Preventing Injection Attacks Through Automated Randomization of Keywords

by

Daniel M. Willenson

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology

May 21, 2012

Signature of Author: _____
Department of Electrical Engineering and Computer Science May 21, 2012

Certified by: _____
Jeff Perkins, Thesis Supervisor

Certified by: _____
Prof. Martin Rinard, Thesis Supervisor

Accepted by: _____
Prof. Dennis M. Freeman, Chairman, Masters of Engineering Thesis Committee

Preventing Injection Attacks Through Automated Randomization of Keywords

by

Daniel M. Willenson

Submitted to the
Department of Electrical Engineering and Computer Science
May 21, 2012

In Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

SQL injection attacks are a major security issue for database-backed web applications, yet the most common approaches to prevention require a great deal of programmer effort and attention. Even one unchecked vulnerability can lead to the compromise of an entire application and its data. We present a fully automated system for securing applications against SQL injection which can be applied at runtime. Our system mutates SQL keywords in the program's string constants as they are loaded, and instruments the program's database accesses so that we can verify that all keywords in the final query string have been properly mutated, before passing it to the database. We instrument other method calls within the program to ensure correct program operation, despite the fact that its string constants have been mutated. Additionally, we instrument places where the program generates user-visible output to ensure that randomized keyword mutations are never revealed to an attacker.

Chapter 1

Introduction

SQL injection attacks are a major security issue for database-backed web applications, yet the most common approaches to prevention require a great deal of programmer effort and attention. Even one unchecked vulnerability can lead to the compromise of an entire application and its data. We present a fully automated system for securing applications against SQL injection which can be applied at runtime. Our system mutates SQL keywords in the program's string constants as they are loaded, and instruments the program's database accesses so that we can verify that all keywords in the final query string have been properly mutated, before passing it to the database. We instrument other method calls within the program to ensure correct program operation, despite the fact that its string constants have been mutated. Additionally, we instrument places where the program generates user-visible output to ensure that randomized keyword mutations are never revealed to an attacker.

This paper is organized as follows. In Chapter 1, we present background context, describing what an injection attack is and how it works. In Chapter 2, we survey other methods for injection attack detection and prevention. In Chapter 3, we discuss the goals our system tries to achieve and strat-

egy we use to achieve them. In Chapter 4, we outline the implementation of the system. In Chapter 5, we discuss the ways we evaluated the system for security, correctness, and performance. Finally, in Chapter 6, we review limitations of the current implementation, explore avenues for further work, and present a short summary.

1.1 Injection Attacks

Injection attacks are a form of attack in which malicious users of a program craft input strings in a way that subverts normal program operation. Vulnerabilities may occur any time a program accepts user input and later uses that input as part of the source code for an executable program. For example, a program might prompt the user to enter his name, and subsequently use the string that the user provided as the name of a file in a shell program. If the user knows how the shell program is structured, he may be able to provide, instead of his name, a string that changes the structure of the shell program. By doing so, he can cause the program to execute arbitrary commands on his behalf, using its privileges.

1.1.1 Web Applications and SQL

The web, because of its public nature, its popularity, and its programming model, provides particularly fertile ground for injection attacks. Many web sites are publicly accessible, and many others that require registration do not ask for much verification of their users' identities. This makes it easy for potential attackers to gain access to the application. There are a large number of web sites with valuable data, which means there are many attractive targets for attackers. Finally, the fact that many web sites use database backends to store their data creates the potential for injection attacks.

Typically, the communication channel from a web program, or application, to its database is text-based: the application presents the source code for a program (known as a query) to the database. The database then compiles and executes that program, returning the results to the application. The application provides the front-end interaction with a user's web browser, receiving input from the user. For example, the application might present a form to the user, in which she can enter free text (e.g. "first name", "comments"), select choices from a list ("quantity", "state/province"), or use some more complicated mechanism ("date"). All of these values are translated into string data (text) for the browser-to-application communication path, even if they are numbers (`quantity = "35"`) or structured data (`date = "2012-04-05"`).

When the application receives a request from a user and needs to store those values in its database, it constructs a query using those values. Figure 1.1 shows an example query written in SQL (Structured Query Language). SQL is the *lingua franca* for most databases in use today. In this query, the values `B-`, `08634`, `ps6`, and `CSCI 100` came from a web form on which the user, presumably a professor, typed in the grade and the student ID, and selected the assignment and class from a drop-down. Update queries usually return either no results or a single result indicating the number of rows updated.

```
UPDATE grades
SET grade = 'B-'
WHERE student_id = '08634' AND
class = 'CSCI 100' AND
assignment = 'ps6'
```

Figure 1.1: Update query

Another page of the same application might show a particular student all

of his grades for a class. Figure 1.2 shows a query that returns one row for each graded assignment the student submitted. In this case, the application doesn't allow the student to enter his student ID; otherwise any student could look at anyone else's grades. Instead, it looks up his student ID based on his login account. The value `CSCI 100` originates from a drop-down selection.

```
SELECT student_id, class, assignment, grade
FROM grades
WHERE student_id = '08634' AND
class = 'CSCI 100'
```

Figure 1.2: Retrieval query

1.1.2 SQL Injections

To see how a injection attack works, we consider a curious student who would like to know everyone's grades for the class. Instead of using the standard form with the class drop-down to submit his request, he manually creates a request with a specially crafted value, such as `xx' OR class = 'CSCI 100`. Notice that he has added quotes into the middle of his value. This allows him to insert non-value text into the query, modifying its structure and meaning. The full query now looks as in Figure 1.3, and it retrieves all of the grades for every student in the class, in violation to the original policy that students can only see their own grades.

```
SELECT student_id, class, assignment, grade
FROM grades
WHERE student_id = '08634' AND
class = 'xx' OR class = 'CSCI 100'
```

Figure 1.3: Query with injection attack, revealing grades for all students

Let us consider another attack, in which a disgruntled student wishes to change all of her grades to A+. The application does not allow her to access the grade-setting page with her login, but she knows about the injection vulnerability in the grade listing page. She also knows that the semicolon character can be used to separate multiple statements in a query. Thus, she can add a new statement to the query by using the attack string `xx'; UPDATE grades SET grade = 'A+' WHERE student_id = '09928' AND class = 'CSCI 100`. The full query now looks as in Figure 1.4, and it accomplishes the attacker's goal, changing all of her grades.

```
SELECT student_id, class, assignment, grade
FROM grades
WHERE student_id = '08634' AND
class = 'xx'; UPDATE grades
SET grade = 'A+'
WHERE student_id = '09928'
AND class = 'CSCI 100'
```

Figure 1.4: Query with injection attack, changing a student's grades

1.1.3 Comment Injection

Some injection attacks require the attacker to not only add his own program structures to the query, but also remove part of the original program. In certain cases, the attacker can accomplish his goal just by removing part of the original query. Attackers can accomplish this by inserting comment markers to delimit the areas he needs to remove. In SQL there are two kinds of comment markers: the `--` delimiter indicates that the rest of the line is a comment, while `/*` and `*/` mark the beginning and end of a multi-line comment. When the database interprets the query, it ignores the commented

areas.

