Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks

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

ABSTRACT

Our dependence on web applications has steadily increased, and we continue to integrate them into our everyday routine activities. When we are making reservations, paying bills, and shopping on-line, we expect these web applications to be secure and reliable. However, as the availability of these services has increased, there has been a corresponding increase in the number and sophistication of attacks that target them. One of the most serious types of attack against web applications is SQL injection. SQL injection is a class of code-injection attacks in which user input is included in a SQL query in such a way that part of the input is treated as code. Using SQL injection, attackers can leak confidential information, such as credit card numbers, from web applications' databases and even corrupt the database. In this paper, we propose a novel technique to counter SQL-injection. The technique combines conservative static analysis and runtime monitoring to detect and stop illegal queries before they are executed on the database. In its static part, the technique builds a conservative model of the legitimate queries that could be generated by the application. In its dynamic part, the technique inspects the dynamically generated queries for compliance with the statically-built model. We also present a preliminary evaluation of the technique performed on two small web applications. The results of the evaluation are promising—our technique was able to prevent all of the attacks that we performed on the two applications.

1. INTRODUCTION

Database-driven web applications have become widely deployed on the Internet, and organizations use them to provide a broad range of services to their customers. These applications, and their underlying databases, often contain confidential, or even sensitive, information, such as customer and financial records. This information can be highly valuable and makes web application an ideal target for attacks. In fact, in recent years there has been an increase in attacks against these online databases. One type of attacks in particular, *SQL-Injection Attack*s (*SQLIA*s), is especially harmful. SQLIAs can give attackers direct access to the underlying databases of a web application and, with that, the power to leak, modify, or even delete information that is stored on them. Recently, there have been many SQLIAs with serious consequences, and the list of victims of such attacks includes high-profile companies and associations, such as Travelocity, FTD.com, Creditcards.com, Tower Records, and RIAA. Even more alarming is a study performed by the Gartner Group on over 300 web sites, which reported that 97% of the sites audited were found vulnerable to this kind of web attacks.

The root cause of SQLIAs is insufficient input validation. SQLIAs occur when data provided by a user is not properly validated and is included directly in a SQL query. By leveraging inadequate input validation, an attacker can submit input with embedded SQL commands directly to the database. This kind of vulnerability represents a serious threat to any web application that reads inputs from the user (e.g., through web forms or through web APIs) and makes SQL queries to an underlying database using these inputs. Most web applications used by corporations and government agencies work this way and could therefore be vulnerable to SQL-injection attacks.

Even though the vulnerabilities that lead to SQLIAs are well understood, SQL injection continues to be a problem due to a lack of effective techniques for detecting and preventing it. Improved coding practices could theoretically prevent SQLIAs. However, in practice, techniques such as defensive programming have been less than effective in addressing the problem (e.g., [16, 17, 20]). Furthermore, attackers continue to find new exploits to circumvent the input checks used by programmers. Finally, improved coding practices do little to help protect large legacy systems, as the human effort required to identify and recode all of the vulnerable sections of such systems often makes this approach impractical in realistic settings. SQLIAs can also generally elude traditional tools such as firewalls and Intrusion Detection Systems (IDSs). SQLIAs are performed through ports used for regular web traffic, which are usually open in firewalls, and work at the application layer, whereas most IDSs tend to focus on the network and IP layers. Finally, there are very few analysis-based techniques for vulnerability detection that target SQLIAs directly, and they provide only partial solutions to the problem. Dynamic techniques, such as penetration testing, suffer from issues related to completeness that often result in false negatives being produced.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Workshop on Dynamic Analysis (WODA 2005) 17 May 2005, St. Louis, MO, USA
Copyright 2005 ACM ISBN # 1-59593-126-0 ...$5.00.

Conversely, existing techniques based on static analysis are either too imprecise or focus only on a specific aspect of the problem.

In this paper, we introduce a novel technique for the detection and prevention of SQLIAs. Our technique consists of a model-based approach specifically designed to target SQLIAs and combines conservative static analysis and runtime monitoring. The key intuition behind our technique is twofold: (1) the information needed to predict the possible structure of the queries generated by a web application is contained within the application's code; (2) an SQLIA, by injecting additional SQL statements into a query, would violate that structure. Our technique consists of two phases. In the static phase, the technique leverages an existing, conservative string analysis [5, 9] to analyze the web-application code and automatically build a conservative model of the legitimate queries that could be generated by the application. In the dynamic phase the technique monitors all dynamically-generated queries at runtime 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 reports information on the characteristics of the attack.

