社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

インジェクション系攻撃防止ライブラリの評価

大久保隆夫[†,††]　　田中　英彦[††]

† (株) 富士通研究所　〒 211–8588 川崎市中原区上小田中 4–1–1
†† 情報セキュリティ大学院大学　〒 221–0835 神奈川県横浜市神奈川区鶴屋町 2–14–1
E-mail: †okubo@jp.fujitsu.com, ††tanaka@iisec.ac.jp

あらまし　ソフトウェアの脆弱性の中で、入力の中にプログラムへの指令を挿入することにより攻撃を可能にするインジェクション脆弱性は大きな割合を占める。インジェクション脆弱性の解決は、その確認手段までを含めると既存のプログラミング技法やライブラリでは不十分であった。筆者らは対策の確認容易性を考慮した、インジェクション攻撃全般に適用可能なコーディング規約セットとライブラリを提案している。本報告では、提案したライブラリを Web アプリケーションのサンプルプログラムに適用し、他の既存手法と比較する形で評価を行った。また、評価によって抽出された課題に基づきライブラリの改善を行った。
キーワード　ソフトウェア, 脆弱性, インジェクション攻撃, コーディング規約, ライブラリ

Evaluation of a library against injection attacks

Takao OKUBO[†,††] and Hidehiko TANAKA[††]

† Fujitsu Laboratories ltd.　Kamikodanaka 4–1–1, Nakahara-ku, Kawasaki, 211–8588 Japan
†† Institute of Information Security　Tsuruyacho 2–14–1, Kanagawa-ku, Yokohama, 221–0835 Japan
E-mail: †okubo@jp.fujitsu.com, ††tanaka@iisec.ac.jp

Abstract　Injection vulnerabilities, those enable attacks done by injecting command string into input data, have big percentage of total software vulnerabilities. Existing Programming techniques and libraries are not sufficient to present complete solution and easy testing methods. We have proposed general convention set and a secure library ”SafeStatement” for preventing and testing injection attacks. In this papaer, the library are evalluated with some sample application programs, compared with other existing methods. And some improvement has done based on the results.
Key words　pLaTeX 2_ε class file, typesetting

1. はじめに

近年、ソフトウェアの脆弱性が問題となっている。IPA が発表した「情報セキュリティ白書」[IPA07] によれば、2006 年末までの Web アプリケーションの脆弱性の届出は累計 750 件 (受理 704 件) に及んでいる。クロスサイトスクリプティングや SQL インジェクションのような脆弱性は、世間にも周知されつつあり、これらの脆弱性を防止するためのプログラミング手法も Web や書籍などで紹介されている。それにも関わらず、上記白書における 2006 年の脆弱性届出のデータは、クロスサイトスクリプティング (43%) と SQL インジェクション (23%) が全体の 7 割近くを占めており、問題が解決されていないことを示している。また、最近では著名な Web アプリケーションの基盤ソフトである Apache Tomcat にもクロスサイトスクリプティング脆弱性が発見され、衝撃を与えた [CVE07]。

ソフトウェアの既知の脆弱性を防止する対策としては、大別して 2 種類あると考えられる。

- 開発者にある程度のセキュリティ知識があることを前提とした対策。脆弱性を起こさないためのプログラミングノウハウがこれに相当する。
- 開発者にセキュリティの知識がなくても、開発の課程で機械的に安全にできる仕組み。

筆者らは後者の対策について研究を行っている。理由は、(1) 前者は開発者にある程度のセキュリティ知識を要求するが、現状はすべての開発者にセキュリティ知識が浸透しているとはいえない。このことは、前述のセキュリティ白書において未だに脆弱性の報告が減少していない事実が示している。セキュリティ知識を浸透させるには、セキュリティ教育を行っていく地道な努力が必要になる。(2) 後者であれば、セキュリティの知識が浸透していない大規模なソフトウェア開発においても、コーディ

