specified at runtime). Second, there are limitations in the type of errors that can be detected by these techniques. For instance, consider the third error in our example from Section 3. This type of fault would not be detected by Christensen, Møller, and Schwartzbach's approach because their syntax checking is not expressive enough to represent the constraints violated by the error. Although our approach uses similar models as these techniques, it uses them for measuring the thoroughness of a test suite with respect to command forms instead of for verifying them. Our technique, although less complete than the ones based on static verification, does not have the limitations of these techniques and may reveal faults that these techniques cannot reveal.

Other approaches, such as SQL DOM [14] and Safe Query Objects [7], propose to change the way developers construct SQL commands. Instead of having developers create SQL commands using string concatenation, they offer a specialized API that handles all aspects of creating and issuing SQL command strings. The main benefit of these approaches is that they can enforce a more disciplined usage of SQL, and thus prevent many errors. However, these approaches require developers to learn a new API and development paradigm and, most importantly, cannot be easily applied to legacy code.

Finally, other related work focuses on test case generation for database applications and regression testing. Although related to our approach, they have different goals and are mostly orthogonal to our work. AGENDA [3, 4, 9] is a framework for automatic generation and execution of unit tests for database applications that is loosely based on the category partition testing method [15]. AGENDA takes as input information about the logical database model (e.g., schema information, database states, and logical constraints) and combines it with tester input to generate test cases for the database. Zhang, Xu, and Cheung propose a technique for generating database instances to support testing [24]. The technique uses a constraint solver to identify which values a database should contain to ensure that the different conditions and predicates in an application's SQL commands will be exercised. Similarly, Willmor and Embury [23] propose a mechanism that allows developers to specify database states that are relevant for a test suite and can then appropriately populate the database. Daou and colleagues use a firewall-based approach for regression testing of database applications [8], while Willmor and Embury propose regression testing based on definition-use analysis of the SQL commands in an application [21].

9 Conclusion

In this paper, we addressed a common problem that arises when testing database applications: how to adequately test

the interactions between an application and its underlying database. To address this problem, we introduced an approach based on a new test adequacy criterion called command form coverage. This criterion requires the coverage of all of the command forms that a given application can issue to its database.

We also presented DITTO, a prototype tool that implements our approach. DITTO generates testing requirements for our criterion, measures the adequacy of a test suite with respect to the criterion, and provides feedback to testers about which requirements were not covered during testing.

Finally, we presented two preliminary studies. The first one is a feasibility study that shows that DITTO can successfully extract testing requirements and measure coverage for a real database application. The second study provides analytical evidence that traditional code-based testing criteria may be inadequate in the case of database applications. The results of the studies, although preliminary, are encouraging and motivate further research.

There are several possible directions for future work. *First*, we will perform a more extensive empirical evaluation of our approach. We will identify additional subjects and fault information for these subjects by performing a survey of existing database applications. We will then use these subjects to (1) assess the effectiveness of our criterion in revealing database-application-specific errors, (2) further compare our criterion and traditional code-based criteria, and (3) study infeasibility and other analysis-related issues for our approach. *Second*, we will investigate whether we can improve the effectiveness of our approach by leveraging information about the database used by the application under test (e.g., the database schema). *Finally*, we will investigate the application of our technique to other domains, such as dynamic web applications.

Acknowledgments

This work was partially supported by NSF awards CCR-0205422 and CCR-0306372 to Georgia Tech and by the Department of Homeland Security and US Air Force under Contract No. FA8750-05-C-0179.

References

[1] T. Ball and J. R. Larus. Efficient Path Profiling. In *Proc. of Micro 96*, pages 46–57, Dec. 1996.

[2] M. Chan and S. Cheung. Testing Database Applications with SQL Semantics. In *CODAS'99: Proc. of 2nd International Symposium on Cooperative Database Systems for Advanced Applications*, pages 363–374, Dec. 1999.

[3] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weyuker. A Framework for Testing Database Applications. In *ISSTA '00: Proc. of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 147–157, Aug. 2000.

[4] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker An AGENDA for Testing Relational Database Applications. *Journal of Software Testing, Verification and Reliability*, Mar. 2004.

[5] J. Chilenski and S. Miller. Applicability of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal*, 9(5):193–200, Sep. 1994.

[6] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise Analysis of String Expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, pages 1–18, Jun. 2003.

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

[8] B. Daou, R. A. Haraty, and N. Mansour. Regression Testing of Database Applications. In *SAC '01: Proc. of the 2001 ACM Symposium on Applied Computing*, pages 285–289, 2001.

[9] Y. Deng, P. Frankl, and D. Chays. Testing Database Transactions with AGENDA. In *ICSE '05: Proc. of the 27th International Conference on Software Engineering*, pages 78–87, 2005.

[10] P. G. Frankl and E. J. Weyuker. An Applicable Family of Data Flow Testing Criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, Oct. 1988.

[11] C. Gould, Z. Su, and P. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In *Proc. of the 26th International Conference on Software Engineering (ICSE 04)*, pages 645–654, May 2004.

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

[13] G. M. Kapfhammer and M. L. Soffa. A Family of Test Adequacy Criteria for Database-Driven Applications. In *ESEC/FSE-11: Proc. of the 9th European Software Engineering Conference, held jointly with, 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 98–107, Sep. 2003.

[14] R. McClure and I. Krüger. SQL DOM: Compile Time Checking of Dynamic SQL Statements. In *Proc. of the 27th International Conference on Software Engineering (ICSE 05)*, pages 88–96, May 2005.

[15] T. J. Ostrand and M. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of the ACM*, 31(6), Jun. 1988.

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

[17] M. J. Suárez-Cabal and J. Tuya. Using an SQL Coverage Measurement for Testing Database Applications. In *Proc. of the 12th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 253–262, Oct. 2004.

[18] F. Tip. A Survey of Program Slicing Techniques. *Journal Of Programming Languages*, 31(5):32–40, May 1998.

[19] A. H. Watson and T. J. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. Technical Report 500-235, NIST Special Publication, Aug. 1996.

[20] E. J. Weyuker. The Applicability of Program Schema Results to Programs. *International Journal of Parallel Programming*, 8(5):387–403, Oct. 1979.

[21] D. Willmor and S. M. Embury. A safe regression test selection technique for database-driven applications. In *Proc. of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 421–430, Sep. 2005.

[22] D. Willmor and S. M. Embury. Exploring test adequacy for database systems. In *Proceedings of the 3rd UK Software Testing Research Workshop (UKTest 2005)*, pages 123–133, Sep. 2005.

[23] D. Willmor and S. M. Embury. An Intensional Approach to the Specification of Test Cases for Database Systems. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*, pages 102–111, May 2006.

[24] J. Zhang, C. Xu, and S. C. Cheung. Automatic Generation of Database Instances for White-box Testing. In *COMPSAC '01: Proc. of the 25th International Computer Software and Applications Conference*, pages 161–165, Oct. 2001.

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 e ective and e cient 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 insu cient input validation. More precisely, SQL Injection Attacks (SQLIAs) 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 e ective 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 e ective 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-

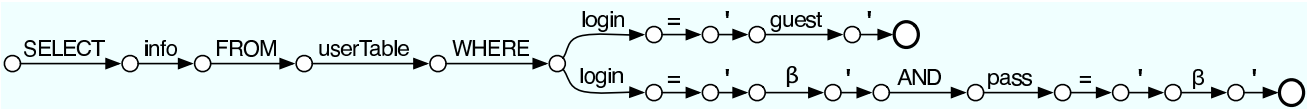


Figure 2: SQL-query model for the servlet in Figure 1.

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 e ects, 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 di erent dialects of SQL.) In our model, we mark transitions that correspond to externally defined strings with the symbol .

To illustrate, Figure 2 shows the SQL-query model for the hotspot in the example provided in Section 2. The model reflects the two di erent 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, 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,) can match either an identical literal value or a 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.

