[2-1.] SQL 組み立て時の引数チェック

ユーザからの入力を埋め込んで検索の SQL 文を組み立てるということはしばしば行われる。このとき入力データのチェックが甘いと，ユーザは自分の都合の良い SQL 文を混入でき，データベースに干渉できるという問題が起こる。

● ● ●

パスワード変更コマンド

ログイン認証用にデータベースを使用するアプリケーションを想定する。このアプリケーションは**表 1** のような USER_ACL テーブル（ユーザアクセス制御テーブル）でユーザ名とパスワードを管理するものである。テーブル定義は**リスト 1** の SQL 文のとおりである。

表 1　USER_ACL テーブル（ユーザアクセス制御テーブル）のデータ例

USER	PASSWORD
admin	admin
john	lovelydog
michael	foo
smith	paul

リスト 1　USER_ACL テーブルの定義

```
1  CREATE TABLE USER_ACL (
2      USER        CHAR(20) PRIMARY KEY NOT NULL,
3      PASSWORD    CHAR(20)
4  )
```

各ユーザはこのアプリケーションの userpwd というプログラムでパスワードを変更できるものとする。**実行例 1** の 1 行目のように，userpwd コマンドに引数としてユーザ名と現在のパスワードと新しいパスワードを渡す。現在のパスワードが必要なので，（それを知っているのなら別だが）他人のパスワードは変更できない仕組みになっている。2 行目は userpwd コマンド内部で実行される SQL 文を表示しているが，これは本稿の例題として分かりやすくするために，意図的に出力させているものである。この結果，**表 2** のように USER_ACL テーブルの michael のレコードが変更された。

実行例 1　userpwd コマンドの実行例（michael のパスワードを bar へ変更）

```
1  $ ./userpwd michael foo bar                    ← michael のパスワードを bar に変更
2  SQL: UPDATE USER_ACL SET PASSWORD='bar' WHERE USER='michael' and PASSWORD='foo'
3  $
```

表 2　michael のパスワード変更後の USER_ACL テーブル内容

USER	PASSWORD	
admin	admin	
john	lovelydog	
michael	bar	← bar に変更されている
smith	paul	

リスト 2 は userpwd プログラムのソースリストである。Perl で書かれたプログラムで，DBI インタフェース（注1）を使用して MySQL（注2）データベースへアクセスする。

5 ～ 7 行目では，コマンドライン引数からユーザ名，現在のパスワード，新しいパスワードをそれぞれ変数 $user，$curpwd，$newpwd に受け取る。10 行目は DBI インタフェースを使用してデータベースへ接続する（注3）。13 ～ 14 行目で受け取ったパラメタをもとに SQL 文を組み立てて $statement 変数へ代入する。WHERE 句にて USER='$user' と PASSWORD='$curpwd' を and することにより，（そのパスワードそのものを知らない限り）他人のパスワードを変更できないようにしている。

15 行目で組み立てた SQL 文を表示する。18 ～ 19 行目で組み立てた SQL 文を実行する。DBI インタフェースでは SQL 文の実行に，予め SQL 文を prepare( ) 関数で準備しておき，これを execute( ) 関数で実行させるという手順を踏む。22 行目でデータベースとの接続を終了している。

リスト 2　パスワード変更コマンド userpwd

```
 1  #!/usr/local/bin/perl
 2  use DBI;
 3
 4  # コマンド引数を受け取る
 5  $user   = $ARGV[0];                                    # ユーザ名
 6  $curpwd = $ARGV[1];                                    # 現在のパスワード
 7  $newpwd = $ARGV[2];                                    # 新しいパスワード
 8
 9  # データベースへ接続する
10  $dbh = DBI->connect("DBI:mysql:ipa", "user", "password") or die;
11
12  # SQL 文を組み立てる
13  $statement = "UPDATE USER_ACL SET PASSWORD='$newpwd' "
14               ."WHERE USER='$user' and PASSWORD='$curpwd'";
15  print "SQL: $statement¥n";                              # SQL 文を表示
16
17  # SQL 文を実行する
18  $sth = $dbh->prepare($statement) or die;               # SQL 文を準備
19  $sth->execute() or die;                                # SQL 文を実行
20
21  # データベースとの接続を終了する
22  $dbh->disconnect() or die;
```

（**注 1**・通常データベースエンジンごとにプログラミングインタフェースは異なるが，DBI インタフェースを使用すると DBI が対応しているデータベースならどれにでも，統一したプログラミングインタフェースでアクセスできるようになる。プログラマがデータベースごとのインタフェースを覚える必要が無く，またプログラムをほとんど変更することなくデータベースエンジンだけを変更するといったことも可能である。こういったメリットがあるため，DBI インタフェースはデータベースを扱う Perl プログラムでは広く使用されている。）

