IPSJ Transactions on Advanced Computing Systems Vol. 4 No. 1 69–82 (Feb. 2011)

Regular Paper

Generating Effective Attacks for Efficient and Precise Penetration Testing against SQL Injection

Yuji Kosuga,[†1] Miyuki Hanaoka[†1]
and Kenji Kono[†1,†2]

An SQL injection attack is one of the most serious security threats to web applications. It allows an attacker to access the underlying database and execute arbitrary commands, which may lead to sensitive information disclosure. The primary way to prevent SQL injection attacks is to sanitize the user-supplied input. However, this is usually performed manually by developers and so is a laborious and error-prone task. Although security tools assist the developers in verifying the security of their web applications, they often generate a number of false positives/negatives. In this paper, we present our technique called Sania, which performs efficient and precise penetration testing by dynamically generating effective attacks through investigating SQL queries. Since Sania is designed to be used in the development phase of web applications, it can intercept SQL queries. By analyzing the SQL queries, Sania automatically generates precise attacks and assesses the security according to the context of the potentially vulnerable slots in the SQL queries. We evaluated our technique using real-world web applications and found that our solution is efficient. Sania generated more accurate attacks and less false positives than popular web application vulnerability scanners. We also found previously unknown vulnerabilities in a commercial product that was just about to be released and in open-source web applications.

1. Introduction

Web applications have become prevalent around the world with the success of a wide range of web services, such as on-line stores, e-commerce, social network services, etc. However, web applications designed to interact with back-end databases are threatened by SQL injection attacks. SQL injection is a technique used to obtain unrestricted access to databases through the insertion of maliciously crafted strings into the SQL queries through a web application. It allows an attacker to spoof his identity, expose and tamper with existing data in the database, and control the database with the same privileges as its administrator. This is caused by a semantic gap in the manipulation of the user inputs between a database and a web application. Although a web application handles the user inputs as a simple sequence of characters, a database handles them as query-strings and interprets them as a meaningfully structured command. According to Cenzic [1], SQL injection was the second most frequently reported web application vulnerability during the second half of 2009, which made up for 16% of the vulnerabilities found in web applications that they investigated.

Sanitizing is a technique that is used to prevent SQL injection attacks by escaping potentially harmful characters in client request messages. Suppose a database contains `name` and `password` fields in a `users` table, and a web application contains the following code to authenticate a user's login.

```
sql = "SELECT * FROM users WHERE name = '"
    + request.getParameter(name)
    + "' AND password = '"
    + request.getParameter(password) + "'";
```

This code generates a query to obtain the authentication data from a database. If an attacker inputs "`' or '1'='1`" into the `password` field, the query becomes:

```
SELECT * FROM users WHERE name = 'xxx' AND password = '' or '1'='1'.
```

The `WHERE` clause of this query is always evaluated true, and thus an attacker can bypass the authentication, regardless of the data entered in the `name` field. To prevent this SQL injection, the web application must sanitize every single quote by replacing it with a double quote. If properly sanitized, the query becomes:

```
SELECT * FROM users WHERE name = 'xxx' AND password = '" or "1"="1',
```

where the entered values are regarded as a string. This technique prevents an SQL injection from changing the syntax of the SQL queries.

Although sanitizing all the client inputs is a sufficient measure for preventing SQL injection attacks, it is not implemented well, judging from the fact that SQL injection is the most frequently reported vulnerability, as highlighted in the Cenzic's report [1]. This is because sanitizing is often done manually by developers in a process subject to mistakes and oversights even when attempted by a skilled and educated programmer.

†1 Keio University
†2 CREST, Japan Science and Technology Agency

In this situation, dynamic analysis tools are widely used for detecting vulnerabilities in web applications. Dynamic analysis is a technique that attempts to analyze the runtime behavior of a program. This technique is able to identify vulnerabilities introduced by programs loaded at runtime, such as plugins and dynamic libraries. In addition, this technique can be conducted on web applications written in any programming language because it does not require the source code of the web applications. It is also useful for checking legacy web applications because it does not require any modification to the source code of the web application.

Existing dynamic analysis tools for discovering SQL injection vulnerabilities are based on penetration testing, which evaluates the security of web applications by simulating an attack from a malicious user. The attack is created by embedding a predefined malicious string into an HTTP request. If the response differs from the expected one or has an error introduced by the database, they regard it as a vulnerability. It executes many unsuccessful attacks on web applications because each of predefined malicious strings is applied to every potentially vulnerable slot without considering the structure of SQL queries. In addition, the judgement of the success of an attack is perfunctory because they only inspect the response without observing the structure of the SQL queries induced by the attack.

This paper presents Sania, which *efficiently* and *precisely* perform penetration testing to discover SQL injection vulnerabilities. Sania is designed to be used in the development and debugging phases. Thus, Sania can intercept SQL queries as well as HTTP requests for discovering SQL injection vulnerabilities, unlike the existing tools that rely on only HTTP requests and responses. By investigating SQL queries, Sania dynamically generates effective attacks according to the context of each SQL query issued by the web application. By means of using the context of SQL queries, Sania can dynamically generate effective attacks suited for each web application, although it is difficult to use with existing techniques that only utilize HTTP requests and responses. By using the context of SQL queries, Sania reduces unsuccessful attacks as well as creates precise attacks that pinpoint vulnerabilities.

For example, Sania generates an attack that exploits two potentially vulnerable slots in the following SQL query at the same time:

`SELECT * FROM users WHERE name='`ϕ_1`' and password='`ϕ_2`'` (ϕ_i: potentially vulnerable slot).

In this example, Sania inserts a backslash to the first potentially vulnerable slot (ϕ_1) and a string " `or 1=1--`" to the second (ϕ_2). If these are not properly sanitized, this attack can successfully change the syntax of the `where` clause (the `name` parameter is identified as "`' and password=`" and the latter part " `or 1=1--'`" becomes always true). In this way, by investigating the context of the SQL query, Sania can generate context-sensitive attacks.

Additionally, Sania offers the users a way to optimize the penetration testing by optionally providing application-specific information. For example, a web page requires clients to enter the same data to several input fields (such as a password field and its verification field). If the entered data do not match then the web application returns a page that we did not expect. To reach the expected webpage for executing the testing, Sania allows the user to specify which fields must have the same value.

We evaluated Sania using six real-world web applications. Sania proved to be efficient, finding 416 vulnerabilities and generating only 7 false positives. Paros [2], a popular web application scanner, found only 7 vulnerabilities and generated 70 false positives under the same evaluation conditions. Moreover, we tested a production-quality web application, which was in the final testing phase before being shipped to the customer. Sania successfully found one vulnerability in the application. In addition, Sania found an unknown vulnerability in a free open-source web application.

The remainder of this paper is organized as follows. We begin the next section by reviewing and discussing related work. Section 3 describes the Sania design. Section 4 presents the technique to improve the accuracy of the testing. Section 5 describes the implementation and Section 6 presents our experimental results. In Section 7, we provide results from tests made on actual products. Finally, we conclude the paper in Section 8.

