PachyRand: SQL Randomization for the PostgreSQL JDBC Driver

Extended Abstract

Michael E. Locasto Angelos D. Keromytis
Department of Computer Science
Columbia University in the City of New York
{*locasto, angelos*}*@cs.columbia.edu*

Abstract

Many websites are driven by web applications that deliver dynamic content stored in SQL databases. Such systems take input directly from the client via HTML forms. Without proper input validation, these systems are vulnerable to SQL injection attacks.

The predominant defense against such attacks is to implement better input validation. This strategy is unlikely to succeed on its own. A better approach is to protect systems against SQL injection automatically and not rely on manual supervision or testing strategies (which are incomplete by nature). *SQL randomization* is a technique that defeats SQL injection attacks by transforming the language of SQL statements in a web application such that an attacker needs to guess the transformation in order to successfully inject his code.

We present *PachyRand*, an extension to the PostgreSQL JDBC driver that performs SQL randomization. Our system is easily portable to most other JDBC drivers, has a small performance impact, and makes SQL injection attacks infeasible.

1 Introduction

Websites that utilize database-driven content are vulnerable to SQL injection [1] attacks. Web access to a database is traditionally provided by a middleware component that translates client requests to SQL queries and returns dynamically created HTML based on the HTTP request parameters. This component often takes input directly from HTML forms. By taking advantage of weak or non-existent input validation, malicious users can modify or retrieve private information or bypass access control mechanisms.

Due to the widespread use of web application systems, SQL injection attacks represent a serious threat to both organizations that have deployed web applications as well as the users that trust these systems to store confidential data. The contribution of this paper is the design and deployment of a system to automatically counter such attacks. The system's name is taken from a combination of *randomization* and *pachyderm*, which is derived from the obsolete order name (itself from the Greek "pakhudermos", meaning thick-skinned) for PostgreSQL's mascot, the elephant.

1.1 SQL Injection Attacks

Since many SQL queries follow predictable patterns, and the input (HTML form elements that correspond to HTTP parameters) is under the direct control of an attacker, attackers can insert arbitrary pieces of SQL code into these form fields. For example, rather than supplying a username to an element called "`username`," the attacker can supply a specially crafted piece of SQL code that, when merged with the query template, causes the query to exhibit radically different behavior than the query designer intended.

Even if the attacker does not know the underlying table structure of the database, injecting SQL or

supplying bad input can cause the middleware to return error messages. Such messages are meant to aid debugging efforts, but in this instance they help an attacker gain knowledge about system internals. This technique has been shown to be effective [2, 6].

The current defense against such attacks is to employ better coding practices that aim at producing more effective input validation routines. Since this strategy does not provide a quantifiable solution to the problem, we employ a variation of instruction set randomization called SQL randomization. SQL randomization enables applications to automatically defend against SQL injection attacks, even when input is not properly validated.

1.2 Instruction Set Randomization

In order to understand how SQL randomization can protect web applications, it is useful to understand the concept it is based on: instruction set randomization (ISR). The basic premise of ISR is to create unique execution environments for individual processes. This environment is created by performing some reversible transformation (*e.g., XOR*) on the instruction stream; the transformation is driven by an unique random key for each executable binary object. The binary is then decoded during runtime with the appropriate key. This technique forces an attacker to guess the key to exploit a code-injection vulnerability.

Since an attacker crafts an exploit to match some expected execution environment (*e.g.* x86 machine instructions) and the attacker cannot easily reproduce the transformation for his exploit code, the injected exploit code will be invalid for the specialized execution environment. The mismatch between the language of the exploit code and the language of the execution environment causes the exploit to fail. Without knowledge of the key, otherwise valid (from the attacker's point of view) machine instructions resolve to invalid opcodes or eventually crash the program by accessing illegal memory addresses. Previous approaches to ISR [3, 5] have proved successful in defeating code injection attacks.

1.3 SQL Randomization

The ISR concept can be extended to SQL [4]. The SQL code that a web application executes is transformed from standard SQL to a specialized version. For example, the key "YOU_CANT_GUESS_THE_KEY0986532" might be appended to every SQL keyword. The web application knows that keywords have this particular key string appended to them – the attacker does not. The web application is able to de-randomize the SQL statement by stripping the appropriate key. Any SQL statements that the attacker injects into the web application will have keywords that do not match the expected key and will fail parsing. The machinery that accomplishes this process in our system is described in Section 2 and Section 3.

