安全なSQLの呼び出し方

「安全なウェブサイトの作り方」別冊

IPA® 独立行政法人 情報処理推進機構
セキュリティセンター

2010 年 3 月

本書は、以下の URL からダウンロードできます。

「安全な SQL の呼び出し方」

http://www.ipa.go.jp/security/vuln/websecurity.html

目次

目次 2
はじめに 3
1. 本書の位置づけ 4
2. リテラルとSQLインジェクション 5
2.1. SQL文の構造 5
2.2. リテラルとは 5
2.3. 文字列リテラルのエスケープ 6
2.4. 数値リテラル 6
2.5. SQLインジェクションの原因 7
2.5.1. 文字列リテラルに対するSQLインジェクション 7
2.5.2. 数値リテラルに対するSQLインジェクション 8
2.5.3. 安全なSQL呼び出しの要件 8
3. SQLの呼び出し方 9
3.1. 文字列連結によるSQL文の組み立て 9
3.2. プレースホルダによるSQL文の組み立て 9
3.2.1. 静的プレースホルダ 10
3.2.2. 動的プレースホルダ 11
3.3. まとめ 12
4. データベースと連動したSQL文生成 13
4.1. 文字列連結にquoteメソッドを使う 13
4.2. データベースと連動した動的プレースホルダ 14
5. DBMS製品の実態調査 16
5.1. 調査内容 16
5.2. Java + Oracle 16
5.2.1. サンプルコード 17
5.3. PHP + PostgreSQL 18
5.3.1. サンプルコード 19
5.4. Perl + MySQL 21
5.4.1. プレースホルダの実装 21
5.4.2. エスケープ対象となる文字の扱い 21
5.4.3. quoteメソッドにおける数値リテラルの扱い 21
5.4.4. 文字エンコーディングの扱い 22
5.4.5. サンプルコード 23
5.5. Java + MySQL 25
5.5.1. プレースホルダの実装 25
5.5.2. エスケープ対象となる文字の扱い 25
5.5.3. 文字エンコーディングの扱い 25

5.5.4. サンプルコード 26
5.6. ASP.NET + Microsoft SQL Server 27
5.6.1. サンプルコード 28
5.7. まとめ 30
付録A. 技術情報 31
A.1. バックスラッシュのエスケープ 31
A.2. Shift_JISによるSQLインジェクション 32
A.3. UnicodeによるSQLインジェクション 33
A.4. Oracleのデータベースを Unicodeで作成する方法 35
A.5. Microsoft SQL Serverと文字コード指定 36

はじめに

2005 年以来、SQL インジェクション攻撃による情報漏洩やサイト改ざんの事件が頻発するようになりました。「安全なウェブサイトの作り方」では 9 種類の脆弱性を取り上げていますが、その中でも SQL インジェクションは影響度の大きさや被害の多さの点から、特に対策を急がなければならない脆弱性です。

一方、ここ数年の間に、ウェブアプリケーション開発の分野でオープンソースソフトウェアの活用が盛んになりました。しかしながら、そうしたソフトウェア製品の一部には、十分に安全性が検証されることなく使用されているものもあり、プログラミング言語やデータベースエンジンの種類によっては、製品固有の仕様を悪用されたり、文字エンコーディングの問題によって、一般的に知られている SQL インジェクション対策を施していても、隙を突いて攻撃されかねない場合があることがわかってきました。

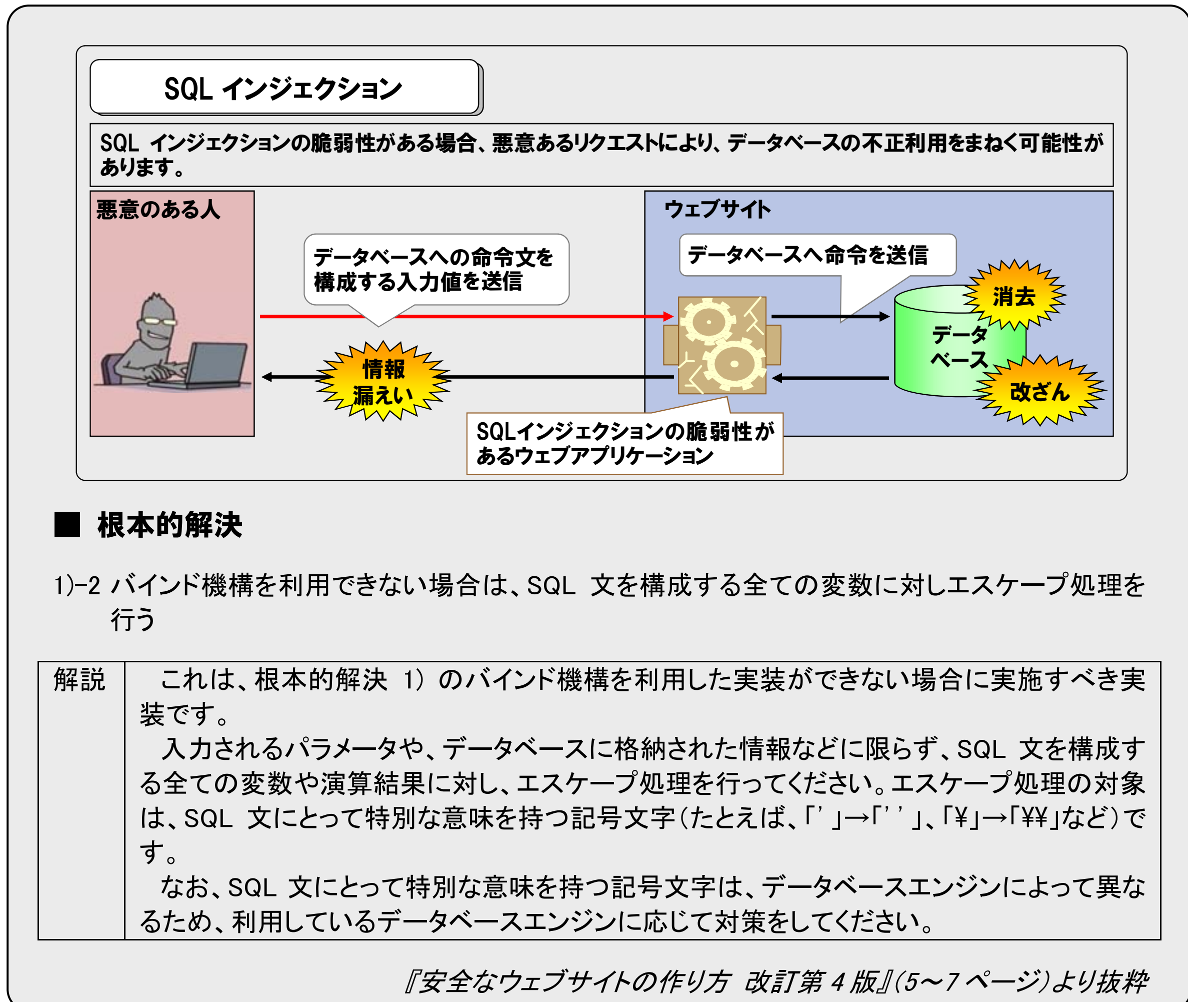
「安全なウェブサイトの作り方」でも、発行当初から SQL インジェクション対策の重要性を訴求し、根本的な解決策としてバインド機構の利用を推奨してきましたが、使用するソフトウェア製品によっては、バインド機構の安全性が疑われる事例も出てきました。また、バインド機構を用いずにエスケープ処理によって対策する手法が、書籍や記事等でも散見されますが、データベースエンジンの種類や設定によっては、その手法では安全な SQL 呼び出しを実現できない場合があることが明らかになってきました。

本書は、このような現状をふまえ、バインド機構やエスケープ処理による対策手法が安全なものであるための要件を原理から掘り下げて検討し、どの DBMS 製品をどのように使えば安全な SQL 呼び出しを実現できるのか、その考え方を整理しながら、いくつかの具体的ケースについて調査結果を示します。

本書が、ウェブサイトにおける SQL インジェクション対策の一助となれば幸いです。

1. 本書の位置づけ

「安全なウェブサイトの作り方 改訂第 4 版」では、SQL インジェクションの対策方法について以下のように説明しています。



■ **根本的解決**

1)-2 バインド機構を利用できない場合は、SQL 文を構成する全ての変数に対しエスケープ処理を行う

解説	これは、根本的解決 1) のバインド機構を利用した実装ができない場合に実施すべき実装です。 入力されるパラメータや、データベースに格納された情報などに限らず、SQL 文を構成する全ての変数や演算結果に対し、エスケープ処理を行ってください。エスケープ処理の対象は、SQL 文にとって特別な意味を持つ記号文字(たとえば、「'」→「''」、「¥」→「¥¥」など)です。 なお、SQL 文にとって特別な意味を持つ記号文字は、データベースエンジンによって異なるため、利用しているデータベースエンジンに応じて対策をしてください。