（**注 2**・MySQL はメジャーなオープンソースデータベースの一つである。MySQL は Linux だけでなく Windows 系ホストでも使用でき，しかも高速で安定しているなどの定評がある。DBI や MySQL については参考文献『Programming the Perl DBI』『MySQL』を参照していただきたい。）

（**注 3**・例題として分かりやすくする目的で，意図的にデータベース接続用アカウント "user" とそのパスワード "password" をプログラム中に直接書いている。本来であれば関連記事『2-2. スクリプトに埋め込まれた DB パスワード』で紹介するようにこれらの情報は別ファイルなどに格納すべきである。）

ダイレクト SQL コマンドインジェクション

リスト 2 の userpwd プログラムは，（そのパスワードそのものを盗まない限り）他人のパスワードの変更はできないように見える。しかし実際には，ダイレクト SQL コマンドインジェクションと呼ばれる攻撃により他人のパスワードも変更できてしまうのである。

ダイレクト SQL コマンドインジェクション攻撃とは，引数などのパラメタに SQL 文を混ぜ込んでおき（インジェクション），プログラム内部でその SQL 文を実行させてしまう攻撃手法である。**実行例 2** の 1 行目がその攻撃例だ。2 つ目の少々複雑なコマンド引数が攻撃文字列で，ダブルクオートでくくられた次の文字列である。

```
"paul' or USER='admin"
```

ダブルクオートでくくることによりシェルに 1 つのパラメタ文字列として認識させている。スペース文字やシングルクオート文字「'」などを含む文字列をプログラムに渡すときに使用するシェルの記述法である。この攻撃文字列により，**リスト 1** の 14 行目の

```
PASSWORD='$curpwd'
```

は次のように展開される。

```
PASSWORD='paul' or USER='admin'
```

結果として**実行例 2** の 2 行目の SQL 文が示しているように WHERE 句は

```
WHERE USER='smith' and PASSWORD='paul' or USER='admin'
```

となり，「USER が smith で PASSWORD が paul であるレコード」および「USER が admin であるレコード」を UPDATE の対象としてしまう。つまり admin ユーザのパスワードが分からなくても，admin ユーザのパスワードを変更できてしまうのだ。実際に**表 3** の USER_ACL テーブルの内容が示すように，admin ユーザのパスワードも変更されてしまった。

実行例 2　ダイレクト SQL コマンドインジェクション（2 行目は折り返している）

```
                引数 1          引数 2          引数 3
                  ↓               ↓               ↓
1  $ ./userpwd smith "paul' or USER='admin" hell
                     ^^^^^^^^^^^^^^^^^^^^
2  SQL: UPDATE USER_ACL SET PASSWORD='hell'
       WHERE USER='smith' and PASSWORD='paul' or USER='admin'
3  $                                    ^^^^^^^^^^^^^^^^^^^^
```

表 3　ダイレクト SQL コマンドインジェクション後の USER_ACL テーブル内容

USER	PASSWORD	
admin	hell	← 他人のパスワードまで変更された
john	lovelydog	
michael	bar	
smith	hell	← 通常どおり変更された

ダイレクト SQL コマンドインジェクションにはこの例の他にも，セミコロン「;」を混入させて任意の SQL 文を続けて実行させたり，ストアドプロシジャ呼び出し文を混入させて外部コマンドを実行させたりなど，さまざまな応用がある。ダイレクト SQL コマンドインジェクションについては参考文献『Direct SQL Com-

mand Injection』に詳しい。

リスト 2 のように，ユーザ入力値をチェックせずにそのまま SQL 文組み立てに使用してしまうと，任意の SQL 文を混入され実行されてしまう。

入力値チェックを徹底しよう

任意の SQL 文を混入されないためには，入力値チェックを徹底する必要がある。たとえば，予算金額を表す入力値 $yosan を使って

```
SELECT HINMEI, KAKAKU FROM SHOUHIN_TABLE WHERE KAKAKU<=$yosan
```

のような SQL 文を組み立てる場合は次のようにすべきである。

```
1  die if($yosan=~/[^0-9]/);      ←数字以外の文字が含まれていたら強制終了

2  $statement = "SELECT HINMEI, KAKAKU FROM SHOUHIN_TABLE "
3               ."WHERE KAKAKU<=$yosan";
```

