2008－CSEC－41 (18)
2008／5／23

SQLインジェクション攻撃の脆弱性の効果的な自動検出手法

小菅 祐史　　花岡 美幸　　河野 健二

慶應義塾大学 理工学部 情報工学科

E-mail: yuji@sslab.ics.keio.ac.jp, hanayuki@sslab.ics.keio.ac.jp, kono@ics.keio.ac.jp

SQLインジェクション攻撃からWebアプリケーションを守る手段として，サニタイズは非常に有効である．しかし，サニタイズの処理を行うコードの挿入やサニタイズ忘れの検査は開発者によって手作業で行われるため，間違いや見落としが発生しやすい．我々の提案手法Saniaはリクエストから実際に生成されるSQLクエリを解析することによって，Webアプリケーションに正確な攻撃テストを行い，SQLインジェクション攻撃の脆弱性を検出する手法である．本論文では，Saniaを用いて，Webアプリケーションに潜むSQLインジェクション攻撃の脆弱性を検出できることを示す．

Effective Automated Testing for Detecting SQL Injection vulnerabilities

Yuji Kosuga　　Miyuki Hanaoka　　Kenji Kono

Department of Information and Computer Science, Keio University

E-mail: yamada@sslab.ics.keio.ac.jp, hanayuki@sslab.ics.keio.ac.jp, kono@ics.keio.ac.jp

Sanitization is a sufficient measure to prevent SQL injection attacks. However, implementing it is done by hand and results in error-prone. Our approach, Sania, is a technique that can detect SQL injection vulnerabilities by analyzing SQL queries generated from HTTP requests and performing effective penetration testings. In this paper, we demonstrate that Sania can discover the sanitization flaws in web applications.

1 はじめに

クライアントの要求に応じて動的にコンテンツを提供するWebアプリケーションの普及とともに，データベースに発行するSQLクエリを偽装したSQLインジェクション攻撃の被害が問題となっている．Armorize Technologies [2] によると，2006年に報告されたSQLインジェクション攻撃は376件で，Webアプリケーションの脆弱性の28%を占める．また，Mitre [14] の報告でも14 %を占めている．

SQLインジェクション攻撃対策として，クライアントの入力に含まれる危険な文字を無害化する方法がある．これはサニタイズと呼ばれる．例えば，データベースがusersテーブルにnameカラムとpasswordカラムを持ち，Webアプリケーションはユーザのログイン時に次のプログラムを使用して，SQLクエリを生成すると仮定する．

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

ここで，攻撃者がpasswordに"' or '1'='1"を挿入すると，次に示すSQLクエリが生成される．

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

この攻撃ではシングル・クオートが構文を不正に変化させるため，WHERE節が"or '1'='1'"により常に真と評価される．そのため，攻撃者はusersテーブルの全ての情報を取り出すことができてしまう．この攻撃へのサニタイズとしては，シングル・クオートにバック・スラッシュを付加することによって単なる文字として認識させるなどの方法などがある．

サニタイズを適切に行うことによって，SQLインジェクション攻撃を防ぐことができる．しかし，サニタイズの処理を行うコードの挿入やサニタイズ忘れの検査は，開発者によって手作業で行われる．そのため間違いや見落としが発生しやすい．

SQLインジェクション攻撃の脆弱性を自動検出する既存のツールは，テスト用の攻撃リクエストを基にWebアプリケーションが発行するレスポンスの内容を検証することによって実際に攻撃が成功するかを判断する．この手法では，たとえ攻撃が成功していなくても，Webアプリケーションが毎回内容が異なるレスポンスページを生成する場合，正確な検出を行うことができない．また，考えられる全ての攻撃を実行するので，多くの時間を要する．

我々の提案手法Saniaは，安全なリクエストとそのときに生成されるSQLクエリの文脈情報から効果的な攻撃を自動生成し，Webアプリケーションに対してテストを行う．SQLクエリの構文と文脈を解析することによって，効果的に攻撃の生成や脆弱性の有無の検証を行う．

実際に使用されている6つのWebアプリケーションに対して，実験を行ったところ，Saniaは脆弱性を39個検出し，false positiveを13個発生させた．

既存のツールとして評価の高いParosは，脆弱性を5個発見し，false positiveを67個引き起こしたことから，Saniaの手法が有効だと言える．また，出荷前の商用Webアプリケーションにテストを行ったところ，脆弱性を1個発見することができた．