『安全なウェブサイトの作り方 改訂第 4 版』(5~7 ページ)より抜粋

SQL インジェクションの根本的解決の基本はバインド機構を使用することですが、何らかの理由によりバインド機構で実装できない場合は、エスケープ処理による対策も根本的解決の一つになるとしています。しかし、上の引用部にもあるように、SQL にとって特別な意味を持つ記号(メタ文字)は、データベースエンジンによって異なるものであり、環境に応じて対策する必要があります。「安全なウェブサイトの作り方」は、その実現方法について詳しく説明していません。

本書は、その実現方法について解説します。データベースエンジンに応じた対策の必要性を、原理まで掘り下げて解説し、各種の DBMS 製品が期待される正しい動作をするかどうかの調査結果を示します。加えて、文字エンコーディングが SQL インジェクションに及ぼす影響についても検討します。それらの結果から、どの DBMS 製品をどのように使えば安全な SQL 呼び出しを実現できるかを明らかにします。

次章では、SQL インジェクションが発生する原理と関係が深い、SQL におけるリテラルの説明をします。

2. リテラルとSQLインジェクション

2.1. SQL文の構造

以下の例を用いて、SQL 文の構造を説明します。

```
SELECT a,b,c FROM atable WHERE name='YAMADA' and age>=20
```

SQL

SQL 文を構成する要素には、キーワード、演算子、識別子、リテラルなどがあります。

キーワード（予約語）	SELECT FROM WHERE AND
演算子など	= >= ,
識別子	a b c atable name age
リテラル	'YAMADA' 20

2.2. リテラルとは

たとえば SQL で、列「employee_id」（社員番号）が「052312」である社員を検索するには、以下のような SQL 文を実行します。

```
SELECT name, age FROM employee WHERE employee_id = '052312'
```

SQL

ここで、SQL 文中の「'052312'」のような定数のことをリテラルと呼び、文字列としてのリテラルのことを文字列リテラルと呼びます。リテラルには文字列リテラルの他、数値リテラル、論理値リテラル、日時リテラルなどがあります。

【数値リテラルの例】

```
20
-17
0
3.14159
6.0221415E+23
```

SQL

【文字列リテラルの例】

```
'情報処理推進機構'
'052312'
'O''Reilly'
```

SQL

【日時リテラルの例】

```
DATE '2009-11-04'
TIME '13:59:26'
```

SQL

2.3. 文字列リテラルのエスケープ

文字列リテラルや日時リテラルは全体をシングルクォートで囲んで記述します。これをクォートするといいます。クォートする対象の文字列の中にシングルクォートが現れる場合が問題になります。文字としてのシングルクォートなのか、リテラルの終端としてのシングルクォートなのかを区別しなければならないからです。

たとえば、人名「O'Reilly」を SQL で以下のように検索しようとすると、SQL の構文エラーになります。

```
SELECT * FROM employee WHERE name = 'O'Reilly'
```

SQL

これは、「'O'」までの部分で文字列リテラルが終端し、その後ろに続く「Reilly'」がリテラルの外にはみ出すからです。このため、文字列リテラル中にシングルクォートが現れる場合には、シングルクォートを重ねて記述することで文字としてのシングルクォートを表すというのが SQL の文法です。先の SQL 文は以下のように記述する必要があります。この処理を文字列リテラルのエスケープ処理と呼びます。

```
SELECT * FROM employee WHERE name = 'O''Reilly'
```

SQL

なお、エスケープが必要な文字は、シングルクォートだけとは限りません。データベースエンジンの種類や設定によって異なります。これについては 4.1 節で述べます。

2.4. 数値リテラル

数値リテラルはクォートしません。数値リテラルの一種である整数リテラルの文法は、概ね以下のようになっています。

整数	← 符号 符号なし整数 または 符号なし整数
符号	← プラス符号(+) または マイナス符号(-)
符号なし整数	← 1 個以上の数字
数字	← 0～9 のいずれか

上記のように、整数リテラルは符号または数字で始まり、数字以外の文字が出現したところで、その手前の文字までということになります。ただし、SQL の JIS/ISO の規格(JIS X 3005、ISO/IEC 9075)では、数値リテラルの次には、記号または空白、コメントが続かなければならないと規定されています。

したがって、以下の SQL 文は JIS/ISO に準拠しています。

```sql
SELECT * FROM employee WHERE age >= 25--comment
```

SQL

一方、以下の SQL 文は JIS/ISO に違反しています。

```sql
SELECT * FROM employee WHERE age >= 25and age <= 60
```

SQL

25 と and の間には空白あるいはコメントが必要です。しかし、一部のデータベースエンジンの実装（Microsoft SQL Server や PostgreSQL）はこのような SQL 文を受け入れます。

数値リテラルはクォートしないため、文字列リテラルのエスケープに相当する処理はありません。

2.5. SQLインジェクションの原因

SQL をアプリケーションから利用する場合、SQL 文のリテラル部分をパラメータ化することが一般的です。パラメータ化された部分を実際の値に展開するとき、リテラルとして文法的に正しく文を生成しないと、パラメータに与えられた値がリテラルの外にはみ出した状態になり、リテラルの後ろに続く文として解釈されることになります。この現象が SQL インジェクションです。

2.5.1. 文字列リテラルに対するSQLインジェクション

以下の Perl による SQL 文生成の例を用いて、SQL インジェクションを説明します。SQL の id 列は文字列型を想定しています。$id は Perl の変数で、外部から与えられるものとします。

```perl
$q = "SELECT * FROM atable WHERE id='$id'";
```

Perl

ここで、$id に以下の値を与える場合、

```
';DELETE FROM atable--
```

テキスト

パラメータを展開した後の SQL 文は以下のようになります。

```
SELECT * FROM atable WHERE id='';DELETE FROM atable--'
```
SQL

SELECT 文の後ろに DELETE 文が追加され、データベースの内容がすべて削除される結果になります。「--」以降はコメントとして無視されます。

このように、SQL 文の文字列リテラルをパラメータ化しているときに、そこに別の SQL 文の断片を含ませることで、元の SQL 文の意味を変更できる場合があります。これが SQL インジェクションの脆弱性です。

2.5.2. 数値リテラルに対するSQLインジェクション

数値リテラルについては、Perl や PHP など、変数に型のない言語を使用している場合に注意が必要です。アプリケーション開発者は変数に数値が入っているつもりでも、数値以外の文字が入力された場合、変数に型のない言語では、それを文字列として扱ってしまいます。

前節と同様の例で、id 列が数値型の場合で説明します。$id は数値リテラルを構成するため、シングルクォートで囲みません。

```
$q = "SELECT * FROM atable WHERE id=$id";
```
Perl

このような SQL 呼び出しがあった際に、$id として以下の値を与える場合、

```
0;DELETE FROM atable
```
テキスト

パラメータを展開した後の SQL 文は以下のようになります。

```
SELECT * FROM atable WHERE id=0;DELETE FROM atable
```
SQL

SELECT 文の後ろに DELETE 文が追加され、データベースの内容がすべて削除される結果になります。

2.5.3. 安全なSQL呼び出しの要件

このように、安全な SQL 呼び出しを実現するには、パラメータを正しくリテラルとして展開することが必要です。具体的には以下を実施してください。

- 文字列リテラルに対しては、エスケープすべき文字をエスケープすること
- 数値リテラルに対しては、数値以外の文字を混入させないこと

次章では、アプリケーションから SQL を呼び出す方法を 3 種類に分類し、それぞれの方法でリテラルがどのように扱われるかを検討します。

3. SQLの呼び出し方

アプリケーションから SQL を呼び出す際には、検索条件などをパラメータ化することが一般的です。前章の例のように、列が人名や社員番号などを表している場合に、検索したい人名文字列や社員番号の数値を、アプリケーションから SQL に渡す必要があります。その実現方法は、まず大きく分けて次の 2 つの方法に分類できます。

・文字列連結による SQL 文の組み立て
・プレースホルダによる SQL 文の組み立て

3.1. 文字列連結によるSQL文の組み立て

文字列連結による組み立てとは、アプリケーションのプログラミング言語で、SQL 文を生成するとき、パラメータ部分への値の埋め込みを文字列連結によって実現する方法です。以下は、CGI パラメータ「name」に指定された社員を検索する SQL 文を、PHP で組み立てる処理の例です。ただし、このプログラムには SQL インジェクションの脆弱性があります。

```
$name = $_POST['name'];
//...
$sql = "SELECT * FROM employee WHERE name='" . $name . "'";
```