1 行目で入力値 $yosan が数値定数であること（数字のみで構成されていること）を確認し，もしそうでなければ die 文（注 4）でプログラムを終了させている。2 ～ 3 行目で SQL 文を組み立てているが，このとき $yosan は数値定数であることが保証されているので，任意の SQL 文を混入される危険はない。

（**注 4**・ここでは分かりやすくするために die 文を使用しているが，実際のアプリケーションでは適切なエラー処理を組み込む必要があることに注意していただきたい。）

また商品コードを扱うような場合はまず商品コード形式にしたがい正しい商品コードであるかどうかをチェックする。たとえば次のような形式であるとする。

```
例：ABT-9800134
```

- 11 文字で構成される
- 先頭 3 文字は大文字のアルファベット
- 4 文字目はハイフン「-」
- 残りの 7 文字は数字

この場合は次のようにすべきである。1 行目では商品コードの入力値 $code を正しい形式になっているかどかを確認し，そうでない場合は die 文でプログラムを終了させている。

```
1  die if($code!~/¥A[A-Z]{3}-[0-9]{7}¥z/);  ←形式の規則に合わないときは強制終了

2  $statement = "SELECT HINMEI, KAKAKU FROM SHOUHIN_TABLE "
3               ."WHERE CODE='$code'";
```

このように入力値の形式に規則があるものは，「その入力値が正しい形式であるかどうか」を判断することにより，任意の SQL 文の混入を避けることができる。入力値チェックの一般的手法については関連記事『9-2. 入力値チェックの手法』を参照していただきたい。

しかし人名などの漢字文字コードを扱う場合，本節で紹介した手法では正しい形式かどうかを判断するのは難しい。次節では正しい形式かどうかの判断が困難な入力値を扱う場合について触れる。

入力文字列はエスケープしよう

人名など任意文字を許可する入力文字列を扱う場合，これが任意の SQL 文として機能しないようにエスケープする必要がある。SQL 文の文字列定数では次のような文字が特別な意味を持つ。**実行例 2** では「'」を悪用して SQL 文を混入させていた。

- 「'」　文字列定数の終端
- 「¥」　エスケープ文字

これらの特殊文字を SQL 文の文字列定数中で「'」や「¥」というそのままの文字として表現するためには，それぞれ次のように表記する必要がある。この文字の置き換えをエスケープと呼ぶ。

- 「'」　→　「''」
- 「¥」　→　「¥¥」

たとえば「That's price is ¥200.」という文字列を INSERT するときは，

```
INSERT INTO MSG_TABLE VALUES ( 'That''s price is ¥¥200.' )
```

という SQL 文でなければならない。Perl プログラムでは次のようになる。

```
1  $msg =~ s/'/''/g;              # ←「'」を「''」に置換
2  $msg =~ s/¥¥/¥¥¥¥/g;          # ←「¥」を「¥¥」に置換
3  $statement = "INSERT INTO MSG_TABLE VALUES ( '$msg' )";
```

以上一般的な SQL におけるエスケープ処理について説明したが，実は不十分な対策である。残念なことにデータベースエンジンごとに SQL が独自拡張されていて，特殊文字が「'」「¥」以外にも存在する。

たとえば MS SQL Server では「'」「¥」に加えて，次の文字も特殊文字として扱われる。

- 「|」　外部コマンド実行文字　←シェルを起動できてしまう

また MySQL では，次の文字も特殊文字として扱われる。

- 「¥0」 ナルバイト文字
- 「¥n」 改行文字
- 「¥r」 キャリッジリターン文字
- 「¥z」 ファイル終端文字（Windows のみ）

このためデータベースエンジンごとに適切なエスケープ処理の仕方は異なる。使用するデータベースエンジンごとに，特殊文字をリストアップし，正しくエスケープ処理を実装する必要がある。

Perl の DBI インタフェースには，データベースエンジンごとの違いを吸収するエスケープ機能がある。次節ではこれについて触れる。

DBI インタフェースの quote( ) 関数

Perl の DBI インタフェース（注1）の quote( ) 関数を使用すると，各種データベースエンジンの違いを意識しなくても，適切なエスケープ処理が可能だ。DBI インタフェースが各種データベースエンジンの違いを吸収してくれるため，エスケープしたいときには quote( ) 関数を単純に呼び出すだけでよい。**リスト 3** は quote( ) 関数を使用したサンプルプログラムである。5 行目で，このプログラムへの引数を quote( ) 関数に渡し，そのエスケープ結果を $string 変数で受け取っている。6 行目の print 文で $string の文字列を表示している。