In the paper, we also present a preliminary evaluation of the proposed technique. We implemented the technique in a prototype tool, and used the tool to evaluate the technique on two small web applications. The results of the evaluation are promising. For all the cases considered, our tool was able to prevent and report the attacks.

The main contributions of the paper are:

- The definition of a novel technique against SQL-injection that combines static analysis and runtime monitoring.
- A feasibility study that shows the effectiveness of the technique on two small applications.

2. RELATED WORK

Many existing techniques, such as filtering, information-flow analysis, penetration testing, and defensive coding, can detect and prevent a subset of the vulnerabilities that lead to SQLIAs. In this section, we list the most relevant techniques and discuss their limitations with relation to SQLIAs.

Defensive Programming. Ultimately, web application vulnerabilities exist because developers do not adequately check user input. Even though the cause of these vulnerabilities is well understood, defensive coding has generally not been successful in preventing them (e.g., [16, 17, 20]). Attackers continue to find new attack strings or subtle variations on old attacks that are able to avoid the checks programmers put in place. While improved coding practices (e.g., [11]) can help mitigate the problem, they are limited by the developer's ability to accurately generate appropriate input validation code and recognize all situations in which they are needed.

General Techniques for Input Validation. Some research work treats SQLIA as part of a more general problem of information flow and input validation. Unfortunately, the generalization of SQLIAs as a type of input validation problem misses key parts of the problem. Engler and Ashcraft propose MC [1], an automated checker that identifies input validation problems. MC is not directly applicable in this context because (1) it primarily focuses on bounds checking for integers rather than string values, (2) it assumes that any type of input sanitizing function is sufficient to ensure that an input value is not tainted, whereas often input sanitizing is performed, but inadequately. Larson and Austin [15] extend MC to address some of its shortcomings. However, their improvements are mostly targeted towards bounds checking, whereas SQLIAs occur because of the content, rather than the size of the input strings. Scott [19] uses a proxy to filter input and output data streams for a web-enabled application based on policy rules defined at the enterprise level. Although this technique may be effective against some basic types of attacks, it is too coarse grained for most systems—it is difficult to specify general, application-independent validation rules.

Specific Techniques Against SQLIAs. Other researchers have developed techniques specifically targeted at SQLIAs. Huang and colleagues [12] explored a black-box technique for testing web applications for SQLIA vulnerabilities. This techniques improves over general penetration-testing techniques, but like all testing-based techniques, it cannot provide guarantees of completeness. Huang and colleagues also define a white-box approach for input validation [13] that relies on user-provided annotations. Besides the fact that relying on user-provided annotations limits considerably the practical applicability of the approach, the technique makes assumptions that might not be practical. For example, it assumes that preconditions for all sensitive functions can be accurately expressed ahead of time using regular expressions. Boyd and colleagues propose an approach based on randomization of SQL instructions using a key [3], which extends a previous approach to counter general code-injections attacks [14]. In this approach, SQL code injected by an attacker would result in a syntactically incorrect query. Although effective, this technique could be circumvented if the key used for the randomization were exposed. Moreover, the approach imposes a significant overhead in terms of infrastructure since a special proxy has to be integrated into the application and it would be very difficult to integrate into existing infrastructure. Wasserman and Su propose an approach that uses static analysis to verify that the SQL queries generated in the application layer cannot contain a tautology [22]. Although useful, this approach does not exclude the possibility of a user inserting a tautology (see example in Section 3.1) and many other kinds of SQLIAs.

Security Techniques Based on Program Models. Our research is also related to the larger field of model-based anomaly detection or behavior-based detection, where execution models are used to monitor program behavior [6]. There has been a large amount of research in model-based anomaly detection (e.g., [7, 21]). Current application-security techniques are very effective in detecting traditional attacks such as buffer overflows. However, to be successful against web-application attacks, we must shift the focus from *system* security policies (e.g., execution of injected/unauthorized code) to *application* security policies (e.g., access to parts of the database not allowed for users of the application).