2. Related Work

In this section, we list work closely related to ours and discuss their pros and cons, and broadly classify them under three headings: (i) defensive coding tech-

niques for programmers, (ii) techniques for vulnerability discovery, and (iii) monitoring and prevention techniques at runtime.

2.1 Defensive Coding

Currently, the use of a `prepared statement` is widely recommended for eliminating SQL injection vulnerabilities. A prepared statement separates the values in a query from the structure of the SQL query. The programmer defines the skeleton of an SQL query and the actual value is applied to the skeleton at runtime. For example, the following program written in Java creates a prepared statement that has two placeholders represented with "?", to which actual values are applied.

```
String sql="SELECT * FROM users WHERE name=?  AND age=?"; /* example in Java */
PreparedStatement ps = connection.prepareStatement(sql);
```

The following program applies actual values to the placeholders.

```
ps.setString(1, request.getParameter(name));
ps.setInt(2, request.getParameter(age));
```

Since this program binds the first placeholder to a string and the second to an integer, it is impossible to change the structure of an SQL query. However, to use prepared statements, web applications must be modified and all the legacy web applications rewritten. Sania is useful for checking for SQL injection vulnerabilities in legacy applications because it does not require them to be rewritten.

2.2 Vulnerability Discovery

Several research efforts have been conducted into detecting SQL injection vulnerabilities. We classify them into three categories: (a) dynamic analysis, (b) static analysis, and (c) the combination of dynamic and static analyses. Sania belongs to the dynamic analysis category.

2.2.1 Dynamic Analysis

Dynamic analysis is a technique to evaluate an application at runtime. By leveraging dynamic analysis, we can observe how a web application behaves in response to attacks.

Sania belongs to this category and employs a dynamic analysis technique for detecting SQL injection vulnerabilities. Sania is designed to be used in the development phase of web applications, it can capture SQL queries. By investigating SQL queries, Sania dynamically generates precise attacks with using the context of SQL queries.

Existing vulnerability scanners [2)–7)] also employ dynamic analysis techniques for detecting SQL injection vulnerabilities. Unlike Sania, they do not use the context of the SQL queries in penetration testing. Since they have no knowledge on the context of SQL queries, it is difficult for them to dynamically create context-sensitive attacks such as an attack that exploits two potentially vulnerable slots introduced in Section 1. Rather, instead of dynamically creating attacks, they use predefined attacks and execute many unsuccessful attacks to a web application because each of predefined malicious strings is applied to every potentially vulnerable slot without considering the structure of SQL queries. Sania creates fewer numbers of more precise attacks that cover possible exploitations since it generate accurate attacks by making use of the structure of SQL queries.

2.2.2 Static Analysis

Static analysis is a technique that examines the source code of a web application without executing the program. By leveraging static analysis, we can perform security checks with high coverage rates. However, this analysis is typically unable to detect vulnerabilities introduced at runtime.

A static analysis tool called Pixy [8)] and the approach by Xie and Aiken [9)] use flow-sensitive taint analysis to detect several kinds of vulnerabilities in PHP web applications, including SQL injection vulnerabilities. They check whether sanitizing is performed along each path from a source (point of input) to a sink (query issuing point). QED [10)] also checks if sanitizing function is used within every path, but is based on flow-*in*sensitive technique. In QED, vulnerability patterns of interest are described in a Java-like language, and QED finds all potential matches of the vulnerability patterns from a target web application. Even though these techniques can detect the existence of sanitizing blocks of code, they do not examine the correctness of the sanitizing code. The user must manually examine them, which is always liable to make mistakes and oversights. Sania checks SQL queries that passed through those sanitizing block of code, and thus minimizes the risk of such errors.

The approach by Wassermann and Su [11)] uses static analysis combined with automated reasoning. It generates finite state automata and verifies that the SQL queries generated by the application do not contain a tautology such as

"`1=1`". Although a tautology is often used by a naive SQL injection as shown in the example in Section 1, there are other types of SQL injection that do not contain a tautology. For example, it is possible to insert a statement to drop a table, "`DROP TABLE users`". On the other hand, Sania can detect SQL injection that does not contain a tautology because it checks the syntax of SQL queries.

2.2.3 Combination of Dynamic and Static Analyses

Saner [12] combines static and dynamic analyses. In the static analysis phase, it analyzes how an application modifies user inputs along each path from a source to a sink. Since the static analysis can incorrectly flag a correct sanitizing code as suspicious, Saner executes the suspicious code with predefined malicious inputs in the dynamic analysis phase. This set of malicious inputs are predefined and thus it is difficult to test the inputs sensitive to the SQL context.

2.3 Monitoring and Prevention at Runtime

Several research efforts [13)–16)] use model checking to prevent SQL injection attacks. They build models of intended SQL queries before running a web application and monitor the application at runtime to identify queries that do not match the model. To create the models, SQLCheck [15], SQLGuard [14], and CANDID [13] statically analyze the source code of the web application. The approach by Valeur [16] uses machine learning in which typical application queries are used as a training set. The effectiveness of these approaches tends to be limited by the precision of the models.

SQLrand [17] provides a framework that allows developers to create SQL queries from randomized keywords instead of normal SQL keywords. A proxy between the web application and the database intercepts SQL queries and de-randomizes the keywords. The SQL keywords injected by an attacker would not have been constructed by the randomized keywords, and thus the injected commands would result in a syntactically incorrect query. Since SQLrand uses a secret key to modify keywords, its security relies on attackers not being able to discover this key.

Some techniques use dynamic taint analysis to prevent SQL injection attacks [18)–20)]. They use context-sensitive analysis to reject SQL queries if a suspicious input was used to create certain types of SQL tokens. A common drawback of these approaches is that legacy web applications must be modified to incorporate these techniques.

As a whole, web application service providers are reluctant to use a tool for preventing SQL injection at runtime because it would impose runtime overhead and would generate false positives that might badly effect the service itself. Since sanitizing is a sufficient measure for preventing SQL injection attacks, tools that identify and fix vulnerabilities before web applications provide a service are highly expected.

3. Design of Sania

3.1 Overview

An interactive web application ordinarily accesses its back-end database through a restricted private network by issuing SQL queries. Because Sania is designed to be used in the development phase of web applications, Sania can intercept the SQL queries between the web application and the database as well as HTTP requests between a browser and the web application. In Sania, the user sends *innocuous* HTTP requests through a web browser. Sania intercepts those innocuous requests and SQL queries issued from the web application. This is illustrated in the left-side of **Fig. 1**. Sania then begins penetration testing with the following three steps.