Consider a query that checks whether a user has supplied a valid username and password combination at login time, as in Figure 1.5. If this query returns a result row, the login was valid. Otherwise, the login was incorrect and the user should be prompted again. By injecting a comment marker into the `name` field, an attacker can log in to this application as any user, without needing a password. For example, if he provides `dwillens' --` for `name`, the query will look as in Figure 1.6, and it will return dwillens's user ID no matter what he provides for `password`. Notice that even though the quotes become imbalanced, the comment marker prevents the extra quotes from breaking the query.

```
SELECT user_id
FROM users
WHERE name = 'dwillens' AND password = 'abc123'
```

Figure 1.5: Login query requiring valid name and password combination

```
SELECT user_id
FROM users
WHERE name = 'dwillens' -- 'AND password = 'anything'
```

Figure 1.6: Login query with injection attack

1.2 Prevalence and Consequences

The Online Web Application Security project (OWASP) consistently lists SQL injection on their Top Ten Most Critical Web Application Security Risks report. In the 2010 report, they describe it as common, easy to perform, and with severe technical consequences including loss of data or system corruption [6]. Security company Imperva recently spent 9 months monitoring 30

web applications, reporting that on average, they observed 71 attempted attacks per hour on each application, up to 1300 per hour when particular applications were subject to a focused attack. Based on data from PrivacyRights.org, Imperva estimate that nearly 300 million data records have been compromised through SQL injection since 2005 [10].

These compromised data records can range from as simple as a name and e-mail address, to credit card and social security numbers. An attacker who gains access to a system thorough SQL injection can often expose or destroy any or all of data stored on it. Among the most famous successful attacks based on SQL injection was the breach of Heartland Payment Systems in 2009, resulting in the compromise of 130 million credit card numbers [2].

Chapter 2

Prior and Related Work

Injection attacks can be prevented. Whenever a program creates a string that represents executable code, it must check to be sure that the meaning of the generated code has the same meaning as originally intended. Many ways exist to perform this check, but most or all of them require some programmer intervention to be successfully applied. The rest of this section explores several existing SQL injection-prevention strategies.

2.1 Manual Prevention Techniques

Most web applications deployed today prevent SQL injections through careful programming. The programmers of these applications keep track of the source of all the data in the program, either in their heads or explicitly in the program's code. Whenever the program needs to execute a database query, the programmer working on that section of the code determines the source each string that makes up part the query text. If there is a possibility that that string could include user-entered data, the programmer must make sure that the program neutralizes potential attacks by sanitizing the string.

2.1.1 Sanitization

For SQL query text, the most common injection vulnerabilities happen when a program accepts string data from a user, then inserts that string into query text. Typically, the string data is surrounded by quotes, to make sure that it is interpreted as data and not as executable code. However, a malicious string that includes its own quote can effectively cause the rest of its data to be interpreted as code.

Sanitizing the user-entered string data can eliminate these vulnerabilities. The sanitization process involves examining the string for any bare quotes and replacing them with escaped quotes, which will cause them to be interpreted as part of the string data, rather than ending the string.

Even though many common attack examples involve subverting quoted string data, it is worth remembering that almost all user interaction data originates from strings. Web forms, for example, transmit almost all of their data as strings, over HTTP. Even a select box containing only numeric values still transmits the selected number to the server using its ASCII representation–a string of digits. Moreover, an attacker can easily create a program that sends any string data as the value of any field on a web form.

SQL, on the other hand, treats non-string data differently from string data. Queries which include numbers are not required to include quotes around the numbers. Thus, before including user-entered "numeric" value in a query, the server must check that the string represents a valid number; for example, an integer value should contain only digits. If a program fails to perform this check, then an attacker can include any code he wants to, without having to end any quoted string. Sanitization cannot prevent this kind of attack.

2.1.2 Database Access Layers

Many programs that access a database use a special layer of software to manage the connections, construct and dispatch queries, and process the results. This layer usually includes facilities for handling the sanitization process described in Section 2.1.1. Usually, the way this works is that the program prepares a query using a library call provided by the data access layer. The query template passed to this call contains placeholders for parameter data, instead of actual parameter values. To execute the prepared query, the program provides it to a separate library call, along with the actual parameter values. Since the database access layer knows that the values passed in are parameters, and should be interpreted only as data, it can automatically apply a sanitization technique to the string data.

Some access layers also require the programmer to declare the type of each parameter in a query. This can help to address the issue of numeric data types. The access layer can automatically apply its own type checking logic to ensure that the supplied values have the correct type, so that an attacker cannot supply non-numeric data where a numeric value is expected.

Though access layers provide some automated security, their use still requires careful programming. When the access layer accepts a query template, it has no way to determine which parts of the template originated from the program and which parts are user-entered data. For example, imagine that the program allows a user to specify the name of a database table to use as part of the query. In this context, the programmer cannot use a query parameter to specify the name of the table–query parameters are only for data values. Thus, the user-entered data must be included as part of the template, and if the programmer does not check that it represents a valid table name before including it, an attacker can inject any code he wants into the query.

2.1.3 Disadvantages of Manual Prevention Techniques

One of the main disadvantages of manual prevention is the amount of care and attention required on the part of the programmer to mitigate any and all vulnerabilities. The problem is twofold. Preventing injection attacks manually requires care and attention, which means it consumes programmer time, which costs money. Nonetheless, even one unchecked vulnerability can be enough for an attacker to compromise the functionality of an application, or even worse, to gain access to the data stored in the application's database. These kinds of consequences can be disastrous.

2.2 Automated Prevention Techniques

Automated prevention techniques attempt to relieve the programmer of some of the burden of remembering to check every query for injection attacks before sending it to the database. This can both reduce the amount of development time that must be spent on prevention, and serve as a backstop, foiling attacks in places where little to no effort was spent to secure the program.

In order to automatically prevent injection attacks, a transparent software layer must be inserted between the program and database. This layer intercepts every query the program sends to the database, and examines it to determine whether or not it contains a potential attack. If an attack is detected, the prevention layer rejects the query, sending an error back to the program. Otherwise, it forwards the query on to the database.

