Transparent SQL Injection Defense Method using Programming Language Constructs

YUKI MUKASA[1,a)] AKIHITO NAKAMURA[1,b)]

Abstract: SQL injection (SQLI) is a type of the most serious and well-known vulnerability for any server-side application with a back-end database. It can typically lead to confidentiality and integrity failures: exposure, defacement, or destruction of information. An attacker passes malicious strings as inputs to the application; they are injected into SQL statements and unexpected commands are executed. In this paper, we discuss how to defend against SQLI attacks in a fundamental way. The proposed method utilizes programming language constructs to detoxify dangerous SQL statements even if the application code is vulnerable to SQLI. Input strings dynamically passed to applications are marked and tracked for any concatenation by the language runtime. If an externally-influenced string is injected into an SQL statement, the statement is automatically converted into a parameterized statement and the string is treated as just a value and can't be an SQL fragment. We demonstrate the feasibility by two different types of programming language and construct: metaprogramming in Ruby and source code transformation in JavaScript. Our method eliminates the problems of programmer involvement, false negatives or positives, and additional infrastructures, while it is defensible against the most types of SQLI attacks. The performance degradation is negligible for common Web application components and environments.

Keywords: SQL injection attack, software vulnerability, unauthorized access, information leakage

1. Introduction

SQL injection (SQLI) is a type of the most serious and well-known vulnerability for any server-side application with a back-end SQL database [19,20]. SQLI vulnerabilities can typically lead to confidentiality and integrity failures: exposure, defacement, or destruction of information. In addition, user authentication and/or authorization could be ruined. In 2010, 2013, and 2017, injection vulnerability, including SQLI, was rated the number one attack on the OWASP Top 10: the 10 most critical Web application security risks [1]. Also, SQLI is ranked 6th in the 2019 Common Weakness Enumeration (CWE) Top 25 -- Most Dangerous Software Errors list [2]. These facts show the severity of the SQLI vulnerabilities and need for defense efforts.

SQLI vulnerabilities are caused by incorrect filtering of user inputs which could be used as parts of SQL statements. If inputs from users or external systems are injected into SQL statements, attackers can potentially abuse the application to execute malicious commands on the database. A considerable amount of research on SQLI defense has been conducted [3,4]. Those methods can be classified into three categories: *defensive coding*, *vulnerability detection*, and *runtime prevention*. Defensive coding, also known as *secure coding*, is a practice of developing software in a way that guards against the accidental introduction of vulnerabilities. Parameterized statements are well-known technique in this category [20]. The practice depends heavily on programmers' knowledge and skill and thus, error-prone. Vulnerability detection is a method to detect SQLI vulnerabilities in source code [5,6,7,8,9,10,11,12]. In general, this type of method has limited scalability or could result in false negatives or positives. Runtime prevention methods involve mechanisms to mitigate SQLI attacks by checking runtime SQL statements [13,14,15,16,17]. The difficulties in this type of method include the exhaustive identification of inputs and the modeling of legitimate statements.

In this paper, we discuss how to defend Web applications against SQLI attacks in a fundamental way. A new runtime method is proposed and its implementations in different types of programming languages are demonstrated. The method utilizes language constructs to detoxify dangerous SQL statements even if the application code is vulnerable to SQLI. Input strings dynamically passed to applications are marked and tracked by the language runtime. If an externally-influenced string is injected into an SQL statement, it is automatically converted into a parameterized statement and the string is treated as just a value and can't be an SQL fragment. We demonstrate the feasibility by two different types of programming language and construct: metaprogramming in Ruby and source code transformation in JavaScript. Our method eliminates the problems of programmer involvement for defensive coding, false negatives or positives caused by incompletion of taint-based vulnerability detection, and additional infrastructures.

The remainder of this paper is organized as follows. In section 2, we introduce some background concepts. Section 3 and 4 describe our method for SQLI defense and implementations in specific programming languages and environments, respectively. Section 5 evaluates the method in quantitative and qualitative forms. Section 6 concludes the paper.

2. Background

Here, we briefly explain how the SQLI vulnerabilities are exploited and how the vulnerabilities are fixed.