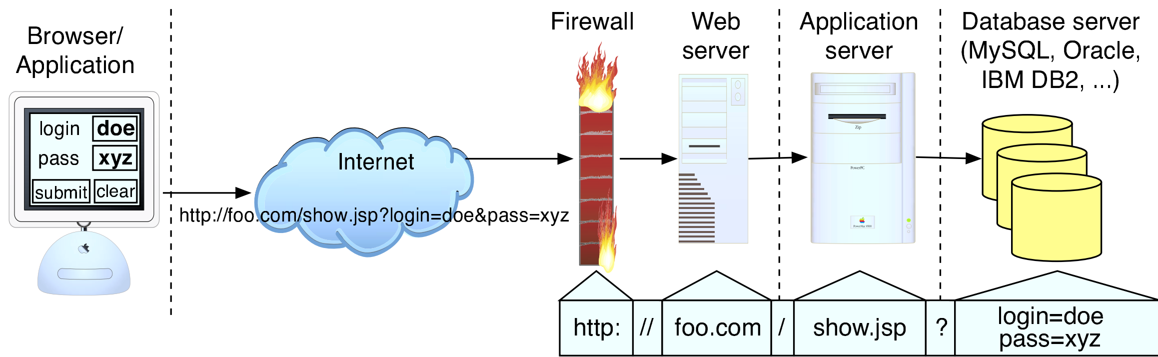


Figure 1: Example of interaction between a user and a typical web application.

Static Analysis Techniques. Although not directly related to SQLIAs, two static analysis techniques are related to our research because we leverage them in our static-analysis phase. The first technique is a string analysis technique for Java developed by Christensen, Müller, and Schwartzbach [5]. Their technique performs a conservative string analysis of an application and creates automata that express all the possible values a specific string can have at a given point in the application. The second technique is a technique for statically checking the type correctness of dynamically-generated SQL queries, by Gould, Su, and Devanbu [8, 9]. Their technique does not address SQLIAs and could only be used to prevent attacks that take advantage of type mismatches to crash the database underlying the web application. Their technique also relies on the string analysis by Christensen and colleagues [5], and we use a slightly modified version of their approach to build our SQL-query models.

3. TECHNIQUE

To better motivate our technique, we introduce an example of SQLIA that we will use throughout the rest of the technical description.

3.1 Example of SQL-Injection Attack

Figure 1 shows a simple example of how a user on a client machine can access services provided by an application server and an underlying database. When the user enters a login and a password in the web form and presses the `submit` button, a URL is generated (`http://foo.com/show.jsp?login=doe&pass=xyz`) and sent to the Web server. The figure illustrates which components of the web application handle the different parts of the URL.

In the example, the user input is interpreted by a servlet (`show.jsp`). Servlets are Java applications that operate in conjunction with a Web server. In this scenario they would typically (1) use the user input to build a dynamic SQL query, (2) submit the query to the database, and (3) use the response from the database to generate HTML-pages that are then sent back to the client. Figure 2 shows an excerpt of a possible implementation of servlet `show.jsp`. Method `getUserInfo` is called with the login and the password provided by the user in string format. If both `login` and `password` are empty, the method submits the following query to the database:

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

If `login` and `password` are defined by the user, the method embeds the submitted credentials in the query. So, if a user

```
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.executeQuery(queryString);
11.   return tempSet;
    }
    ...
}
```

Figure 2: Example servlet.

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 site that uses this servlet would be vulnerable to a SQLIA. 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 returns the `info` records for all users in the database. An attacker could insert a wide range of SQL commands via this exploit, including commands to modify or destroy database tables.

This particular exploit is a simple example of SQLIA. Current attacks have progressed to the point where they are automated, supported by various tools, and can evade many types of input validation. In general, any application that accepts input from a user, embeds it into a SQL query, and then submits it to a database, could be vulnerable.

3.2 Proposed Approach

Our approach addresses SQLIAs by combining static analysis and runtime monitoring. The key insights behind the approach are that (1) the source code contains enough in-

formation to infer models of the expected, legitimate SQL queries generated by the application, and (2) an SQLIA, by injecting additional SQL statements into a query, would violate such a model. In its static part, our technique uses program analysis to automatically build a conservative model of the legitimate queries that could be generated by the application. (This analysis leverages previous work on static string analysis [5, 9].) In its dynamic part, our technique monitors the dynamically generated queries at runtime and checks them for compliance with the statically-generated model. Queries that violate the model represent potential SQLIAs and are prevented from executing on the database and reported. The technique 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 the possible SQL queries that may be generated at that hotspot. A *SQL-query model* is a non-deterministic finite-state automaton in which the transition labels are either SQL tokens (SQL keywords and operators), delimiters, or string tokens.