We believe that using *PachyRand* does not impose a significant performance overhead. Response times for requests are likely dominated by the network latency between the web server and the client, the processing time of the web application's "business logic," and the processing time of the database itself. We plan to validate this performance hypothesis by observing how long the driver takes to derandomize SQL statements of various lengths and parsing complexity.

1.4 Open Source Security Research

PachyRand represents a good example of the benefits of being able to implement an innovative research idea on an open and widely-used platform. Development methods that provide unfettered access to the source code allows the research community to rapidly apply novel techniques and concepts to real systems. This kind of unencumbered contribution is especially important in implementing security measures. SQL randomization is one such security feature that will have a large impact on the community involved in delivering and consuming web-accessible database content.

1.5 Paper Organization

The remainder of this paper abstract is dedicated to explaining the design (Section 2) and partial implementation (Section 3) of *PachyRand*. Implementation continues, and the system is freely available from our website[1]. We plan to submit the system to the PostgreSQL JDBC driver team for consideration. Hopefully, its adoption can encourage the implementation of SQL randomization for other languages with database interface libraries.

2 PachyRand Design

Our system consists of two components: an offline SQL randomization tool used to randomize the SQL statements inserted into a Java application (for example, a Java servlet) and an online module that is part of the JDBC driver (as shown in Figure 1). This second component is responsible for parsing SQL statements and queries and de-randomizing them. If de-randomization does not succeed, then the driver throws an exception.

The operation of the system utilizing our modified driver proceeds as detailed in Figure 1. The client (1) sends an HTTP request to the web server. The web server recognizes that this request should be handled by the application server and passes it off to that logic (2). The application server then utilizes the services of the JDBC driver to obtain data from the database (3). The database answers the query (4) and the application server passes an HTTP response back to the client (5) via the web server. In our system, the SQL statements in the application server have been randomized, and are de-randomized by the JDBC driver before being sent to the database. Any SQL injected by the attacker as part of (1) will fail to de-randomize correctly, causing the driver to raise an exception; the injected SQL never gets to the database.

[1] `http://nsl.cs.columbia.edu/projects/pachyrand/pachyrand.tar.gz`

2.1 Benefits

Our approach has four benefits:

1. The system **protects both validated and non-validated** SQL commands against SQL injection attacks.
2. A **quantifiable security property**, whereas previous systems only provide *ad hoc* methods of preventing or detecting SQL injection. The difficulty of successfully injecting SQL is comparable to the brute force guessing of a cryptographic key.
3. The anticipated **performance impact is small**. This property is especially important, because the system operates online and is invoked for each request.
4. Provides the previous benefits **without the need for complex validation logic** that may have subtle bugs and miss corner cases. Such code often clutters up the source, making it difficult to maintain and debug the application.

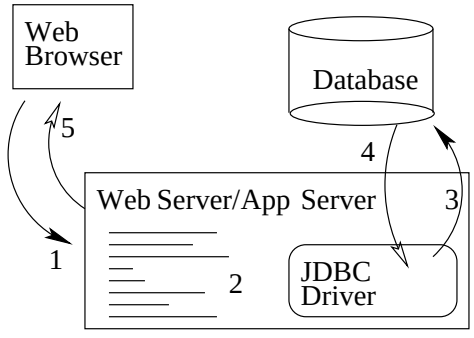

Figure 1: **General architecture for JDBC-based web applications. The application server includes processing logic that invokes the functionality of the JDBC driver to pass SQL statements to the database.**

2.2 System Limitations

Dealing with stored procedures is a difficult issue, as these SQL statements are stored directly in and

invoked by the database itself. It is not possible to de-randomize them without changing the SQL parsing logic in the database.

Changing the randomization key is a potential issue. This problem can be addressed by storing queries in an external data source (*e.g.*, an XML file) that the application reads in during execution. This content can easily be randomized during runtime under a different key.

One interesting issue is that SQL randomization is meant to work in an environment where the attacker does not have direct control over the de-randomization process. If an attacker controls this component (in our case, the JDBC driver), they can easily discover the randomization key.

Finally, note that *PachyRand* does not handle any error generated by a failure to de-randomize an SQL statement; rather, its purpose is to detect an attack and then notify the web application by generating an `SQLException`. Instances of `SQLException` occur normally during JDBC operations, and applications generally expect to handle them. Web applications should be wary about displaying the error messages. Doing so may reveal the randomization key.