(1) **Identifying target slots**

Sania analyzes the syntax of the SQL queries to identify *target slots*. A target slot is a slot in an SQL query, in which an attacker can embed malicious strings to cause SQL injection. For example, when a client tries to log into a web application, a browser sends an HTTP request with

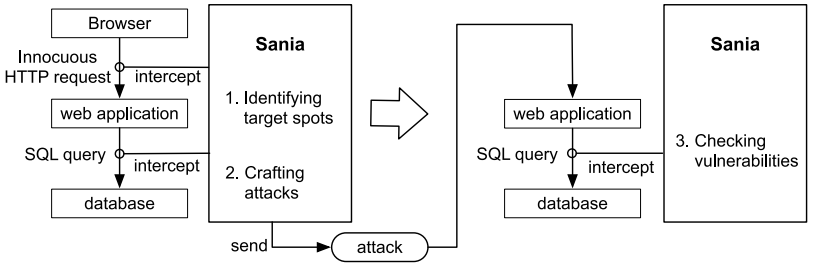


Fig. 1 Fundamental design of Sania.

parameters p_1, p_2, and p_3. If the parameters p_1 and p_2 appear at the slots $\emptyset_1$ and $\emptyset_2$ in the following SQL query respectively, we refer to the slots as target slots.

`SELECT * FROM users WHERE name='`$\emptyset_1$`' AND (password='`$\emptyset_2$`')` ($\emptyset_i$: target slot)

Then p_1 and p_2 are used for embedding attacks to cause SQL injection, but p_3 is not.

(2) **Crafting attacks**

Sania analyzes the context of each target slot to generate context-dependent attacks that can successfully change the syntax of the SQL query. In the previous example, by analyzing the context of the target slot "$\emptyset_2$", Sania generates an attack code such as "`') or 1=1--`", which contains a right-parenthesis to close the left-parenthesis. This avoids breaking the syntax of the SQL query.

(3) **Checking vulnerabilities**

After sending the attacks generated from the second step, Sania checks if there are any SQL injection vulnerability within the web application. Sania uses the well-known tree validation technique [14]; if an attack successfully injects malicious strings into an SQL query, the parse tree of the SQL query differs from that generated from the innocuous HTTP request.

The technical originality is described in Section 3.3. We also describe another optional technique to improve the accuracy of Sania's testing in Section 4.

3.2 Identifying Target Slots

In SQL injection attacks, an attacker embeds malicious strings at certain points in HTTP requests whose values may appear in target slots in SQL queries. The malicious strings can be embedded into query-strings, cookies, or any other parameters in an HTTP header.

Suppose that an HTTP request has a query-string such as "`id=555&cat=book`" and the generated SQL query is "`SELECT * FROM users WHERE user_id=555`". This query-string has two sets of data separated by an ampersand (&), and the equality sign (=) divides each data set into two elements: *parameter* and *value*. In this case, the parameters are `id` and `cat`, and their values are respectively `555` and `book`. A parameter element is fixed, but an attacker can freely alter a value element. In Sania, a target slot is identified by checking whether a value element appears in a leaf node of the parse trees of the SQL queries generated from the innocuous HTTP request.

Sania may identify a potentially safe slot as a target slot if the value of a *stateful* parameter appears in an SQL query. A stateful parameter is an HTTP parameter whose value is not embedded into any SQL query even though the same value happens to appear in an SQL query. Suppose a web page accepts the query-string, `name=xxx&action=yyy`, and the `action` parameter is a stateful parameter to determine the action of the web page. When the value of the `action` parameter is:

- `select`; the web page issues the following SQL query:
  `select * from users where name='`σ`'` (σ: the value of `name` parameter).
- others; the web page issues no SQL query.

If the query-string in an HTTP request contains `name=xxx&action=select`, the SQL query becomes for example "`select * from users where name='xxx'`". Sania determines the slots in which the values, `xxx` and `select`, appear are both target slots, even though the HTTP parameter `select` does not appear in a target slot. It results in an unsuccessful attempt to exploit a stateful parameter since its value never appears in target slots. Note that it is inappropriate to remove the reserved SQL keywords from testing, because a reserved word, such as "`select`", can appear in a target slot. To exclude stateful parameters, Sania allows the users to specify which parameters never appear in target slots. The details are discussed in Section 4.

3.3 Crafting Attacks

Sania dynamically generates attacks by analyzing SQL queries generated from an innocuous request. It embeds a malicious string, called an *attack code* into a target slot. Sania generates two types of attacks: *singular* and *combination*. In a singular attack, Sania inserts an attack code into a single target slot. In a combination attack, it inserts attack codes into two target slots at the same time.

3.3.1 Singular Attack

A singular attack attempts to exploit a target slot at a time. To create an effective singular attack, Sania generates attack codes according to the context of the target slot in the SQL query. The context is obtained by tracing back to the ancestor nodes of the target slot in the parse tree of the SQL query, since

each node in the parse tree represents the syntax in the SQL grammar. Suppose a web application issues the following SQL query to authenticate a user's log in.

`SELECT * FROM users WHERE name = '`$\emptyset_1$`' AND (password =` $\emptyset_2$`)` ($\emptyset_i$: target slot)

The parse tree reveals that the target slots $\emptyset_1$ and $\emptyset_2$ are a *string* and an *integer* in the SQL grammar respectively. In addition, Sania can learn that $\emptyset_2$ is enclosed in parentheses by tracing the ancestor nodes of $\emptyset_2$ in the parse tree.

With the information about the context of a target slot, Sania can create effective attacks. In the above example, an attack code for the string ($\emptyset_1$) is for example "`' or 1=1--`", which contains a single quote to end the `name` value in the SQL query. By ignoring the characters after double-hyphens "`--`", the attack code successfully changes the syntax of the SQL query. On the other hand, an attack code for the integer ($\emptyset_2$) is for example "`123) or 1=1--`", which does not contain a single quote to end the `password` value since $\emptyset_2$ is not enclosed in quotes. Note that the attack code contains a right-parenthesis "`)`" to end the left parenthesis. This is possible because Sania creates attack codes according to the context of a target slot.

To create context-dependent attack codes, Sania uses an *attack rule* that we prepared for each data type in the SQL grammar. An attack rule is a specification about how to create attack codes according to the data type. Sania refers to the attack rule corresponding to the SQL data type of target slot when creating attack codes. In the above example, the attack codes are created by referring to the attack rule for a string or that for an integer. We found the SQL data type of target slots can be classified into 95 patterns in the SQL grammar and defined an attack rule for each data type by thoroughly investigating SQL injection techniques [21)–28)].

Each attack rule is represented as a four-element tuple:

`(metaCharacter, userInput, insertedSQL, parentheses)`.

A `metaCharacter` holds a boolean value that represents whether to insert a quote, which *ends* the user input (usually a string) in a target slot to divide the target slot into two parts. The first part, called a `userInput`, contains a normal string that mimics the input from an ordinary user. The second part, called an `insertedSQL`, contains a part of the SQL query that an attacker attempts to inject. Since the quote inserted into a target slot represents the end of a `userInput`, the string in the `insertedSQL` is interpreted as SQL keywords. In addition, a `parentheses` also holds a boolean value that determines whether or not to insert parentheses to make an SQL query syntactically correct.