2.1 Three-Tier Architecture for Web-based Systems

A typical Web-based system has three-tier architecture: *presentation* (user interface), *domain logic* (application processing), and *data management*. The presentation tier displays information related to the application. It runs in a Web browser deployed on a user's device. The domain logic tier controls

1 University of Aizu, Aizu-Wakamatsu, Fukushima 965-8580, Japan
a) m5221150@u-aizu.ac.jp
b) nakamura@u-aizu.ac.jp

application functionality on an application server. The data management tier includes data storage and access functions which are provided by a database management system on a database server. The most widespread data model and query language for databases are the relational model and SQL [18].

2.2 SQLI Vulnerabilities and Attacks

One of the root causes of SQLI is the creation of SQL statements as strings in the application code without correct neutralization of special elements [19,20]. This behavior, commonly known as *dynamic string building* or *dynamic SQL*, allows the injection of externally-influenced input that could modify the intended SQL statements.

As an example, we show an application component for user authentication which is vulnerable to SQLI. In Figure 1, the left and right parts represent the user interface and the SQL statement which is created by the code shown in Listing 1, respectively.

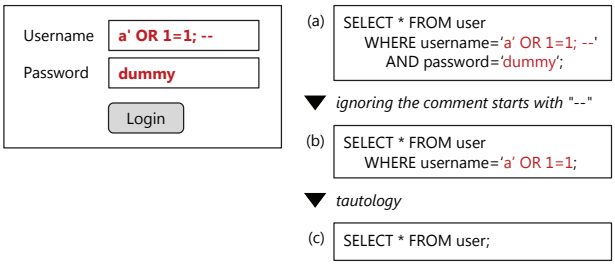

Figure 1: Example SQLI to bypass authentication

Listing 1: SQLI-vulnerable authentication code

```
1  def login(username, password):
2    str_sql = "SELECT * FROM user "
3              + "WHERE username='" + username
4              + "' AND password='" + password + "';"
5    r = execute_query(str_sql)
6    if r == NULL:
7      throw Exception('Login Failed')
8    return MyPageRender(r)
```

The SQL statement is optimized as follows.

(a) A SELECT statement is created by dynamic string building (Listing 1, lines 2-4). The input strings "`a' OR 1=1; --`" and "`dummy`", passed as the arguments `username` and `password`, are concatenated with the SQL fragments.

(b) Since `--` is the mark of a start of comment in SQL syntax, the rest of the statement is ignored.

(c) The evaluation of `1=1` is always true; the condition given by the WHERE clause becomes empty by a tautology.

The resultant SQL statement is "`SELECT * FROM user;`" by which all the user accounts are retrieved. If the database contains at least one user account, the condition of the `if` block is false (line 6), and therefore, the authentication is bypassed.

2.3 Parameterized Statements: An SQLI Defense Method

One of the defensive coding practices against SQLI is the use of *parameterized statements* (or *prepared statements*) [20]. It defines the structure of an SQL statement and the structure does not change after the combination of inputs. As a result, it eliminates injections that change the structure of expected statements. The method is the best solution to SQLI defense but is highly dependent on developers' knowledge and care.

Listing 2 shows a revised code of that in Listing 1 using a parameterized statement. An SQL statement on lines 2-4 is a template with *parameters* (or *placeholders*) specified by the question marks "`?`". A function call `prepared` on line 5 sends the template to a database server to precompile. The next call `execute_query` binds values for the parameters; two input values `username` and `password` are passed to the database server and the completed statement is executed.

A parameter can only store a value and not an SQL fragment. For example, a string "`a' OR 1=1; --`" shown in Figure 1 is interpreted as a string value, neither a logical disjunction `OR` nor a start of comment `--`. As a result, the parameterized statement prevents the construction of unintended SQL statements by enforcing the separation between data and code.

Listing 2: SQLI-defensed authentication code

```
1  def login(username, password):
2    str_sql = "SELECT * FROM user "
3              + "WHERE username=?"
4              + " AND password=?;"
5    ps_sql = prepared(str_sql)
6    r = execute_query(ps_sql,[username, password])
7    if r == NULL:
8      throw Exception('Login Failed')
9    return MyPageRender(r)
```

3. Transparent SQLI Defense Method

In this section, we present the proposed SQLI defense method.

3.1 User Input Tracking

To implement taint-based SQLI detection, first we assume an *abstract language construct* which extends the existing data structure for the string type object and the related operations in the target programming language. This construct is referred to as *extended string system* (ESS). It has two functions: *user input marking* and *mark propagation*.