Unfortunately, it is not generally possible to determine by examining a query string alone whether it contains an injection attack or not, because the attacker's goal is to create a valid query string which the database can execute without knowing that it performs the wrong operation. The query in Figure 1.3, for example, is syntactically a valid SQL query; only the semantic

meaning of the query has been subverted. Since an automated system cannot guess what the semantic meaning of the query should be, we cannot detect this injection without some other information. This means that an automated prevention mechanisms must maintain some extra data (metadata) about the program's query strings that allows it to detect when user-entered data changes a query's structure.

2.2.1 Randomization

Many changes to the structure of a query require the attacker to insert new keywords into the string. SQLRand [3] tries to prevent these changes by detecting when one of a query's keywords did not exist in the original program. If the keyword did not exist in the original program, it is likely that it was inserted by the user, and so the query may represent an attack and should not be executed.

The metadata required by SQLRand is embedded in the query strings themselves. When a programmer creates a new query template, she must modify the keywords in the template, appending a random string of digits to the end of each one. SQLRand provides a tool that applies the modification to a single query, but the programmer is responsible for making sure it is applied to each query. In Figure 2.1, we show how the randomized query template appears in the program's source code.

```
SELECT123 student_id, class, assignment, grade
FROM123 grades
WHERE123 student_id = '%s' AND123
class = '%s'
```

Figure 2.1: Randomized retrieval query

SQLRand uses a proxy as its transparent software layer. The programmer

modifies the program so that instead of connecting to the database directly, it connects to SQLRand's proxy, which implements the database network protocol. The proxy maintains its own connection to the real database. When the program sends a query to the proxy, it parses it using a grammar that expects randomized keywords instead of standard ones. If the proxy can parse the grammar successfully, then it produces an unrandomized version and forwards it on to the database. If it cannot parse the query, then it is possible that it contains an injection attack, so the proxy simply returns an error to the program without forwarding anything to the database. In our example, when the curious student uses the attack string `xx' OR class = 'CSCI 100`, the parser rejects it because the token `OR`, without the key appended to it, doesn't have any meaning.

2.2.2 Parse Tree Matching

Many automated systems make use of the observation that all successful injections change the structure of the query they attack. SQLGuard [4] requires programs to build query strings using `SQLGuard`, a static class they provide. When a programmer places user input in the query string, she is required to declare it as such. SQLGuard wraps the user-entered data in random keys similar to those used by SQLRand; so all of its metadata is also carried in the string itself. At query execution time, SQLGuard generates two query strings; one, the real query, simply has all of the random tokens removed. The other, a reference query, replaces all of the marked user input with dummy inputs. SQLGuard parses both query strings, and if the parse tree from the real query does not match the from for the reference query, SQLGuard determines that an injection has occurred and stops the query from executing.

CANDID [1] provides a similar verification system; it generates a reference

query by replacing user-entered data in the query string with dummy data, then compares parse trees. Instead of requiring the programmer to declare which data is user-entered, however, CANDID uses a source-to-source transformation to create a parallel data set for strings in the program. Whenever the program assigns to a string variable, CANDID inserts code to assign to a shadow variable which will be used in the reference query. If the real variable is assigned a string constant, then the shadow variable gets the same value. If the real variable receives a value from user input, then the shadow variable gets a dummy value. String operations such as concatenations are performed on both data sets in parallel. At query execution time, CANDID uses the shadow variable corresponding to the real query string as its reference.

2.2.3 Taint Tracking

In contrast to SQLRand and SQLGuard, which embed their metadata within the program strings themselves, several approaches track metadata in separate but related objects. CSSE [13] uses the idea of "taint," wherein each string has some related metadata that describes the source of its data. In CSSE, each string that is instantiated as the result of user input receives a taint marking that indicates it consists of untrusted data. Strings from within the program itself receive no marking. As strings are concatenated or otherwise manipulated, these taint markings are transferred. For example, a string which resulted from concatenating user input to a string constant would have a taint marking indicating which part was copied from the input string. At query execution time, CSSE ensures that there are no unescaped single quotes in tainted portions of the query string. Nguyen-Tuong, Guarnieri, et. al. [11] describe a similar system, in which the verification step consists of tokenizing the string, then checking to make sure that no keyword or operator tokens result from tainted data. Both systems require the use of

a modified PHP runtime system.

WASP [8] takes an opposite approach to the previous two systems, which the authors call positive tainting. WASP applies taint to trusted, instead of untrusted, data. Trusted data comprises string constants, and strings derived from sources listed in a special configuration file. At query execution time, it tokenizes the string, and only accepts the query if all operators and keywords result from trusted data. This is an inherently more conservative estimate, which prevents more injections than negative tainting. However, it also generates a higher false-positive rate, because in situations where a query results from a source that should be marked as trusted but is not, the system will improperly reject it. WASP uses a Java implementation similar to our own. They take advantage of the Java agent library to perform transformations on compiled code, inserting instructions into the bytecode that call their own methods, which keep track of taint and check queries on execution.

Chapter 3

Architecture

3.1 Goals

The goal for this project is to introduce a fully automated system for detection and prevention of SQL injection attacks. All of the automated systems described so far require some manual effort at development time. For example, to use SQLRand, a programmer must change all query templates in a program's source code so that they only contain randomized keywords, and must change the database connection parameters so that the program connects to SQLRand's proxy instead of the database.

There are several reasons to desire a more automated solution than those proposed previously. Some amount of programmer effort must be spent to make the required modifications to the program. In addition, the modified source code may be harder to read and understand, which also contributes to development costs.

Another advantage we gain from fully automating this process is assurance that all code paths leading from user input to the database are checked. With manual approaches any mistake in the program, such as unsanitized string data or a missed data type check, leads to a vulnerability.

Most importantly though, a major disadvantage of any solution that requires source code modification is that the application of the solution requires access to the source code. If a person or organization wishes to deploy a web application developed by a third party, they may not be able to modify query templates or ensure that the program always sanitizes string data before inserting them into a query. We wanted to design a system that could automatically secure a program even without access to its source code.

3.2 Strategy

To protect the database from injection attacks, the protection system will still need to install a transparent access layer between the program and the database. Likewise, in order to be able to decide whether a query the program submits contains an injection, it will need to maintain some metadata about the sources of the program's data.

Our system uses a security model similar to that of SQLRand. It randomizes keywords in query strings so that it can determine the source of string data. When the program wishes to execute a query, it tokenizes the query string to see if there are any unrandomized keywords. If there are none, it passes the query on to the database; otherwise, it sends an error back to the program.