Instrument Application: Add to the application, at each hotspot, calls to the runtime monitor.

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

3.2.1 Identify Hotspots

This step performs a simple scanning of the application code to identify hotspots. For the example servlet in Figure 2, the set of hotspots would contain a single element: the statement at line 10. (In Java-based applications, interactions with the database occur through calls to specific methods in the JDBC library,[1] such as `java.sql.Statement-.executeQuery(String)`.)

3.2.2 Build SQL-Query Models

In this step we build the SQL-query model for each hotspot identified in the previous step. Within each hotspot, we are interested in computing the possible values of the query string passed to the database. To do this, we leverage the Java String Analysis (*JSA*) developed by Christensen, Müller, and Schwartzbach [5]. Their technique constructs a flow graph that abstracts away the control flow of the program and represents string-manipulation operations performed on string variables. For each string of interest, the technique analyzes the flow graph and simulates the string-manipulation operations that are performed on the string. The result of the analysis is a Non-Deterministic Finite Automaton (NDFA) that expresses, at the character level, all the possible values the considered string can assume at the hotspot. The string analysis is conservative, so the NDFA for a string is an overestimate of all the possible values of the string.[2]

[1] `http://java.sun.com/products/jdbc/`

[2] Christensen and colleagues' technique actually generates Deterministic Finite Automata (DFAs) by transforming each NDFA into a corresponding DFA. However, because the transformation to DFAs increases considerably the size of the automata and introduces patterns that complicate the construction of our SQL-query model, we use their technique but skip its determinization step.

To build our SQL-query models, we use an approach that is analogous to the one proposed by Gould, Su, and Devanbu [9], except that we perform it on NDFAs instead of DFAs. We perform a depth first traversal of each hotspot's NDFA and group characters into SQL tokens and string tokens. *SQL tokens* consist of either SQL keywords (e.g., "`WHERE`") or operators (e.g., "`=`" and "`<=`"). For example, a sequence of transitions labeled 'S', 'E', 'L', 'E', 'C', and 'T' would be recognized as the SQL `SELECT` keyword and suitably grouped into a single transition labeled "SELECT". Any token that is not a SQL keyword, SQL operator, or delimiter is recognized as a *string token*. A string token can either be a *constant string*, hard-coded in the application (e.g., value 'guest' for the example servlet in Figure 2), or a *variable string*, a string that corresponds to a variable related to some user input (e.g., variable `password` for the example servlet). In the latter case, we use the generic label "`VAR`" to indicate the string. Note that the parsing of the NDFAs that we just described is configurable, in that it lets us consider different SQL dialects. At the end of the parsing we obtain, for each hotspot, the SQL-query model that is then used by the dynamic part of our technique. Each SQL-query model is an NDFA in which the transitions represent SQL or string tokens.

To illustrate, Figure 3 shows the SQL-query model for the hotspot in our example (Figure 2, line 10). The query model reflects the two types of query strings that can be generated by the code depending on the branch followed after the `if` statement at line 6.

3.2.3 Instrument Application

In this step, we instrument the application by adding calls to the monitor in charge of checking the queries at runtime. For each hotspot, the technique inserts a call to the monitor before the call to the database. The monitor is invoked with two parameters: the string that contains the actual query about to be submitted and a unique identifier for the hotspot. Using the unique identifier, the runtime monitor is able to correlate the hotspot with the specific SQL-query model that was statically generated for that point and check the query against the appropriate model.

Figure 4 shows how the example application would be instrumented by our technique. The hotspot, originally at line 10 in Figure 2, 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 4: Example hotspot after instrumentation.

3.2.4 Runtime Monitoring

During execution, when the application reaches a hotspot, the runtime monitor is invoked and the string that is about to be submitted as a query is passed as a parameter. The monitor parses the query string into a sequence of SQL tokens, delimiters, and string tokens (similar to what the technique does when generating SQL-query models). Figure 5 shows how the last two queries discussed in Section 3.1 would be parsed during runtime monitoring.

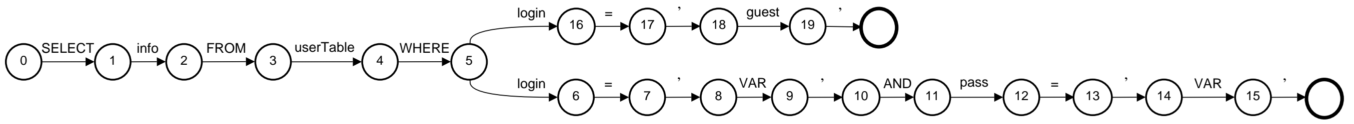
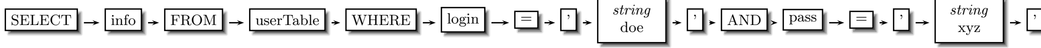


Figure 3: SQL-level model for the servlet in Figure 2.