3.1.1 User Input Marking

The ESS data structure has two parts: a string and *marks* as a list of *slice indexes*. Strings are indexed with the first character having index 1. For a string S and indexes a and b ($a < b$), a *slice* $S[a,b]$ is a substring of S which is started from position a (included) to b (excluded). For example, "abcdefg"[2,4] is "bc".

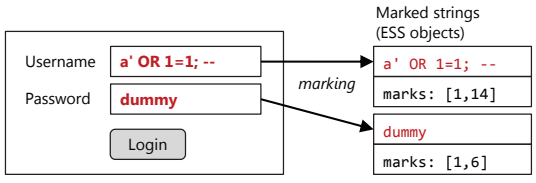

Figure 2: User input marking and marked strings

Figure 2 shows the ESS data structure and examples of user input marking. ESS marks two inputs which are passed as username and password from a Web browser. They are stored in two ESS objects and their marks are [1,14] and [1,6].

3.1.2 Mark Propagation

SQLI happens when the application code builds SQL statements

by concatenating strings, including SQL fragments and user inputs. While the strings are concatenated, the ESS marks should be kept to check if the statement includes user inputs at a later time. This function of ESS is referred to as *mark propagation*.

Figure 3 shows an example. Suppose that an SQL statement is built from two user inputs shown in Figure 2. First, in step (a), a fragment of SELECT clause and the user input for username, which has a mark [1,14], are concatenated. The resulted ESS object, in step (b), has a mark [36,49] because the user input is started from position 36 to 49. Then, the second input is concatenated for password. The resultant ESS object, in step (c), has two marks [36,49] and [65,70].

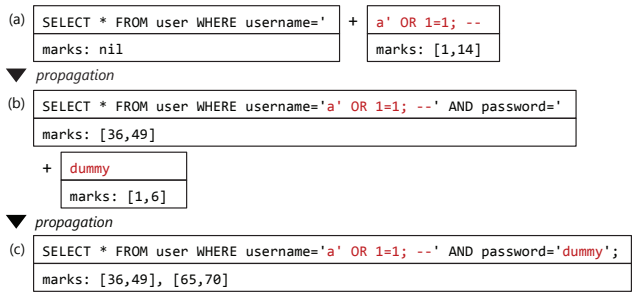

Figure 3: String concatenation and mark propagation

3.2 Defense Procedure

Figure 4 shows the procedure to mitigate SQLI. A Web application interface receives requests from clients and parses them for further processing. Query generation logic is a part of application code to build SQL statements passed to the database server via database driver.

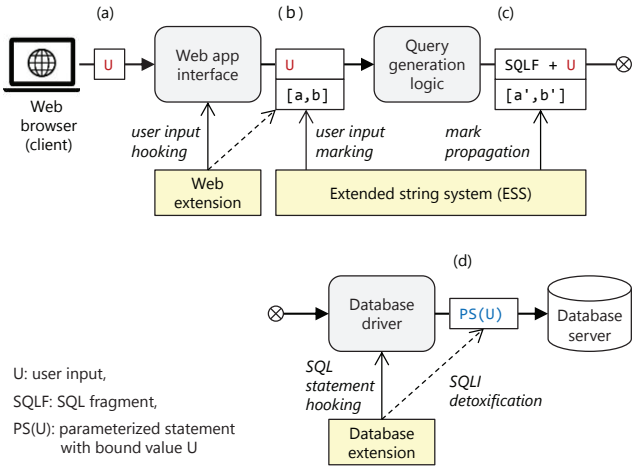

Figure 4: SQLI defense procedure

In addition to ESS, two small extensions are installed to intercept user inputs and SQL statements in order to mark the strings and detoxify the statements. The former extends the Web application interface for *user input hooking* and the latter extends the database driver for *SQL statement hooking*.

The procedure is executed through the following steps.

(a) *User input hooking*: When the Web application interface module receives user inputs, it parses them and creates string objects.

(b) *User input marking*: The ESS marking function is called by the Web extension. ESS objects are created for each input.

(c) *Mark propagation*: When a new string is created by concatenating two strings, the marks are integrated and updated properly.

(d) *SQLI detoxification*: When an SQL statement is passed to the database driver, the extension generates a corresponding parameterized statement, binds the parameter values, and executes the statement.