PHP

PHP の式「$name」の値が「山田」の場合、実行後は以下の SQL 文が生成されます。

```
SELECT * FROM employee WHERE name='山田'
```

SQL

SQL インジェクションの脆弱性をなくすには、式「$name」をクォートして文字列連結する際に、「$name」の値に対するエスケープ処理が必要です。エスケープが必要な文字は、データベースエンジンの種類や設定によって異なります。

3.2. プレースホルダによるSQL文の組み立て

プレースホルダによる組み立てとは、パラメータ部分を「?」などの記号で示しておき、後に、そこへ実際の値を機械的な処理で割り当てる方法です。以下は Java におけるプレースホルダによる組み立ての例です。

```
PreparedStatement prep = conn.prepareStatement("SELECT * FROM employee WHERE name=?");
prep.setString(1, "山田");
```

Java

ここで、パラメータ部分を示す記号「?」のことをプレースホルダと呼び、そこへ実際の値を割り当てることを「バインドする」と呼びます。プレースホルダのことを「バインド変数」と呼ぶこともあります。

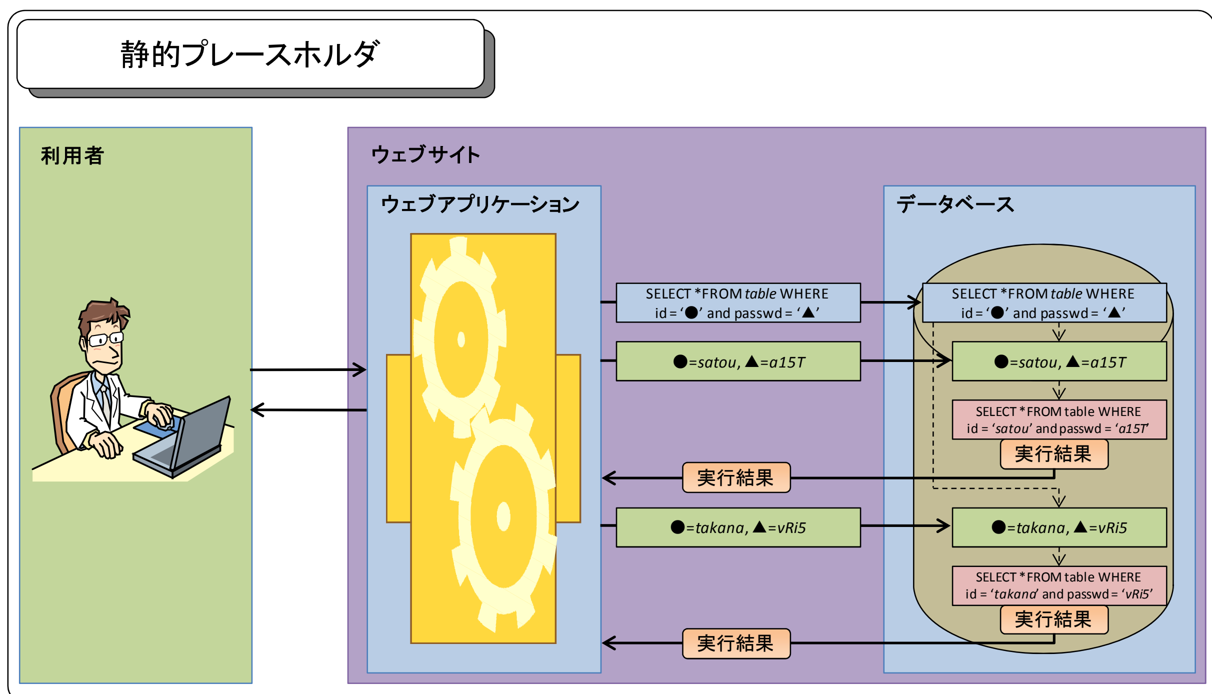
プレースホルダによる組み立ては、バインドをいつ行うのかによって2種類に分けることができます。ここでは次のように呼ぶことにします。

・静的プレースホルダ
・動的プレースホルダ

3.2.1. 静的プレースホルダ

静的プレースホルダは、JIS/ISO の規格では「準備された文(Prepared Statement)」と規定されています。これは、プレースホルダのままの SQL 文をデータベースエンジン側にあらかじめ送信して、実行前に、SQL 文の構文解析などの準備をしておく方式です。SQL 実行の段階で、実際のパラメータの値をデータベースエンジン側に送信し、データベースエンジン側がバインド処理します。

アプリケーションでは同じ SQL 文をパラメータの値を変えながら繰り返し実行することが多いため、構文解析などの準備を事前にしておくことによって、実行効率を高くできる利点があります。ただし、静的プレースホルダは、データベースエンジンやライブラリによってはサポートされていない場合があります。

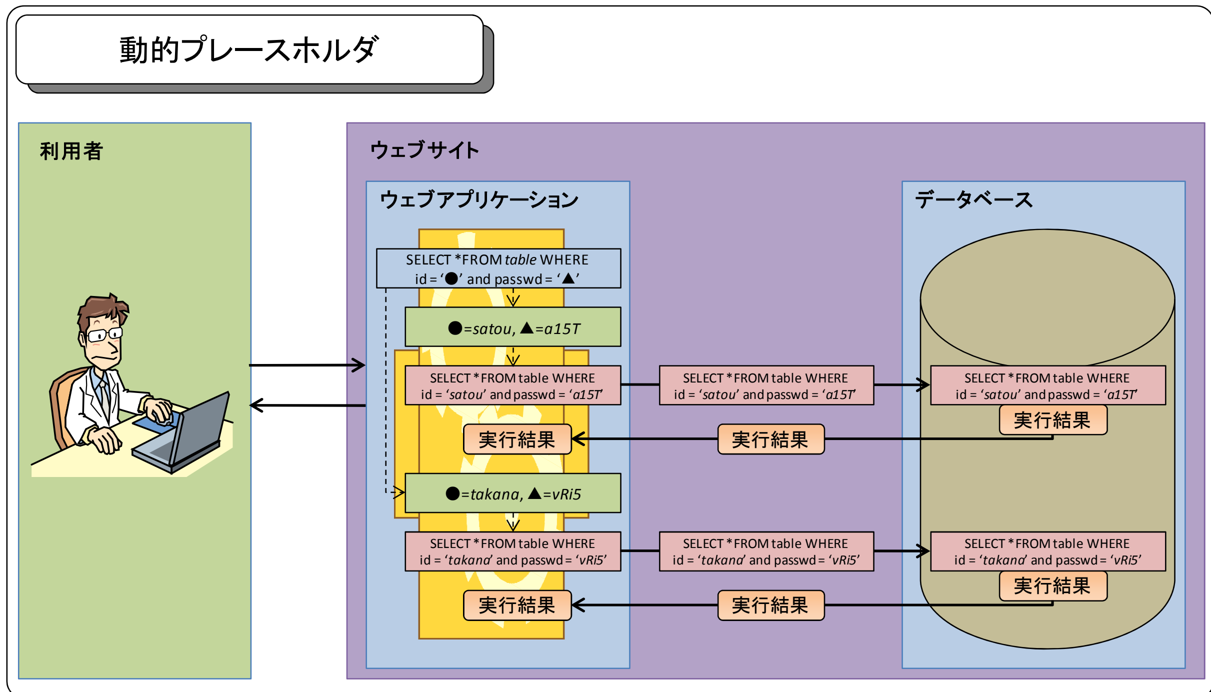


静的プレースホルダでは、SQL 文の構文がバインド前に確定することから、プレースホルダに渡す文字列はクォートして記述する必要がありません。そのため、シングルクォートのエスケープ処理も必要ありません。また、数値リテラルもそのまま適切にバインドされます。

このことから、セキュリティの観点で、静的プレースホルダは最も安全です。静的プレースホルダでは、SQL を準備する段階で SQL 文の構文が確定し、後から SQL 構文が変化することがないため、パラメータの値がリテラルの外にはみ出す現象が起きません。その結果として、SQL インジェクションの脆弱性が生じません。

3.2.2. 動的プレースホルダ

動的プレースホルダは準備された文(Prepared Statement)とは異なり、プレースホルダを利用するものの、パラメータのバインド処理をデータベースエンジン側で行うのではなく、アプリケーション側のライブラリ内で実行する方式です。