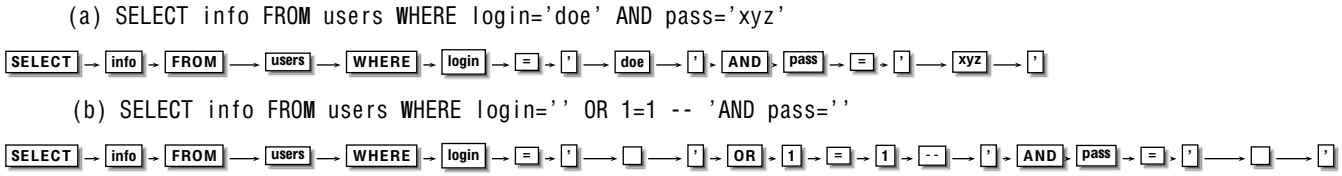


Figure 4: Example of parsed runtime queries.

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.

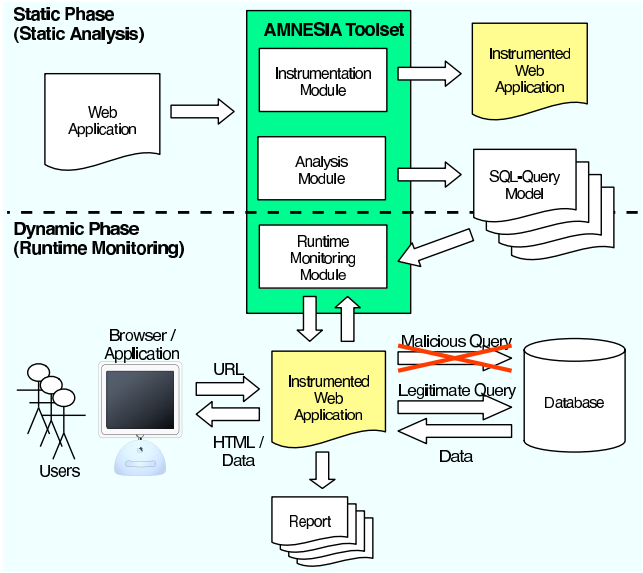


Figure 5: High-level overview of AMNESIA.

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 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 e ective, these approaches have limitations that affect their ability to provide general detection and prevention capabilities against SQLIAs [9]. Furthermore, some of these approaches are di cult 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 o ering an e ective solution to SQLIAs, has shown to be di cult to apply e ectively 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 e ectiveness of AMNESIA. The results of this evaluation show that AMNESIA can be very e ective and e cient 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, 2nd 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 11th 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.

A Classification of SQL Injection Attacks and Countermeasures

William G.J. Halfond, Jeremy Viegas, and Alessandro Orso
College of Computing
Georgia Institute of Technology
{whalfondljeremyvlorso}@cc.gatech.edu

ABSTRACT

SQL injection attacks pose a serious security threat to Web applications: they allow attackers to obtain unrestricted access to the databases underlying the applications and to the potentially sensitive information these databases contain. Although researchers and practitioners have proposed various methods to address the SQL injection problem, current approaches either fail to address the full scope of the problem or have limitations that prevent their use and adoption. Many researchers and practitioners are familiar with only a subset of the wide range of techniques available to attackers who are trying to take advantage of SQL injection vulnerabilities. As a consequence, many solutions proposed in the literature address only some of the issues related to SQL injection. To address this problem, we present an extensive review of the different types of SQL injection attacks known to date. For each type of attack, we provide descriptions and examples of how attacks of that type could be performed. We also present and analyze existing detection and prevention techniques against SQL injection attacks. For each technique, we discuss its strengths and weaknesses in addressing the entire range of SQL injection attacks.

1. INTRODUCTION

SQL injection vulnerabilities have been described as one of the most serious threats for Web applications [3, 11]. Web applications that are vulnerable to SQL injection may allow an attacker to gain complete access to their underlying databases. Because these databases often contain sensitive consumer or user information, the resulting security violations can include identity theft, loss of confidential information, and fraud. In some cases, attackers can even use an SQL injection vulnerability to take control of and corrupt the system that hosts the Web application. Web applications that are vulnerable to SQL Injection Attacks (SQLIAs) are widespread—a study by Gartner Group on over 300 Internet Web sites has shown that most of them could be vulnerable to SQLIAs. In fact, SQLIAs have successfully targeted high-profile victims such as Travelocity, FTD.com, and Guess Inc.

SQL injection refers to a class of code-injection attacks in which data provided by the user is included in an SQL query in such a way that part of the user's input is treated as SQL code. By leveraging these vulnerabilities, an attacker can submit SQL commands directly to the database. These attacks are a serious threat to any Web application that receives input from users and incorporates it into SQL queries to an underlying database. Most Web applications used on the Internet or within enterprise systems work this way and could therefore be vulnerable to SQL injection.

The cause of SQL injection vulnerabilities is relatively simple and well understood: insufficient validation of user input. To address this problem, developers have proposed a range of coding guidelines (e.g., [18]) that promote defensive coding practices, such as encoding user input and validation. A rigorous and systematic application of these techniques is an effective solution for preventing SQL injection vulnerabilities. However, in practice, the application of such techniques is human-based and, thus, prone to errors. Furthermore, fixing legacy code-bases that might contain SQL injection vulnerabilities can be an extremely labor-intensive task.

Although recently there has been a great deal of attention to the problem of SQL injection vulnerabilities, many proposed solutions fail to address the full scope of the problem. There are many types of SQLIAs and countless variations on these basic types. Researchers and practitioners are often unaware of the myriad of different techniques that can be used to perform SQLIAs. Therefore, most of the solutions proposed detect or prevent only a subset of the possible SQLIAs. To address this problem, we present a comprehensive survey of SQL injection attacks known to date. To compile the survey, we used information gathered from various sources, such as papers, Web sites, mailing lists, and experts in the area. For each attack type considered, we give a characterization of the attack, illustrate its effect, and provide examples of how that type of attack could be performed. This set of attack types is then used to evaluate state of the art detection and prevention techniques and compare their strengths and weaknesses. The results of this comparison show the effectiveness of these techniques.

The rest of this paper is organized as follows: Section 2 provides background information on SQLIAs and related concepts. Section 4 defines and presents the different attack types. Sections 5 and 6 review and evaluate current techniques against SQLIAs. Finally, we provide summary and conclusions in Section 7.

2. BACKGROUND ON SQLIAS

Intuitively, an *SQL Injection Attack (SQLIA)* occurs when an attacker changes the intended effect of an SQL query by inserting new SQL keywords or operators into the query. This informal definition is intended to include all of the variants of SQLIAs reported in literature and presented in this paper. Interested readers can refer to [35] for a more formal definition of SQLIAs. In the rest of this section, we define two important characteristics of SQLIAs that we use for describing attacks: injection mechanism and attack intent.

2.1 Injection Mechanisms

Malicious SQL statements can be introduced into a vulnerable application using many different input mechanisms. In this section, we explain the most common mechanisms.

Injection through user input: In this case, attackers inject SQL commands by providing suitably crafted user input. A Web application can read user input in several ways based on the environment in which the application is deployed. In most SQLIAs that target Web applications, user input typically comes from form submissions that are sent to the Web application via HTTP `GET` or `POST` requests [14]. Web applications are generally able to access the user input contained in these requests as they would access any other variable in the environment.

Injection through cookies: Cookies are files that contain state information generated by Web applications and stored on the client machine. When a client returns to a Web application, cookies can be used to restore the client's state information. Since the client has control over the storage of the cookie, a malicious client could tamper with the cookie's contents. If a Web application uses the cookie's contents to build SQL queries, an attacker could easily submit an attack by embedding it in the cookie [8].

Injection through server variables: Server variables are a collection of variables that contain HTTP, network headers, and environmental variables. Web applications use these server variables in a variety of ways, such as logging usage statistics and identifying browsing trends. If these variables are logged to a database without sanitization, this could create an SQL injection vulnerability [30]. Because attackers can forge the values that are placed in HTTP and network headers, they can exploit this vulnerability by placing an SQLIA directly into the headers. When the query to log the server variable is issued to the database, the attack in the forged header is then triggered.