3 Implementation

The implementation of *PachyRand* has proceeded in two phases. The first phase was a proof-of-concept system that utilized a simple but incomplete parsing strategy for SQL statements. The prototype first system is available for download[2]. We are working on an improved version of the system that employs code generated by Antlr[3] 2.7.4 to properly parse SQL statements. The grammar that Antlr works from was translated by hand from the PostgreSQL 7.4.5 grammar source (meant for flex/bison). The only difference between the two implementations is the parsing code.

Our implementation is quite compact. It contains a utility to randomize SQL statement strings (from either standard input or a file) and modifications to the JDBC driver to configure the driver with the randomization key and de-randomize the text of SQL statements passed to it via its API. Finally, our system includes a sample test application that uses the modified driver and accepts SQL statements from an interactive command prompt. This test utility can be used to interactively attempt SQL injection and observe how the driver refuses to accept the modified statements.

We constructed a Java class `jdbcrand.JSQLRandomize` to take SQL statements as input and randomize them with a particular key. This tool can also generate a random key using the `java.security.SecureRandom` class. It should be used by web application developers to randomize their SQL queries before they are inserted into the application's code.

We modified the JDBC driver to accept a new configuration parameter that contains the randomization key. When the driver is asked to execute a statement or query, the driver checks if this configuration parameter exists. If it does exist, the driver parses the SQL statement and attempts to strip off the key from each SQL keyword. If this process fails, the driver raises an exception for the calling code. Except for the configuration parameter, the driver should be included in the `CLASSPATH`, loaded, and invoked normally.

The software is compiled via a standard makefile. Wrapper scripts written for a Unix shell environment are provided to run the randomizer and the sample test application. A patch against the current CVS head of the JDBC driver is provided, as well as a binary version of the modified driver.

4 Related Work

The work most closely related to ours is Boyd *et al.* [4], which presents the concept of SQL randomization for MySQL. *PachyRand* is an extension of SQL randomization to another platform and slightly more compact organization.

[2] `http://nsl.cs.columbia.edu/projects/pachyrand/pachyrand-poc.tar.gz`

[3] `http://www.antlr.org/`

Although SQL injection attacks represent a critical threat to web applications and databases, the work to address this type of vulnerability has focused on providing better input validation logic. There is some work [7] being done to modify Intrusion Detection systems to detect SQL injection abuses, but these efforts still have two shortcomings. First, they rely on writing complex regular expressions to formulate a rule that can match certain suspicious SQL data. Such rules likely do not cover all possible cases of injection strings. Second, the IDS merely notes the possibility of an attack; it does nothing to prevent one. Even if combined with a tool that rewrites the data or denies the request, such systems are susceptible to false positives.

Work that relates to the general instruction set randomization technique is RISE [3], which applies a randomization technique similar to our Instruction-Set Randomization [5] for binary code only, and uses an emulator attached to specific processes.

Because the methods of injection are quite similar, cross-site scripting[4] is closely related to SQL injection. Cross-site scripting is an example of an attack that can be performed on web applications. An attacker supplies HTML code as the input to a form element, and the web server then renders that input as part of a results page. For a non-malicious request, displaying such data (*e.g.*, a username) is usually harmless. CSS attacks rely on the developer not employing proper input validation by escaping special HTML characters.

5 Conclusions

We have presented *PachyRand*, an extension to the PostgreSQL JDBC driver that performs SQL randomization. This system is broadly applicable (web-based dynamic content systems are widely deployed), addresses a critical security problem (SQL injection attacks), implements an innovative idea (SQL randomization), and is an improvement over previous efforts. Our system is easy to use, portable to other JDBC drivers, should have a small performance impact, and makes SQL injection attacks infeasible.

[4] `http://httpd.apache.org/info/css-security/`

References

[1] CERT Vulnerability Note VU#282403. `http://www.kb.cert.org/vuls/id/282403`, September 2002.

[2] C. Anley. Advanced SQL Injection In SQL Server Applications. `http://www.nextgenss.com/papers/advanced_sql_injection.pdf`, 2002.

[3] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 281–289, October 2003.

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

[5] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the ACM Computer and Communications Security (CCS) Conference*, pages 272–280, October 2003.

[6] D. Litchfield. Web Application Disassembly wth ODBC Error Messages. `http://www.nextgenss.com/papers/webappdis.doc`.

[7] K. Mookhey and N. Burghate. Detection of SQL Injection and Cross-site Scripting Attacks. `http://www.securityfocus.com/printable/infocus/1768`, March 2004.