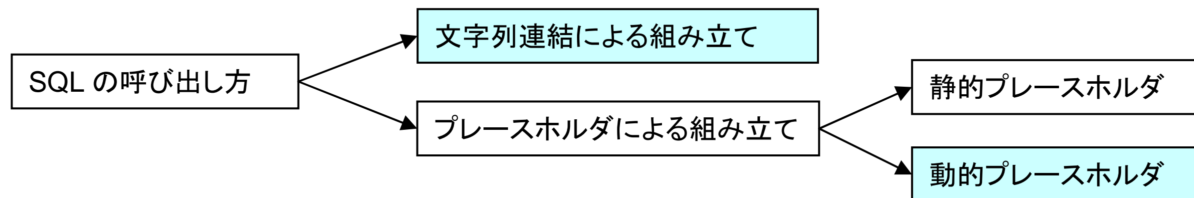
図のように、SQL 呼び出しごとに、パラメータをバインドした後の SQL 文がデータベースエンジンに送られるため、実行効率の点で静的プレースホルダに劣りますが、一部のデータベースエンジンは静的プレースホルダの代わりにこの機能を提供しています。これを俗に、「クライアントサイドのプリペアドステートメント」と呼ぶことがありますが、JIS/ISO で規定された意味での「準備された文(Prepared Statement)」ではないので注意が必要です。

セキュリティの観点では、プレースホルダを用いたバインド処理によってパラメータの値の埋め込みがライブラリで機械的に処理されることから、文字列連結による組み立てに比べてアプリケーション開発者のミスによるエスケープ漏れを防止できると期待されます。

ただし、動的プレースホルダは静的プレースホルダとは異なり、バインド処理を実現するライブラリによっては、SQL 構文を変化させるような SQL インジェクションを許してしまう、脆弱な実装のものが存在する可能性を否定できません。

3.3. まとめ

本章での SQL の呼び出し方の分類をまとめると下図のようになります。



このうち、静的プレースホルダは、SQL を準備する段階で SQL 文の構文が確定し、あとから SQL 構文が変化することがないため、SQL インジェクションの脆弱性が生じません。

それに対して、動的プレースホルダは、バインド処理を実現するライブラリの実装に問題があると、SQL 構文が変化する可能性があり、SQL インジェクションの脆弱性が生じる可能性を否定できません。

また、文字列連結による組み立ては、アプリケーション開発者のミスによるエスケープ漏れが生ずる危険性のほか、エスケープすべき文字がデータベースエンジンの種類や設定によって異なるため、それに合わせた開発が容易でないという問題をかかえています。

次章では、「文字列連結による組み立て」と「動的プレースホルダ」のこれらの問題について、詳しく検討します。

4. データベースと連動したSQL文生成

本章では、「文字列連結による組み立て」と「動的プレースホルダ」について、安全な SQL 呼び出しのためには何が必要かを検討します。

4.1. 文字列連結にquoteメソッドを使う

文字列連結による SQL 文の組み立てで安全な SQL 呼び出しを実現するには、以下の要件を満たさなければなければなりません。

・文字列リテラルに対しては、エスケープすべき文字をエスケープすること
・数値リテラルに対しては、数値以外の文字を混入させないこと

しかし、付録 A.1 に示すように、文字列リテラル生成時にエスケープが必要な文字は、データベースエンジンの種類によって異なり、さらにデータベースの設定によって異なる場合があります。データベースの種類や設定に対応したエスケープ処理を行わないと、SQL インジェクションの脆弱性を生じる場合があります。

そのような処理は煩雑であるため、アプリケーションのプログラミング言語やデータベースエンジンによっては、SQL におけるリテラルを文字列として生成する専用のメソッドや関数を用意している場合があります。

Perl の DBI、PHP の Pear::MDB2、PDO (PHP Data Objects) などで用意されている quote メソッドは、SQL のリテラルを生成する処理を抽象化したメソッドであり、アプリケーション開発者がデータベースエンジンの種類や設定を意識することなく、正しいエスケープ処理を実現するのに利用できます。

【PHP の Pear::MDB2 における quote の呼び出し】

```
require_once 'MDB2.php';//ライブラリのロード

// DB への接続（PostgreSQL の場合）
$db = MDB2::connect('pgsql://dbuser:password@hostname/dbname?charset=utf8');

// 文字列型を指定して、文字列リテラルのクォート済み文字列を得る
（略）$db->quote($s, 'text') （略）

// 数値型を指定して、数値リテラルの文字列を得る
（略）$db->quote($n, 'decimal') （略）
```
PHP

上のように、quote メソッドの第 2 引数で、生成するリテラルの型を指定します。「'text'」で文字列型を指定すると、データベースエンジンの種類や設定に応じて適切なエスケープ処理を施し、それをシング

ルクォートで括った文字列を返します。「'decimal'」などの数値型を指定すると、数値リテラルとして相応しい文字列を返します。

PHP の Pear::MDB2 で、様々なデータを quote メソッドに与えた場合の例を以下に示します。

データ	型指定	戻り値
abc	'text'	'abc'（PHP の文字列型の値、クォートを含む）
O'Reilly	'text'	'O''Reilly'（PHP の文字列型の値、クォートを含む）
-123	'decimal'	-123（PHP の文字列型の値）
123abc	'decimal'	123（PHP の文字列型の値）
-123	'integer'	-123（PHP の整数型の値）
123abc	'integer'	123（PHP の整数型の値）

これらは正しく処理されたリテラルの生成[1]であり 、これを用いることによって、SQL文の構文エラーを防ぎ、結果としてSQLインジェクションの脆弱性を排除できます。

しかし、調査の結果、アプリケーションのプログラミング言語とデータベースエンジンの組み合わせによっては、quote メソッドが期待される正しい結果を返さない場合があることが判明しました。この問題について、5 章で検証しています。

4.2. データベースと連動した動的プレースホルダ

動的プレースホルダは、文字列連結をアプリケーションで行う代わりに、ライブラリやドライバ側で機械的に行うものです。以下に、動的プレースホルダを使用して、SQL 文がどのように組み立てられるかを説明します。

動的プレースホルダを実現している例として、Perl 言語の DBI モジュールから呼び出される DBD::mysql があります。DBD::mysql の提供するプレースホルダ機能は、データベース接続時のオプションにより、動的プレースホルダを使用するか静的プレースホルダを使用するかを、明示的に指定することができます。

```
my $db = DBI->connect("DBI:mysql:$dbname:$host;mysql_server_prepare=0",
                      $user, $pwd) || die $DBI::errstr;
my $sql = "SELECT * FROM test3 where age >= ? and name = ?";
my $sth = $db->prepare($sql);
$sth->bind_param(1, 27, SQL_INTEGER);
$sth->bind_param(2, '山本', SQL_VARCHAR);
my $rt = $sth->execute();
```

Perl

この例では、データベース接続時のオプション「mysql_server_prepare=0」によって、動的プレースホルダを指定しています。

[1] PHP の Pear::MDB2 では、quote メソッドの型指定で 'integer' を指定すると、戻り値は文字列ではなく PHP の整数型の値となりますが、SQL 文として文字列連結する際に文字列に変換されます。

このプログラムによって生成される SQL 文は以下のようになります。

```
SELECT * FROM test3 where age >= 27 and name = '山本'
```

SQL

同じプログラムで、name 列に「O'reilly」および「\100」を与えた場合、生成される SQL 文はそれぞれ以下のようになります。

```
SELECT * FROM test3 where age >= 27 and name = 'O\'reilly'
SELECT * FROM test3 where age >= 27 and name = '\\100'
```

SQL

MySQL のデフォルト設定では、「'」と「\」をエスケープする必要があるため、このような挙動となります。しかし、MySQL に「NO_BACKSLASH_ESCAPES」オプションを指定した場合、JIS/ISO の規格通り「'」のみをエスケープする設定となり（付録 A.1 参照）、生成される SQL 文は以下のようになります。

```
SELECT * FROM test3 where age >= 27 and name = 'O''reilly'
SELECT * FROM test3 where age >= 27 and name = '\100'
```

SQL

このように、DBD::mysql は MySQL 側の設定に応じた適切なエスケープ処理を実現します。

次に、1 番目のプレースホルダに、数値以外の文字を入れた場合を検討します。「1 or 1=1」を与えた場合、以下のようになります。

【生成される SQL】

```
SELECT * FROM aTable where age >= 1 and name = '山本'
```

SQL

【エラーメッセージ】

```
DBD::mysql::st bind_param failed: Binding non-numeric field 1, value '1 or 1=1' as
a numeric! at C:¥・・・ファイル名・・・ line 23.
```