Second-order injection: In second-order injections, attackers seed malicious inputs into a system or database to indirectly trigger an SQLIA when that input is used at a later time. The objective of this kind of attack differs significantly from a regular (i.e., first-order) injection attack. Second-order injections are not trying to cause the attack to occur when the malicious input initially reaches the database. Instead, attackers rely on knowledge of where the input will be subsequently used and craft their attack so that it occurs during that usage. To clarify, we present a classic example of a second order injection attack (taken from [1]). In the example, a user registers on a website using a seeded user name, such as "`admin' --` ". The application properly escapes the single quote in the input before storing it in the database, preventing its potentially malicious effect. At this point, the user modifies his or her password, an operation that typically involves (1) checking that the user knows the current password and (2) changing the password if the check is successful. To do this, the Web application might construct an SQL command as follows:

```
queryString="UPDATE users SET password='" + newPassword +
"' WHERE userName='" + userName + "' AND password='" +
oldPassword + "'"
```

`newPassword` and `oldPassword` are the new and old passwords, respectively, and `userName` is the name of the user currently logged-in (i.e., `''admin'--''`). Therefore, the query string that is sent to the database is (assume that `newPassword` and `oldPas-sword` are "newpwd" and"oldpwd"):

```
UPDATE users SET password='newpwd'
WHERE userName= 'admin'--' AND password='oldpwd'
```

Because "`--`" is the SQL comment operator, everything after it is ignored by the database. Therefore, the result of this query is that the database changes the password of the administrator ("admin") to an attacker-specified value.

Second-order injections can be especially difficult to detect and prevent because the point of injection is different from the point where the attack actually manifests itself. A developer may properly escape, type-check, and filter input that comes from the user and assume it is safe. Later on, when that data is used in a different context, or to build a different type of query, the previously sanitized input may result in an injection attack.

2.2 Attack Intent

Attacks can also be characterized based on the goal, or *intent*, of the attacker. Therefore, each of the attack type definitions that we provide in Section 4 includes a list of one or more of the attack intents defined in this section.

Identifying injectable parameters: The attacker wants to probe a Web application to discover which parameters and user-input fields are vulnerable to SQLIA.

Performing database finger-printing: The attacker wants to discover the type and version of database that a Web application is using. Certain types of databases respond differently to different queries and attacks, and this information can be used to "fingerprint" the database. Knowing the type and version of the database used by a Web application allows an attacker to craft database-specific attacks.

Determining database schema: To correctly extract data from a database, the attacker often needs to know database schema information, such as table names, column names, and column data types. Attacks with this intent are created to collect or infer this kind of information.

Extracting data: These types of attacks employ techniques that will extract data values from the database. Depending on the type of the Web application, this information could be sensitive and highly desirable to the attacker. Attacks with this intent are the most common type of SQLIA.

Adding or modifying data: The goal of these attacks is to add or change information in a database.

Performing denial of service: These attacks are performed to shut down the database of a Web application, thus denying service to other users. Attacks involving locking or dropping database tables also fall under this category.

Evading detection: This category refers to certain attack techniques that are employed to avoid auditing and detection by system protection mechanisms.

Bypassing authentication: The goal of these types of attacks is to allow the attacker to bypass database and application authentication mechanisms. Bypassing such mechanisms could allow the attacker to assume the rights and privileges associated with another application user.

Executing remote commands: These types of attacks attempt to execute arbitrary commands on the database. These commands can be stored procedures or functions available to database users.

Performing privilege escalation: These attacks take advantage of implementation errors or logical flaws in the database in order to escalate the privileges of the attacker. As opposed to bypassing authentication attacks, these attacks focus on exploiting the database user privileges.

3. EXAMPLE APPLICATION

Before discussing the various attack types, we introduce an example application that contains an SQL injection vulnerability. We use this example in the next section to provide attack examples.

```
 1.  String login, password, pin, query
 2.  login = getParameter("login");
 3.  password = getParameter("pass");
 3.  pin = getParameter("pin");
 4.  Connection conn.createConnection("MyDataBase");
 5.  query = "SELECT accounts FROM users WHERE login='" +
 6.          login + "' AND pass='" + password +
 7.          "' AND pin=" + pin;
 8.  ResultSet result = conn.executeQuery(query);
 9.  if (result!=NULL)
10.      displayAccounts(result);
11.  else
12.      displayAuthFailed();
```

Figure 1: Excerpt of servlet implementation.

Note that the example refers to a fairly simple vulnerability that could be prevented using a straightforward coding fix. We use this example simply for illustrative purposes because it is easy to understand and general enough to illustrate many different types of attacks.

The code excerpt in Figure 1 implements the login functionality for an application. It is based on similar implementations of login functionality that we have found in existing Web-based applications. The code in the example uses the input parameters `login`, `pass`, and `pin` to dynamically build an SQL query and submit it to a database.

For example, if a user submits login, password, and pin as "`doe,`" "`secret,`" and "`123,`" the application dynamically builds and submits the query:

```
SELECT accounts FROM users WHERE
login='doe' AND pass='secret' AND pin=123
```

If the login, password, and pin match the corresponding entry in the database, `doe`'s account information is returned and then displayed by function `displayAccounts()`. If there is no match in the database, function `displayAuthFailed()` displays an appropriate error message.

4. SQLIA TYPES

In this section, we present and discuss the different kinds of SQLIAs known to date. For each attack type, we provide a descriptive *name*, one or more *attack intents*, a *description* of the attack, an attack *example*, and a set of *references* to publications and Web sites that discuss the attack technique and its variations in greater detail.

The different types of attacks are generally not performed in isolation; many of them are used together or sequentially, depending on the specific goals of the attacker. Note also that there are countless variations of each attack type. For space reasons, we do not present all of the possible attack variations but instead present a single representative example.

Tautologies

Attack Intent: Bypassing authentication, identifying injectable parameters, extracting data.
Description: The general goal of a tautology-based attack is to inject code in one or more conditional statements so that they always evaluate to true. The consequences of this attack depend on how the results of the query are used within the application. The most common usages are to bypass authentication pages and extract data. In this type of injection, an attacker exploits an injectable field that is used in a query's `WHERE` conditional. Transforming the conditional into a tautology causes all of the rows in the database table targeted by the query to be returned. In general, for a tautology-based attack to work, an attacker must consider not only the injectable/vulnerable parameters, but also the coding constructs that evaluate the query results. Typically, the attack is successful when the code either displays all of the returned records or performs some action if at least one record is returned.
Example: In this example attack, an attacker submits " ' or 1=1 - - " for the *login* input field (the input submitted for the other fields is irrelevant). The resulting query is:

```
SELECT accounts FROM users WHERE
login='' or 1=1 -- AND pass='' AND pin=
```

The code injected in the conditional (`OR 1=1`) transforms the entire `WHERE` clause into a tautology. The database uses the conditional as the basis for evaluating each row and deciding which ones to return to the application. Because the conditional is a tautology, the query evaluates to true for each row in the table and returns all of them. In our example, the returned set evaluates to a non-null value, which causes the application to conclude that the user authentication was successful. Therefore, the application would invoke method `displayAccounts()` and show all of the accounts in the set returned by the database.
References: [1, 28, 21, 18]