Nevertheless, the manner in which our system accomplishes these tasks is significantly different from that of SQLRand. As in WASP, we use instrumentation to automatically apply our changes to a compiled program when the runtime system loads it, eliminating the need for a programmer to manually change all of the query templates in the program. It also allows us to secure a program even without access to its source code. This way, companies can safely deploy third-party code, without having to worry whether it contains injection vulnerabilities.

Applying changes to compiled code has other systemic security benefits. For example, in SQLRand, the only way to change the randomization key is to edit the source code, rebuild the program, and test it to make sure nothing has broken. In our system, the key can be changed at will, and could even be chosen randomly when the system starts.

While automatic randomization has benefits, it also creates complications. For example, without potentially complicated dataflow analysis, we cannot tell, when we encounter a string constant in the program, whether it is part of a query template or not. This means that we are forced to replace keywords in every string in the program, even if it will never be used as part of a query. We have to implement extra functionality to ensure that the program operates correctly when strings with randomized keywords are used in a non-query context.

Chapter 4

Implementation

4.1 Instrumentation

The Java agent infrastructure allows the examination and modification of Java class bytecode at load time. Our system uses a Java agent to change the value of strings in a class when it is loaded; this is how it modifies query templates, replacing standard SQL keywords with randomized keywords. The agent also replaces certain method calls in the class with calls to methods we have written. These method call replacements allow us to insert our own code between the program and the Java libraries, which we do for two reasons. First, it serves as our transparent query verification layer, where we can decide whether a given query string contains an injection or not. Second, it allows us to correct program functioning in places where automated keyword randomization has mutated a string used outside of the context of a database query.

4.1.1 Bytecode Manipulation Tool

Since the Java agent infrastructure only provides access to the raw bytecode of a class file when it is loaded, we use the ASM bytecode manipulation tool from the OW2 consortium to implement our bytecode modifications. ASM reads the raw bytecode, and provides a simple interface for iterating over each instruction in each method in the class file. It also provides an easy interface for inserting and removing instructions from a method's bytecode.

4.1.2 `premain()`

The Java agent infrastructure gives us the ability to execute some code before the target program's `main` function begins. This allows us to register our bytecode transformers and specify which classes to transform. In some cases, our code depends on classes we would like to transform, which means they are loaded before our transformer is registered. In these cases, we use the `Instrumentation.retransformClasses` method to ask the infrastructure to reload these classes, allowing our transformer a chance to modify their code.

4.2 String Mutation

Whenever a Java program uses a string constant, it contains an `ldc` (load constant) instruction, which pushes the value of the constant on the stack. For example, the compiled code for the statement in Figure 4.1 would contain the three instructions in Figure 4.2. Note that the string data are not contained directly within the instruction. Rather, the instruction contains an index into a constant table included in the compiled program.

As each class is loaded, our Java agent examines all of the `ldc` instructions in the bytecode. When it finds one that loads a string constant, it mutates the value, replacing any SQL keywords within the string with ran-

```
query = "SELECT * FROM users WHERE name = '" + name + "' AND
password = '" + password + "'";
```

Figure 4.1: Program fragment using string constants

```
ldc #1; //String SELECT * FROM users WHERE name = '
...
ldc #2; //String ' AND password = '
...
ldc #3; //String '
```

Figure 4.2: Compiled bytecode referencing string constants

domized versions. It adds the mutated string to the class's constant table, and updates the `ldc` instruction's operand. The bytecode from Figure 4.2 might be transformed into the bytecode in Figure 4.3.

```
ldc #7; //String SELECT123 * FROM123 users WHERE123 name = '
...
ldc #8; //String ' AND123 password = '
...
ldc #9; //String '
```

Figure 4.3: Compiled bytecode with mutated string constants

4.2.1 Keyword Randomization

Our approach to mutation appends a random string of digits to the end of each keyword. It has the disadvantage that it changes the final length of the string constant, as well as the positions of data within the string. For example, in Figure 4.1, string #1 is originally 34 characters long, but after mutation, in Figure 4.2, string #7 has 43 characters. Additionally, the

keyword FROM originally extended from positions 9 to 13 in the string, but after mutation it extends from positions 12 to 19. These changes can cause challenges in maintaining program correctness, as we will discuss later.

Another approach to mutation would be to replace each keyword with a same-length string of random characters. This would have the advantage of maintaining overall string length and data positions, but we chose not to use this approach because we wanted to be able to make the possibility of collisions arbitrarily small. For example, imagine the keyword AND. In order to mutate this keyword, we have to choose a random three-character string of characters to replace it. Imagine that we chose the replacement string ZMC. If a legitimate user enters his password, baZMC8tW, then the query verification layer will produce something like Figure 4.5 to send to the database. Note that the user's password has been garbled, and as a result he will not be able to log in.

```
RDKDBS id EQNL users VIDQD name = 'john' ZMC password =
'baZMC8tW';
```

Figure 4.4: Randomized query string with password

```
SELECT id FROM users WHERE name = 'john' AND password =
'baAND8tW';
```

Figure 4.5: Derandomized query string with garbled password

It might seem that we could reduce the probability of collision by trying to pick character combinations that are unlikely to occur in a normal program. Nevertheless, it may be very difficult to pick such strings, as we can see from the password example. Additionally, to maintain security, we have to change the replacement keywords regularly; possibly even as often as every day, or every time the program is loaded. Each time we do so, it becomes more likely that we will cause a collision.

Another solution would be to choose the replacement keywords' characters from a set unlikely to occur in a program's regular operation. Unicode's large character set facilitates this–we could choose to replace `OR` with ΩΠ . This may work, but only as long as we don't try to target a system which happens to try to use these characters. It may be impossible to find a character range that will never occur in any program. Additionally, we may run into character set support issues when debugging the system's operation–examining randomized queries may become more difficult or impossible.

Our randomization solution, on the other hand, always allows us to reduce the chance of a collision. Clearly, the chances that we will encounter another string that collides with a randomized keyword go down as the randomized keyword gets longer. Even if we only use digits as the key character set, we can always reduce the probability of a collision by making the key longer. For example, it may be likely that the string `123` will occur in other parts of a program, but it is much less likely that the string `829357038497025` will occur elsewhere. By allowing ourselves flexibility on string length, we gain the ability to make the probability of a collision arbitrarily small, even if we choose new randomization keys daily.

4.2.2 Keyword Matching and Replacement

Our string mutation implementation is based on regular expressions. During `premain()`, we construct a regular expression that matches any SQL keyword, as long as it is a whole word. For example, we match the the characters `OR` in `"a OR b"` but not in `"MORK FROM ORK"`. Our transformer uses ASM to scan each instruction in each loaded class file. Each time it encounters an `ldc` instruction, it matches it with this regular expression. Any matches are replaced with the matched text, followed by the randomization key.