The attack code for a string in the above example "`' or 1=1--`" is generated from the attack rule for a string, which is defined as:

`(true,` λ `|` ϵ`, or 1=1-- | ;select x from z--, true)`.

This represents `metaCharacter` is required, `userInput` is either the input from the user (`λ`) or a blank (`ϵ`), `insertedSQL` is "`or 1=1--`" or "`;select x from z--`", and `parentheses` are required to create an attack code. When the `metaCharacter` is required, Sania also chooses the proper quote type, a single quote (`'`) or a double quote (`"`), according to the context of target slots. In the above example, the single quote is chosen because $\emptyset_1$ is enclosed in single quotes. The attack rule also indicates `parentheses` are required. In this case, no parenthesis is used because none of the ancestor nodes of $\emptyset_1$ holds any parenthesis.

The attack code for an integer in the above example "`123) or 1=1--`" is generated from the attack rule for an integer, which is defined as:

`(false,` λ`, or 1=1-- | ;select x from z--, true)`.

The `false` for `metaCharacter` indicates no quote is required to end the `password` value since it is not enclosed in any quotes. `userInput` has to hold the value of the `password`. Since the value of `parenthesis` is true, a right-parenthesis is embedded into the attack code to close the left-parenthesis. Sania counts the appropriate number of parentheses by tracing the ancestor nodes.

3.3.2 Combination Attack

A combination attack exploits two target slots at the same time. Sania inserts a special character into the first target slot and an SQL keyword into the second to cause an SQL injection. Suppose a web application issues the following SQL query:

`SELECT * FROM users WHERE name='`$\emptyset_1$`' and password='`$\emptyset_2$`'` ($\emptyset_i$: target slot).

Sania inserts a backslash to the first target slot ($\emptyset_1$) and a string "` or 1=1--`" to the second ($\emptyset_2$). If they are not sanitized correctly, the resulting SQL becomes:

`SELECT * FROM users WHERE name='\' and password=' or 1=1--'`,

The `name` parameter is identified as "`' and password=`" because the injected

backslash escapes the single quote. Thus, the `where` clause is evaluated to be true because "`1=1`" is always true and the single quote at the end of the query is commented out by the double-hyphens "`--`".

Sania executes a combination attack only when the first target slot is enclosed in quotes. This is because Sania can detect the vulnerability by a singular attack if the first target slot that is not enclosed in quotes is vulnerable to a combination attack. As shown above, to activate the SQL keyword injected into $\emptyset_2$, the quote that indicates the beginning of $\emptyset_2$ should be forced to indicate the end of $\emptyset_1$. To this end, if no quote encloses $\emptyset_1$, at least one quote should be injected into $\emptyset_1$. Since Sania checks if a quote can be injected to every target slot with singular attacks, Sania can detect the vulnerability without executing combination attacks.

We defined two attack rules for combination attacks:

(1) `(true, \, or 1=1-- | ;select x from z--, true)`, which is used if the second target slot is *not* enclosed in quotes, such as:
`SELECT * FROM users WHERE name='`$\emptyset_1$`' and id=`$\emptyset_2$ ($\emptyset_i$: target slot).

(2) `(false, \, or 1=1-- | ;select x from z--, true)`, which is used if the second target slot is enclosed in quotes, such as:
`SELECT * FROM users WHERE name='`$\emptyset_1$`' and passwd='`$\emptyset_2$`'` ($\emptyset_i$: target slot).

In a combination attack, Sania chooses two target slots even if there are more than two target slots in an SQL query. This is because attacking two target slots is enough to detect a vulnerability against a combination attack.

3.4 Checking Vulnerabilities

To check for an SQL injection vulnerability, Sania uses the well-known technique called tree validation [14]. In the tree validation, the structure of a query generated from an innocuous request is juxtaposed with that of an actual query generated from an attack. This technique determines the attack was successful if those structures are different.

In addition to the tree validation, Sania also checks for *multi-byte* SQL injection [29], which can bypass sanitizing through the abuse of multibyte character sets. For example, since the `addslashes()` function in PHP changes "`'`" into "`\'`", we expect the input "`0x97' or 1=1`" will be changed into "`0x97\' or 1=1`". When the Shift-JIS character set is used, "`0x97' or 1=1`" is changed into "予`' or 1=1`", in which the single quote is not sanitized properly. Ascii code level, "`0x97'`" becomes `0x9727` ("`'`" is `0x27`). `addslashes()` changes `0x9727` to `0x975c27` because "`'`" (`0x27`) is changed to "`\'`" (`0x5c27`). When this byte sequence (`0x975c27`) is interpreted in Shift-JIS, it becomes "予`'`" because `0x975c` represents a multi-byte character "予" in Shift-JIS. In this way, a multi-byte character hides a backslash. Similarly, multi-byte SQL injection is possible in UTF-8, UTF-16, BIG5, and GBK character sets, if a backslash is preceded by a byte code that does not represent a letter or a digit or a symbol (`0x20`∼`0x7e`). To detect multi-byte SQL injection vulnerabilities, Sania creates attack codes containing these type of suspicious bytes and checks those bytes do not precede a backslash in an SQL query.

4. Improving Accuracy of the Testing

Sania allows the users to specify additional information about target web applications to improve the accuracy of the testing. This information is called *Sania-attributes*. We prepared five Sania-attributes as shown in **Table 1**, and introduce them in order.

4.1 Length Attribute

A database defines the maximum length of a string that can be inserted via a web application. An attack code that is longer than the length will be rejected by the database. To suppress the creation of such attacks, Sania allows the users to specify the maximum length of the attack code. The Sania-attribute, *length-attribute*, is used to specify the maximum length so that Sania does not create attack codes whose length is longer than that specified by the length-attribute.

4.2 Equivalent Attribute

In some web pages, a client needs to enter the same data into several input

Table 1 Sania-attributes to improve the accuracy of testing.

Name	Purpose
length-attribute	To limit the maximum length of attack code
equivalent-attribute	To apply the same value to multiple fields
skip-attribute	To skip user-specified parameters
preserve-attribute	To detect a vulnerability of second-order SQL injection
structure-attribute	To accept the change of tree structure of SQL query

fields. For example, a web page has a password field and its verification field to which the same password must be entered. If these do not match, the web application rejects the request. Sania allows the user to attach Sania-attribute, *equivalent-attribute*, to HTTP parameters. By attaching equivalent-attribute, Sania inserts the same data to the parameters.