Figure 5 shows the details of SQLI detoxification process. Here, let's suppose that a string SQL statement *S* is created by the query generation logic in the application. As same as the previous examples, inappropriate username and password are passed and embedded in *S*. The positions of the user inputs are marked in the ESS object: marks [36,49] and [65,70].

The detoxification process is executed as follows.

(d1) *Parameterization*: ESS replaces the user input slices with parameter placeholders ("?"). *PS* is the resultant parameterized statement.

(d2) *Parameter binding*: The database extension calls a function, provided by the database driver, to prepare the parameterized statement. Then, every user input slice is bound to *PS* in sequence by calling a bound function also provided by the database driver.

(d3) *Execution*: Finally, the database extension executes *PS* with the bound parameters by passing it to the SQL execution function provided by the database driver.

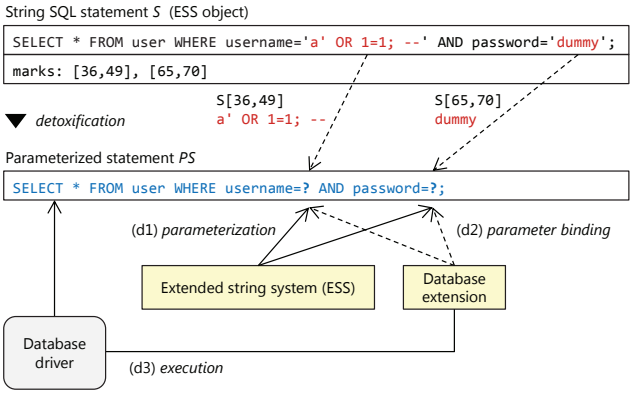

Figure 5: SQLI detoxification

4. Implementation

In this section, we describe how to implement the proposed method in specific programming languages and application environments. In order to demonstrate the feasibility of the method, two different types of programming language widely used for Web applications were chosen: Ruby and JavaScript.

4.1 Metaprogramming in Ruby

A straightforward way to track user input in code is to modify a data structure for string objects and the related operations in a programming language. There are a few languages in which such feature extension is possible at user level, and even at runtime. Ruby has an innate *dynamic metaprogramming* construct; it is possible to write code that manipulates itself at runtime [21].

4.1.1 Extended String System (ESS)

Listing 3 shows code in which the original `String` class in Ruby is extended. `StringEx` redefines a method of the `String` class: a concatenation operator `+` (lines 5-9). In fact, the operator is syntactic sugar for the `String#+` method in Ruby.

A method named `exMark` realizes the user input marking. The marks in ESS are stored in the instance variable `marks`. When strings are concatenated, the `exConcat` method is invoked instead of `String#+` to implement the mark propagation.

4.1.2 Defense Procedure

As described in section 3.2, the installation of Ruby ESS requires two extensions to an application environment: user input hooking and SQL statement hooking. In this implementation, we chose frequently used components: Rack [22] and Mysql2 [23] for Web application interface and database driver, respectively.

Listing 4 shows the extensions. `RequestEx` extends `Request` which is the request parser of Rack. It intercepts user inputs given by `params` and calls the ESS `exMark` method to mark them (lines 9-11). Next, `ClientEx` extends `Client` of Mysql2. The `query` method invokes an execution of SQL statement, given as a string parameter `sql`, which could be created by dynamic string building. If the string is marked, it should be intercepted (line 22). If so, `ClientEx` creates a parameterized statement, binds the parameters, and execute the statement (lines 24-26).

4.2 Source Code Transformation in JavaScript

We utilize a source code transformation technique for JavaScript because of lack of a metaprogramming construct.

4.2.1 Extended String System (ESS)

JavaScript supports an abstract syntax tree (AST) manipulation. This feature can be used for source code transformation, i.e. compiler, in static and dynamic manners [24]. This construct is used to implement ESS.

Listing 5 shows the implementation of ESS in JavaScript. The positions of user inputs are stored in the `marks` variable. The marking and propagation are implemented as `exMark` and `exConcat` functions, respectively.