テキスト

エラーメッセージに failed とあることから、SQL 文が実行されていないようにも見えますが、実際には SQL 文は実行されています。しかし、このケースでは、1 番目のプレースホルダには、文字列「1 or 1=1」を数値に変換した「1」がバインドされて実行されています。文字列が数値に変換されたことにより、SQL インジェクションは防止できています。

こうした動的プレースホルダの処理が、アプリケーションのプログラミング言語とデータベースエンジンの組み合わせそれぞれにおいて、期待される正しい挙動となっているかどうか、次の 5 章で検証しています。

5. DBMS製品の実態調査

5.1. 調査内容

前章までの検討をふまえ、ウェブアプリケーション開発で実際に使用されることの多い DBMS とプログラミング言語の組み合わせを対象に、以下の点について実態を調査しました。

・プレースホルダの実装は静的プレースホルダ(準備された文)か、動的プレースホルダか
・動的プレースホルダのエスケープ処理は正しく処理されるか
・quote メソッドは正しく処理されるか
・文字エンコーディングの扱い

5.2. Java + Oracle

Java から Oracle を呼び出すには通常、JDBC を使用します。Oracle 用 JDBC は数種類提供されていますが、この調査では、Oracle 社が提供する ojdbc6.jar を用い、データベースの文字エンコーディングを UTF-8 に設定しました(付録 A.4 参照)。

【調査結果】

項目	調査結果
プレースホルダの実装	静的プレースホルダのみ
動的プレースホルダのエスケープ処理	調査対象外(動的プレースホルダは提供されていない)
quote メソッドの処理	調査対象外(quote メソッドは提供されていない)
文字エンコーディングの扱い	DB 接続には UTF-8 が使用される

Java + Oracle + ojdbc6.jar では、常に静的プレースホルダが使用されるため、Java の PreparedStatement インターフェースを使っている限り、注意点はありません。

Java では quote メソッドに該当するライブラリが提供されておらず、データベースエンジンの種類や設定に連動したエスケープ処理ができないため、文字列連結による SQL 文の組み立ては推奨されません。

5.2.1. サンプルコード

```
import java.sql.*;

public class OraclePrepared {
  public static void main(String[] args) {
    String url = "jdbc:oracle:thin:@SERVERNAME:1521:ORCL";
    try {
      // JDBC ドライバのロード
      Class.forName("oracle.jdbc.OracleDriver");
      // データベース接続
      db = DriverManager.getConnection(url, userName, password);

      String param = ....

      String sql = "SELECT * FROM atable WHERE name=?";
      PreparedStatement stmt = con.prepareStatement(sql);
      stmt.setString(1, param);   // ? の場所に値を埋め込む
      ResultSet rs = stmt.executeQuery();
      while(rs.next()){
        int id = rs.getInt("id");
        String name = rs.getString("name");
        String address = rs.getString("address");
        String comment = rs.getString("comment");
        System.out.printf("id = %d  name = %s  address = %s  comment = %s\n",
 id, name, address, comment);
      }
      rs.close();
      stmt.close();
      con.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Java

5.3. PHP + PostgreSQL

PHP から PostgreSQL を呼び出すライブラリは数多くありますが、下記の点から、PEAR::MDB2 について調査しました。

- MDB2 は PostgreSQL の他、MySQL、Oracle 等の複数の DBMS に対して、SQL 呼び出しを抽象化したインターフェースを提供している
- Pear::DB、Pear::MDB などの同種のモジュールは既に開発が終了しているのに対し、MDB2 は開発が継続されている
- 文字エンコーディングを考慮している
- 静的プレースホルダを利用できる
- プレースホルダへの値のバインドとクォートの際に、データの型を考慮している

以下、調査結果について説明します。

【調査結果】

項目	調査結果
プレースホルダの実装	静的プレースホルダのみ
動的プレースホルダのエスケープ処理	調査対象外（動的プレースホルダは提供されていない）
quote メソッドの処理（文字列リテラルの生成）	正しく処理される
quote メソッドの処理（数値リテラルの生成）	正しく処理される
文字エンコーディングの扱い	DB 接続時に文字エンコーディングを指定できる

PHP + MDB2 + PostgreSQL の組み合わせでは、常に静的プレースホルダが使用されるため、プレースホルダを使っている限り、注意点はありません。

プレースホルダの代わりに quote メソッドを使うことも可能で、quote メソッドは、文字列リテラル、数値リテラルともに、正しく生成します。

5.3.1. サンプルコード

MDB2 を用いた SQL 呼び出しのサンプルコードを以下に示します。

```
<?php
require_once 'MDB2.php';//ライブラリのロード

$db = MDB2::connect('pgsql://username:password@hostname/dbname' .
'?charset=utf8');
if(PEAR::isError($db)) {
   //エラー処理
}

$stmt = $db->prepare('SELECT * FROM atable WHERE name=? and num=?',
            array('text', 'integer'), array('text', 'text', 'integer'));
$rs = $stmt->execute(array($name, $num)); // 文字列型と整数型の変数
if(PEAR::isError($rs)) {
    //エラー処理
}

// 以下は検索結果の表示
while($row = $rs->fetchRow()) {
    printf("%s:%s:%s\n", $row[0], $row[1], $row[2]);
}
```

PHP

【DB 接続時の説明】

1. 文字エンコーディングの指定
   PostgreSQL との通信に用いる文字エンコーディングとして、UTF-8 を使用することを指定します。

   ```
   charset=utf8
   ```

   テキスト

   PHP では、設定ファイル php.ini で、プログラム内部で使用する文字エンコーディングを指定できます。この文字エンコーディングとして Shift_JIS を指定すると、Shift_JIS の 2 バイト目に存在し得るコード 5C が「\」と解釈されて、SQL インジェクションの問題が生ずる場合があります。このため、文字エンコーディングとして、UTF-8 または EUC-JP を指定することを推奨します。また、PHP プログラムとデータベースで文字エンコーディングを一致させることを推奨します。

2. プレースホルダの型を指定する
   プレースホルダに型を指定することができます。型を指定しないと、varchar(可変長文字列型)と

みなされ、SQL 実行時に文字列型から実際の型への「暗黙の型変換」が起こり、精度が損なわれるなどの予期せぬ不具合や、性能低下の原因になります。

```
$stmt = $db->prepare('SELECT * FROM atable WHERE name=? and num=?',
             array('text', 'integer'), array('text', 'text', 'integer'));
```

PHP

上の例で、第 2 引数に与えている配列がプレースホルダの型指定です。第 3 引数は、結果の型を指定するもので、これは省略可能です。

【プレースホルダを利用できない場合のエスケープ処理】

何らかの理由でプレースホルダを利用できない場合、quote メソッドを使うことで、テーブルの列の型とデータベースエンジンの設定状況に連動した、正しいエスケープ処理を実現できます。

PostgreSQL では、標準設定ではバックスラッシュ「\」がエスケープの必要なメタ文字として扱われますが、standard_conforming_strings を on に設定した場合には、バックスラッシュがメタ文字ではなくなります(付録 A.1 参照)。このため、standard_conforming_strings の値に応じて、エスケープ方法を切り替えなければなりません。MDB2 の quote メソッドは、この切り替えを自動的に行います。

また、MDB2 の quote メソッドは型の指定が可能です。

```
$sql = 'SELECT * FROM atable WHERE name=' . $db->quote($name, 'text') . ' and num=' .
$db->quote($num, 'integer');
```

PHP

$db->quote('10', 'integer')は、SQL の数値リテラル「10」を表す PHP の整数型の値「10」を返し、文字列連結の際に PHP の文字列「10」に変換されます。$db->quote('10', 'text')は、SQL の文字列リテラルを表す PHP の文字列「'10'」を返します。この機能を利用して、テーブルの列の型に合わせたエスケープ処理を実現できます。

5.4. Perl + MySQL

Perl から SQL を呼び出す際には、通常、DBI モジュールを使用します。DBI モジュールは SQL 呼び出しを抽象化したインターフェースであり、DBMS 毎のドライバとして DBD モジュールが用意されています。

DBI による SQL 呼び出しでは、prepare メソッドによるプレースホルダを利用できるほか、quote メソッドも用意されています。

Perl DBI と DBD::MySQL の組み合わせについて調査した結果を下記に示します。

【調査結果】

項目	調査結果
プレースホルダの実装	動的プレースホルダまたは静的プレースホルダ
動的プレースホルダのエスケープ処理	正しく処理される
quote メソッドの処理（文字列リテラルの生成）	正しく処理される
quote メソッドの処理（数値リテラルの生成）	正しく処理されない（入力をそのまま返す）
文字エンコーディングの扱い	DB 接続時に UTF-8 を明示的に指定できる

5.4.1. プレースホルダの実装

デフォルトでは動的プレースホルダが選択されるので、静的プレースホルダを使用するには、DB 接続時に mysql_server_prepare=1 というパラメータを指定する必要があります。

5.4.2. エスケープ対象となる文字の扱い

MySQL では、標準設定ではバックスラッシュ「\」がエスケープの必要なメタ文字として扱われますが、NO_BACKSLASH_ESCAPES を設定すると、バックスラッシュがメタ文字ではなくなります（付録 A.1 参照）。このため、この設定に応じて、エスケープ方法を切り替えなければなりません。DBD::MySQL の quote メソッドは、この切り替えを自動的に行い、文字列リテラルのエスケープ処理を正しく行います。

5.4.3. quoteメソッドにおける数値リテラルの扱い

DBD::MySQL の quote メソッドは、第 2 引数で数値型を指定した場合、入力文字列が数値として妥当かのチェックを行わず、また数値への変換をすることもなく、入力文字列をそのまま返します。

【例】