4.3 Skip Attribute

Sania excludes HTTP parameters from testing if the Sania-attribute, *skip-attribute* is attached to the HTTP parameters. This attribute is useful for stateful parameters described in Section 3.2. It is also useful for excluding a *volatile* parameter. A volatile parameter is an HTTP parameter whose value appears in the SQL query only when it has a certain value. For example, the following SQL query is issued only when the value for $\emptyset_2$ is an integer:

`SELECT * FROM books WHERE category='`$\emptyset_1$`' and price<`$\emptyset_2$ ($\emptyset_i$: target slot).

If $\emptyset_2$ is not an integer, the SQL query will be:

`SELECT * FROM books WHERE category='`$\emptyset_1$`'` ($\emptyset_i$: target slot).

In this example, $\emptyset_2$ is not vulnerable because no attack code can be created only with integers. By attaching a skip-attribute to the volatile parameter, Sania can skip the testing.

4.4 Preserve Attribute

To deal with a *second-order* SQL injection attack [21], Sania introduces *preserve-attribute*. A second-order SQL injection is an attack that targets user-supplied data temporarily stored in the web application. The attack succeeds when a certain request triggers the issue of the SQL query that contains the user-supplied data. Since Sania observes the SQL query just after a request is sent, it can not identify the user-supplied data in the SQL query. For example, request R_1 contains parameter p_1 but does not trigger any SQL query. The second request R_2 neither contains any parameter nor triggers any issue of an SQL query. The third request R_3 has no parameter but issues an SQL query that contains p_1. In this example, Sania can not identify p_1 in the SQL query triggered by R_3.

A preserve-attribute is attached to the parameter whose value appears in a later SQL query triggered by another request. Sania records all the requests between the request attached preserve-attribute and the request that triggers the SQL query. To send an attack, Sania sends all the recorded requests and checks for a vulnerability in the SQL query of interest. In the above example, Sania records the requests (from R_1 to R_3), and checks the SQL query after R_3.

Table 2 Structure-attributes and their acceptable expressions.

Name	Acceptable expressions
arithmeticExpression	Number/mathematical statements
conditionalExpression	Conditional statements such as AND/OR statements
relationalExpression	Relational statements used to compare two values, such as LIKE and IS NULL statements
notExpression	Statements that can accept NOT expression, such as BETWEEN, IN, and LIKE statements
subSelectExpression	Statements that can accept sub-SELECT expressions, such as JOIN and FROM statements

4.5 Structure Attribute

We also added another Sania-attribute to optimize the tree validation for a special case we encountered during the preliminary experiments. We found the structure of a dynamically generated query depends on the client's inputs, even though there was no vulnerability. The web application issues the following SQL query and the ø can hold an arbitrary arithmetic expression including a number:

`SELECT * FROM users WHERE id=`ø (ø: target slot).

The structure of this SQL query changes according to the value of ø, because an arithmetic expression, for example "1 + 2", is expressed as a subtree composed of two number nodes. Because of this, Sania judges the application to be vulnerable to SQL injection. To avoid this problem, Sania allows the user to attach a *structure-attribute* to an HTTP parameter, which enables the user to specify several acceptable subtrees. **Table 2** lists structure-attributes. In the above example, the user can associate an `arithmeticExpression` attribute with the `id` field to let it contain an arbitrary arithmetic expression.

4.6 Automated Deletion of Inserted Data

Additionally, we also found a case where Sania needs to delete successfully injected attack code before executing the subsequent attacks. A web page, such as a user registration page, issues an SQL query to insert user-supplied data into the database. If Sania embeds an attack code into the data, the attack code is stored in the database and will adversely affect the subsequent attack results. For example, the web site initially checks the database for the user ID specified

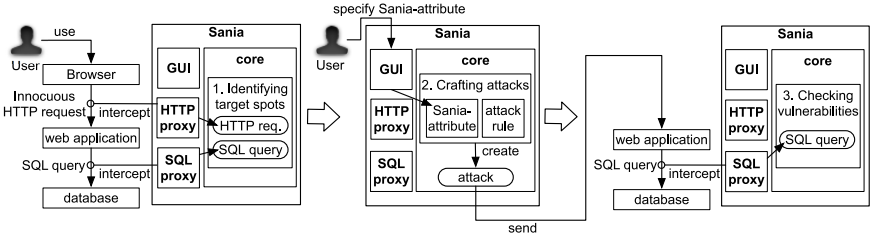


Fig. 2 Implementation and vulnerability detection process of Sania.

in an HTTP request. If the user ID is not in the database, an SQL query is issued to insert the new user information. Otherwise, the SQL query is not issued and we cannot execute testing of any value in the SQL query. To avoid this, every data inserted into the database has to be deleted before the next attack starts. Sania automatically issues an SQL query that deletes the inserted data.

5. Implementation

We implemented a prototype of Sania in Java that had 25,000 lines of code. In addition, it had a list of attack rules in XML that had 1,800 lines of code. As shown in **Fig. 2**, Sania consists of GUI interface, an HTTP proxy, an SQL proxy, and a core component. The GUI interface supplies a panel to control Sania, and the HTTP and SQL proxies intercept HTTP and SQL packets respectively. The core component performs a task to detect SQL injection vulnerabilities, such as identifying target slots, crafting attacks, and checking vulnerabilities.

In this implementation, Sania requires two user involvements; accessing the target web application with a browser and optionally providing Sania-attributes. By accessing the web application, the browser sends an HTTP request, and the request triggers SQL queries. Sania needs these packets for initializing the testing. In addition, by providing Sania-attributes, Sania can optimize the testing. Since the phase of providing the attributes is after identifying target slots and before crafting attacks, Sania can display detailed information on the GUI panel about target slots, such as which syntax they belong to in the SQL query, and create effective attacks according to the provided attributes as shown in **Fig. 3**.

A test result is output in HTML or XML. The target slots, attack codes, and

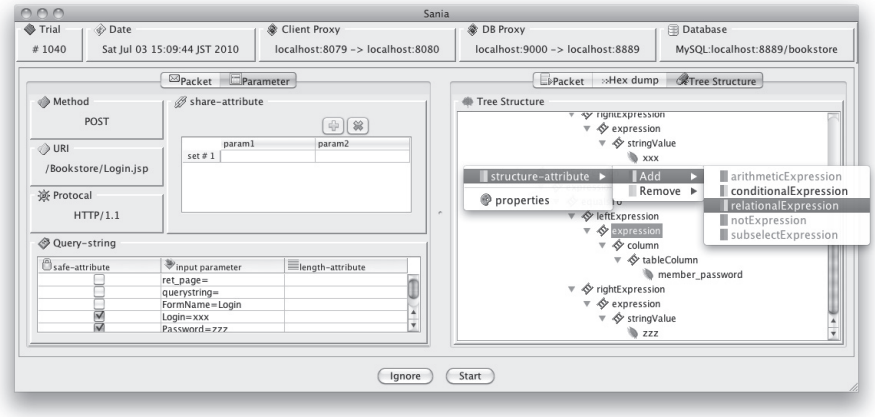


Fig. 3 Snapshot of Sania at work (selecting Sania-attributes for improving attack codes and flexible tree validation).

the structures of SQL queries are output in the document, so that the user can easily see how the SQL injection succeeded.