ング規約やライブラリによって制約を加えることで一定のセキュリティ品質を確保することが容易になる。また制約を加えることで、一般には困難なセキュリティの確認作業を制約の遵守の確認に単純化できる。筆者らは Web アプリケーションのインジェクション系攻撃 (SQL インジェクション、OS コマンドインジェクション、クロスサイトスクリプティング (XSS)) に着目し、その性質と可能な制約について検討を行い、インジェクションの防止と確認に適したコーディング規約とライブラリの提案を行った。本報告では、提案したライブラリ Web アプリケーションのサンプルプログラムに適用し、他の既存手法と比較する形で評価を行った。また、評価によって抽出された課題に基づきライブラリの改善を行った。以降、2. 節ではインジェクション攻撃の防止に関する関連研究について、3. 節では筆者らの提案方式について述べる。4. 節では提案方式で提案したライブラリを SQL インジェクション対策として実装したものを、アプリケーションサンプルを用いて評価した結果について述べる。5. 節では評価で明らかになった課題と、それに基づきライブラリに行った改善について述べ、6. 節で結論を述べる。

2. 関 連 研 究

インジェクション系攻撃のうち SQL インジェクションについては、近年では様々な視点からの検出、防御手法が提案されている。

（1） 入力値を検査し、SQL において攻撃に悪用される危険な文字 (引用符など) を安全な文字にエスケープする方法
個別の入力値に対してプログラマが行う必要がある。入力値の対処漏れ、記号の対処漏れなどを原因とする脆弱性を生じやすい。

（2） Serialization API を用いる方法
動的なパラメータを埋めこむ位置があらかじめ指定された固定のテンプレートのクエリ文字列を事前に準備する。動的な値は、定められた位置のパラメータに値をバインドすることによって行う。Java の PreparedStatement、Perl の DBI、PHP,Python の ADOldb、VisualBASIC の SQLCommand などがこれに相当する。

（3） ブラックリスト、またはホワイトリストによる異常な SQL クエリの排除
リストに登録されたクエリパターンに基づき、疑わしいパターンを遮断する方式である。遮断に用いるリストには、ブラックリストとホワイトリストの 2 通りがある。また、検査手法については、SQLBlock のようにデータベースの直前でクエリそのものを検査する手法と、Web アプリケーションファイアウォールのように HTTP リクエストのパラメータを検査する手法とがある。ブラックリスト、ホワイトリストいずれの手法についても、クエリの種類が少なければ効果を発揮するが、クエリが複雑化すると、禁止パターンと正常パターンの衝突が発生し、制限を厳密にすると正常なクエリが拒否されたり、逆に制限を緩めると攻撃のパターンを検出できなくなってしまう可能性がある。また、プログラムの変更によりクエリのパターンが増える場合はリストを変更する必要がある。

（4） プログラムで利用されるクエリの構文を解析し、許容されるクエリの構文を定義する
この手法の代表的な例としては AMNESIA [HO06] がある。AMNESIA は、ソースプログラムを静的に解析し、プログラムが生成する可能性のある SQL クエリの構文を非決定性有限オートマトン (NDFA) としてすべて抽出し構文モデルとして保存しておく。そしてプログラム実行時に実行されるクエリを監視し、構文モデルから外れるものを検出する。この手法は、前述のホワイトリスト、ボラックリストよりも精度は高く、特に正常なクエリが拒否される可能性は低い。しかし、静的解析が持つ曖昧さが原因で、抽出された構文パターンが攻撃可能なクエリパターンを排除できない場合がある。

上記に挙げた手法の中では、Java の PreparedStatement などの Serialization API を使う方法が広範に普及しており、セキュアプログラミングのガイドラインでもこの API の利用が推奨されているものが多い。しかし、Serialization API には次に挙げる問題がある。

- パラメータのバインド機構を利用しないプログラミングが脆弱性を生じる

Java の PreparedStatement のクエリ文字列を構築する際、入力値などをパラメータ指定せずに通常のクエリとして構築すると、パラメータのバインド時の無害化処理が機能せず、入力値の操作により SQL インジェクション攻撃が可能になってしまう。このような脆弱性を生じる Java プログラムの例を Source code1 に示す。

```
1   String query = "SELECT␣*␣FROM␣user_info␣WHERE␣
        userid=\""␣+␣userid␣+␣"\"␣AND␣password=\""␣+␣
        password␣+␣"\"";
    PreparedStatement pstmnt = con.prepareStatement(
        query);
    ResultSet rs = pstmnt.executeQuery();
```