```
$dbh->quote("1 or 1=1", SQL_INTEGER);  # → "1 or 1=1" を返す
```

Perl

したがって、SQL インジェクション対策として使用できません。このため、現状の DBD::MySQL では、文字列連結による SQL 文の組み立ては推奨されません。既存のアプリケーションに対する SQL インジェクション対策の場合など、やむを得ない場合を除き、プレースホルダの利用を推奨します。

5.4.4. 文字エンコーディングの扱い

Perl 5.8 以降では、プログラム内部で使用する文字エンコーディングとして UTF-8 の利用が推奨されています。DB 接続時に mysql_enable_utf8=1 を指定することで、Perl プログラムとデータベースで文字エンコーディングを一致させることができます。

5.4.5. サンプルコード

DBI/DBD を用いた SQL 呼び出しのサンプルコードを以下に示します。

```
#!/usr/bin/perl
use CGI;
use DBI;
use DBI qw(:sql_types);
use strict;
use utf8;
use Encode 'decode', 'encode';

my $db = DBI->connect(
'DBI:mysql:database=xxxx;host=xxxx;mysql_server_prepare=1;mysql_enable_utf8=1',
'xxxx', 'xxxx');
if (! $db) {
    # 接続失敗のエラー処理
}
my $sql = 'SELECT * FROM antable WHERE num=? AND name=?';
my $sth = $db->prepare($sql);

$sth->bind_param(1, $num, SQL_INTEGER);
$sth->bind_param(2, $name, SQL_VARCHAR);

my $rt = $sth->execute();
if (! $rt) {
    # SQL 呼び出しエラー時の処理
}
# 検索した結果のフェッチ処理など
$sth->finish;
$db->disconnect;
```

Perl

【DB 接続時の説明】

1. 静的プレースホルダの指定

   静的プレースホルダを指定します。これを指定しないと、動的プレースホルダとなり、データベース接続ドライバ内部でエスケープ処理されて SQL が呼び出されます。

   ```
   mysql_server_prepare=1
   ```

   テキスト

静的プレースホルダを利用することで、SQL 文とパラメータの値は別々にデータベースエンジンに送られます。プレースホルダ記号「?」のままで SQL 文が構文解析されることから、原理的に SQL インジェクションの危険性がありません。

2. 文字エンコーディングの指定
文字エンコーディングとして UTF-8 を使用することを指定します。

```
mysql_enable_utf8=1
```

テキスト

DB 接続に用いる文字エンコーディングとして UTF-8 を使用するという指定です。
Perl 5.8 以降では、Encode.pm を利用することにより、処理系内部の文字エンコーディングは UTF-8 で処理されます。DB 接続にも UTF-8 を使用することで、文字化けや、文字エンコーディングに起因するセキュリティ問題を回避することができます。

3. バインド処理で型を指定する
バインド処理には、bind_param メソッドを使います。bind_param メソッドの第 3 引数にデータの型を指定します。

```
$sth->bind_param(1, $num, SQL_INTEGER);
$sth->bind_param(2, $name, SQL_VARCHAR);
```

Perl

この方法以外に、execute メソッドでパラメータの値を与えることもできますが、その場合、パラメータはすべて varchar 型とみなされ、SQL 実行時に文字列型から実際の型への「暗黙の型変換」が生じます。MySQL の場合、文字列型から数値型への変換時に浮動小数点数型を経由するため、精度が損なわれるなどの不具合の原因になります。このことから、MySQL では、bind_param メソッドで型を明示してバインドする方法を推奨します。

5.5. Java + MySQL

Java から MySQL にアクセスする場合は JDBC(MySQL Connector/J)を使用します。JDBC では quote メソッドは用意されておらず、PrepeardStatement インターフェースを使用して SQL を呼び出します。

【調査結果】

項目	調査結果
プレースホルダの実装	動的プレースホルダまたは静的プレースホルダ
動的プレースホルダのエスケープ処理	正しく処理される(ただし、バージョンによっては付録 A.3 の問題がある)
quote メソッドの処理	調査対象外(quote メソッドは用意されていない)
文字エンコーディングの扱い	DB 接続時に文字エンコーディングを指定できる

5.5.1. プレースホルダの実装

デフォルトでは動的プレースホルダが選択されるので、静的プレースホルダを使用するには、DB 接続時に useServerPrepStmts=true というパラメータを指定する必要があります。

5.5.2. エスケープ対象となる文字の扱い

MySQL では、標準設定ではバックスラッシュ「\」がエスケープの必要なメタ文字として扱われますが、NO_BACKSLASH_ESCAPES を設定すると、バックスラッシュがメタ文字ではなくなります(付録 A.1 参照)。このため、この設定に応じて、エスケープ方法を切り替えなければなりません。MySQL Connector/J で動的プレースホルダを使用する場合、この切り替えは自動的に行われ、文字列リテラルのエスケープ処理は正しく行われます。

5.5.3. 文字エンコーディングの扱い

DB 接続時に characterEncoding パラメータにて文字エンコーディングを指定できます。付録 A.3 では、UTF-8 の使用を推奨しています。

5.5.4. サンプルコード

```
import java.sql.*;

public class MysqlPrepared {
  public static void main(String[] args) {
    String url = "jdbc:mysql://HOSTNAME/DBNAME?user=USERNAME&password=PASSWORD&"
              + "useUnicode=true&characterEncoding=utf8&useServerPrepStmts=true";
    try {
      Class.forName("com.mysql.jdbc.Driver");
      Connection con = DriverManager.getConnection(url);

      String param = ....

      String sql = "SELECT * FROM atable WHERE name=?";
      PreparedStatement stmt = con.prepareStatement(sql);
      stmt.setString(1, param);   // ? の場所に値を埋め込む
      ResultSet rs = stmt.executeQuery();
      while(rs.next()){
        int id = rs.getInt("id");
        String name = rs.getString("name");
        String address = rs.getString("address");
        String comment = rs.getString("comment");
        System.out.printf("id = %d  name = %s  address = %s  comment = %s\n",
                          id, name, address, comment);
      }
      stmt.close();
      con.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Java

5.6. ASP.NET + Microsoft SQL Server

ASP.NET と Microsoft SQL Server の組み合わせから SQL を呼び出す場合は、ADO.NET を利用できます。ADO.NET は、従来の ODBC や OLE DB 経由での SQL 呼び出しが可能であるほか、Microsoft SQL Server や Oracle 用のドライバを利用できます。

【調査結果】

項目	調査結果
プレースホルダの実装	静的プレースホルダのみ
動的プレースホルダのエスケープ処理	調査対象外（動的プレースホルダは提供されていない）
quote メソッドの処理	調査対象外（quote メソッドは提供されていない）
文字エンコーディングの扱い	DB サーバへの接続には UTF-16 が使用される

ASP.NET + Microsoft SQL Server の組み合わせでは、常に静的プレースホルダが使用されるため、プレースホルダを利用している限り、注意点はありません。

5.6.1. サンプルコード

ADO.NET で、Microsoft SQL Server 用のドライバ(.NET Framework Data Provider for SQL Server)を利用する例を、Visual Basic .NET による記述で示します。

```
Imports System.Data.SqlClient

Partial Class SqlSample
Inherits System.Web.UI.Page

'ページロード時の処理
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles
Me.Load

Dim dbcon As SqlConnection
Dim dbcmd As SqlCommand
Dim dataRead As SqlDataReader
Dim sqlStr As String

'DB コネクション作成
dbcon = New SqlConnection(
       "Server=HOSTNAME; database=DBNAME;userid=USERID;password=PASSWORD")

'DB 接続
dbcon.Open()

'SQL 文
sqlStr = "select * from aTalbe where name=@s1"

'SQL コマンド作成
dbcmd = New SqlCommand(sqlStr, dbcon)

Dim param As String = ....

'パラメータセット
Dim p1 As SqlParameter = New SqlParameter("@s1",param)
dbcmd.Parameters.Add(p1)

'SQL 文実行
dataRead = dbcmd.ExecuteReader()

'結果を読み込み(省略)

'クローズ処理
dataRead.Close()
dbcmd.Dispose()
dbcon.Close()
dbcon.Dispose()

end sub
End class
```