以下，2章では関連研究について述べる．3章では本研究の提案手法Saniaの詳細について説明する．4章ではSaniaの実装を述べる．5章ではSaniaの実験と評価を示し，最後に6章で本論文をまとめる．

2 関連研究

SQLインジェクション攻撃の被害の拡大とともに，様々な対策が研究されている．本章ではそれらの研究について述べる．

フレームワークの利用 SQLで用意されている*prepare*文を使用することで，SQLインジェクション攻撃を防ぐことができる．prepare文はあらかじめ定義した型を持つ入力のみをSQLクエリに組み込むため，攻撃コードが入力されたとしても，実際に生成されるSQLクエリの構文は変化しない．しかし，既存のWebアプリケーションに使用するには，コードを記述し直す必要があり，コストがかかってしまう．

Struts [1] の*validator*は，クライアントの入力があらかじめ定義された規則に従っているかを検証する．しかし，この規則の記述は手作業で行われるため，ミスが混入しやすい．

静的解析 Wassermannら[17]は，静的解析と自動推論を組み合わせた手法で，SQLクエリに“1=1”のような常に真を返す式(トートロジー)が含まれないことを保証する．しかし，トートロジーを含まないSQLインジェクション攻撃を検出することができない．JDBC Checker [9]はクライアントの入力の型チェックを静的に行うことによって，脆弱性を検出する．しかし，多くの攻撃では構文や型が正しいため，検出できない脆弱性が多く存在する．

動的解析 Paros [5]はフリーの脆弱性検出ツールである．あらかじめ定義された攻撃コードをHTTPリクエストに埋め込み，Webアプリケーションに送信し，レスポンス内容を見ることによって，実際に攻撃が成功するかを判断する．5章でSaniaとの実験結果の比較に用いた．

静的解析と動的解析の組み合わせ SQLCheck [16]やAMNESIA [10]，SQLGuard [4]は，静的解析時に正しいSQLクエリのモデルを定義する．そして動的解析時に実際に生成されたSQLクエリとモデルを比較し，一致する場合のみ，データベースへの発行を許可する．これらの手法の検知能力は，静的解析時の学習の質に依存する．

SQLrand [3]は，標準のSQLキーワードを，ランダムな値で構成されたキーワードに置き換え，攻撃者から入力されたSQLキーワードと区別することで攻撃を防ぐ．しかし，ランダムなキーワードを標準のSQLキーワードに戻すときに使用するキーが攻撃者に見つかると攻撃を防ぐことはできない．

Pietraszekら[15]はユーザが入力した文字列を文字単位で追跡する手法を提案している．文脈に基いて解析を行うことにより，安全ではない入力がSQLキーワードとして使用されることを防ぐ，そのためSQLインジェクション攻撃を防ぐことができるものの，実行環境の修正が必要である．

3 Sania

本章では，SQLインジェクション攻撃の脆弱性を自動検出する手法Saniaを提案する．SaniaはWebアプリケーションが発行したSQLクエリの構文に従って適切な攻撃を生成し，検証を行う機構である．

3.1 Saniaの概要

Saniaは従来のテストツールとは異なり，HTTPリクエストだけでなくSQLクエリを取得し，次に示す3段階の処理を行うことで，サニタイズの正確さを検証する．

1. 安全なリクエストとそのときに生成されるSQLクエリの構文を用いて、攻撃ポイントの特定を行う．攻撃ポイントとは，攻撃者がSQLインジェクション攻撃を引き起こすような悪意のあるコードを埋め込むHTTPリクエスト内のパラメータである．
2. それぞれの攻撃ポイントに適した攻撃を自動生成する．SQLクエリ内に出現する攻撃ポイントの文脈を利用することによって，SQL文法に応じた攻撃生成を行う．
3. 攻撃をWebアプリケーションに送り，脆弱性の検証を行う．検証には，SQLGuard [4]で提案されたSQLクエリの構文木を比較する手法を用いる．

3.2 攻撃ポイントの特定

SQLインジェクション攻撃では，攻撃者は悪意ある文字列を攻撃ポイントに埋め込む．攻撃ポイントとは，SQLクエリに組み込まれるHTTPリクエスト内のクエリ・ストリングやクッキー等のパラメー

タの値であり，SQL クエリの構文木上では葉 (終端) ノードとして現れる．例えば，“=” で区切られたパラメータ要素と値要素を構成しているクエリ・ストリングでは，パラメータ要素はアプリケーションで定義された固定値であり，値要素にはクライアントの入力が埋め込まれる．例えばクエリ・ストリング “cat=book” では，“book” が攻撃ポイントである．