Source Code 1　バインド機構を利用しないプログラム

- クエリが動的に変化する場合、静的なテンプレートを用意するのが困難になる

条件分岐、ループによって発行するクエリが実行ごとに動的に変化する場合。分岐やループの組み合わせの数だけ、静的なクエリのテンプレートを用意しなければならず、コーディング量が増加し、保守性低下、バグ発生率の増加を招く。下記の Java プログラム (Source code1) はキーワード検索の例であるが、キーの種類が 6 種類だったとすると、1 から 5 までのキーの組み合わせ総数 62 通りの SQL クエリテンプレートを用意する必要がある。

上記で述べたように、現在までに提案されている SQL インジェクション対策では、検出精度、対応可能なアプリケーションの網羅性などで問題がある。

3. 提 案 方 式

筆者らは、セキュリティの制約と確認容易性の関連に着目し、コーディング規約とライブラリの両面から制約をかけることを

```
Statement stmnt = con.createStatement();
String query = "SELECT␣*␣FROM␣user_info␣";
Set keyset = map.keySet();
Iterator iter = keyset.iterator();
boolean first = true;
while (iter.hasNext()) {
        String key =(String) iter.next();
        String[] value = (String[]) map.get(key);
        if(value[0] != ""){
                if(!first) {
                        query+= "␣OR␣";
                } else {
                        query+= "␣WHERE␣";
                }
                query+= key + "=\""␣+␣value[0]␣+␣"\"";
                first = false;
        }
}
ResultSet rs = stmnt.executeQuery(query);
```

Source Code 2　クエリが動的に変化するプログラム

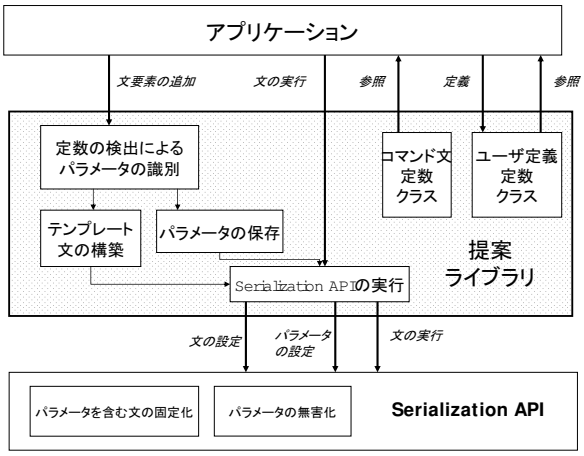

Figure 1　提案ライブラリの構成

検討し、インジェクション攻撃の防止、確認に有効な規約、フレームワークについて提案を行った。以下ではその概要について述べる。

インジェクション攻撃とは、サーバプログラムがクライアントから入力されたデータに基づいて、外部プログラムに何かのコマンドを送信する際に、攻撃者が入力データにコマンドの一部を挿入することにより、外部プログラムに意図しない動作をさせる攻撃をいう。外部プログラムがリレーショナルデータベースの場合、コマンドは SQL であり、それを利用した攻撃は SQL インジェション攻撃と呼ばれる。同様に、OS コマンドインジェクション攻撃では外部プログラムが OS で、コマンドが OS コマンドに相当する。一般には別の攻撃に分類されるクロスサイトスクリプティング (XSS) 攻撃も、外部プログラムを HTML レンダラ (ブラウザ)、コマンドを HTML と考えれば、本質的な構造はインジェクション攻撃と同じである。SQL インジェクションの典型的な例を以下に示す。サーバプログラムが、認証時にデータベースに問い合わせを行い、ユーザ ID, パスワードに一致するデータが存在すればそれを返すことを想定して構築されたクエリである。。

```
SELECT * FROM utable WHERE userid="(ユーザ ID)"
 AND password="(パスワード)"
```

上記の () で括った部分には入力データが入る。ここで、攻撃者が" or "a"="a という文字列をパスワードとして入力させると、サーバプログラムが発行するクエリは以下のように変更されてしまう。

```
SELECT * FROM utable WHERE userid="dummy"
 AND password="" OR "a"="a"
```

上記のクエリは、条件が常に真になるため、データベースの utable 上の任意のユーザにマッチしてしまい、認証を回避してログインすることが可能になってしまう。

筆者らは、インジェクション攻撃の本質は「ユーザ入力によってコマンドの構文が変えられてしまうこと」と定義し、それに従って攻撃を防止するための要件を「入力値によって、元のコマンドの構文が変えられないこと」と規定した。次に、要件を満たす制約の中から、確認が最も容易と考えられる次の制約を規約案とした。