4.3 Method Call Replacement

Whenever a Java program calls a method on an object, it contains instructions that load that object and all of the method arguments onto the stack, followed by an `invoke...` instruction. The opcode of the instruction depends on the type of method to be called. For example, methods specified in a regular class definition are called using the `invokevirtual` instruction, whereas methods specified in an interface definition are called using the `invokeinterface` instruction. Figure 4.6 shows a typical method call for one of the JDBC query-execution methods. Figure 4.7 shows the corresponding bytecode.

```
s.execute("SELECT * FROM users");
```

Figure 4.6: Program fragment calling JDBC `Statement.execute()` method

```
aload_2 //Statement s
ldc #14; //String SELECT  FROM users
invokeinterface #15, 2; //InterfaceMethod
java/sql/Statement.execute:(Ljava/lang/String;)Z
```

Figure 4.7: Bytecode fragment calling JDBC `Statement.execute()` method

Notice that this code first loads the `Statement` instance owning the `execute` method, followed by the `String` argument to the method. For static methods, there is no owning instance to load, so a static method call simply loads the arguments before executing the `invokestatic` instruction.

We take advantage of this difference to easily replace method calls in the compiled code with calls to our own methods. For each method we want to replace, we define a static method that takes, as its first argument, an instance of the class whose method we are replacing. The static method's second argument is the same as the replaced method's first argument, the

third argument is the same as the replaced method's second, and so on. This way, when our Java agent encounters an `invokeinstance` or `invokevirtual` instruction referring to one of the methods we wish to replace, it simply replaces that instruction with an `invokestatic` referring to our static replacement method.

4.3.1 Instrumentation Annotations

Unlike the string mutation pass, which examines and potentially modifies every `ldc` instruction in a target class file, we only want to replace certain `invoke...` instructions. To make it easier to determine which method calls to replace, we devised a system of annotations with which we mark our replacement methods.

When we want to replace calls to methods in a particular class or interface, we create an instrumentation class to contain them. For example, to replace methods in `java.sql.Statement`, we define a new class, `StatementInstrumentation`. We apply the `@InstrumentationClass` annotation to `StatementInstrumentation`, as shown in Figure 4.8. This annotation requires the name of the class or interface owning the original method. If the original method owner is an interface, we also set `isInterface = true`, so that our transformer replaces `invokeinterface` instructions instead of `invokevirtual` instructions.

In `StatementInstrumentation`, we implement methods designed to replace calls to the `Statement`'s methods. These replacement methods are all static, but for non-static calls to `Statement`, each replacement method takes an instance of `Statement` as its first argument, followed by the normal parameters for that call. We mark these methods using the `@InstrumentationMethod` annotation. Using this annotation, we can specify everything about the target `invoke...` instruction we would like to

modify. This includes the type of `invoke...` instruction, the name of the target method, and its type signature. By default, we assume that it is invoked with `invokeinterface` or `invokevirtual`, depending on the value of `isInterface` in the
`@InstrumentationClass` annotation. The default name is the same as the replacement method's, and the default type signature depends on the invocation instruction. If we are replacing an `invokeinterface` or `invokevirtual` instruction, the parameters are the same as the replacement method's without the first parameter. If we are replacing an `invokestatic` method, then the targeted type signature is the same as the replacement method's.

```
@InstrumentationClass(value = "java/sql/Statement",
  isInterface = true)
public class StatementInstrumentation {

  @InstrumentationMethod
  public static boolean execute(Statement s, String sql)
      throws SQLException {

    sql = SQLRandomizer.getInstance().intervene(sql);
    return s.execute(sql);
  }

...
```

Figure 4.8: Part of `StatementInstrumentation`

4.3.2 Replacement

During `premain()`, we create our transformer, and register with it of all of the defined instrumentation classes. It uses reflection to scan all of the instrumentation classes' annotations, producing a `Map` from target method invocations to the instrumentation invocations that should replace them. Then, as classes are loaded, the transformer scans their bytecode. When it finds a method invocation, it checks whether it is a call to a targeted methods. If it is, then the transformer replaces that invocation instruction with a call to the instrumentation method.

4.4 Instrumentation Methods

We replace method calls in the target program for two reasons. First, prior to query execution, we need to insert the logic that determines whether the provided query text might contain an injection attack. Secondly, in other places where strings are used, we may need to de-mutate the string data or otherwise correct the operation of the replaced methods, so that the program continues to work.

4.4.1 Query Verification

The Java Database Connectivity (JDBC) interface is the API most Java programs use to interact with a database. Using JDBC, a program establishes a connection to its database, represented in the program by an instance of the `Connection` interface. By calling methods on this object, the program obtains `Statement` objects that represent queries it can send to the database. All of the SQL query text the program sends to the database passes through these method calls, so they are a natural place to install the transparent software layer that intercepts queries. To imple-

ment our query verification layer, we replace calls to `Statement.execute()` and `Statement.executeUpdate()`, as well as `Connection.prepareCall()` and `Connection.prepareStatement()`. Our replacement calls perform the query verification before calling the original method to send the query to the database.

Tokenizer

To implement query verification, we initially modified an existing SQL92 parser, written in Java, called JSqlParser. Unfortunately, there are many different dialects of SQL, because nearly every database management system (DBMS) has extended the language, usually to support DBMS-specific features. The plain SQL92 parser failed to recognize several syntactic structures from our initial tests.

One possible solution to this problem would be to extend the parser until it supported each individual SQL dialect completely. We find, though, that we do not necessarily need a full parser to determine whether a query string contains a potential injection attack or not. Instead, it is sufficient to tokenize the string, breaking it up into program elements such as identifiers, keywords, numbers, strings, and operators. In the example query string in Figure 4.9, there are eight tokens, including the keyword `FROM123`, the identifier `users`, the string `'Dave'`, and the operator =.

```
SELECT123 * FROM123 users WHERE123 name = 'Dave'
```

Figure 4.9: Example query string containing 8 tokens

One complication with the tokenization approach is that in SQL, not all keywords are reserved. This means that it may be impossible to tell, without fully parsing its context, whether a particular token represents a keyword or an identifier. In order for tokenization to work, we make the simplifying

assumption that keywords are not used as identifiers; therefore, we apply our check to any keyword or identifier in the token stream.

Verification

Once the query string is tokenized according to SQL's syntax, we examine the keywords and identifiers. For each one, we check whether it is an plain keyword, without the random mutation applied. This would indicate that it did not come from one of the program's string constants. If we find such a keyword, we reject the query, reporting a database error to the program.