6. Experiments

This section presents our evaluation of Sania. We compare Sania with a public web application scanner from the two points of view: efficiency and false positives.

6.1 Experimental Setup

We selected six subject web applications to evaluate Sania. All of them are interactive web applications that accept HTTP requests from a client, generate SQL queries, and issue them to the database. **Table 3** lists the subject programs. Five of them (Bookstore, Portal, Event, Classifieds and EmplDir) are free open source applications from GotoCode [30]. We found some of them have already been used to provide real world services. Each web application is provided in multiple programming languages. We chose the JSP and PHP versions, but we show only the result of the JSP version because there is no difference in the test result. The remaining one, E-learning, is a JSP and Java Servlet application provided by IX Knowledge Inc. [31]. It was previously put to actual use on an intranet but no longer used since a newer version has been released.

Table 3 Subject programs used in our evaluation.

Subject	Description	Language	LOC	Target slot
E-learning	Online Learning System	Java (Servlet) & JSP	3,682	13 (24)
Bookstore	Online Bookstore	JSP	11,078	71 (117)
Portal	Portal for club	JSP	10,051	98 (136)
Event	Event tracking system	JSP	4,737	29 (50)
Classifieds	Online Classifieds System	JSP	6,540	40 (64)
EmplDir	Online Employee Directory	JSP	3,526	24 (38)

Table 4 Sania-attributes specified for evaluation.

Subject	length-attribute	equivalent-attribute	skip-attribute	structure-attribute
E-learning	0	0	0	1 parameter (2 tree elements)
Bookstore	0	2 (1 pair)	30	0
Portal	25	0	28	0
Event	2	0	12	0
Classifieds	0	0	17	0
EmplDir	0	0	7	0

Table 3 shows each subject's name (Subject), a brief description (Description), the languages in which the application was written (Language), the number of lines of code (LOC), and the number of target slots (Target slot) with the total number of HTTP parameters into which an attacker can attempt to inject malicious strings in parentheses.

Before Sania started to craft attacks, we had manually provided Sania-attributes. **Table 4** shows the number of attributes specified for each web application. Even though we are not the authors of the subject web applications, it was easy for us to know application-specific information for providing Sania-attributes because we only looked for some information about the maximum character length allowed at the database, input fields that must have the same value, and so on. Specifying them would be even easier for a developer of a web application.

We compare Sania with Paros [2]. We carefully investigated 25 vulnerability scanners: 15 scanners from Security-Hacks.com [32] and 10 scanners from Insecure.Org [33]. Free scanners that we could use were 21 out of the 25 scanners. 20 out of the 21 scanners can detect SQL injection vulnerabilities. Four out of the 20 can detect *unknown* vulnerabilities. Two out of the 4 scanners are automated tools. We used both of the two remaining scanners for our subjects and found Paros detected more vulnerabilities than the other. The popularity of Paros is quite high; more than 7,000 copies of Paros are downloaded every month from August 2009 to June 2010 at SourceForge.net [34]. We therefore use Paros for our comparison.

Table 5 Results for Sania and Paros.

Subject	Sania				Paros			
	trials	warns	vuls.	f.p.	trials	warns	vuls.	f.p.
E-learning	214	210	210 (21)	0	362	9	7 (5)	2 (2)
Bookstore	708	52	52 (26)	0	4,802	8	0	8 (7)
Portal	1,080	93	88 (44)	5 (5)	5,477	20	0	20 (20)
Event	276	18	16 (8)	2 (2)	1,698	21	0	21 (20)
Classifieds	498	32	32 (16)	0	1,210	6	0	6 (6)
EmplDir	290	18	18 (9)	0	1,924	13	0	13 (11)
total	3,064	423	416 (124)	7 (7)	15,473	77	7 (5)	70 (66)

Paros indiscriminately applies an attack code to every HTTP parameter. The attack code is a predefined string listed in the program. After sending an attack, Paros determines the success of the attack in two ways. First, Paros determines that the attack is successful if the response after the attack differs from the normal response. Second, it determines an attack is successful if the response message contains predefined strings that indicate the existence of a vulnerability, such as "`JDBC.Driver.error`". This technique is also employed in three commercial systems [3),4),7)].

There is no significant difference in testing time between Sania and Paros. It took around 15 minutes to perform testing for each subject application.

6.2 Results

Table 5 shows the experimental results for Sania and Paros. The table presents the number of trials (trials), the number of warning messages that a tool reported they are vulnerable (warns), total vulnerabilities with the number of actual vulnerable slots in parentheses (vuls.), and false positives with the number of target slots that were not actually vulnerable in parentheses (f.p.) for each subject. We checked whether each warning was truly vulnerable. This table reveals that Sania found, using fewer trials, more vulnerabilities for every subject and generated

Table 6 Details of vulnerabilities for Sania and Paros.

Attack Type	Sania		Paros	
Singular	194	(13)	7	(5)
Combination	222	(111)	0	
Total	416	(124)	7	(5)

fewer false positives than Paros did.

6.2.1 Accuracy of Attacks

Table 6 shows the total number of warnings for attack types. Note that there is no vulnerability that Paros could find but Sania could not. The table reveals that Sania can execute:

- Precise singular attacks. It found more vulnerabilities (194 vuls.) than Paros (7 vuls.). This is because Sania generates an elaborate attack according to the context of a target slot. It was necessary to embed a parenthesis into an attack code to detect the vulnerabilities that only Sania could detect.
- Powerful combination attacks. It found 222 vulnerabilities. A combination attack requires knowledge about target slots in an SQL query. Therefore, it is hard for Paros to work out a combination attack.

E-learning does not sanitize at all and is the only subject where Paros could successfully find vulnerabilities. All vulnerabilities in the GotoCode applications are revealed by combination attacks. For example, a web page of the `Event` application accepts a query-string, `Login=xxx&Password=zzz`, to authenticate a user's log in, and issues the following SQL query:

```
SELECT count(*) FROM users WHERE user_login ='xxx' and user_password='zzz'.
```

When Sania sets a backslash to the value in the `Login` parameter, it can easily change the syntax of the resulting SQL query. Paros cannot find them because it does not support any function to attack several target slots at the same time.

6.2.2 False Positives

Table 7 shows the number of false positives with the number of target slots that were not actually vulnerable in parentheses. In total, Sania and Paros respectively raised 7 and 70 false positives.

6.2.2.1 Data Length Error

The maximum length of data is defined at the database. Sania generated 7 false positives as a result of making attacks longer than the limitation, and Paros also generated 11 false positives as shown in Table 7.

Table 7 Details of false positives for Sania and Paros.