- 外部プログラムに渡す引数の文字列は、定数 (事前に定義され、変更がない) のみで構成されていなければならない。

次に、既存のライブラリについて、規約を遵守する場合のアプリケーションの実現可能性という観点から検討した。既存のアプリケーションについて調査した結果、クエリのテンプレートとバインド機構を用いた Serialization API を用いれば、大部分のアプリケーションは規約に遵守する形で構築できることが分かった。しかし、前節で述べたように、Serialization API では対応が困難なアプリケーション (Source Code1、Source Code2 等のパターン) が存在することも分かった。

そこで筆者らは、Serialization API で対応が困難なアプリケーションでも上記規約を満たすような実装を可能とするようなライブラリを提案した。提案ライブラリは以下の特徴を持つ。

- Serialization API を内部で利用するが、固定のテンプレートを事前に定義するのではなく、実行ごとに動的に生成する。
- コマンド文字列の構成時に、構成要素が静的要素か、パラメータ要素かの区別を、定数として定義されているかを識別することによって行なう。

提案ライブラリを SQL インジェクション向けに、Java で実装したものの構成を Figure 1 に示す。提案ライブラリは、文字列追加の API と、コマンド文実行の API を持つ。文字列追加の引数には、通常の文字列クラスか、ライブラリによって許可された型の定数しか利用できない。許可されている定数は、予め定義されている SQL 文の構文要素か、プログラマが独自に追加できるユーザ定数のいずれかである。各 API はそれぞれ下記のように動作する。

（1） 文字列追加 API は、追加する文字列の型を検査し、定数であればクエリの文字列にそのまま追加する。定数でない文字列はパラメータとみなし、パラメータのリストに保存し、クエリの文字列にはパラメータが挿入される位置を示す'?' を挿入する。

（2） コマンド文実行 API は蓄積されているクエリ文字

列をもとに PreparedStatement を生成し、別に保存されているパラメータを順に PreparedStatement にバインドし、`PreparedStatement#executeQuery()` を実行する。

4. 提案ライブラリの評価

提案したライブラリの SQL インジェクション向け実装を Java を用いて試作し、サンプルの Web アプリケーションによる動作を検証、評価を行った。評価の目的は、筆者らの提案するライブラリが他の実装方式と比較して、SQL インジェクションの防止、検出の点でより有効であるかを検証することである。比較対象とする実装方式は、(1)Statement クラスによる実装、(2)PreparedStatement による実装、(3) 本提案ライブラリによる実装、(4)AMNESIA による実装の 4 種類である。ただし、AMNESIA のみは今回入手できなかったため、机上による検証のみ行った。

動作検証に用いた環境は以下の通りである。

- JavaVM:JDK1.5.11
- Servlet コンテナ:Tomcat5.5.23
- データベース:MYSQL 5.0.41

4.1 認証機能を持つアプリケーションを用いた検証

ユーザ ID とパスワードを入力させ、ログインする機能を持つアプリケーションを各実装方法により実装後動作させ、Figure 2 に示すブラウザ画面からパスワード入力欄に SQL インジェクション攻撃を行う文字列を実際に入力して、攻撃が成功するかどうかの検証を行った。

検証に用いた攻撃パターンは以下の 3 通りである。攻撃が成功したかどうかの確認方法はそれぞれの場合で異なる。

（1） 秘密の属性による検索

攻撃手法：パスワード欄に「” or sex=”female」と入力

攻撃の確認方法：ユーザ情報の中で、sex が female に該当するユーザのみが表示される。

（2） 秘密データの検索

攻撃手法：パスワード欄に「”;SELECT * from admininfo WHERE ”a”=”a」と入力

攻撃の確認方法：管理者テーブル (admininfo) にある管理者情報がすべて表示される。

（3） データの破壊

攻撃手法：パスワード欄に「”;DELETE * FROM userinfo WHERE ”a”=”a」と入力

攻撃の確認方法：実行後、データベース上のユーザ情報テーブル (userinfo) のデータがすべて消去される

（1） Statement クラスによる実装

Statement クラスによる実装コードの一部を Source Code3 以下に示す。Source Code3 の動作検証では、攻撃パターン 1〜3

```
1 Statement stmnt = con.createStatement();
  String query = "SELECT␣*␣FROM␣userinfo␣WHERE␣userid=\""+␣
      userid␣+␣"\"␣AND␣password=\""␣+␣password␣+␣"\"";
  ResultSet rs = stmnt.executeQuery(query);
```