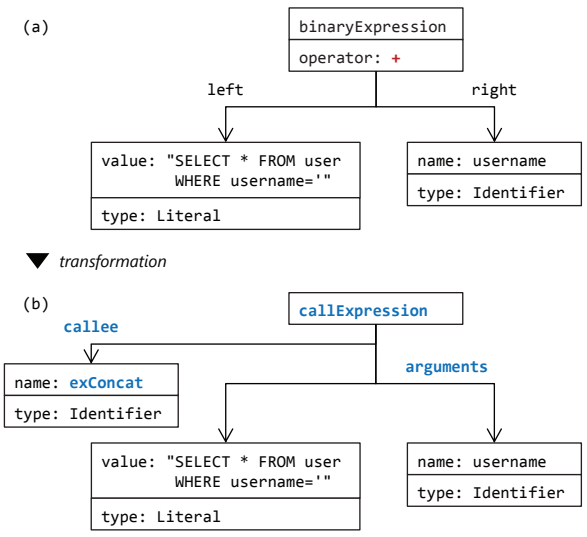

Figure 6: Source code transformation in JavaScript

4.2.2 Source Code Transformation

Figure 6 shows the transformation of source code as AST. In JavaScript, a concatenation of two strings is a binary expression whose operator is `+` and operands are the `left` and `right` strings. Figure 6 (a) represents an AST for this expression.

To implement the user input tracking, the binary expression is replaced with a call expression to a function named `exConcat`. A transformed AST is shown in Figure 6 (b). This function implements the mark propagation as shown in Listing 5. We adopt ESTree [25] as the specification of JavaScript AST.

4.2.3 Defense Procedure

Listing 6 shows the code to implement the defense procedure. We adopted Express [26] and Mysql [27] for Web application interface and database driver, respectively.

An Express application is essentially a series of *middleware function* calls, including access to request and response objects. This scheme is convenient to implement the user input hooking. A middleware intercepts user inputs stored in the `req` request object, then creates an object of `StringEx` type and calls the `exMark` method to mark each (lines 4-8). The `createQuery` function is a database extension using Mysql driver. If the type of the SQL statement, given as a parameter `sql`, is ESS (line 16), the statement includes at least one user input. Therefore, a parameterized statement is created and the bound parameters are extracted (lines 22-23).

5. Evaluation

In this section, we show the evaluation results of the proposed method and its implementations. A test application is prepared; it only has an authentication function in which combinations of username and password are checked against user accounts in a database as shown in Figure 1. There are two versions of the application: Ruby and JavaScript. In addition, there are two implementations for each language: SQLI-vulnerable and SQLI-defensed. The latter incorporate the proposed method.

5.1 Correctness

First, we tested how the system is accurate at preventing SQLI attacks. One way to evaluate this is to create real SQLI attacks. We utilized sqlmap [28], a penetration testing tool that automates the process of detecting SQLI vulnerabilities. The results show that no SQLI vulnerability was detected in the SQLI-defensed implementations in both Ruby and JavaScript, while a few vulnerabilities were certainly detected in the vulnerable ones. That is, the method can provide protection against the attacks.

5.2 Performance

Here, we show the overhead introduced by the proposed method. The performance measurement employed the platform and components shown in Table 1. Both the client and servers run on the same host. In JavaScript, the application source code was transformed before the execution using Babel compiler [24].

5.2.1 Response Time

First, the performance of the application was measured by response time experienced by the users. The response time is the time duration from initiating an HTTP request to receiving the HTTP response from the application.

Table 2 shows the measurements; the average of 10 thousand measurements. The overhead of the SQLI-defensed Ruby application, 13.81%, is conspicuous but the absolute value 0.39ms is small and negligible in most human-interactive applications. In JavaScript, the proposed method brought a good outcome because the precompilation of parameterized statements and static transformation of source code probably overcomes the increase of code.

Table 1: Performance evaluation environment

Platform	Google Compute Engine, n1-standard-1 CPU: Intel Xeon 2.20GHz 1Core, RAM: 3.75GB
OS	Debian GNU/Linux 9, kernel 4.9.0-9-amd64
Ruby	Ruby 2.3.8
Web app I/F	Rack 2.0.7 [22]
Database driver	Mysql2 0.5.2 [23]
JavaScript	Node.js 10.16.2, V8 [29,30]
Web app I/F	Express 4.17.1 [26]
Database driver	Mysql 2.17.1 [27]
DBMS	Maria DB 10.1.38

Table 2: Response time (ms)

	Ruby		JavaScript	
Original	2.85		1.78	
Proposed	3.24		1.68	
Overhead	0.39	(13.81%)	-0.10	(-5.62%)

5.2.2 Memory Usage

Next, we show the memory overhead; how much memory is being used by the application under test. Resident set size (RSS) was measured as a quantitative metric. RSS is the real memory size of the process, including the heap, code segment, and stack.