Illegal/Logically Incorrect Queries

Attack Intent: Identifying injectable parameters, performing database finger-printing, extracting data.
Description: This attack lets an attacker gather important information about the type and structure of the back-end database of a Web application. The attack is considered a preliminary, information-gathering step for other attacks. The vulnerability leveraged by this attack is that the default error page returned by application servers is often overly descriptive. In fact, the simple fact that an error messages is generated can often reveal vulnerable/injectable parameters to an attacker. Additional error information, originally intended to help programmers debug their applications, further helps attackers gain information about the schema of the back-end database. When performing this attack, an attacker tries to inject statements that cause a syntax, type conversion, or logical error into the database. Syntax errors can be used to identify injectable parameters. Type errors can be used to deduce the data types of certain columns or to extract data. Logical errors often reveal the names of the tables and columns that caused the error.
Example: This example attack's goal is to cause a type conversion error that can reveal relevant data. To do this, the attacker injects the following text into input field *pin*: "convert(int,(select top 1 name from sysobjects where xtype='u'))". The resulting query is:

```
SELECT accounts FROM users WHERE login='' AND
pass='' AND pin= convert (int,(select top 1 name from
sysobjects where xtype='u'))
```

In the attack string, the injected select query attempts to extract the first user table (`xtype='u'`) from the database's metadata table (assume the application is using Microsoft SQL Server, for which the metadata table is called `sysobjects`). The query then tries to convert this table name into an integer. Because this is not a legal type conversion, the database throws an error. For Microsoft SQL Server, the error would be: *"Microsoft OLE DB Provider for SQL Server (0x80040E07) Error converting nvarchar value 'CreditCards' to a column of data type int."*
There are two useful pieces of information in this message that aid an attacker. First, the attacker can see that the database is an SQL Server database, as the error message explicitly states this fact. Second, the error message reveals the value of the string that caused the type conversion to occur. In this case, this value is also the name of

the first user-defined table in the database: "CreditCards." A similar strategy can be used to systematically extract the name and type of each column in the database. Using this information about the schema of the database, an attacker can then create further attacks that target specific pieces of information.
References: [1, 22, 28]

Union Query

Attack Intent: Bypassing Authentication, extracting data.
Description: In union-query attacks, an attacker exploits a vulnerable parameter to change the data set returned for a given query. With this technique, an attacker can trick the application into returning data from a table different from the one that was intended by the developer. Attackers do this by injecting a statement of the form: `UNION SELECT <rest of injected query>`. Because the attackers completely control the second/injected query, they can use that query to retrieve information from a specified table. The result of this attack is that the database returns a dataset that is the union of the results of the original first query and the results of the injected second query.
Example: Referring to the running example, an attacker could inject the text "' UNION SELECT cardNo from CreditCards where acctNo=10032 - -" into the login field, which produces the following query:

```
SELECT accounts FROM users WHERE login='' UNION
SELECT cardNo from CreditCards where
acctNo=10032 -- AND pass='' AND pin=
```

Assuming that there is no login equal to "", the original first query returns the null set, whereas the second query returns data from the "CreditCards" table. In this case, the database would return column "cardNo" for account "10032." The database takes the results of these two queries, unions them, and returns them to the application. In many applications, the effect of this operation is that the value for "cardNo" is displayed along with the account information.
References: [1, 28, 21]

Piggy-Backed Queries

Attack Intent: Extracting data, adding or modifying data, performing denial of service, executing remote commands.
Description: In this attack type, an attacker tries to inject additional queries into the original query. We distinguish this type from others because, in this case, attackers are not trying to modify the original intended query; instead, they are trying to include new and distinct queries that "piggy-back" on the original query. As a result, the database receives multiple SQL queries. The first is the intended query which is executed as normal; the subsequent ones are the injected queries, which are executed in addition to the first. This type of attack can be extremely harmful. If successful, attackers can insert virtually any type of SQL command, including stored procedures,[1] into the additional queries and have them executed along with the original query. Vulnerability to this type of attack is often dependent on having a database configuration that allows multiple statements to be contained in a single string.
Example: If the attacker inputs "'; drop table users - -" into the *pass* field, the application generates the query:

```
SELECT accounts FROM users WHERE login='doe' AND
pass=''; drop table users -- ' AND pin=123
```

After completing the first query, the database would recognize the query delimiter ("`;`") and execute the injected second query. The result of executing the second query would be to drop table `users`, which would likely destroy valuable information. Other types of queries could insert new users into the database or execute stored procedures. Note that many databases do not require a special character to separate distinct queries, so simply scanning for a query separator is not an effective way to prevent this type of attack.
References: [1, 28, 18]

[1]Stored procedures are routines stored in the database and run by the database engine. These procedures can be either user-defined procedures or procedures provided by the database by default.

Stored Procedures

Attack Intent: Performing privilege escalation, performing denial of service, executing remote commands.
Description: SQLIAs of this type try to execute stored procedures present in the database. Today, most database vendors ship databases with a standard set of stored procedures that extend the functionality of the database and allow for interaction with the operating system. Therefore, once an attacker determines which backend-database is in use, SQLIAs can be crafted to execute stored procedures provided by that specific database, including procedures that interact with the operating system.

It is a common misconception that using stored procedures to write Web applications renders them invulnerable to SQLIAs. Developers are often surprised to find that their stored procedures can be just as vulnerable to attacks as their normal applications [18, 24]. Additionally, because stored procedures are often written in special scripting languages, they can contain other types of vulnerabilities, such as buffer overflows, that allow attackers to run arbitrary code on the server or escalate their privileges [9].

```
CREATE PROCEDURE DBO.isAuthenticated
    @userName varchar2, @pass varchar2, @pin int
AS
    EXEC("SELECT accounts FROM users
    WHERE login='" +@userName+ "' and pass='" +@password+
          "' and pin=" +@pin);
GO
```

Figure 2: Stored procedure for checking credentials.

Example: This example demonstrates how a parameterized stored procedure can be exploited via an SQLIA. In the example, we assume that the query string constructed at lines 5, 6 and 7 of our example has been replaced by a call to the stored procedure defined in Figure 2. The stored procedure returns a true/false value to indicate whether the user's credentials authenticated correctly. To launch an SQLIA, the attacker simply injects " ' ; SHUTDOWN; - -" into either the `userName` or `password` fields. This injection causes the stored procedure to generate the following query:

```
SELECT accounts FROM users WHERE
login='doe' AND pass=' '; SHUTDOWN; -- AND pin=
```

At this point, this attack works like a piggy-back attack. The first query is executed normally, and then the second, malicious query is executed, which results in a database shut down. This example shows that stored procedures can be vulnerable to the same range of attacks as traditional application code.
References: [1, 4, 9, 10, 24, 28, 21, 18]

Inference

Attack Intent: Identifying injectable parameters, extracting data, determining database schema.
Description: In this attack, the query is modified to recast it in the form of an action that is executed based on the answer to a true/-false question about data values in the database. In this type of injection, attackers are generally trying to attack a site that has been secured enough so that, when an injection has succeeded, there is

no usable feedback via database error messages. Since database error messages are unavailable to provide the attacker with feedback, attackers must use a different method of obtaining a response from the database. In this situation, the attacker injects commands into the site and then observes how the function/response of the website changes. By carefully noting when the site behaves the same and when its behavior changes, the attacker can deduce not only whether certain parameters are vulnerable, but also additional information about the values in the database. There are two well-known attack techniques that are based on inference. They allow an attacker to extract data from a database and detect vulnerable parameters. Researchers have reported that with these techniques they have been able to achieve a data extraction rate of 1B/s [2].

Blind Injection: In this technique, the information must be inferred from the behavior of the page by asking the server true/-false questions. If the injected statement evaluates to true, the site continues to function normally. If the statement evaluates to false, although there is no descriptive error message, the page differs significantly from the normally-functioning page.

Timing Attacks: A timing attack allows an attacker to gain information from a database by observing timing delays in the response of the database. This attack is very similar to blind injection, but uses a different method of inference. To perform a timing attack, attackers structure their injected query in the form of an if/then statement, whose branch predicate corresponds to an unknown about the contents of the database. Along one of the branches, the attacker uses a SQL construct that takes a known amount of time to execute, (e.g. the `WAITFOR` keyword, which causes the database to delay its response by a specified time). By measuring the increase or decrease in response time of the database, the attacker can infer which branch was taken in his injection and therefore the answer to the injected question.