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



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

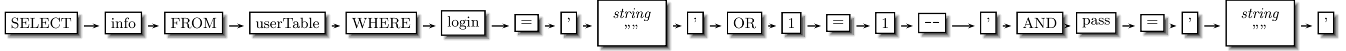


Figure 5: Example of parsed runtime queries.

After the query has been parsed, the runtime monitor checks whether the query violates the SQL-query model associated with the hotspot from which the monitor has been called. As discussed above, an SQL-query model is an NDFA whose alphabet consists of all the SQL keywords and operators, a set of constant strings, delimiters, and the placeholder string "VAR". Therefore, to check whether a query is compliant with the model, the runtime monitor simply checks whether the model (i.e., the automaton) accepts the query (i.e., whether at least a series of valid transitions imposed by the query reaches an accepting state). Note that, in the checking, the monitor considers "VAR" as a special symbol that can match any single non-SQL token[3] in the query . If the model accepts the query, then the monitor lets the execution of the query continue. Otherwise, the monitor identifies the query as an SQLIA, prevents the query from executing on the database, and reports the attack.

To illustrate, consider again the queries from Section 3.1 shown in Figure 5 and recall that the first query is legitimate, whereas the second one corresponds to a SQLIA. When checking query (a), the analysis would start matching from token SELECT and from the initial state (State 0) of the SQL-query model in Figure 3. Because the token matches the label of the only transition from the initial state, the automaton reaches State 1. The automaton continues to reach new states until it reaches State 5. At this point, the monitor would either continue with the higher branch and backtrack, or continue with the lower branch. On the lower branch, all the remaining tokens in the query suitably match transitions' labels, and the automaton reaches an accept state after consuming the last token in the query ("'"). Note that, during the matching, the monitor successfully matches token "doe" with label "VAR" on the transition from State 8 because "doe" is a non-SQL symbol. The same holds for token "xyz" and label "VAR" on the transition from State 14.

The checking of query (b) would proceed in an analogous way until token OR is reached, and the automaton is in State 10. Because the node does not match the label of the only outgoing transition from State 10 (AND), the query is not accepted by the automaton, and the monitor identifies the query as a SQLIA.

Once a SQLIA has been detected, our technique stops the query before it is executed on the database and reports relevant information about the attack in a way that can be leveraged by developers. For example, in our implementation of the technique for Java, we throw an exception when the attack is detected and encode information about the attack in the exception. If developers want to access the information at runtime, they can simply leverage the exception-handling mechanism of the language and integrate their handling code into the application. Having this attack information available at runtime is useful because it allows developers to react to an attack right after it is detected and develop an appropriate customized response. For example, developers may decide to avoid any risk and shut-down the part of the application involved in the attack. Alternatively, a developer could handle the attack by converting the information into a format that is usable by another tool, such as an IDS, and report it to that tool. Because this mechanism integrates with the application's language, it allows developers flexibility in choosing a response to the SQLIAs that matches the specific needs of their organization.

Currently, the information reported by our technique includes the time of the attack, the location of the hotspot that was exploited, and the attempted-attack query. We are currently considering additional information that could be useful for the developer (e.g., information correlating program execution paths with specific parts of the query model) and investigating ways in which we can modify the static analysis to collect this information.

3.3 Considerations on the Technique

Because the static analysis that we use to build the SQL-query models is conservative [5], the models are guaranteed to represent an overestimate of the set of legitimate query strings that the application can generate. Therefore, at runtime, all legitimate queries necessarily match the corresponding model, and our technique does not generate false positives.

As far as false negatives are concerned, our technique could produce false negatives only in cases in which (1) the imprecision of the static analysis results in SQL models that include more spurious queries than the one that can actually be generated by the application, and (2) the malicious query happens to match one of these spurious queries. We expect this case to be rare, mostly because queries resulting from SQL injection are typically fairly convoluted.