3.3 攻撃の生成

Sania は攻撃ポイントの現れる文脈に基づいて攻撃コードを生成する．攻撃コードは HTTP リクエストに埋め込まれ，実際に SQL インジェクション攻撃を引き起こす文字列である．さらに Sania は，単一攻撃と組合せ攻撃の 2 種類の攻撃を用いて，テストを行う．

3.3.1 単一攻撃

単一攻撃は一度に 1 つの攻撃ポイントにインジェクションを行う．例えば，Web アプリケーションが次の SQL クエリを発行すると仮定する．

```
SELECT * FROM users WHERE
  (name = 'xxx' AND (ø)) (øi: 攻撃ポイント)
```

さらにこのとき，Sania が攻撃ポイント ø への攻撃コードとして “yyy')) or 1=1--” を生成する場合を考える．

まず Sania は，この単一攻撃の攻撃コードを攻撃コード生成規則に基づいて動的に生成する．攻撃コード生成規則は次に示す 4 つの要素から構成される．

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

まず，metaCharacter はメタ・キャラクタ (シングル・クオート (') あるいはダブル・クオート (")) を表し，攻撃ポイントを userInput と insertedSQL に 2 分割する．前述の攻撃コードにおける userInput は “yyy” に相当し，クライアントの入力を模した文字列として使用される．insertedSQL は “ or 1=1--” 相当し，実際に攻撃を実行するような文字列が使用される．また，括弧の非対応によって SQL クエリの構文が壊れることによる無効な攻撃コードの生成を防ぐために，parentheses 要素を使用することによって 2 つの括弧を適切に埋め込んだ．

この攻撃コードを生成するための攻撃コード生成規則は次のように記述される．

```
(λ | ε,  //(λ): 入力文字列, (ε): 空白文字列
 true,   //使用の有無
 true,   //使用の有無
 or '1'='1 | or "1"="1 | or 1=1-- |
 or 1=1;-- | or 1=1/*)
```

生成規則のそれぞれの要素は論理和で表され，実際に生成される攻撃コードはそれぞれの要素を結合した文字列である．ただし，metaCharacter におけるメタ・キャラクタの種類と parentheses における括弧の数は，攻撃ポイントが現れる SQL クエリの文脈に基づいて決定する．

我々は SQL インジェクション攻撃に関する文献やメーリング・リストを調査した結果，95 種類の非終端ノードに対し，21 種類の攻撃コード生成規則が存在することを確認し，記述した．攻撃コード生成規則は XML で記述されているため，新たな攻撃の追加を容易に行うことができる．

3.3.2 組合せ攻撃

組合せ攻撃は一度に 2 つ以上の攻撃ポイントにインジェクションを行う．Sania は，2 つの攻撃ポイントが SQL 構文木上で隣接する文字列要素として出現するとき，組合せ攻撃を行う．1 つ目の攻撃ポイントにバック・スラッシュを挿入することによって文字列の終端を意味するクオートをエスケープし，2 つ目の攻撃ポイントに SQL キーワードを含む文字列を挿入する．

例えば，Web アプリケーションが次の SQL クエリを発行するとする．

```
SELECT * FROM users WHERE
  name='ø1' AND password='ø2' (øi: 攻撃ポイント)
```

このとき，Sania は 1 つ目の攻撃ポイント ($ø_1$) にバック・スラッシュ，2 つ目の攻撃ポイント ($ø_2$) に “ or 1=1--” を挿入する．バック・スラッシュが適切にサニタイズされていない場合，1 つ目の文字列の終端を示すシングル・クオートがバック・スラッシュによってエスケープされるため，name の値が “' AND password=” となる．その結果，where 節は “1=1” となり，常に真と評価される．このように，組合せ攻撃は 2 つの攻撃ポイントに対して動作する．

3.4 脆弱性の検知

Sania では，SQLGuard [4] と同じ手法を用いて脆弱性の検証を行う．この手法は，SQL インジェクション攻撃が成功すると，意図した木構造とは異なる構造を持つ SQL クエリが生成される性質を利用し，木構造の比較を行うことによって攻撃の成功を検出している．

しかし，この手法では常に攻撃の成功と失敗を正しく区別できるわけではない．ユーザの入力に応じて異なる木構造を持つ SQL を生成する Web アプリケーションも存在する．例えば，次の SQL クエリの id として任意の数値，または数式を許可する場