Visual Basic .NET

.NET フレームワークでは quote メソッドに該当するライブラリが提供されておらず、データベースエンジ

ンの種類や設定に連動したエスケープ処理ができないため、文字列連結による SQL 文の組み立ては推奨されません。

5.7. まとめ

本章で調査した結果を下の表にまとめます。

	Java + Oracle	PHP + MDB2 + PostgreSQL	Perl + MySQL	Java + MySQL	ASP.NET + SQL Server
プレースホルダの実装	静的のみ	静的のみ	静的または動的	静的または動的	静的のみ
動的プレースホルダの処理	－	－	○	△	－
quote メソッドの処理（文字列）	－	○	○	－	－
quote メソッドの処理（数値）	－	○	×	－	－
文字エンコーディング	UTF-8 が使われる	指定可能	UTF-8 を明示可能	指定可能	UTF-16 が使われる

プレースホルダの実装が静的プレースホルダに限定される環境では、プレースホルダを使っている限り、開発者が注意すべき点はありません。

一方、「prepare」など、一見、静的プレースホルダ（準備された文）による実装であるかのような名称のメソッドであっても、デフォルトでは動的プレースホルダが使用される環境もありますので、設定に注意して使用する必要があります。

動的プレースホルダを使用する場合、Java + MySQL のケース(MySQL Connector/J のバージョンが古い場合)のように、ドライバやライブラリの実装によっては、文字エンコーディングの処理が原因となって、SQL インジェクションの脆弱性が生ずることもあるので、注意が必要です。

何らかの理由で、文字列連結による SQL 文の組み立てが必要な場合、quote メソッドを使用することで、データベースエンジンの種類や設定に連動したエスケープ処理を実現できると期待されるところですが、Perl + MySQL のケースのように、数値リテラルの生成が正しく行われないものもあるようですので、注意が必要です。

また、今回の調査では結果を掲載していませんが、Perl 用の DBD::PgPP (Pure Perl PostgreSQL driver for the DBI)など、動的プレースホルダと quote メソッドにおいて、データベースエンジンの種類や設定に連動したエスケープ処理が行われない環境がある（DBD::PgPP バージョン 0.08 において確認）ようですので、注意が必要です。

文字エンコーディングについては、DB 接続において、常に UTF-8 や UTF-16 が使用される環境もありますが、環境によっては、文字エンコーディングを明示的に指定することが望ましい場合があります。また、今回の調査では結果を掲載していませんが、PHP の PDO (PHP Data Objects)のように、文字エンコーディング指定を無視するものもある（PDO version 1.0.4dev において確認）ようですので、注意が必要です。

付録A. 技術情報

A.1. バックスラッシュのエスケープ

データベースの製品によっては、シングルクォート以外の文字をエスケープしなければならないものもあります。MySQL と PostgreSQL がその代表例です。これらデータベースの場合、バックスラッシュが、シングルクォートなどをエスケープするためのメタ文字として解釈されるため、文字列リテラル中にバックスラッシュが含まれる場合には、バックスラッシュ自身のエスケープも必要になります。

エスケープ対象文字	エスケープ方法
'	''（\'でもよい）
\	\\

ただし、両製品とも、バックスラッシュをメタ文字として扱わない設定にすることも可能です。

MySQL のオプション	NO_BACKSLASH_ESCAPES バックスラッシュをエスケープ用のメタ文字として扱わない
PostgreSQL のオプション	standard_conforming_strings バックスラッシュをエスケープ用のメタ文字として扱わない

バックスラッシュのエスケープが必要な環境でそれを怠ると、SQL インジェクションの脆弱性となります。以下の SQL 文生成の例を用いて説明します。

```
$q = "SELECT * FROM atable WHERE a='$s'";
```

Perl

ここで、攻撃コードとして$s に以下の文字列を与えるとします。

```
\';DELETE FROM atable--
```

テキスト

この文字列をシングルクォートのみエスケープすると以下になります。

```
\'';DELETE FROM atable--
```

テキスト

このエスケープ処理をした文字列を元の SQL 文に与えると以下の文になります。

```
SELECT * FROM atable WHERE a='\'';DELETE FROM atable--'
```

SQL

前述のように、「\'」は「シングルクォートのエスケープ」とみなされるので、続くもう一つのシングルクォートがエスケープされない状態になります。

```
SELECT * FROM atable WHERE a='\'';DELETE FROM atable--'
                             ↑エスケープされていないシングルクォート
```

SQL

この結果、「;DELETE」以降がリテラルの外にはみ出して、SQL 文として解釈されてしまいます。

一方、バックスラッシュをエスケープする必要がない環境でバックスラッシュをエスケープした場合、二重のバックスラッシュがデータベースに格納されることになり、正しい処理となりません。

A.2. Shift_JISによるSQLインジェクション

文字列リテラルのエスケープ処理では、文字エンコーディングに配慮する必要があります。特に、文字エンコーディングとして Shift_JIS を用いていて、かつバックスラッシュ「\」をエスケープしなければならない環境で発生しやすいのですが、この組み合わせ以外の場合でも問題が生じる場合があります。

以下の SQL 文生成の例で説明します。

```
$q = "SELECT * FROM atable WHERE a='$id'";
```

Perl

ここで、$id に以下の文字列を与える場合、

```
表';DELETE FROM atable--
```

テキスト

「表'」の部分の文字コードは以下のようになります。

表		'
0x95	0x5c	0x27

0x5c は、US-ASCII や ISO-8859-1 ではバックスラッシュに該当します。この文字列に対し、文字エンコーディングを考慮せずにエスケープ処理を施した場合、0x5c がエスケープ対象と解釈されて、以下のようにエスケープされることがあります。

0x95	0x5c	0x5c	0x27	0x27

これを Shift_JIS として解釈すると以下のようになります。

表		\	'	'
0x95	0x5c	0x5c	0x27	0x27

これを元の SQL 文に与えると以下の文になります。

```
SELECT * FROM atable WHERE a='表\'';DELETE FROM atable--'
```

SQL

「\'」が「シングルクォートをエスケープしたもの」として解釈されるため、続くもう一つのシングルクォー

トで文字列リテラルが終端し、セミコロン以降がリテラルの外にはみ出し、SQL 文として解釈されてしまいます。

このことから、文字エンコーディングについて、以下の点に注意するべきです。

- データベースエンジン、プレースホルダ、quote メソッド等のライブラリ、アプリケーションのプログラムプログラミング言語それぞれについて、文字列処理において文字エンコーディングを考慮した処理を正しく実装しているものを使用する
- 文字エンコーディングに起因する SQL インジェクションの脆弱性が発生しやすい Shift_JIS の使用を避ける

A.3. UnicodeによるSQLインジェクション

MySQL と Java の組み合わせ(MySQL Connector/J)の場合、プレースホルダを使っていても、以下の条件が揃うと、SQL インジェクションが生じます。

- MySQL Connector/J の 5.1.7 以前のバージョンを使用している
- データベースとアプリケーション間の接続の文字エンコーディングとして Shift_JIS あるいは EUC-JP を使用している
- 動的プレースホルダを使用している

以下、具体的な例で示します。

```
Connection con = DriverManager.getConnection(
"jdbc:mysql://HOSTNAME/DBNAME?user=USERNAME&password=PASSWORD&useUnicode=true&ch
aracterEncoding=sjis");
```

Java

データベースへの接続で、useServerPrepStmts を指定しない(あるいは false を指定する)ことで、動的プレースホルダを使用しており、characterEncoding=sjis を指定(あるいは、characterEncoding を指定せず、my.ini のサーバ側文字エンコーディング設定を sjis あるいは ujis を指定)することで、データベースとアプリケーション間の接続の文字エンコーディングとして Shift_JIS あるいは EUC-JP を使用しています。

この状況で、以下の文字列をバインドすると SQL インジェクションが発生します。

```
"\u00a5'or 1=1#"
```

Java

Java 言語の文字列リテラル中の「\u00a5」は、Unicode のコードポイント U+00A5 を意味し、円記号「¥」が割り当てられています。Unicode ではバックスラッシュ「\」(U+005C)とは別に円記号が用意されています。しかし、Shift_JIS や EUC-JP ではこれらの区別がなく、どちらも 0x5C「¥」に変換されます。このため SQL 文を組み立てるときに、以下の現象が起きます。