Table 3 shows the memory usage of Ruby and JavaScript implementations. There are two measurements; "i" means the initial state, i.e. before receiving the first request and "r" means the state after execution of 10 thousand requests. The overhead of the method is very little: 4.33% and 0.72%. This is mainly caused by the ESS data structure and additional code. The results show that the performance degradation is negligible for Web applications made of common components.

Table 3: Memory usage (Kbytes)

	Ruby		JavaScript	
	RSSi	RSSr	RSSi	RSSr
Original	24.65	27.41	40.73	45.99
Proposed	24.69	28.60	40.80	46.32
Overhead	0.04	1.19	0.07	0.33
	(0.17%)	(4.33%)	(0.17%)	(0.72%)

6. Concluding Remarks

In this paper, we discussed how to defend against SQLI attacks in a fundamental way. We have developed a practical method in an application transparent way. The proposed method utilizes programming language constructs to detoxify dangerous SQL statements even if the application code is vulnerable to SQLI. We also demonstrated the feasibility by two different types of programming language and construct: metaprogramming in Ruby and source code transformation in JavaScript. The proposed method and implementations successfully eliminate the problem. Our plans for future work include implementation in other programming language, including PHP and Python.

Reference

[1] OWASP Top 10, 2010, 2013, 2017. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
[2] MITRE: 2019 CWE Top 25 Most Dangerous Software Errors. https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html
[3] Shar, L. K., Tan, H. B. K.: Defeating SQL Injection, *IEEE Computer*, vol.46, no.3, 2013, pp. 69-77.
[4] Steiner, S., et al.: A Structured Analysis of SQL Injection Runtime Mitigation Techniques, *Proc. of the 50th Hawaii International Conference on System Sciences*, 2017, pp. 2887-2895.
[5] Shin, Y., et al.: SQLUnitGen: Test Case Generation for SQL Injection Detection, *TR-2006-21*, North Carolina State University.
[6] Fu, X., Li, C.-C.: A String Constraint Solver for Detecting Web Application Vulnerability, *Proc. of the 22nd Int'l Conf. on Software Engineering and Knowledge Engineering*, 2010, pp.535-542.
[7] Kieyzun, A., et al.: Automatic Creation of SQL Injection and Cross-Site Scripting Attacks, *Proc. of the 31st Int'l Conf. on Software Engineering*, 2009, pp.199-209.
[8] Alshahwan, N., Harman, M.: Automated Web Application Testing Using Search Based Software Engineering, *Proc. of the 26th Int'l Conf. on Automated Software Engineering*, 2011, pp.3-12.
[9] Livshits, V. B., Lam, M. S.: Finding Security Vulnerabilities in Java Applications with Static Analysis, *Proc. of the 14th USENIX Security Symposium*, 2005, pp.271-286.
[10] Xie, Y., Aiken, A.: Static Detection of Security Vulnerabilities in Scripting Languages, *Proc. of the 15th USENIX Security Symposium*, 2006, pp.179-192.
[11] Wassermann, G., Su, Z.: Sound and Precise Analysis of Web Applications for Injection Vulnerabilities, *Proc. of the 28th SIGPLAN Conf. on Programming Language Design and Implementation*, 2007, pp.32-41.
[12] Pietraszek, T., Berghe, C. V.: Defending Against Injection Attacks Through Context-Sensitive String Evaluation, *Proc. of the Int'l Workshop on Recent Advances in Intrusion Detection*, LNCS 3858, Springer, 2005, pp.124-145.
[13] Boyd, S. W., Keromytis, A. D.: SQLrand: Preventing SQL Injection Attacks, *Proc. of the Int'l Conf. on Applied Cryptography and Network Security*, LNCS 3089, Springer, 2004, pp. 292-302.
[14] Buehrer, G., et al.: Using Parse Tree Validation to Prevent SQL Injection Attacks, *Proc. of the 5th Int'l Workshop on Software Engineering and Middleware*, 2005, pp.106-113.
[15] Halfond, W. G. J., et al.: WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation, *IEEE Trans. on Software Engineering*, vol. 34, no.1, 2008, pp. 65-81.
[16] Liu, A., et al.: SQLProb: A Proxy-based Architecture towards Preventing SQL Injection Attacks, *Proc. of the 2009 Symposium on Applied Computing*, 2009, pp.2054-2061.
[17] Su, Z., Wassermann, G.: The Essence of Command Injection Attacks in Web Applications, *Conf. Record of the 33rd SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006, pp.372-382.
[18] Date, C. J., Darwen, H.: *A Guide to SQL Standard* (*4th ed.*), Addison Wesley, 1996.
[19] MITRE: CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection').