Source Code 3　Statement クラスによる実装

Figure 2　サンプルアプリケーションのログイン画面

による攻撃はすべて成功したことが確認された。

（2） PreparedStatement による正しい実装

PreparedStatement クラスのパラメータのバインド機構を正しく使い、実装したコードの一部を Source Code4 に示す。Source

```
String query = "SELECT␣*␣from␣userinfo␣WHERE␣userid=?␣AND
    ␣password=?";
2 PreparedStatement pstmnt = con.prepareStatement(query);
  pstmnt.setString(1, userid);
  pstmnt.setString(2, password);
  ResultSet rs = pstmnt.executeQuery();
```

Source Code 4　PreapredStatement による実装 (1)

Code4 の動作検証では、攻撃パターン 1〜3 による攻撃は失敗した。これは、パラメータとして入力されるデータは単なる文字列として扱われるため、攻撃文字列はクエリの一部としては機能しないためである。

（3） PreparedStatement による誤った実装

PreparedStatement クラスを用いているものの、パラメータのバインド機構を使っていない、誤った実装コードを用いて検証を行う。用いたサンプルは、前述の Source Code1 と同じものである。

動作検証では、攻撃パターン (1)〜(3) による攻撃は成功が確認された。PreparedStatement が SQL インジェクション防止として機能しない理由は、このコードではパラメータのバインド機能を使っていないために無害化機能が働かないことと、テンプレートとなる SQL のクエリの構文が、入力値による変更を許してしまっているためである。

（4） 提案ライブラリによる正しい実装

提案ライブラリを用い、静的要素は定数を用いて正しく実装したコードの一部を Source Code5 に示す。Source Code5 で、”SELECT * FROM userinfo WHERE userid=”という文字列が MYQUERY1、” AND password=”という文字列が MYQUERY2 という、ユーザ定義の定数として定義されており、提案ライブラリでは、SafeStatement クラスへの静的要素の追加はこの定数クラスを用いて行われている。それ以外の、userid やパスワードなどの動的要素はそのまま追加されてい

```
SafeStatement sstmnt = new SafeStatement();
sstmnt.add(UserConstValue.MYQUERY1); //SELECT * FROM
    userinfo WHERE userid=
sstmnt.add(userid);
sstmnt.add(UserConstValue.MYQUERY2); // AND password=
sstmnt.add(password);
ResultSet rs = sstmnt.prepareAndExecute(con);
```

Source Code 5　提案ライブラリによる実装 (1)

る。このコードの動作検証では、攻撃パターン (1)〜(3) による攻撃は失敗した。SafeStatement クラスが、追加要素の型の識別によって自動的に静的要素とパラメータに振り分けて PreparedStatement に渡しているためである。

（5） 提案ライブラリによる誤った実装

提案ライブラリを用いているものの、静的要素に定数を用いていない誤った実装コードの一部を Source Code6 に示す。Source Code6 では、本来定数として追加されるべき”SELECT

```
SafeStatement sstmnt = new SafeStatement();
sstmnt.add("SELECT␣*␣FROM␣user_info␣WHERE␣userid=");
sstmnt.add(userid);
sstmnt.add("␣AND␣password=");
sstmnt.add(password);
ResultSet rs = sstmnt.prepareAndExecute(con);
```

Source Code 6　提案ライブラリによる実装 (2)

* FROM userinfo WHERE userid=”、および” AND password=”が、文字列リテラルとして追加されている。このコードの動作検証では、PreparedStatement の場合と異なり、攻撃は失敗した。この場合、すべてが動的要素とみなされるため、SafeStatement クラスが構築するクエリ文字列は攻撃パターン (1) では”？？？？”となる。これが PreparedStatement のクエリとして文法的に正しくないため、JDBC コネクタによるシンタックスエラー (SQLException 例外) が発生し、認証処理は成功しないのである。