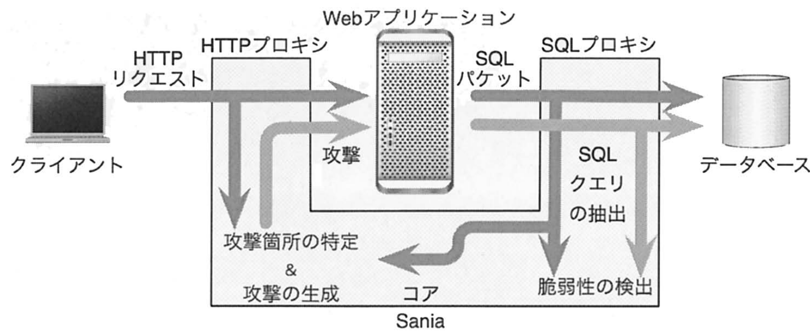

図 1: Sania の構造

合，数値は 1 つの葉要素として現れるのに対し，数式は 2 つの葉要素を子要素として持つ部分木として出現する．

```
SELECT * FROM users WHERE id=ø (ø: 攻撃ポイント)
```

このように，入力に応じて動的に構造が変化する SQL クエリに対応するため，Sania は攻撃ポイントに属性を付加することによって，任意の構文を持つことを許可する．前述の例では id に数値または数式を許可する属性を指定をすると，正しい構文の SQL クエリであると判断される．

4 実装

図 1 に Sania の実装を示す．Sania はプロキシ，SQL プロキシ，コア・モジュールで構成されている．プロキシと SQL プロキシはそれぞれ HTTP パケットと SQL パケットを取得する．コア・モジュールは前章で述べた処理を行う．

Sania は 13,000 行の Java プログラムと 1,800 行の XML で記述された攻撃コード生成規則で構成されている．SQL パーサは JSqlParser [13] に JavaCC [12] で改良を加えたものを使用した．

5 実験

5.1 実験方法

6 つの Web アプリケーション (表 1) を対象に実験を行った．5 つのアプリケーション (Bookstore, Portal, Event, Classifieds, Employee Directory) は GotoCode [8] の Web サイトからダウンロードした，フリーのアプリケーションであり，他の文献 [10, 16] においても実験環境として使用されている．我々はこれらのアプリケーションを基に作成された Web アプリケーションが実際に実運用されていることを確認している．残り 1 つのアプリケーション (E-learning) は，IX Knowledge Inc. [11] から提供を受けた JSP と Java Servlet で記述されたアプリケーションである．このアプリケーションは実際にイントラネットで使用されていたものである．表 1 にそれぞれの実験対象のアプリケーションの詳細を示す．

Sania の有効性を示すため，Insecure.Org [7] の Web アプリケーション脆弱性スキャナランキングで 2 位の Paros [5] との比較を行った．同ランキングでは Nikto [6] が首位であったが，Nikto は未知の SQL インジェクション攻撃の脆弱性を検出できないため，Sania との比較に用いるのは適切ではない．Paros では，すべてのパラメータを攻撃ポイントと考え，攻撃リクエストを生成して攻撃を行う．脆弱性の検出はレスポンスの内容を見ることによって行う．レスポンスの内容が変化した場合，攻撃が成功，つまり脆弱性が存在すると判断する．

5.2 結果

表 2 に Sania と Paros の実験結果を示す．全てのアプリケーションにおいて，Sania は Paros より少ない攻撃数で多くの脆弱性を検出し，少ない誤検知を発生したことが分かる．

5.2.1 攻撃の正確さ

Sania と Paros が検出した脆弱性の詳細を表 3 に示す．Sania と Paros は E-learning に単一攻撃に対する脆弱性があることを検出し，GotoCode のアプリケーションでは Sania だけが組合せ攻撃に対する脆弱性を検出している．単一攻撃だけを見ても，Sania は 14 個の脆弱性を検出したのに対し，Paros は 5 個である．これは，Sania が SQL の文脈に従った効果的な攻撃コードを生成することができるからである．また，組合せ攻撃に着目すると，Sania は 25 個の脆弱性を検出している．組合せ攻撃を行うためには，攻撃ポイントが現れる SQL クエリの場所や文脈が必要である．ゆえに，Paros は組合せ攻撃を行うことができず，Sania だけがこの脆弱性を検出できた．

5.2.2 False Positives

Sania と Paros の手法の違いにより，両者は異なる種類の false positive を発生させた．表 4 に false positive の詳細を示す．