【入力文字列（Unicode）】

コードポイント	00A5	0027	006F	0072	0020	0031	003D	0031	0023
文字	¥	'	o	r	SP	1	=	1	#

【エスケープ処理後の文字列（Unicode）】

コードポイント	00A5	005C	0027	006F	0072	0020	0031	003D	0031	0023
文字	¥	\	'	o	r	SP	1	=	1	#

【Shift_JIS に変換した文字列】

文字コード	5C	5C	27	6F	72	20	31	3D	31	23
文字	¥	¥	'	o	r	SP	1	=	1	#

【動的プレースホルダにバインドした SQL 文】

```
SELECT * FROM test WHERE name='¥¥' or 1=1#'
```

SQL

ここで、¥¥で示した部分が「バックスラッシュをエスケープした一文字」と見なされ、続くシングルクォートで文字列リテラルが終端されます。その結果、続く「or 1=1#」が SQL 文の一部として見なされ、SQL インジェクションとなります。

この問題が発生する原因は、SQL 文の動的プレースホルダによる組み立てでエスケープ処理時に仮定した文字エンコーディングと、SQL 文の実行時の文字コードが異なっており、前者から後者への文字エンコーディング変換において、異なる文字が同一の文字に割り当てられていることにあります。

この問題は、MySQL Connector/J（5.1.7 およびそれ以前）の脆弱性として 2009 年 7 月に修正されました[2]。

対策は以下の一つ以上を実施することです。全てを実施することを推奨します。

- MySQL との接続に使用する文字エンコーディングとして Unicode(UTF-8)を指定する
  （接続文字列に characterEncoding=utf8 を指定する）
- 静的プレースホルダを使用する
  （接続文字列に useServerPrepStmts=true を指定する）
- MySQL Connector/J の最新版を使用する

[2] JVN#59748723: MySQL Connector/J における SQL インジェクションの脆弱性
http://jvn.jp/jp/JVN59748723/index.html

A.4. OracleのデータベースをUnicodeで作成する方法

Oracle は、テーブル単位や列単位で文字エンコーディングを指定することができず、データベース単位で文字エンコーディングを指定します。既存データベースの文字エンコーディングを変更することはできないため、データベース新規作成時に文字エンコーディングに対する考慮を済ませておく必要があります。

上図は、Oracle の Database Configuration Assistant にて、「キャラクタ・セット」を指定しているところです。デフォルトではデータベース・キャラクタ・セットが Shift_JIS（JA16SJISTILDE）となっているところを、Unicode（AL32UTF8）に変更しています。

データベース・キャラクタ・セットを Unicode に設定していないことが、必ず SQL インジェクションの問題を発生させるわけではありませんが、データベースに格納した文字が別の文字に変換される場合があり、思わぬバグの原因となる場合があります。このため、Unicode で統一して処理することを推奨します。

A.5. Microsoft SQL Serverと文字コード指定

Microsoft .NET では、内部処理に使用する文字エンコーディングは UTF-16 で統一されています。一方、Microsoft SQL Server 上のテーブルの文字列を格納する際の文字エンコーディングは、Microsoft SQL Server をインストールしたサーバの動作環境のコードページによって決まり、日本語環境では、コードページ 932(CP932)が使用されます。

アプリケーションから Microsoft SQL Server に渡された UTF-16 エンコーディングの文字列は、データベースに格納される際に CP932 に変換されます。したがって、CP932 にない文字を格納することはできず、文字化けの原因になります。

このため Microsoft SQL Server では、Unicode の文字列を格納する手段が用意されています。Unicode 文字列を格納する場合は、列の型として、char や varchar の代わりに、nchar や nvarcar を使用します。また、文字列リテラルには、Unicode を示す識別子 N を前置します。

【NVARCHAR 型の使用例】

```sql
CREATE TABLE aTable (name NVARCHAR(30), city NVARCHAR(30));
```

SQL

【Unicode 文字列リテラルの使用例】

```sql
INSERT INTO aTable VALUES (N'佐藤', N'横浜市');
```

SQL

文字化けの例として、Unicode の U+00A5「¥」(円記号)が CP932 の文字コード 5C「\」(バックスラッシュ)に変換されてしまう事例などが挙げられます。

この問題が SQL インジェクションの原因となることはありません。この文字エンコーディング変換が SQL 文の構文解析の後に処理されるものだからです。しかし、予期しないバグの原因となったり、別の脆弱性の原因となる可能性があります。

【参考】

マイクロソフト MSDN より「Unicode データの使用」
http://msdn.microsoft.com/ja-jp/library/ms191200.aspx

著作・制作　独立行政法人情報処理推進機構（IPA）

編集責任　小林 偉昭　山岸 正

執筆者　徳丸 浩　永安 佑希允　相馬 基邦　勝海 直人

高木 浩光　独立行政法人産業技術総合研究所

協力者　板橋 博之　谷口 隼祐　大谷 槙吾　大森 雅司

※独立行政法人情報処理推進機構の職員については所属組織名を省略しました。

「安全なウェブサイトの作り方」別冊

安全な SQL の呼び出し方

［発　　行］２０１０年　３月１８日　第１版　第１刷

［著作・制作］独立行政法人 情報処理推進機構 セキュリティセンター
［協　　力］独立行政法人 産業技術総合研究所 情報セキュリティ研究センター

情報セキュリティに関する届出について

IPA セキュリティセンターでは、経済産業省の告示に基づき、コンピュータウイルス・不正アクセス・脆弱性関連情報に関する発見・被害の届出を受け付けています。

ウェブフォームやメールで届出ができます。詳しくは下記のサイトを御覧ください。
URL: http://www.ipa.go.jp/security/todoke/

コンピュータウイルス情報

　コンピュータウイルスを発見、またはコンピュータウイルスに感染した場合に届け出てください。

不正アクセス情報

　ネットワーク（インターネット、LAN、WAN、パソコン通信など）に接続されたコンピュータへの不正アクセスによる被害を受けた場合に届け出てください。

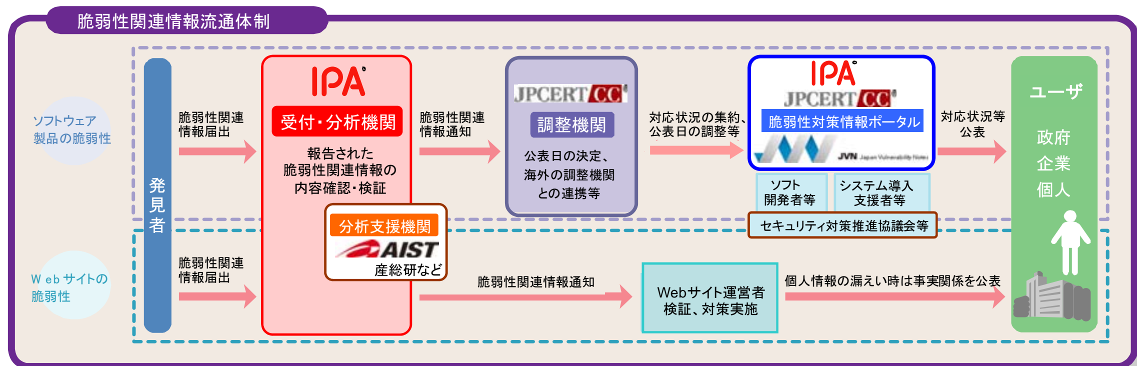
ソフトウエア製品脆弱性関連情報

　OSやブラウザ等のクライアント上のソフトウエア、ウェブサーバ等のサーバ上のソフトウエア、プリンタやICカード等のソフトウエアを組み込んだハードウエア等に対する脆弱性を発見した場合に届け出てください。

ウェブアプリケーション脆弱性関連情報

　インターネットのウェブサイトなどで、公衆に向けて提供するそのサイト固有のサービスを構成するシステムに対する脆弱性を発見した場合に届け出てください。

脆弱性関連情報流通の基本枠組み「情報セキュリティ早期警戒パートナーシップ」



※IPA：独立行政法人 情報処理推進機構、JPCERT/CC：一般社団法人 JPCERT コーディネーションセンター、産総研：独立行政法人　産業技術総合研究所

独立行政法人 情報処理推進機構
〒113-6591
東京都文京区本駒込二丁目28番8号
文京グリーンコートセンターオフィス16階
http://www.ipa.go.jp

セキュリティセンター
TEL: 03-5978-7527　FAX 03-5978-7518
http://www.ipa.go.jp/security/