Subject	Sania					Paros				
	f1	f2	f3	f4	f5	f1	f2	f3	f4	f5
E-learning	0	0	0	0	0	0	0	0	0	2 (2)
Bookstore	0	0	0	0	0	0	7 (6)	1 (1)	0	0
Portal	5 (5)	0	0	0	0	2 (2)	7 (7)	6 (6)	5 (5)	0
Event	2 (2)	0	0	0	0	1 (1)	8 (7)	8 (8)	4 (4)	0
Classifieds	0	0	0	0	0	3 (3)	3 (3)	0	0	0
EmplDir	0	0	0	0	0	5 (4)	4 (3)	4 (4)	0	0
total	7 (7)	0	0	0	0	11 (10)	29 (26)	19 (19)	9 (9)	2 (2)

f1: Data length error, f2: Attacking potentially safe slots, f3: Mishandling of dynamic contents, f4: Data type error, f5: Duplicate warning

In our subject, `Portal` limits the length of the `member_password` to 15 characters at the database, and a web page in the application has a sanitizing function that translates a single-quote into two single-quotes. We used length-attributes for limiting attack codes to `member_password` to be less than 15 characters. However, the sanitizing operation converted the attack codes to longer than the length defined by the length-attribute. For example, the sanitizing operation converted an attack code "`xxx' or '1'='1`" (14 characters) into "`xxx'' or ''1''=''1`" (18 characters). The database rejected the attack code. After handling an error message from the database, the web application generated a response page that is different from the expected one. Since the web application issued an SQL query that was not the expected SQL query, Sania raised an alert. This happened 7 times.

On the other hand, Paros does not recognize the acceptable maximum length. It generated improperly long attack codes and the web application returned an unintended response page. Then, Paros determined the attack was successful because the response page was different from the intended one. This happened 11 times in total.

6.2.2.2 Attacking Potentially Safe Slots

A parameter is potentially safe when it is a stateful or volatile parameter. We attached a skip-attribute to such a parameter, so that Sania could skip testing it. On the other hand, Paros executed the testing and wrongly evaluated it which generated 29 false positives as shown in Table 7.

Paros embedded attack codes to the value of stateful parameters, and generated 16 false positives. A stateful parameter is potentially safe because its value is not embedded into any SQL query, but the same value happens to appear in an SQL query. In our evaluation, a `FormAction` parameter is used as a stateful parameter in `Bookstore`. The parameter requires its value to be "`insert`" to insert new member information, and to be "`delete`" to delete the existing user information. If a value other than "`insert`" and "`delete`" is used, the web application makes the client go back to the original page with an error, "`java.sql.SQLException: Can not issue empty query`", without issuing any SQL queries. Therefore, if an attack code is applied to this stateful parameter, no SQL query is issued and an unintended page is returned. Paros recognizes this as a successful attack, which resulted in false positive.

Paros also embedded attack codes to the values of volatile parameters, and generated 13 false positives. A volatile parameter is an HTTP parameter whose value appears in the SQL query only when it has a certain value. In our experiments, a web page of `EmplDir` checks if the user input is a number. If the user input is a number, an SQL query is issued to retrieve the user information whose user-ID corresponds to the number. If an attacker attempts to insert an attack code composed of non-digit characters, the application rejects it, issues no SQL query, and returns an unintended response page. Paros recognized it as the success of an attack, which resulted in false positive.

6.2.2.3 Mishandling of Dynamic Contents

Paros generated 19 false positives as a result of mishandling of dynamic contents, while Sania generated no false positives, as shown in Table 7. Some web applications dynamically generate web pages that contain the values entered by a user. For example, in a web page in `Classifieds`, the user can add a new category name. After the new category name is added, the web application returns a page containing a list of all registered category names. When an attacker attempts to inject an attack code to the category name field, the content of the response page always changes even when the attack fails. Paros always misjudged this as vulnerable because it regards the change in the response page as implying a successful attack, and generated false positives. Sania generated no false positives because it judges the success of an attack by looking at the structure of the SQL query.

6.2.2.4 Data Type Error

Paros generated 9 false positives by injecting improper types of attack codes, while Sania generated no false positive, as shown in Table 7. If the type of attack code is not equivalent to that of a corresponding column in a database, the database returns an error to its host web application. When handling this error message, some web applications generate a response page that is different from the intended one. For example, in a web page in `Portal`, the user enters a date-formatted string to a "`date_added`" parameter. The corresponding column in the database accepts only a date expression. Since Paros has no way of knowing the type of target slot, it executed inappropriate attacks, and generated 9 false positives. On the other hand, since Sania properly recognizes the type of a target slot by looking into the structure of the SQL query, it did not inject incorrect data type attack codes.

6.2.2.5 Duplicated Warnings

Paros generated 2 duplicated warnings, as shown in Table 7. A duplicated warning is not false alert but a redundant warning. For example, *servlet alias* enables clients to use a shortcut URL to call a servlet. In `E-learning`, accessing the URL:

`http://hostname:port/E-learning/Security`

is the same as accessing the following URL:

`http://hostname:port/E-learning/user/jsp/login.jsp`.

While Paros tests all the pages indiscriminately, Sania users can choose the page of interest to test, which suppresses this type of warning duplication.

7. Testing Real Products

After Sania was proven effective in our experiments, we had a chance to test a production-quality commercial web application developed by IX Knowledge Inc. [31] on March 28, 2007. This application, RSS-Dripper, provides RSS information to users based on their previous choices. It is written in Java Servlet and JSP, developed on Struts [35], and was in the final stage of development just before being shipped when we tested it.

The login page in RSS-Dripper accepts two parameter, `userid` and `password`.

When a query-string, `userid=xxx&password=zzz`, is supplied, it issues the following SQL query:

```
SELECT USERID, USERNAME, PASSWORD, MAILADDRESS, TIMESTAMP
            FROM USERMST WHERE USERID = 'xxx' AND TRIM(PASSWORD) = 'zzz'.
```

After Sania executed 32 attacks, it detected one SQL injection vulnerability against a combination attack with a query-string, `userid=\&password= or 1=1--`. After the testing, we confirmed that it was truly vulnerable. By analyzing the source code of RSS-Dripper, we found that it did not sanitize the backslash.

Additionally, we found an SQL injection vulnerability in Schoorbs [36], an open-source web application. This vulnerability resided in an HTML hidden field, which was supposed to only accept a number. Since it was not properly sanitized, an attacker can inject an arbitrary SQL statement preceded by a number, such as "`2;delete from schoorbs_room--`". We provided indepth information on this vulnerability to the developer, and confirmed that the vulnerability was readily fixed in the next version.

8. Conclusion

We presented Sania that dynamically generates effective attacks for penetration testing to efficiently detect SQL injection vulnerabilities. Because it is designed to be used in the development phase of web applications, it can intercept SQL queries. By investigating the context of potentially vulnerable slots in the SQL queries, Sania dynamically generates precise, context-dependent attacks. We evaluated our technique using real-world web applications and Sania was found to prove effective. It found 416 SQL injection vulnerabilities and generated only 7 false positives when evaluated. In contrast, Paros, a popular web application scanner, found only 7 vulnerabilities and generated 70 false positives. We also found vulnerabilities in a production-quality commercial web application and in an open source web application. Finally, we plan to adapt our technique to detect other injection vulnerabilities, such as cross-site scripting, XPath injection, and OS injection.