表 1: 実験対象の Web アプリケーション

名称	概要	記述言語	LOC	攻撃ポイント
E-learning	Online Learning System	java(Servlet) & jsp	3682	14
Bookstore	Online Bookstore	jsp	11078	70
Portal	Portal for club	jsp	10051	96
Event	Event tracking system	jsp	4737	29
Classifieds	Online management system for classifieds	jsp	6540	35
Employee Directory	Online Employee Directory	jsp	3526	24

表 2: Sania と Paros の実験結果

名称	Sania				Paros			
	攻撃数	警告数	検知数	誤検知数	攻撃数	警告数	検知数	誤検知数
E-learning	18	18	18	0	362	9	5	4
Bookstore	616	7	6	1	4802	8	0	8
Portal	831	16	7	9	5477	15	0	15
Event	250	4	4	0	1698	21	0	21
Classifieds	279	5	3	2	1210	6	0	6
Employee Directory	194	2	1	1	1924	13	0	13
計	2188	52	39	13	15473	72	5	67

長さと型のエラー　攻撃コードの長さが不正であったり型が不適切であったために，データベースがエラーを返したため，Web アプリケーションのページ遷移が異なり，予期しない SQL クエリを生成する場合がある．Sania はこの時 SQL インジェクション攻撃の脆弱性があると判断する．Sania は不正な長さにより false positive を 8 個発生させた．Paros は長さが不正だった場合に 10 個，型が不正だった場合に 9 個の false positive を発生させた．

構文の破壊　不適切なインジェクションを行って構文を壊してしまうケースがあった．`Portal` では攻撃ポイントϕ_1 を持つ SQL クエリを発行した後，攻撃ポイントϕ_1 とϕ_2 を持つ SQL クエリを発行する．2 つ目の SQL クエリに攻撃を行うため，ϕ_1 にバック・スラッシュを挿入すると，1 つ目のϕ_1 にもバック・スラッシュが挿入され，SQL クエリの構文を壊してしまう．このようなケースにより Sania は false positive を 3 個発生させた．

重複データによる攻撃失敗　ある攻撃を試した際にデータベースにデータが追加される場合，次の攻撃の前にそのデータを消去する必要がある．もし，データを消去しないと，その後の検証結果に影響を与える場合がある．`Classifieds` では入力文字列がデータベースに存在するか否か確認を行い，存在しなければその文字列をデータベースに追加する．しかしデータが存在すればエラー扱いとなり，エラーページに遷移する．このエラーページが予期しない SQL クエリを発行すると Sania は警告を発生する．このような場合に Sania は false positive を 1 個発生させた．

認証の失敗　ユーザからの 2 つの入力が一致しなくてはならない場合がある．例えば，Web ページ内のパスワード入力欄とその確認用の入力欄では，両者の文字列が一致する必要がある．Sania はこの 2 つが同じ値を持たなくてはならないことを認識できないため，異なる値を挿入し，エラーページへと遷移する場合がある．その結果，予期しない SQL クエリを取得し，警告を発した．Sania ではこのような false positive を 1 個発生させた．

安全な箇所への攻撃　SQL クエリには組み込まれず，ページ遷移などに使用されるパラメータに攻撃コードを挿入すると，予期しないページ遷移をし，意図したレスポンスと異なるものが発行される．例えば，`Bookstore` では，パラメータ“`FormAction=insert`”は新しいデータを追加する際に使用され，追加が成功するとユーザを元のページへ誘導する．しかしこのパラメータの値が `insert` 以外だと他のページに遷移する．この結果，Paros は false positive を 16 個発生させた．

また，アプリケーションが攻撃を検知したときにレスポンスが変化した場合，Paros は攻撃成功と判断し，false positive を 13 個発生させた．

編集後のページでの誤検出　クライアントの入力を反映したレスポンスページを返す Web アプリケーションでは，Paros はレスポンス内容の変化を攻撃の成功と判断するため，常に攻撃が成功したと考える．そのため，実際に攻撃が成功していない場合

表 3: 検出した脆弱性の詳細

Tool	数		概要
Sania	39	14	単一攻撃 (E-learning で検出)
		25	組合せ攻撃
Paros	5	5	単一攻撃 (E-learning で検出)

表 4: False positive の詳細

Tool	数		概要
Sania	13	8	不正な長さの攻撃コード
		3	構文の破壊
		1	重複データによる攻撃失敗
		1	認証の失敗
Paros	67	16	State パラメータへの攻撃
		15	編集後のページでの誤検出
		13	アプリケーションによる検出
		10	不正な長さの攻撃コード
		9	不正な型の攻撃コード
		4	重複した警告