（6） AMNESIA

Statement クラスによる実装コード (3 を AMNESIA で解析した結果、得られる SQL 構文モデルは Figure 3 に示す通りになる。図中の”β”記号は、その要素がプログラム外部から来ていることを示す。検証に用いる攻撃パターン (1)(認証回避) によって生成されるクエリは、”SELECT * FROM ustable WHERE userid=”dummy” OR ”a”=”a”” となり、その構文は Figure 3 のモデルを逸脱しているため、攻撃の検出が可能と考えられる。。

4.2　動的なクエリ構築を行うアプリケーションを用いた検証

次に、動的なクエリ構築を伴うアプリケーションによる検証を行う。用いるサンプルは、ブラウザで 5 種類の検索キーワードから任意の数のキーワードを選んで検索するアプリケーションである。ブラウザの入力画面を Figure 4 に示す。検証に用いた攻撃パターンは以下の 3 通りである。攻撃が成功したかどうかの確認方法はそれぞれの場合で異なる。

（1） 認証回避

攻撃手法：パスワード欄に「” or ”a”=”a」と入力

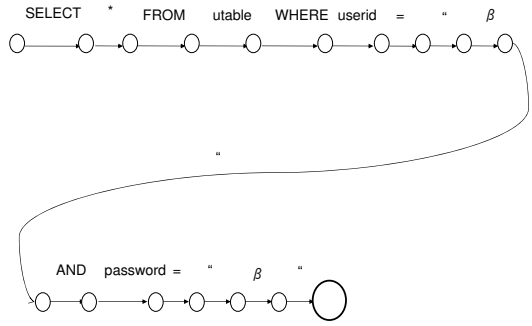

Figure 3　AMNESIA が生成する SQL 構文モデル

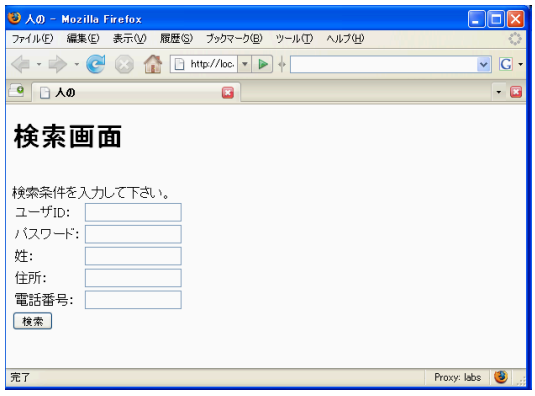

Figure 4　サンプルアプリケーションのブラウザ画面

攻撃の確認方法：不正なパスワードで認証が回避され、ログイン後の画面が表示される。

（2） 認証回避 (2)

攻撃手法：パスワード欄に「”;SELECT * from admininfo WHERE ”a”=”a」と入力

攻撃の確認方法：管理者テーブル (admininfo) にある管理者で認証に成功し、ログイン後の画面が表示される。

（3） データの破壊

攻撃手法：パスワード欄に「”;DELETE * FROM userinfo WHERE ”a”=”a」と入力

攻撃の確認方法：実行後、データベース上のユーザ情報テーブル (userinfo) のデータがすべて消去される

（1） Statement クラスによる実装

Statement クラスによる実装コードは前述の Source Code2 と同じものを用いている。HTTP リクエストパラメータのキーの数だけクエリの条件が for ループによって追加される構造になっているため、実行ごとにクエリの構文が変化することが分かる。

動作検証では、攻撃パターン (1)〜(3) による攻撃は成功し、テーブルのすべてのデータが表示された。

（2） PreparedStatement による実装

PreparedStatement による実装を行う場合、「クエリが定数の

みから構成される」という制約条件を満たすためには、まず、5 種類のキーワードの 1〜5 組の組み合わせのため、31 通りのテンプレートのクエリを用意する必要がある。次に、動作時のキーワードの数と種類を判定し、上記制約条件を満たした実装を行い、動作検証を行えば攻撃を防止することは不可能ではない。しかし、それを実現するためには著しくコード量が増加することになる。また一方、制約条件を満たさなければ、動的にクエリを構築する実装を行うことも可能である。しかしその場合、脆弱性を防止できているかどうかの確認が困難になってしまう。確認を容易に行うためには、「入力値によって、元のコマンドの構文が変えられないこと」という要件を満たす別の制約が必要になる。