Example: Using the code from our running example, we illustrate two ways in which Inference based attacks can be used. The first of these is identifying injectable parameters using blind injection. Consider two possible injections into the *login* field. The first being "legalUser' and 1=0 - -" and the second, "legalUser' and 1=1 - -". These injections result in the following two queries:

```
SELECT accounts FROM users WHERE login='legalUser'
and 1=0 -- ' AND pass='' AND pin=0
```

```
SELECT accounts FROM users WHERE login='legalUser'
and 1=1 -- ' AND pass='' AND pin=0
```

Now, let us consider two scenarios. In the first scenario, we have a secure application, and the input for *login* is validated correctly. In this case, both injections would return login error messages, and the attacker would know that the *login* parameter is not vulnerable. In the second scenario, we have an insecure application and the *login* parameter is vulnerable to injection. The attacker submits the first injection and, because it always evaluates to false, the application returns a login error message. At this point however, the attacker does not know if this is because the application validated the input correctly and blocked the attack attempt or because the attack itself caused the login error. The attacker then submits the second query, which always evaluates to true. If in this case there is no login error message, then the attacker knows that the attack went through and that the *login* parameter is vulnerable to injection.

The second way inference based attacks can be used is to perform data extraction. Here we illustrate how to use a Timing based inference attack to extract a table name from the database. In this attack, the following is injected into the *login* parameter:

```
''legalUser' and ASCII(SUBSTRING((select top 1 name from
sysobjects),1,1)) > X WAITFOR 5 --''.
```

This produces the following query:

```
SELECT accounts FROM users WHERE login='legalUser' and
ASCII(SUBSTRING((select top 1 name from sysobjects),1,1))
> X WAITFOR 5 -- ' AND pass='' AND pin=0
```

In this attack the `SUBSTRING` function is used to extract the first character of the first table's name. Using a binary search strategy, the attacker can then ask a series of questions about this character. In this case, the attacker is asking if the ASCII value of the character is greater-than or less-than-or-equal-to the value of *X*. If the value is greater, the attacker knows this by observing an additional 5 second delay in the response of the database. The attacker can then use a binary search by varying the value of *X* to identify the value of the first character.

References: [34, 2]

Alternate Encodings

Attack Intent: Evading detection.

Description: In this attack, the injected text is modified so as to avoid detection by defensive coding practices and also many automated prevention techniques. This attack type is used in conjunction with other attacks. In other words, alternate encodings do not provide any unique way to attack an application; they are simply an enabling technique that allows attackers to evade detection and prevention techniques and exploit vulnerabilities that might not otherwise be exploitable. These evasion techniques are often necessary because a common defensive coding practice is to scan for certain known "bad characters," such as single quotes and comment operators.

To evade this defense, attackers have employed alternate methods of encoding their attack strings (e.g., using hexadecimal, ASCII, and Unicode character encoding). Common scanning and detection techniques do not try to evaluate all specially encoded strings, thus allowing these attacks to go undetected. Contributing to the problem is that different layers in an application have different ways of handling alternate encodings. The application may scan for certain types of escape characters that represent alternate encodings in its language domain. Another layer (e.g., the database) may use different escape characters or even completely different ways of encoding. For example, a database could use the expression `char(120)` to represent an alternately-encoded character "`x`", but `char(120)` has no special meaning in the application language's context. An effective code-based defense against alternate encodings is difficult to implement in practice because it requires developers to consider of all of the possible encodings that could affect a given query string as it passes through the different application layers. Therefore, attackers have been very successful in using alternate encodings to conceal their attack strings.

Example: Because every type of attack could be represented using an alternate encoding, here we simply provide an example (see [18]) of how esoteric an alternatively-encoded attack could appear. In this attack, the following text is injected into the *login* field: "legalUser'; exec(0x73687574646f776e) - - ". The resulting query generated by the application is:

```
SELECT accounts FROM users WHERE login='legalUser';
exec(char(0x73687574646f776e)) -- AND pass='' AND pin=
```

This example makes use of the `char()` function and of ASCII hexadecimal encoding. The `char()` function takes as a parameter an integer or hexadecimal encoding of a character and returns an instance of that character. The stream of numbers in the second part of the injection is the ASCII hexadecimal encoding of the string "SHUTDOWN." Therefore, when the query is interpreted by the

database, it would result in the execution, by the database, of the `SHUTDOWN` command.
References: [1, 18]

5. PREVENTION OF SQLIAS

Researchers have proposed a wide range of techniques to address the problem of SQL injection. These techniques range from development best practices to fully automated frameworks for detecting and preventing SQLIAs. In this section, we review these proposed techniques and summarize the advantages and disadvantages associated with each technique.

5.1 Defensive Coding Practices

The root cause of SQL injection vulnerabilities is insufficient input validation. Therefore, the straightforward solution for eliminating these vulnerabilities is to apply suitable defensive coding practices. Here, we summarize some of the best practices proposed in the literature for preventing SQL injection vulnerabilities.

Input type checking: SQLIAs can be performed by injecting commands into either a string or numeric parameter. Even a simple check of such inputs can prevent many attacks. For example, in the case of numeric inputs, the developer can simply reject any input that contains characters other than digits. Many developers omit this kind of check by accident because user input is almost always represented in the form of a string, regardless of its content or intended use.

Encoding of inputs: Injection into a string parameter is often accomplished through the use of meta-characters that trick the SQL parser into interpreting user input as SQL tokens. While it is possible to prohibit any usage of these meta-characters, doing so would restrict a non-malicious user's ability to specify legal inputs that contain such characters. A better solution is to use functions that encode a string in such a way that all meta-characters are specially encoded and interpreted by the database as normal characters.

Positive pattern matching: Developers should establish input validation routines that identify *good* input as opposed to *bad* input. This approach is generally called *positive validation*, as opposed to negative validation, which searches input for forbidden patterns or SQL tokens. Because developers might not be able to envision every type of attack that could be launched against their application, but should be able to specify all the forms of legal input, positive validation is a safer way to check inputs.

Identification of all input sources: Developers must check all input to their application. As we outlined in Section 2.1, there are many possible sources of input to an application. If used to construct a query, these input sources can be a way for an attacker to introduce an SQLIA. Simply put, all input sources must be checked.

Although defensive coding practices remain the best way to prevent SQL injection vulnerabilities, their application is problematic in practice. Defensive coding is prone to human error and is not as rigorously and completely applied as automated techniques. While most developers do make an effort to code safely, it is extremely difficult to apply defensive coding practices rigorously and correctly to all sources of input. In fact, many of the SQL injection vulnerabilities discovered in real applications are due to human errors: developers forgot to add checks or did not perform adequate input validation [20, 23, 33]. In other words, in these applications, developers were making an effort to detect and prevent SQLIAs, but failed to do so adequately and in every needed location. These examples provide further evidence of the problems associated with depending on developer's use of defensive coding.

Moreover, approaches based on defensive coding are weakened by the widespread promotion and acceptance of so-called "pseudo-remedies" [18]. We discuss two of the most commonly-proposed pseudo-remedies. The first of such remedies consists of checking user input for SQL keywords, such as "FROM," "WHERE," and "SELECT," and SQL operators, such as the single quote or comment operator. The rationale behind this suggestion is that the presence of such keywords and operators may indicate an attempted SQLIA. This approach clearly results in a high rate of false positives because, in many applications, SQL keywords can be part of a normal text entry, and SQL operators can be used to express formulas or even names (e.g., O'Brian). The second commonly suggested pseudo-remedy is to use stored procedures or prepared statements to prevent SQLIAs. Unfortunately, stored procedures and prepared statements can also be vulnerable to SQLIAs unless developers rigorously apply defensive coding guidelines. Interested readers may refer to [1, 25, 28, 29] for examples of how these pseudo-remedies can be subverted.

5.2 Detection and Prevention Techniques

Researchers have proposed a range of techniques to assist developers and compensate for the shortcomings in the application of defensive coding.