If the query text contains no unmutated keywords, we remove all instances of the randomization key from the query. Since we only remove instances of the key in this step, an attacker cannot break the system by using his own random key. If he did so, his random key wouldn't be removed, and the database itself would reject the query as invalid.

4.4.2 Program Correctness

As mentioned earlier, one issue with a fully automated approach is that it cannot benefit from the careful eye of a programmer when it makes its changes to the program. When it mutates the program's strings, for example, it cannot tell whether a particular string is part of a query template or not. This indiscriminate modification will cause the help text `"Select whether you want guacamole or not"` to be changed to `"Select123 whether you want guacamole or123 not123"`. These modifications can cause user confusion, unexpected behavior, and program crashes. In general, we need a way to preserve program correctness even when its string data have been indiscriminately modified.

Fortunately, all operations that modify or examine string data in Java are implemented in method calls, even the + concatenation operator, which

compiles as a call to the `StringBuilder.append` method. This allows us to use the same method-call replacement technique to insert our own code whenever we want to correct the outcome of an operation on mutated string data.

String Comparison

Many of the cases we had to correct involved comparisons. We wanted to preserve the meaning of string equality, even when strings that occurred as constants have been mutated whereas strings that came from user input have not. For example, the comparison `"ROW".equals(str)` should return true whenever `str` contains `"ROW"` or `"ROW123"`. To implement these semantics, we instrument calls to `String.equals`, `String.hashCode`, and `String.compareTo`. The instrumented methods pass both values to the same routine that removes the randomized key when a query verifies as safe, before calling the original comparison method.

Any classes on which our code depends do not get transformed automatically. In some cases we are able to retransform these classes so that any time they call one of the string comparison methods, our code is invoked instead. In other cases, we were unable to retransform the dependencies. For example, `HashMap` is loaded and used extensively before our transformer can start. In this case, we replace calls to `HashMap`'s methods with calls to our own methods, so that we can undo the mutations before entering `HashMap`'s logic. Other types of `Map` and `Set` are able to use our implementations of `String.hashCode` and `String.equals` to produce correct results.

Because Java `Strings` are immutable, the string library is able to provide a faster way to test for string equality. By calling the `String.intern` method, a program can obtain a reference to a canonical ("interned") representation of the string. This reference can then be compared to other interned strings, and if these references are equal then the strings must be equal. This

can provide a performance increase over standard string comparison, which must iterate over every character in the strings if they are indeed equal.

Testing references for equality does not call a method, so we cannot apply our method call replacement strategy in this case. Instead, we observe that the primary use of interned strings is for comparison. Also, there are no performance benefits to interning a string constant that is part of a query template, i.e. interning does not provide any benefits for concatenation or value interpolation. These reasons make it unlikely that an interned string constant will ever be used in a query context (user-entered strings may be interned and used, but these are not randomized anyway). So, in this case, instead of replacing the reference equality instruction, we instead replace calls to `String.intern`, inserting our own code which removes the random key before calling the real `intern` method.

String Length and Positions

Other string operations may also be affected by our mutations. Because we have elected to append our key to each keyword in the statement, rather than picking same-length replacements, our randomization pass will change both the overall length of string constants and the positions of characters within them. Consider the string in Figure 4.10, which becomes the string in Figure 4.11. when mutated. The original string has length 19, while the mutated string contains 25 characters. The * symbol has moved from (zero-indexed) position 7 to position 10. The keyword `FROM`, which originally extended from positions 9 to 12, now reads `FROM123` and extends from positions 12 to 18. All of these changes affect the correctness of string operations such as `String.length`, `String.charAt`, `String.substring`, and `String.indexOf`, as well ass the correctness of operations that build on these primitives, like regular expression matching and string splitting.

Because these changes affect methods that return string positions as well

```
S E L E C T   *   F R O M   u s e r s
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8
```

Figure 4.10: Original String, with character positions shown below

```
S E L E C T 1 2 3   *   F R O M 1 2 3   u s e r s
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 2 2 2 2 2
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4
```

Figure 4.11: Mutated string

as those that use them as parameters, it is not an unreasonable assumption that many programs will remain correct. For example, a program that uses `indexOf` to find the spaces in our mutated example string will correctly find them at positions 9, 11, and 19. If it subsequently uses substring to extract the text between these positions, it will correctly obtain the strings `SELECT123`, `*`, `FROM123`, and `users`. The randomized keywords obtained may be derandomized when they are used in some other context later. Only if the program tries to find and operate on characters in the random key will it become incorrect. Note also that only string constants are randomized. All of the string-indexing operations continue to operate the same way on strings obtained as input. Many programs do not attempt to use indexing operations on their own string constants, or even on strings built by concatenating them with input. So, it may be possible to simply ignore the string length and position changes, and expect that many programs will still be able to operate properly. We pursued this course, after finding that all of our test programs fell into this category.

Nevertheless, it may be desirable to maintain the consistency of these operations for programs as much as possible. For example, mutation might make a program's regular expressions match incorrectly—the expression `([A-Za-z]+ [A-Za-z]+)`, which matches two sequences of letters separated by a space, will no longer match the string `"FROM users"`, because it has been mutated to `FROM123 users`. In addition, programs which supply string positions as constants will no longer work properly. For example, a program which expects to retrieve `"FROM users"` by calling `substring(9, 18)` will instead receive `" * FROM12"`. (Note that the arguments to substring are the beginning index and the ending index, and that the ending index is not inclusive.)

We implemented a solution to correct these issues, making `indexOf`, `charAt`, `length`, and `substring` consistent with respect to unmutated positions. All four receive and return position values as if they were indexed into the original string. To implement `indexOf`, `charAt`, and `length`, it is sufficient to demutate the string before passing it to the real API method. `substring` requires some extra work.

For `substring`, we would like to preserve the randomization of a keyword if and only if it is fully contained within the requested substring. For example, `substring(7, 15)` should return `"* FROM123 u"`, whereas `substring(7, 11)` should only return `"* FR"`, and `substring(11, 15)` should only return `"OM u"`.

The first step is to build a table describing where the keywords exist in both the unmutated and mutated string. In our example string, this looks as in Figure 4.12. We build this table using the same regular expression we use when derandomizing strings. Each time it matches a randomized keyword, we add its position in the mutated string to the table. The offsets in the original string can be calculated by subtracting the number of randomized keywords we have seen so far, times the length of the random key string

(3 in this example). Keyword extents can never overlap, because only one keyword precedes each instance of the key.