（3） 提案ライブラリによる実装

提案ライブラリを用いたコードの一部を Source Code7 に示す。Source Code7 では、Statement クラスによる実装 (Source Code2) と同様に、for ループを用いて動的にクエリを構築している。Statement との違いは、add() で静的要素の場合は定数を用いていることのにである。コード行数も、Source Code2 の 19 行に対し、27 行と、コード的にも大きな差は見られない。動作検証では、攻撃パターン (1)〜(3) による攻撃は失敗したことが確認された。

```
SafeStatement sstmnt = new SafeStatement();
sstmnt.add(UserConstValue.SELECT);
Set keyset = map.keySet();
Iterator iter = keyset.iterator();
boolean first = true;
while (iter.hasNext()) {
       String key =(String) iter.next();
       String[] value = (String[]) map.get(key);
       if(value[0] != ""){
              if(!first) {
                     sstmnt.add(UserConstValue.OR);
              } else {
                     sstmnt.add(UserConstValue.WHERE);
              }
              //query+= key + "=\"" + value[0] + "\"";
              UserConstValue constKey = findKey(key);
              if(constKey != null) {
                     sstmnt.add(constKey);
              } else {
                     sstmnt.add(key);
              }
              sstmnt.add(UserConstValue.EQ);
              sstmnt.add(value[0]);
              first = false;
       }
}
ResultSet rs = sstmnt.prepareAndExecute(con);
```

Source Code 7　提案ライブラリによる実装

（4） AMNESIA

AMNESIA はソースコードを静的に解析するため、Source Code2 のループ回数の定まらないループでは、有限のオートマトンとして記述することができない。また、仮にループの最大値が 5 個と決定していた場合、その構文モデルは Figure 5 に示す通りになるが、このモデルは動作検証で用いている攻撃 1 によって構築されるクエリパターン”SELECT * FROM utable WHERE userid=”” OR sex=”female”を許容する。従って AMNESIA ではこの攻撃は排除できないと考えられる。

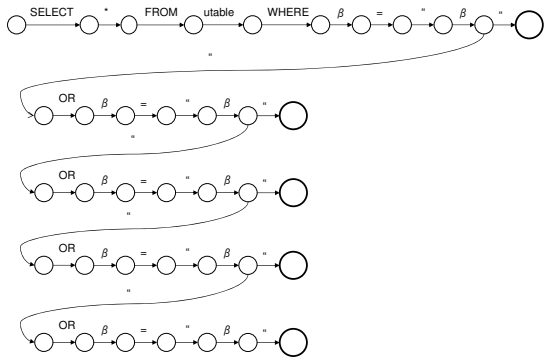

Figure 5　AMNESIA が生成する SQL 構文モデル

4.3 評 価 結 果

動作検証による比較評価結果をまとめたものを Table 1 に示す。表で、○は用意した攻撃データに対しては、攻撃の検出または防止が可能であったことを、△は防止する実装を行うことが困難であることを、×は用意した攻撃データのうち少なくとも一つで攻撃の検出、防止ができない例があったことを示す。PreparedStatement はバインド機構を用いない誤った実装を行った場合に、SQL インジェクションの防止ができなかった。また、動的にクエリを構築する場合では AMNESIA では攻撃パターンの検出ができず、PreparedStatement のみでの実装は不可能ではないがクエリのパターン数に比例してコーディング量が増加するため、実装コストや保守性などの面で問題がある。一方、提案ライブラリを用いた手法では、いずれの場合でも検出または防止が可能であり、比較対象とした既存の SQL インジェクション対策手法のいずれに対しても、対象としたアプリケーションのパターンでは攻撃防止に関してより有効であるという結果になった。

5. 課題とライブラリの改善

5.1 課　　題

今回提案ライブラリを用いて実装を行ってみて、プログラミングの観点からいくつか課題があることが分かった。

（1） リクエストパラメータの key の扱い

Source Code7 では SQL クエリの構築に HTTP リクエストパラメータが用いられている。このアプリケーションではリクエストパラメーターの key には固定の値が使われているが、そのままでは提案ライブラリに追加しても定数とはみなされないため、PreparedStatement においてパラメータ扱いになってしまう。従って提案ライブラリで用いるためには、このパラメータの値を定数の値と比較し、一致していたら定数に変換する必要がある。正しく実装すれば問題はないのだが、単純な定数、非定数の使い分けあけではなく、コーディングに工夫が必要になる。リクエストパラメータの扱いについては改善の余地があると考えられる。