Black Box Testing. Huang and colleagues [19] propose WAVES, a black-box technique for testing Web applications for SQL injection vulnerabilities. The technique uses a Web crawler to identify all points in a Web application that can be used to inject SQLIAs. It then builds attacks that target such points based on a specified list of patterns and attack techniques. WAVES then monitors the application's response to the attacks and uses machine learning techniques to improve its attack methodology. This technique improves over most penetration-testing techniques by using machine learning approaches to guide its testing. However, like all black-box and penetration testing techniques, it cannot provide guarantees of completeness.

Static Code Checkers. JDBC-Checker is a technique for statically checking the type correctness of dynamically-generated SQL queries [12, 13]. This technique was not developed with the intent of detecting and preventing general SQLIAs, but can nevertheless be used to prevent attacks that take advantage of type mismatches in a dynamically-generated query string. JDBC-Checker is able to detect one of the root causes of SQLIA vulnerabilities in code—improper type checking of input. However, this technique would not catch more general forms of SQLIAs because most of these attacks consist of syntactically and type correct queries.

Wassermann and Su propose an approach that uses static analysis combined with automated reasoning to verify that the SQL queries generated in the application layer cannot contain a tautology [37]. The primary drawback of this technique is that its scope is limited to detecting and preventing tautologies and cannot detect other types of attacks.

Combined Static and Dynamic Analysis. AMNESIA is a model-based technique that combines static analysis and runtime monitoring [17, 16]. In its static phase, AMNESIA uses static analysis to build models of the different types of queries an application can legally generate at each point of access to the database. In its dynamic phase, AMNESIA intercepts all queries before they are sent to the database and checks each query against the statically-built models. Queries that violate the model are identified as SQLIAs and prevented from executing on the database. In their evaluation,

the authors have shown that this technique performs well against SQLIAs. The primary limitation of this technique is that its success is dependent on the accuracy of its static analysis for building query models. Certain types of code obfuscation or query development techniques could make this step less precise and result in both false positives and false negatives.

Similarly, two recent related approaches, SQLGuard [6] and SQLCheck [35] also check queries at runtime to see if they conform to a model of expected queries. In these approaches, the model is expressed as a grammar that only accepts legal queries. In SQLGuard, the model is deduced at runtime by examining the structure of the query before and after the addition of user-input. In SQLCheck, the model is specified independently by the developer. Both approaches use a secret key to delimit user input during parsing by the runtime checker, so security of the approach is dependent on attackers not being able to discover the key. Additionally, the use of these two approaches requires the developer to either rewrite code to use a special intermediate library or manually insert special markers into the code where user input is added to a dynamically generated query.

Taint Based Approaches. WebSSARI detects input-validation-related errors using information flow analysis [20]. In this approach, static analysis is used to check taint flows against preconditions for sensitive functions. The analysis detects the points in which preconditions have not been met and can suggest filters and sanitization functions that can be automatically added to the application to satisfy these preconditions. The WebSSARI system works by considering as sanitized input that has passed through a predefined set of filters. In their evaluation, the authors were able to detect security vulnerabilities in a range of existing applications. The primary drawbacks of this technique are that it assumes that adequate preconditions for sensitive functions can be accurately expressed using their typing system and that having input passing through certain types of filters is sufficient to consider it not tainted. For many types of functions and applications, this assumption is too strong.

Livshits and Lam [23] use static analysis techniques to detect vulnerabilities in software. The basic approach is to use information flow techniques to detect when tainted input has been used to construct an SQL query. These queries are then flagged as SQLIA vulnerabilities. The authors demonstrate the viability of their technique by using this approach to find security vulnerabilities in a benchmark suite. The primary limitation of this approach is that it can detect only known patterns of SQLIAs and, because it uses a conservative analysis and has limited support for untainting operations, can generate a relatively high amount of false positives.

Several dynamic taint analysis approaches have been proposed. Two similar approaches by Nguyen-Tuong and colleagues [31] and Pietraszek and Berghe [32] modify a PHP interpreter to track precise per-character taint information. The techniques use a context sensitive analysis to detect and reject queries if untrusted input has been used to create certain types of SQL tokens. A common drawback of these two approaches is that they require modifications to the runtime environment, which affects portability. A technique by Haldar and colleagues [15] and SecuriFly [26] implement a similar approach for Java. However, these techniques do not use the context sensitive analysis employed by the other two approaches and track taint information on a per-string basis (as opposed to per-character). SecuriFly also attempts to sanitize query strings that have been generated using tainted input. However, this sanitization approach does not help if injection is performed into numeric fields. In general, dynamic taint-based techniques have shown a lot of promise in their ability to detect and prevent SQLIAs. The primary drawback of these approaches is that identifying all sources of tainted user input in highly-modular Web applications and accurately propagating taint information is often a difficult task.

New Query Development Paradigms. Two recent approaches, SQL DOM [27] and Safe Query Objects [7], use encapsulation of database queries to provide a safe and reliable way to access databases. These techniques offer an effective way to avoid the SQLIA problem by changing the query-building process from an unregulated one that uses string concatenation to a systematic one that uses a type-checked API. Within their API, they are able to systematically apply coding best practices such as input filtering and rigorous type checking of user input. By changing the development paradigm in which SQL queries are created, these techniques eliminate the coding practices that make most SQLIAs possible. Although effective, these techniques have the drawback that they require developers to learn and use a new programming paradigm or query-development process. Furthermore, because they focus on using a new development process, they do not provide any type of protection or improved security for existing legacy systems.

Intrusion Detection Systems. Valeur and colleagues [36] propose the use of an Intrusion Detection System (IDS) to detect SQLIAs. Their IDS system is based on a machine learning technique that is trained using a set of typical application queries. The technique builds models of the typical queries and then monitors the application at runtime to identify queries that do not match the model. In their evaluation, Valeur and colleagues have shown that their system is able to detect attacks with a high rate of success. However, the fundamental limitation of learning based techniques is that they can provide no guarantees about their detection abilities because their success is dependent on the quality of the training set used. A poor training set would cause the learning technique to generate a large number of false positives and negatives.

Proxy Filters. Security Gateway [33] is a proxy filtering system that enforces input validation rules on the data flowing to a Web application. Using their Security Policy Descriptor Language (SPDL), developers provide constraints and specify transformations to be applied to application parameters as they flow from the Web page to the application server. Because SPDL is highly expressive, it allows developers considerable freedom in expressing their policies. However, this approach is human-based and, like defensive programming, requires developers to know not only which data needs to be filtered, but also what patterns and filters to apply to the data.

Instruction Set Randomization. SQLrand [5] is an approach based on instruction-set randomization. SQLrand provides a framework that allows developers to create queries using randomized instructions instead of normal SQL keywords. A proxy filter intercepts queries to the database and de-randomizes the keywords. SQL code injected by an attacker would not have been constructed using the randomized instruction set. Therefore, injected commands would result in a syntactically incorrect query. While this technique can be very effective, it has several practical drawbacks. First, since it uses a secret key to modify instructions, security of the approach is dependent on attackers not being able to discover the key. Second, the approach imposes a significant infrastructure overhead because it require the integration of a proxy for the database in the system.

Technique	*Taut.*	*Illegal/ Incorrect*	*Piggy-back*	*Union*	*Stored Proc.*	*Infer.*	*Alt. Encodings.*
AMNESIA [16]	•	•	•	•	×	•	•
CSSE [32]	•	•	•	•	×	•	×
IDS [36]							
Java Dynamic Tainting [15]	-	-	-	-	-	-	-
SQLCheck [35]	•	•	•	•	×	•	•
SQLGuard [6]	•	•	•	•	×	•	•
SQLrand [5]	•	×	•	•	×	•	×
Tautology-checker [37]	•	×	×	×	×	×	×
Web App. Hardening [31]	•	•	•	•	×	•	×

Table 1: Comparison of detection-focused techniques with respect to attack types.