Keyword randomization table:

Keyword	Original Start	Original End	Mutated Start	Mutated End
SELECT	0	6	0	9
FROM	9	13	12	19

Figure 4.12: Randomization Table

Once we have the table, we use it to adjust the beginning and ending index of the substring we are to retrieve. The substring's beginning position in the mutated string must be offset by the same amount as the end of the closest preceding keyword. So, we find the keyword with greatest "original end" position smaller than the substring beginning position, and adjust the substring beginning position by the difference between that keyword's "original end" and "mutated end" positions.

The end position shifts similarly. However, if the beginning of the substring falls in the middle of a keyword, the logic is not quite the same. In this case, if the substring end position coincides with that keyword's end position then we shift by the amount for the preceding keyword (as we would if the position were in the middle of this keyword). This prevents us from including the random key when a substring does not pick up the whole keyword. If the substring continues past the end of the keyword, we have to break the substring operation up into two operations, one ending at the end of the keyword, and the other beginning there. We then concatenate their results to produce the correct substring.

`substring(7, 15)` does not begin in the middle of a keyword, so it does not have to be broken into two operations. To adjust the beginning position, we find that SELECT is the closest preceding keyword. SELECT's end position shifts by 3 positions, so the beginning position in the mutated string starts

at 10. The end position does not coincide with the end of a keyword, so we shift it by 6, the same amount as the end position of `FROM`; producing the ending position 21. If we examine Figure 4.11, remembering that `substring` is not inclusive of the end position, we can see that the characters from 10 to 20 represent the substring we wanted to generate, `"* FROM123 u"`.

`substring(7, 11)` proceeds similarly. The beginning position 7 shifts to 10 as before, but the closest preceding keyword for ending position 11 is `SELECT`, so we shift 11 by 3, to 14. The characters from 10 to 13 in the mutated string are, again, the correct substring, `"* FR"`.

`substring(11, 15)` does have to be broken into two operations, because its beginning position is in the middle of the keyword `FROM` (which extends from positions 9 to 13), and its end position is after the end of the keyword. The first sub-operation `substring(11, 13)` begins in the middle of `FROM` and ends at its end, so the end positions shifts according to `SELECT`, by 3 to 16. The beginning position also shifts by 3, to 14. Characters 14 and 15 of the mutated string produce the string `"OM"`. The second sub-operation `substring(13, 15)` shifts both positions by 6 using the offset from `FROM`. Positions 19 and 20 of the mutated string produce `" u"`, so the final result is `"OM u"`, as expected.

One consequence of making the operations consistent in this way is that random keys may be lost in certain cases. For example, consider a program that copies from a string to a character array by looping through it, character by character. Each time it calls `charAt`, it retrieves a single character from the derandomized string; at the end, the character array contains only the derandomized string. Loss of random keys may lead to false positives—our system would detect an injection where none existed. Additionally, the instrumentations we have described add a lot of overhead to string operations. Because of this, and because we suspect that many programs will operate properly even if string operations are inconsistent with constant positions,

we decided not to use this model of string consistency in our final system.

4.4.3 Output Envelope

Besides affecting program correctness, improper handling of the mutated strings can pose a security risk. If a keyword has been properly mutated, then at query verification we assume it is safe, because it must have originated in one of the program's string constants. However, if the attacker can figure out how to mutate his own keywords correctly, he can make them appear to also be safe. If the program outputs `"Select123 whether you want guacamole or123 not123"` to an attacker's display, then he has learned how to mutate the keywords `SELECT` and `OR`, and can then use them as part of future injection attacks.

We should note that this may be one major advantage our approach has over static source code randomization techniques. A standard practice during application development is to print out the text of a failing query to the user's terminal, so that the programmer can see what part of the query is producing a problem. If this debugging output is left in place in production code, or if an attacker can gain access to a development deployment of the code, then all he has to do is create an input that produces an error to reveal all of the randomized keywords in a particular query. With output derandomization, this is not a concern, because random keys will be filtered out of even debugging output.

By examining the Java API, we can identify method calls that can carry string data back to a user. For example, `PrintWriter.print(String)` is usually used to display output to a user. The constructor `File(String)` is used to create a reference to a file; a user may be able to access the file system and see the names of created files. It is unlikely that string data that passes through the output envelope will ever be used as query text.

Chapter 5

Evaluation

5.1 Security

We evaluate our system along three dimensions: security, program correctness, and performance. Security was evaluated as part of IARPA's STONESOUP (Securely Taking On New Executable Software of Uncertain Provenance) program [9]. The goal of STONESOUP is to find a way that the US intelligence community can deploy software developed by third parties without having to worry about its security infrastructure. As part of this evaluation, we were provided insecure test programs to which we applied our system. The evaluation team then ran a battery of inputs against each program. For each attack input, our system was evaluated on whether it was able to render the program unexploitable; that is, that it prevented the expected undesired behavior. For benign inputs, it was evaluated on whether it preserved the correct operation of the program.

In the STONESOUP evaluation, there were 17 separate test programs we were to secure against SQL injections. Across all of the programs, we received 28 benign test inputs, and 27 attack inputs. For all of the benign inputs, our system preserved the correct operation of the test program, that

is, we did not cause the program to break or report an injection incorrectly. For 24 of the 27 attack inputs, we successfully detected and stopped the attack. At the time of the evaluation, we had not yet implemented comment randomization; the three attack inputs we failed to detect involved comment injection.

We also tested our system against the SAMATE (Software Assurance Metrics and Tool Evaluation) Reference Dataset Juliet test suite [12], which contains a number of test programs designed to contain SQL injection vulnerabilities; these are primarily listed under "CWE-089: Failure to Sanitize Data within SQL Queries (SQL injection)." These programs are all fairly simple, but they are designed to test a wide variety of query execution methods (different ways of calling JDBC such as `execute`, `executeBatch()`, `executeQuery()`, and `executeUpdate()`) and untrusted data sources (such as reading from a network socket, file, or environment variable). Against 2024 of the test programs in this suite, we tried 12 generic attack inputs and 7 benign test inputs. We were able to detect all of the attacks, and generated no false positives.

5.2 Program Correctness

For program correctness, we tried to use real-world programs so that we could gauge the likely impact on an actual deployment. One program we tested was Daikon [7], a dynamic invariant detector developed by our group for a previous project. Daikon reads traces of a program execution and uses the data to try to find invariant conditions at various program points. The trace files can be large since they contain a full trace of the execution of a particular program, including all of the data values within the program at each point. Once Daikon has read all of the data, it does a large amount of string processing, including comparisons, concatenation, `substring`, etc. In

fact, our initial tests with Daikon were the impetus for our implementation of interning for comparisons. After applying our system to Daikon, we were able to verify that it produces the same output as it does without our system.