アプリケーション	Statement	PreparedStatement	提案ライブラリ	AMNESIA
正しい実装	×	○	○	○
誤った実装	−	×	○	−
動的なクエリ構築	×	△	○	×

Table 1　評 価 結 果

（2） 誤った実装の場合の検出が不完全

提案ライブラリで誤った実装 (静的要素を定数として追加しなかった) の場合でも、JDBC でシンタックスエラーになる場合とならない場合がある。Source Code6 の場合はクエリ文字列が”? ? ? ?”となりシンタックスエラーになったが、5 で最初の文字列のみ定数を用いた場合、クエリ文字列は”SELECT * FROM utable WHERE userid=? ? ?”となるが、この場合の実行では MySQL の JDBC はシンタックスエラーにならない。ただし、SQL インジェクション攻撃自体は成功しない。このように、プログラミングの誤り方によっては、うまくエラーが検出できない場合もある。この対策としては、デバッグモードで生成クエリの文法チェックを行い、”SELECT * FROM utable WHERE userid=? ? ?”のようなパターンを許容しないようにすればプログラミングの誤りの検出精度は挙がると考えられる。

（3） ライブラリ定義の定数はほぼ不要

ライブラリが提供する定数は要素単位のため、それを繋げたクエリの文字列を構築するには要素数分の add() 処理を行う必要があり、非常にコーディングが煩雑になる。実際には、ユーザ定義の定数だけを使っていれば十分であることが分かった。

（4） 定数を変更した場合、再コンパイルが必要データベースのテーブル名に変更があった場合など、ユーザ定義の定数を変更しなければならない。現在のライブラリの実装では、ユーザ定義変数は Java 5 以降の enum を用いて定義されている。これを変更する場合は Java クラス中の文字列を変更しなければならないため、再コンパイルが必要になる。ただし、この問題は提案ライブラリに特有ではなく、SQL 呼び出しを行うアプリケーション一般について言えることである。

5.2 ライブラリの改善

上記に挙げた課題のうち、最後の課題についてユーザ定義定数クラスを改善し、定数の初期化の際に properties ファイルから値を読みこむようにした。これより、値が変わっても、再コンパイルの必要はなくなった。

ただし、ここで注意しなければならないのは、改善されたユーザ定義定数クラスサーバ側のファイルやクラスに対する攻撃がないという前提条件が必要であるという点である。なんらかの手段で properties ファイルの内容が書きかえられてしまった場合、ファイルに攻撃文字列を挿入することで SQL インジェクションが可能になる恐れがある。また、ユーザ定義定数クラスを攻撃者によって置きかえられてしまうと、やはりインジェクション攻撃が可能になる。

6. おわりに

筆者らは確認容易性に着目したインジェクション防止のためのコーディング規約とライブラリを提案しているが、本報告ではその SQL インジェクション向け実装について、他の既存の手法と比較し、評価を行った。特に他の手法では検出や防止が困難なアプリケーションについて、本提案ライブラリが有効であることが確認できた。

一方、プログラミング容易性については、改善すべきいくつかの課題があることが明らかになった。また、今回のサンプルアプリケーションは筆者らが設計、実装したものであり、プログラミングのし易さ、コーディングの自由度としては第三者的な評価が必要であると考える。また、提案方式の性能的な評価も今後の課題である。

References

[CVE07] National Vulnerability Database, common Vulnerabilities and Exposures, CVE–2007–1355, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-1355(2007).

[HO06] William G. J. Halfond, Alessandro Orso, Preventing SQL injection attacks using AMNESIA, Proceeding of the 28th international conference on Software engineering,pp.795-798(2006).

[IPA07] 情報処理推進機構, 情報セキュリティ白書 2007 年度版, http://www.ipa.go.jp/security/vuln/20070309_ISwhitepaper.html(2007).

[OT07] Takao Okubo, Hidehiko Tanaka, Secure Sofoware Development through Coding Conventions and Frameworks, The Second International Conference on Availability, Reliability and Security (ARES'07), pp.1042-1051(2007).

[SQLB07] SQLBlock, http://www.sqlblock.com/(2007).