https://cwe.mitre.org/data/definitions/89.html
[20] Clarke-Salt, J.: *SQL Injection Attacks and Defense* (*2nd ed.*), Syngress, 2009.
[21] Perrotta, P.: *Metaprogramming Ruby* (*2nd ed.*), Pragmatic Bookshelf, 2014.
[22] Rack. https://github.com/rack/rack
[23] Mysql2. https://github.com/brianmario/mysql2
[24] Babel. https://babeljs.io/
[25] The ESTree Spec. https://github.com/estree/estree
[26] Express. https://expressjs.com/
[27] Mysql. https://github.com/mysqljs/mysql
[28] sqlmap. http://sqlmap.org/
[29] Node.js. https://nodejs.org/
[30] V8 JavaScript Engine. https://v8.dev/

Listing 3: User input tracking in Ruby

```ruby
class String
  prepend StringEx
end
module StringEx
  def +(target)
    r = super target
    exConcat(self, target)
    r
  end
  def exConcat(old, target)  #mark propagation
    ... #omitted
    if not target.getExMark.nil?
      target.getExMark.each { | mark |
        mark = mark.dup
        mark[0] += old.length
        mark[1] += old.length
        self.pushExMark mark
      }
    end
  end
  def exMark  #user input marking
    @marks = [[0, self.length]]
  end
  ... #omitted
end
```

Listing 4: Defense procedure in Ruby

```ruby
module Rack
  class Request
    prepend RequestEx
  end
end
module RequestEx  #Web extension
  def params
    req = super
    req.each { |k, v|
      req[k].exMark()  #user input hooking
    }
    req
  end
end
module Mysql2
  class Client
    prepend ClientEx
  end
end
module ClientEx  #Database extension
  def query(sql, options={})
    #SQL statement hooking
    if not sql.getExMark.nil? then
      #SQLI detoxification
      user_inputs = sql.extractExMark
      stmt = self.prepare(sql.paramStatement)
      result = stmt.execute(*user_inputs)
    else
      result = super
    end
    result
  end
end
```

Listing 5: User input tracking in JavaScript

```javascript
module.exports = function(value) {
  this.value = value
  this.marks = []
  this.length = value.length
  this.valueOf = function() {
    return this.value
  }
  this.exMark = function() {  //user input marking
    this.marks = [[0, this.value.length]]
  }
  ... //omitted
}
function exConcat(left, right) {
  var result = left + right
  if ((typeof result) !== 'string') {
    return result
  }
  result = new StringEx(result)
  if (left instanceof StringEx) {
    if (left.marks.length > 0) {
      result.marks = result.marks.concat(left.marks)
    }
  }
  let left_length = left.length
  if (right instanceof StringEx) {
    if (right.marks.length > 0) { //mark propagation
      result.marks = result.marks.concat(
        right.marks.map(
          i => [i[0] + left_length, i[1] + left_length]
        )
      )
    }
  }
  return result
}
```

Listing 6: Defense procedure in JavaScript

```javascript
const express = require('express');
const app = express();
app.use(function (req, res, next) {  //Web extension
  for (key in req.body) {
    let value = new StringEx(req.body[key]);
    value.exMark();  //user input hooking
    req.body[key] = value;
  }
  next();
});
//Database extension
Connection.createQuery =
 function createQuery(sql, values, callback) {
  //omitted
  //SQL statement hooking
  if (typeof sql === 'object') {
    for (var prop in sql) {
      options[prop] = sql[prop];
    }
    if (typeof sql.extract === 'function') {
      //SQLI detofixication
      options.sql = sql.parameterized();
      options.values = sql.extractExMark();
    }
    return new Query(options, ...);
  }
  //omitted
}
```