In terms of efficiency, our approach requires the execution of the monitor for each database query. The monitor just matches a typically short set of tokens against an NDFA whose size is at worst quadratic in the size of the pro-

[3]Because the parser within the monitor understands the SQL syntax, it is able to correctly distinguish and identify SQL keywords, SQL operators, and strings.

gram [5] (and in practice almost always linear). Conversely, database queries normally involve interactions over a network. Therefore, we expect the overhead for the monitoring to be dominated by the cost of the database transactions and negligible.

4. EVALUATION

To evaluate our approach, we developed a prototype tool that implements our technique and used this tool to detect SQLIAs on two small web applications. The following sections present the tool, illustrate the setup for our evaluation, and discuss our results.

4.1 The Tool

AMNESIA (Analysis and Monitoring for NEutralizing SQL Injection Attacks) is the prototype tool that implements our technique to counter SQLIAs for Java web applications. AMNESIA is developed in Java and consists of three modules that leverage various existing technologies and libraries in their implementation:

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 set of SQL-query models associated with the hotspots. For the implementation of this module, we leveraged the BRICS Java String Analyzer [5] and its extensions by Gould, Su, and Devanbu [9].

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 InsECT, a generic instrumentation and monitoring framework for Java developed at Georgia Tech [4].

Runtime-monitoring module. This module implements Step 4 of our technique. The module inputs 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. For this module, we also leveraged InsECT.

4.2 Experiment Setup

Subjects. For the evaluation we used two experimental subjects, ExampleSQL and Checkers. ExampleSQL is an example program that we wrote for testing purposes and consists of about 100 lines of code. One goal when developing ExampleSQL was to create a subject that contained various control- and data-flow patterns. Such patterns were designed to exercise the technique in meaningful ways and to result in non-trivial SQL-query models. Another goal was for ExampleSQL to be representative, on a smaller scale, of existing web applications. As a result, ExampleSQL is characterized by non-trivial control- and data-flow, the presence of paths that contain multiple string-manipulation operations, and realistic vulnerabilities (we used parameter-handling code similar to that found in real applications).

Our second test subject, Checkers, is a Java servlet that consists of about 5,000 lines of code. Checkers was previously used in related work [8] and was developed as part of a class project. Checkers is a typical example of a web application; it accepts input from the user through a web form and uses the input to build queries to an underlying database.

Evaluation Protocol. To evaluate the effectiveness of our technique, we first manually generated a set of inputs to the two subjects that included both legitimate and malicious inputs. Then, we submitted the inputs to the two subjects, assessed which inputs were actually leading to a SQLIA, and tagged the inputs accordingly. Finally, we used AMNESIA on the subjects, resubmitted all the inputs tagged as "legitimate" to the subjects, and measured the number of queries reported as SQLIAs by AMNESIA. Similarly, we reran all of the "malicious" inputs and measured which queries were blocked and reported.

To define the set of inputs, we first identified one relevant entry point for each of the two subjects. For Checkers, we targeted the login verification function, which inputs a user-provided login and password and inserts them into queries sent to the underlying database. These functions performed only minimal input validation and were therefore likely to be vulnerable. In the case of ExampleSQL, the application has only one entry point, purposefully designed to be vulnerable to SQLIAs. We then defined a set of legitimate and malicious inputs by surveying various web sites (e.g., US-CERT, `http://www.us-cert.gov/`) and security-related mailing lists. We used SQLIAs similar to the ones discussed on these sites as representative examples of the types of attacks that could be performed on ExampleSQL and Checkers. In total, we used 17 legitimate accesses and 10 attacks.

4.3 Results

In the study, AMNESIA achieved a perfect score: For both subjects, it was able to correctly identify all attacks as SQLIAs, while allowing all legitimate queries to be performed. In other words, for the cases considered, AMNESIA generated no false positives and no false negatives. On the one hand, the lack of false positives is guaranteed by the fact that the SQL models are built using a conservative analysis, and this results only provides some initial evidence of the correct implementation of the tool. The lack of false negatives, on the other hand, is not a guarantee (see Section 3.3). Our results, although obtained on two simple examples and for a fairly small number of data points, are very encouraging because we considered realistic attacks. However, more extensive experimentation is needed before drawing more definitive conclusions.

5. CONCLUSION AND FUTURE WORK

In this paper, we presented a novel technique against SQLIAs. Our approach is based on the intuition that the web-application code implicitly contains a policy that allows for distinguishing legitimate and malicious queries. The technique is fully automated and detects, prevents, and reports SQLIAs using a model-based approach that combines sound static analysis and runtime monitoring. In its static phase, our technique uses an existing string analysis technique [5, 9] to extract from the web-application's code a conservative model of all the query strings that could be generated by the application. In the dynamic phase, the technique monitors queries generated at runtime for con-

formance with the statically-built model. Queries that do not match the model are identified as SQLIAs, blocked, and reported, together with relevant information.