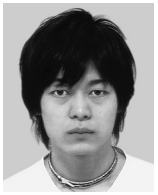
Acknowledgments The authors would like to thank IX Knowledge Inc. for providing us with E-learning and RSS-Dripper for our evaluations. This work is partially supported by funding from Research Fellowships of the Japan Society for the Promotion of Science.

References

1) Cenzic, Inc.: Q3–Q4 2009 Trends Report on Web Security (2009). http://www.cenzic.com/downloads/Cenzic_AppSecTrends_Q3-Q4-2009.pdf
2) Chinotec Technologies Company: Paros. http://www.parosproxy.org/
3) Acunetix: Acunetix WVS. http://www.acunetix.com/
4) Hewlett-Packard: HP WebInspect software.
5) Huang, Y.-W., Huang, S.-K., Lin, T.-P. and Tsai, C.-H.: Web Application Security Assessment by Fault Injection and Behavior Monitoring, *Proc. Int'l Conf. World Wide Web*, pp.148–159 (2003).
6) Kals, S., Kirda, E., Kruegel, C. and Jovanovic, N.: SecuBat: A Web Vulnerability Scanner, *Proc. Int'l Conf. World Wide Web*, pp.247–256 (2006).
7) Watchfire: AppScan. http://www.watchfire.com/
8) Jovanovic, N., Kruegel, C. and Kirda, E.: Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper), *Proc. IEEE Sym. Security and Privacy*, pp.258–263 (2006).
9) Xie, Y. and Aiken, A.: Static Detection of Security Vulnerabilities in Scripting Languages, *Proc. Conf. USENIX Security Sym.*, pp.179–192 (2006).
10) Martin, M. and Lam, M.S.: Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking, *Proc. Conf. USENIX Security Sym.*, pp.31–43 (2008).
11) Wassermann, G. and Su, Z.: An Analysis Framework for Security in Web Applications, *Proc. FSE Workshop on Specification and Verification of Component-Based Systems*, pp.70–78 (2004).
12) Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C. and Vigna, G.: Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications, *Proc. IEEE Sym. Security and Privacy*, pp.387–401 (2008).
13) Bandhakavi, S., Bisht, P., Madhusudan, P. and Venkatakrishnan, V.N.: CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluations, *Proc. ACM Conf. Computer and Communications Security*, pp.12–24 (2007).
14) Buehrer, G., Weide, B.W. and Sivilotti, P.A.G.: Using Parse Tree Validation to Prevent SQL Injection Attacks, *Proc. Int'l Workshop on Software Engineering and Middleware*, pp.106–113 (2005).
15) Su, Z. and Wassermann, G.: The Essence of Command Injection Attacks in Web Applications, *Proc. ACM SIGPLAN-SIGACT Sym. Principles of Programming Languages*, pp.372–382 (2006).
16) Valeur, F., Mutz, D. and Vigna, G.: A Learning-Based Approach to the Detection of SQL Attacks, *Proc. Conf. Detection of Intrusions and Malware Vulnerability Assessment*, pp.123–140 (2005).
17) Boyd, S. and Keromytis, A.: SQLrand: Preventing SQL Injection Attacks, *Proc.*

Applied Cryptography and Network Security Conf., pp.292–304 (2004).
18) Halfond, W.G.J., Orso, A. and Manolios, P.: Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks, *Proc. ACM SIGSOFT Int'l Sym. Foundations of Software Engineering*, pp.175–185 (2006).
19) Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J. and Evans, D.: Automatically Hardening Web Applications using Precise Tainting, *Proc. IFIP Int'l Information Security Conf.*, pp.372–382 (2005).
20) Pietraszek, T. and Berghe, C.V.: Defending Against Injection Attacks through Context-Sensitive String Evaluation, *Proc. Recent Advances in Intrusion Detection*, pp.124–145 (2005).
21) Anley, C.: Advanced SQL Injection In SQL Server Applications, *An NGSSoftware Insight Security Research* (*NISR*) *Publication* (2002).
22) C.A. Mackay: SQL Injection Attacks and Some Tips on How to Prevent Them (2005). http://www.codeproject.com/cs/database/SqlInjectionAttacks.asp
23) Ferruh Mavituna: SQL Injection Cheat Sheet (2007). http://ferruh.mavituna.com/makale/sql-injection-cheatsheet/
24) Friedl, S.: SQL Injection Attacks by Example (2006). http://www.unixwiz.net/techtips/sql-injection.html
25) iMPERVA: Blind SQL Injection (2003). http://www.imperva.com/resources/adc/blind_sql_server_injection.html
26) OWASP: Testing for SQL Injection (2008). http://www.owasp.org/index.php/Testing_for_SQL_Injection
27) SecuriTeam: SQL Injection Walkthrough (2002). http://www.securiteam.com/securityreviews/5DP0N1P76E.html
28) Spett, K.: SQL Injection: Are your web applications vulnerable? (2004). http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf
29) Shiflett, C.: addslashes() Versus mysql_real_escape_string() (2006). http://shiflett.org/blog/2006/jan/addslashes-versus-mysql-real-escape-string
30) GotoCode.com: GotoCode. http://www.gotocode.com/
31) IX Knowledge Inc. http://www.ikic.co.jp/
32) KT: Top 15 free SQL Injection Scanners (2007). http://kalmah03.wordpress.com/2007/05/21/top-15-free-sql-scanner/
33) Gordon Lyon: Top 10 Web Vulnerability Scanners (2006). http://sectools.org/web-scanners.html
34) SourceForge.net. http://sourceforge.net
35) Apache Struts project: Struts. http://struts.apache.org/
36) XhockY: Schoorbs version 1.0.2 (2008). http://schoorbs.xhochy.com/

(Received May 7, 2010)
(Accepted August 28, 2010)



Yuji Kosuga received his B.E. and M.E. degrees from Keio University in 2007 and 2009, respectively. He is currently a Ph.D. student in the Graduate School of Science and Technology, Keio University. His research interests are in web security and system software. He is a student member of IPSJ.

Miyuki Hanaoka received her B.E. degree from the University of Electro-Communications in 2005, and M.E. from Keio University in 2007. Her research interests include network security and system software. She is a student member of IEEE, ACM, and IPSJ.

Kenji Kono received his B.Sc. degree in 1993, M.Sc. degree in 1995, and Ph.D. degree in 2000, all in computer science from the University of Tokyo. He is an associate professor of the Department of Information and Computer Science at Keio University. His research interests include operating systems, system software, and Internet security. He is a member of the IEEE/CS, ACM and USENIX.