には false positive となる．これにより Paros は false positive を 15 個発生させた．Sania はレスポンスページの内容ではなく，SQL クエリの構文で検知を行うため，この false positive は発生しなかった．

重複した警告　Paros は Servlet に用いられる URL のエイリアスによるページ遷移を認識できない．例えば，E-learning において “`/security`” への最初のアクセスは “`/user/jsp/login.jsp`” へのアクセスと同等である．しかし，Paros はこれらのアクセスを異なるアクセスと考え，異なる脆弱性として報告した．このような false positive を 4 個発生させた．Sania は開発者の操作によって動くため，この false positive を避けることができる．

5.3 実製品でのテスト

我々は 2007 年 3 月 28 日に IX Knowledge Inc. [11] の開発した出荷前の Web アプリケーション RSS-Dripper の脆弱性検査を行った．RSS-Dripper はユーザの利用履歴に基づいて RSS 情報を提供する．Java Servlet と JSP で記述されており，Struts [1] 上で動作している．

Sania は 33 回の攻撃の後，1 個の脆弱性を検出した．この脆弱性は組合せ攻撃に対する脆弱性であった．実際に攻撃が成功することを確認した上で，サニタイズのアドバイスを行った．この結果から，近年開発された製品レベルのアプリケーションでも脆弱性が存在する場合があり，Sania がこのような実製品に対しても有効であることが確認できた．

6 まとめ

SQL インジェクション攻撃はサニタイズを適切に行うことによって防ぐことができる．しかし Web アプリケーションの開発現場ではサニタイズが手作業で行われているため，脆弱性が混入しやすい．本論文では Web アプリケーションがサニタイズを適切に行っているかを自動検出する手法 Sania を提案した．Sania は安全なリクエストとそのときに生成される SQL クエリの文脈情報から効果的な攻撃を自動生成し，Web アプリケーションに対してテストを行う．実験では，脆弱性検査ツール Paros との比較によって Sania の有効性を示した．

謝辞

本研究に関して様々なアドバイスや提案に加え，アプリケーションの提供をしていただいた IX Knowledge Inc. の高濱祐氏，菱山美穂氏，及び関係者の方々に深く感謝いたします．

参考文献

[1] Apache Struts project: Struts, http://struts.apache.org/.

[2] Armorize Technologies: Vulnerability Database, http://www.armorize.com/resources/vulnerability.php.

[3] Boyd, S. and Keromytis, A.: SQLrand: Preventing SQL Injection Attacks, *Proceedings of the 2nd Applied Cryptography and Network Security Conference (ACNS)*, pp. 292–304 (2004).

[4] Buehrer, G., Weide, B. W. and Sivilotti, P. A. G.: Using parse tree validation to prevent SQL injection attacks, *Proceedings of the 5th International Workshop on Software Engineering and Middleware (SEM)*, pp. 106–113 (2005).

[5] Chinotec Technologies Company: Paros, http://www.parosproxy.org/.

[6] CIRT.net: Nikto, http://www.cirt.net/code/nikto.shtml.

[7] Gordon Lyon: Top 10 Web Vulnerability Scanners, http://sectools.org/web-scanners.html (2006).

[8] GotoCode.com: GotoCode, http://www.gotocode.com/.

[9] Gould, C., Su, Z. and Devanbu, P.: JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications, *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pp. 697–698 (2004).

[10] Halfond, W. G. J. and Orso, A.: AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks, *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 174–183 (2005).

[11] IX Knowledge Inc.: http://www.ikic.co.jp/.

[12] java.net: JavaCC, https://javacc.dev.java.net/.

[13] Leonardo Francalanci: JSqlParser, http://jsqlparser.sourceforge.net/.

[14] MITRE: Common Vulnerabilities and Exposures (CVE), http://cve.mitre.org/.

[15] Pietraszek, T. and Berghe, C. V.: Defending Against Injection Attacks through Context-Sensitive String Evaluation, *Proceedings of Recent Advances in Intrusion Detection (RAID)*, pp. 124–145 (2005).

[16] Su, Z. and Wassermann, G.: The essence of command injection attacks in web applications, *Proceedings of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pp. 372–382 (2006).

[17] Wassermann, G. and Su, Z.: An Analysis Framework for Security in Web Applications, *Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, pp. 70–78 (2004).