In this paper, we also presented a preliminary evaluation of the technique in which we used a prototype tool that implements our technique on two small subjects. The results of our evaluation are encouraging: our technique was able to correctly identify the attacks that we performed on the applications without blocking legitimate accesses to the database (i.e., the technique produced neither false positives nor false negatives). These results show that our technique represents a promising approach to countering SQLIAs and motivate further work in this direction.

Our first goal in future work is to further evaluate our technique using more subjects and attacks. We have already started the development of a testbed that includes several subjects and a large pool of attacks. Using this testbed, we will measure the effectiveness and efficiency of our technique and use the results to fine tune and refine the approach.

Why the approach may fail. Although our preliminary results are encouraging, and the basic idea behind the technique is sound, we can envision situations in which the technique would not be successful for scalability reasons. We have not tried to use our technique on any medium or large program, and we may find that the approach does not scale. We also have not used the approach on many subjects, and there may be problems with the analysis in the presence of certain coding patterns or program constructs (e.g., massive use of reflection). Another possible issue for the approach is the presence of black-box components in the web application, which the technique may not be able to analyze.

These potential issues will drive additional future work. We will evaluate the technique on subjects of different types and increasing sizes to assess the limits of the approach in terms of practical applicability. If the approach fails in some cases, for example because the static analysis is too expensive or the SQL-query models built are too imprecise, we will identify the causes of the failure and investigate improvements to the technique for handling them. This investigation may also lead to the definition of alternative techniques, static, dynamic, or combined, for building SQL models.

Acknowledgments

This work was supported in part by National Science Foundation awards CCR-0306372, CCR-0205422, and CCR-0209322 to Georgia Tech. Wenke Lee and David Dagon provided useful comments on an early version of the paper. Carl Gould, Zhendong Su, and Premkumar Devanbu provided us with an implementation of their JDBC Checker tool.

6. REFERENCES

[1] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the IEEE Symposium On Security and Privacy*, pages 143–159, May 2002.

[2] D. Aucsmith. Creating and maintaining software that resists malicious attack. http://www.gtisc.gatech.edu/aucsmith_bio.htm, September 2004. Distinguished lecture.

[3] 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.

[4] A. Chawla and A. Orso. A generic instrumentation framework for collecting dynamic information. In *Online Proceeding of the ISSTA Workshop on Empirical Research in Software Testing (WERST 2004)*, Boston, MA, USA, july 2004. http://www.sce.carleton.ca/squall/WERST2004.

[5] A. S. Christensen, A. Möller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th International Static Analysis Symposium, SAS 03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003.

[6] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31:805–822, 1999.

[7] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A.Longstaff. A sense of self for unix processes. In *IEEE Symposium on Security and Privacy*, pages 120–129, Oakland, California, May 1996.

[8] 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.

[9] 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.

[10] A. R. Group. Java Architecture for Bytecode Analysis (JABA), 2004. http://www.cc.gatech.edu/aristotle/Tools/jaba.html.

[11] M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, Redmond, Washington, 2nd ed., 2003.

[12] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. 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.

[13] Y.-W. 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.

[14] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS 03)*, pages 272–280, October 2003.

[15] E. Larson and T. Austin. High coverage detection of input-related security faults. In *Proceedings of the 12th USENIX Security Symposium*, pages 121–136, 2003.

[16] O. Maor and A. Shulman. Sql injection signatures evasion. http://www.imperva.com/application_defense_center/white_papers/sql_inje%ction_signatures_evasion.html, April 2004. White paper.

[17] S. McDonald. Sql injection: Modes of attack, defense, and why it matters. http://www.governmentsecurity.org/articles/SQLInjectionModesofAttackDef%enceandWhyItMatters.php, April 2004. White paper.

[18] OWASPD – Open Web Application Security Project. Top ten most critical web application vulnerabilities. http://www.owasp.org/documentation/topten.html, 2005.

[19] 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.

[20] SecuriTeam. Sql injection walkthrough. http://www.securiteam.com/securityreviews/5DP0N1P76E.html, May 2002. White paper.

[21] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2001.

[22] 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, October 2004.