Technique	*Taut.*	*Illegal/ Incorrect*	*Piggy-back*	*Union*	*Stored Proc.*	*Infer.*	*Alt. Encodings.*
JDBC-Checker [12]	-	-	-	-	-	-	-
Java Static Tainting* [23]	•	•	•	•	•	•	•
Safe Query Objects [7]	•	•	•	•	×	•	•
Security Gateway* [33]	-	-	-	-	-	-	-
SecuriFly [26]	-	-	-	-	-	-	-
SQL DOM [27]	•	•	•	•	×	•	•
WAVES [19]						-	
WebSSARI* [20]	•	•	•	•	•	•	•

Table 2: Comparison of prevention-focused techniques with respect to attack types.

6. TECHNIQUES EVALUATION

In this section, we evaluate the techniques presented in Section 5 using several different criteria. We first consider which attack types each technique is able to address. For the subset of techniques that are based on code improvement, we look at which defensive coding practices the technique helps enforce. We then identify which injection mechanism each technique is able to handle. Finally, we evaluate the deployment requirements of each technique.

6.1 Evaluation with Respect to Attack Types

We evaluated each proposed technique to assess whether it was capable of addressing the different attack types presented in Section 4. For most of the considered techniques, we did not have access to an implementation because either the technique was not implemented or its implementation was not available. Therefore, we evaluated the techniques analytically, as opposed to evaluating them against actual attacks. For *developer-based techniques*, that is, those that required developer intervention, we assumed that the developers were able to correctly apply all required defensive-coding practices. In other words, our assessment of these techniques is optimistic compared to what their performance may be in practice. In our tables, we denote developer-based techniques with the symbol "*".

For the purposes of the comparison, we divide the techniques into two groups: *prevention-focused* and *detection-focused* techniques. Prevention-focused techniques are techniques that statically identify vulnerabilities in the code, propose a different development paradigm for applications that generate SQL queries, or add checks to the application to enforce defensive coding best practices (see Section 5.1). Detection-focused techniques are techniques that detect attacks mostly at runtime.

Tables 1 and 2 summarize the results of our evaluation. We use four different types of markings to indicate how a technique performed with respect to a given attack type. We use the symbol "•" to denote that a technique can successfully stop all attacks of that type. Conversely, we use the symbol "×" to denote that a technique is not able to stop attacks of that type. We used two different symbols to classify techniques that are only partially effective. The symbol " " denotes a technique that can address the attack type considered, but cannot provide any guarantees of completeness. An example of one such technique would be a black-box testing technique such as WAVES [19] or the IDS based approach from Valeur and colleagues [36]. The symbol "-," denotes techniques that address the attack type considered only partially because of intrinsics limitations of the underlying approach. For example, JDBC-Checker [12, 13] detects type-related errors that enable SQL injection vulnerabilities. However, because type-related errors are only one of the many possible causes of SQL injection vulnerabilities, this approach is classified as only partially handling each attack type.

Half of the prevention-focused techniques effectively handle all of the attack types considered. Some techniques are only partially effective: JDBC-Checker by definition addresses only a subset of SQLIAs; Security Gateway, because it can not handle all of the injection sources (See Section 6.2) can not completely address all of the attack profiles; SecuriFly, because its prevention method is to escape all SQL meta-characters, which still would allow injection into numeric fields; and WAVES, which because it is a testing-based technique, can not provide guarantees as to its completeness. We believe that, overall, the prevention-focused techniques performed well because they incorporate the defensive coding practices in their prevention mechanisms. See Section 6.4 for further discussion on this topic.

Most of the detection-focused techniques perform fairly uniformly against the various attack types. The three exceptions are the IDS-based approach by Valeur and colleagues [36], whose effectiveness depends on the quality of the training set used, Java Dynamic Tainting [15], whose performance is negatively affected by the fact that its untainting operations allow input to be used without regard to the quality of the check, and Tautology-checker, which by definition can only address tautology-based attacks.

Two attack types, stored procedures and alternate encodings, caused problems for most techniques. With stored procedures, the code that generates the query is stored and executed on the database.

Technique	*Modify Code Base*	*Detection*	*Prevention*	*Additional Infrastructure*
AMNESIA [16]	No	Automated	Automated	None
CSSE [32]	No	Automated	Automated	Custom PHP Interpreter
IDS [36]	No	Automated	Generate Report	IDS System-Training Set
JDBC-Checker [12]	No	Automated	Code Suggestions	None
Java Dynamic Tainting [15]	No	Automated	Automated	None
Java Static Tainting [23]	No	Automated	Code Suggestions	None
Safe Query Objects [7]	Yes	N/A	Automated	Developer Training
SecuriFly [26]	No	Automated	Automated	None
Security Gateway [33]	No	Manual Specification	Automated	Proxy Filter
SQLCheck [35]	Yes	Semi-Automated	Automated	Key Management
SQLGuard [6]	Yes	Semi-Automated	Automated	None
SQL DOM [27]	Yes	N/A	Automated	Developer Training
SQLrand [5]	Yes	Automated	Automated	Proxy, Developer Training, Key Management
Tautology-checker [37]	No	Automated	Code Suggestions	None
WAVES [19]	No	Automated	Generate Report	None
Web App. Hardening [31]	No	Automated	Automated	Custom PHP Interpreter
WebSSARI [20]	No	Automated	Semi-Automated	None

Table 3: Comparison of techniques with respect to deployment requirements.

Technique	*Input type checking*	*Encoding of input*	*Identification of all input sources*	*Positive pattern matching*
JDBC-Checker [12]	Yes	No	No	No
Java Static Tainting [23]	No	No	Yes	No
Safe Query Objects [7]	Yes	Yes	N/A	No
SecuriFly [26]	No	Yes	Yes	No
Security Gateway [26]	Yes	Yes	No	Yes
SQL DOM [27]	Yes	Yes	N/A	No
WebSSARI [20]	Yes	Yes	Yes	Yes

Table 4: Evaluation of Code Improvement Techniques with Respect to Common Development Errors.

Most of the techniques considered focused only on queries generated within the application. Expanding the techniques to also encompass the queries generated and executed on the database is not straightforward and would, in general, require substantial effort. For this reason, attacks based on stored procedures are problematic for many techniques. Attacks based on alternate encoding are also difficult to handle. Only three techniques, AMNESIA, SQLCheck, and SQLGuard explicitly address these types of attacks. The reason why these techniques are successful against such attacks is that they use the database lexer or parser to interpret a query string in the same way that the database would. Other techniques that score well in this category are either developer-based techniques (i.e., Java Static Tainting and WebSSARI) or techniques that address the problem by using a standard API (i.e., SQL DOM and Safe Query Objects).

It is important to note that we did not take precision into account in our evaluation. Many of the techniques that we consider are based on some conservative analysis or assumptions that may result in false positives. However, because we do not have an accurate way to classify the accuracy of such techniques, short of implementing all of them and assessing their performance on a large set of legitimate inputs, we have not considered this characteristic in our assessment.

6.2 Evaluation with Respect to Injection Mechanisms

We assessed each of the techniques with respect to their handling of the various injection mechanisms that we defined in Section 2.1. Although most of the techniques do not specifically address all of those injection mechanisms, all but two of them could be easily extended to handle all such mechanisms. The two exceptions are Security Gateway and WAVES. Security Gateway can examine only URL parameters and cookie fields. Because it resides on the network between the application and the attacker, it cannot examine server variables and second-order injection sources, which do not pass through the gateway. WAVES can only address injection through user input because it only generates attacks that can be submitted to the application via the Web page forms.

6.3 Evaluation with Respect to Deployment Requirements

Each of the techniques have different deployment requirements. To determine the effort and infrastructure required to use the technique, we examined the author's description of the technique and its current implementation. We evaluated each technique with respect to the following criteria: (1) Does the technique require developers to modify their code base? (2) What is the degree of automation of the detection aspect of the approach? (3) What is the degree of automation of the prevention aspect of the approach? (4) What infrastructure (not including the tool itself) is needed to successfully use the technique? The results of this classification are summarized in Table 3.