Since the primary target for our system is web applications, we also tested it against the open source web application server Tomcat [5]. To test it, we implemented a small servlet which executes a few simple SQL queries. We confirmed that the servlet still worked properly with benign inputs while our system was operational. We also confirmed that attempted injections were caught and reported as errors.

5.3 Performance

To test performance, we first wrote a small driver program which reads some input from the console and uses each line to construct a query, which it executes on a sample database. Without our system enabled, the program's startup time (before it began reading inputs) was 0.5 seconds, and it took 30.1 seconds to complete after that point. With our system enabled, the startup time is 2.9 seconds, and the program takes 31.9 seconds to complete after startup. So, for this simple program, most of the overhead of our system is in the startup time. During startup, our `premain()` is running, and all of the classes that are needed to run our system and the program are loaded and transformed. This is likely because the number of classes is large—over 200 classes even for a small program. Fortunately, our target programs are generally larger, longer running processes such as servers, which only pay their startup cost before serving hundreds or thousands of requests.

To get a sense of what performance is like on larger, string-processing heavy program, we also timed Daikon. Here we found a more significant increase in runtime, from 4.1 seconds to 18.5 seconds. This heavy increase in running time is likely due to the high number of string manipulations used in

Daikon—each time an operation needs to derandomize a string, that represents a regular expression match and replacement. However, we expect that most of our targeted programs will represent a much lower string processing workload than Daikon, so this is likely to be a high upper bound on the performance penalty.

Chapter 6

Conclusion

6.1 Limitations

Our system secures applications against any injection attack that requires the attacker to add a keyword or comment marker to the query string. A classification of injection attack types given in [14] shows that this kind of approach (similar to SQLRand) neutralizes all of the types of attacks that can do damage or reveal data from an application. Some of the injection attacks it cannot prevent include illegal queries, alternate encodings, and stored procedures.

In illegal query injections, the attacker gains information by inducing an error in the query text. Since we do not actually parse the query before sending it to the database, an attacker could cause a syntax error by adding some non-keyword in an illegal place. It is not clear, though, whether he could induce any other kind of error, such as a type-checking or logical error, without introducing a new keyword into the query. In alternate encoding injections, the attacker uses a different way of representing characters to implement the attack; we cannot detect this in our current implementation, because to do so we would have to implement these other character sets.

Stored procedure injections are nearly impossible to prevent in an automated manner, because at the application level there is no way to know what the stored procedure will do with the data we pass to it; it may change data, retrieve it, or do something completely different such as run an operating system command.

Another limitation to our system relates to correctness and consistency of string operations. We described two strategies for dealing with string operations. In one we allow string operations to proceed unchanged on potentially mutated strings, which may cause incorrect program functioning in certain cases. In the other we are careful to maintain the exact semantics of string operations, at the expense of performance and of incurring false positives, wherein legitimate query strings are rejected because of the processing that occurred along the code path that generated them. Both of these solutions represent some amount of compromise, but we think that many programs will be able to operate under the first, more relaxed, model.

6.2 Further Work

The output envelope of the system is not yet fully instrumented. To do so we need to make an inventory of all the methods in the JDK, and decide for each one whether it represents a potential output from the system, adding an instrumentation method for it if so.

Currently, the random key is chosen in the system source code, to ease debugging. For a real deployment, there needs to be a way to change the key on a regular basis, at system startup for example.

Checking for keywords by tokenization works well, but the system could be more robust if it employed a full parser. For example, it could distinguish between identifiers and keywords, alleviating the issue of collisions. The problem we found with this approach is that there are a wide variety of SQL

dialects in use, so it is easier to maintain a common list of keywords than a common grammar for all of them, or one for each.

String operation performance and consistency are certainly weak points, and it is possible that further work in this area might lead to a better model that would make fewer restrictions on the string operations the secured program could employ, or offer better performance and false-positive behavior than the strictest model.

6.3 Summary

We have introduced a new system for securing application programs against SQL injection attacks. This is particularly important for web applications, which by their nature are easy targets and prone to include vulnerabilities. Our system mutates, at load time, the string constants of the program to be secured, changing any SQL keywords it finds by appending a random key. These random keys will be propagated through the program as string constants are combined with other data to form full query strings. At the same time as we mutate the string constants, we replace method calls to SQL query execution methods so that we can insert calls to our own verification layer. The verification layer tokenizes the query string, and if it finds any unrandomized keywords, it assumes they came from user input and rejects the query. Otherwise, it derandomizes the query and passes it on to the database.

Our approach has several advantages. It can be applied to compiled programs without access to their source code, so that institutions can deploy third-party code without worrying about its provenance. It prevents any injection attack that involves adding a keyword or comment marker to the query string, which defeats all injection types that directly harm the database. Finally and most importantly, it is fully automated, so it requires

no intervention by an application developer to make it work. This means that it costs less to implement, and provides more complete security than the manual solutions which are still commonly used in web development.

Bibliography

[1] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. Candid: preventing sql injection attacks using dynamic candidate evaluations. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 12–24, New York, NY, USA, 2007. ACM.

[2] BBC. US man 'stole 130m card numbers'. http://news.bbc.co.uk/2/hi/americas/8206305.stm, August 2009.

[3] Stephen W. Boyd and Angelos D. Keromytis. Sqlrand: Preventing sql injection attacks, 2004.

[4] Gregory Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti. Using parse tree validation to prevent sql injection attacks. In *Proceedings of the 5th international workshop on Software engineering and middleware*, SEM '05, pages 106–113, New York, NY, USA, 2005. ACM.

[5] Apache Foundation. Apache tomcat. http://tomcat.apache.org/, January 2012.

[6] OWASP Foundation. Top 10 most critical web application security risks–a1–injection. https://www.owasp.org/index.php/Top_10_2010-A1, June 2011.

[7] MIT CSAIL Program Analysis Group. The daikon invariant detector. http://groups.csail.mit.edu/pag/daikon/, June 2010.

[8] William G J Halfond, Alessandro Orso, and Panagiotis Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. pages 175–185, 2006.

[9] IARPA. Stonesoup program. http://www.iarpa.gov/manager_vesey.html, September 2009.

[10] Imperva. Hacker intelligence summary report an anatomy of a sql injection attack. Technical report, Imperva, September 2011. http://www.imperva.com/docs/ HII_An_Anatomy_of_a_SQL_Injection_Attack_SQLi.pdf.

[11] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting, 2005.

[12] NIST. Juliet test suite. http://samate.nist.gov/SRD/testsuite.php, April 2011.

[13] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation, 2006.

[14] J. Viegas W. G. Halfond and A. Orso. A classification of sql-injection attacks and countermeasures. In *Proceedings of the International Symposium on Secure Software Engineering*, March 2006.