6.4 Evaluation of Prevention-Focused Techniques with Respect to Defensive Coding Practices

Our initial evaluation of the techniques against the various attack types indicates that the prevention-focused techniques perform very well against most of these attacks. We hypothesize that this result is due to the fact that many of the prevention techniques are actually applying defensive coding best practices to the code base. Therefore, we examine each of the prevention-focused techniques

and classify them with respect to the defensive coding practice that they enforce. Not surprisingly, we find that these techniques enforce many of these practices. Table 4 summarizes, for each technique, which of the defensive coding practices it enforces.

7. CONCLUSION

In this paper, we have presented a survey and comparison of current techniques for detecting and preventing SQLIAs. To perform this evaluation, we first identified the various types of SQLIAs known to date. We then evaluated the considered techniques in terms of their ability to detect and/or prevent such attacks. We also studied the different mechanisms through which SQLIAs can be introduced into an application and identified which techniques were able to handle which mechanisms. Lastly, we summarized the deployment requirements of each technique and evaluated to what extent its detection and prevention mechanisms could be fully automated.

Our evaluation found several general trends in the results. Many of the techniques have problems handling attacks that take advantage of poorly-coded stored procedures and cannot handle attacks that disguise themselves using alternate encodings. We also found a general distinction in prevention abilities based on the difference between prevention-focused and general detection and prevention techniques. Section 6.4 suggests that this difference could be explained by the fact that prevention-focused techniques try to incorporate defensive coding best practices into their attack prevention mechanisms.

Future evaluation work should focus on evaluating the techniques' precision and effectiveness in practice. Empirical evaluations such as those presented in related work (e.g., [17, 36]) would allow for comparing the performance of the different techniques when they are subjected to real-world attacks and legitimate inputs.

Acknowledgements

This work was partially supported by DHS contract FA8750-05-2-0214 and NSF award CCR-0209322 to Georgia Tech. Adam Shostack provided valuable feedback and suggestions that helped improve the paper.

8. REFERENCES

[1] C. Anley. Advanced SQL Injection In SQL Server Applications. White paper, Next Generation Security Software Ltd., 2002.

[2] C. Anley. (more) Advanced SQL Injection. White paper, Next Generation Security Software Ltd., 2002.

[3] D. Aucsmith. Creating and Maintaining Software that Resists Malicious Attack. `http://www.gtisc.gatech.edu/bio_aucsmith.html`, September 2004. Distinguished Lecture Series.

[4] F. Bouma. Stored Procedures are Bad, O'kay? Technical report, Asp.Net Weblogs, November 2003. `http://weblogs.asp.net/fbouma/archive/2003/11/18/38178.aspx`.

[5] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, pages 292–302, June 2004.

[6] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using Parse Tree Validation to Prevent SQL Injection Attacks. In *International Workshop on Software Engineering and Middleware (SEM)*, 2005.

[7] W. R. Cook and S. Rai. Safe Query Objects: Statically Typed Objects as Remotely Executable Queries. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, 2005.

[8] M. Dornseif. Common Failures in Internet Applications, May 2005. `http://md.hudora.de/presentations/2005-common-failures/dornseif-common-failures-2005-05-25.pdf`.

[9] E. M. Fayo. Advanced SQL Injection in Oracle Databases. Technical report, Argeniss Information Security, Black Hat Briefings, Black Hat USA, 2005.

[10] P. Finnigan. SQL Injection and Oracle - Parts 1 & 2. Technical Report, Security Focus, November 2002. `http://securityfocus.com/infocus/1644` `http://securityfocus.com/infocus/1646`.

[11] T. O. Foundation. Top Ten Most Critical Web Application Vulnerabilities, 2005. `http://www.owasp.org/documentation/topten.html`.

[12] C. Gould, Z. Su, and P. Devanbu. JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 04) – Formal Demos*, pages 697–698, 2004.

[13] C. Gould, Z. Su, and P. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 04)*, pages 645–654, 2004.

[14] N. W. Group. RFC 2616 – Hypertext Transfer Protocol – HTTP/1.1. Request for comments, The Internet Society, 1999.

[15] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *Proceedings 21st Annual Computer Security Applications Conference*, Dec. 2005.

[16] W. G. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005)*, Long Beach, CA, USA, Nov 2005. To appear.

[17] W. G. Halfond and A. Orso. Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks. In *Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)*, pages 22–28, St. Louis, MO, USA, May 2005.

[18] M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, Redmond, Washington, second edition, 2003.

[19] Y. Huang, S. Huang, T. Lin, and C. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *Proceedings of the 11th International World Wide Web Conference (WWW 03)*, May 2003.

[20] 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 *Proceedings of the 12th International World Wide Web Conference (WWW 04)*, May 2004.

[21] S. Labs. SQL Injection. White paper, SPI Dynamics, Inc., 2002. `http://www.spidynamics.com/assets/documents/WhitepaperSQLInjection.pdf`.

[22] D. Litchfield. Web Application Disassembly with ODBC Error Messages. Technical document, @Stake, Inc., 2002. http://www.nextgenss.com/papers/webappdis.doc.

[23] V. B. Livshits and M. S. Lam. Finding Security Errors in Java Programs with Static Analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, Aug. 2005.

[24] C. A. Mackay. SQL Injection Attacks and Some Tips on How to Prevent Them. Technical report, The Code Project, January 2005. `http://www.codeproject.com/cs/database/SqlInjectionAttacks.asp`.

[25] O. Maor and A. Shulman. SQL Injection Signatures Evasion. White paper, Imperva, April 2004. `http://www.imperva.com/application_defense_center/white_papers/sql_injection_signatures_evasion.html`.

[26] M. Martin, B. Livshits, and M. S. Lam. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA 2005)*, pages 365–383, 2005.

[27] R. McClure and I. Krüger. SQL DOM: Compile Time Checking of Dynamic SQL Statements. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 05)*, pages 88–96, 2005.

[28] S. McDonald. SQL Injection: Modes of attack, defense, and why it matters. White paper, GovernmentSecurity.org, April 2002.

http://www.governmentsecurity.org/articles/SQLInjectionModesofAttackDefenceandWhyItMatters.php.

[29] S. McDonald. SQL Injection Walkthrough. White paper, SecuriTeam, May 2002. http://www.securiteam.com/securityreviews/5DP0N1P76E.html.

[30] T. M. D. Network. Request.servervariables collection. Technical report, Microsoft Corporation, 2005. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/iissdk/html/9768ecfe-8280-4407-b9c0-844f75508752.asp.

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

[32] T. Pietraszek and C. V. Berghe. Defending Against Injection Attacks through Context-Sensitive String Evaluation. In *Proceedings of Recent Advances in Intrusion Detection (RAID2005)*, 2005.

[33] D. Scott and R. Sharp. Abstracting Application-level Web Security. In *Proceedings of the 11th International Conference on the World Wide Web (WWW 2002)*, pages 396–407, 2002.

[34] K. Spett. Blind sql injection. White paper, SPI Dynamics, Inc., 2003. http://www.spidynamics.com/whitepapers/Blind_SQLInjection.pdf.

[35] 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)*, Jan. 2006.

[36] F. Valeur, D. Mutz, and G. Vigna. A Learning-Based Approach to the Detection of SQL Attacks. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Vienna, Austria, July 2005.

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

Acronyms

SQL	Structured Query Language
SQLIA	SQL Injection Attacks
URL	Uniform Resource Locator
HTML	HyerText Markup Language
LLC	Limited Liability Corporation
CTO	Corporate Technical Officer
VP	Vice President
CEO	Corporate Executive Officer
IP	Internet Protocol
GTRC	Georgia Tech Research Center
NSA	National Security Agency
IEEE	Institute of Educational and Electronic Engineers
FSE	Foundation of Software Engineers
ICSE	International Conference on Software Engineering
ISSSE	International Symposium on Secure Software Engineering
ASE	Automated Software Engineering
TSE	Transactions on Software Engineering
DETER	Defense Technology Experimental Research