リスト 3　DBI インタフェースの quote( ) 関数を使用したサンプルプログラム

```
1  #!/usr/local/bin/perl
2  use DBI;
3
4  $dbh = DBI->connect("DBI:mysql:ipa", "user", "password") or die;
5  $string = $dbh->quote($ARGV[0]) or die; ←データベースに適した方法でエスケープ
6  print "$string¥n";                      ←エスケープ結果を表示
7  $dbh->disconnect() or die;
```

実行例 3 にこのサンプルプログラムの実行例を示す。1 行目でサンプルプログラムに「'」と「¥」を含む文字列を渡しており，2 行目にそのエスケープ結果の文字列が表示されている。「That's」のエスケープ結果が「That''s] ではなく「That¥'s」であることに注意していただきたい。一般的に言えば，SQL では「'」は「''」へエスケープすべきである。しかし「¥'」も同じシングルクオート文字を意味するので，実は「'」を「¥'」に変換してもよい。DBI インタフェースの実装はこのような「くせ」をもつが，その機能としては必要十分である。

実行例 3　DBI インタフェースの quote( ) 関数の実行例

```
1  $ ./quote "That's price is ¥200."
2  'That¥'s price is ¥¥200.'          ←エスケープされている
3  $
```

以上，SQL 文の組み立てにおいて，エスケープ処理の必要性およびその手法について説明した。実はもっと手軽で便利なバインドメカニズムがある。次節ではこれについて説明する。

バインドメカニズムを活用しよう

リスト 4 はバインドメカニズムを利用した**リスト 2** の改良版だ。入力変数を使った SQL 文の組み立てがないことに注意していただきたい。

13 ～ 14 行目の prepare( ) 関数に直接 SQL 文が渡されているが, この SQL 文には入力変数が使われていない。代わりに「?」が配置されている。この「?」を「プレースホルダ」と呼ぶ。execute( ) 関数を利用してこの部分に変数値を当てはめることができる。15 行目の execute( ) 関数に引数として 3 つの入力変数 $newpwd, $user, $curpwd が渡されていることに注意していただきたい。この 3 つの変数の値が 14 行目の SQL 文中のプレースホルダ「?」の部分に自動的に当てはめられる。変数の値がプレースホルダ「?」部分に結び付けられる（バインドされる）ので，これらの変数を「バインド変数」と呼ぶ。

リスト 4　DBI インタフェースにおけるバインドメカニズムの使用例

```
 1  #!/usr/local/bin/perl
 2  use DBI;
 3
 4  # コマンド引数を受け取る
 5  $user   = $ARGV[0];                                    # ユーザ名
 6  $curpwd = $ARGV[1];                                    # 現在のパスワード
 7  $newpwd = $ARGV[2];                                    # 新しいパスワード
 8
 9  # データベースへ接続する
10  $dbh = DBI->connect("DBI:mysql:ipa", "user", "password") or die;
11
12  # SQL 文を実行する
13  $sth = $dbh->prepare(
14          "UPDATE USER_ACL SET PASSWORD=? WHERE USER=? and PASSWORD=?") or die;
15  $sth->execute($newpwd, $user, $curpwd) or die;
16
17  # データベースとの接続を終了する
18  $dbh->disconnect() or die;
```

ここで注意すべきことは，バインド変数の値は文字列として単純に当てはめられるのではなく，完全な数値定数や文字列定数として組み込まれる。たとえ変数の値に「'」や「¥」といった特殊文字が含まれていても，特殊文字としてではなくただの文字として扱われる。

バインド変数の値をプレースホルダに当てはめる場合の内部的な処理は，データベースエンジンによって異なる。Oracle のようにデータベースエンジンそのものがバインドメカニズムを持っている場合は，データベースエンジン側にプレースホルダ用メモリ領域が用意され，そこにバインド変数値が直接渡される。組み立てた SQL 文として渡されるのではなく，SQL 文の要素となる値として渡される。したがってバインド変数の値が SQL 文として機能することはない。一方 MySQL のようなバインドメカニズムを持たないデータベースエンジンの場合，DBI インタフェースなどのデータベースライブラリが内部的にバインド変数をエスケープして SQL 文を組み立て，その SQL 文をデータベースエンジンへ渡す。どちらにしても，プログラマからはプレースホルダにバインド変数が数値定数や文字列定数としてそのまま当てはめられるとだけ理解しておけばよい。

このようにバインドメカニズムは任意の SQL 文の混入を避けることができる大変便利な機能である。本稿では Perl の DBI インタフェースにおけるバインドメカニズムを紹介したが，この他にも独自のバインドメカニズムを持つデータベースエンジンもある。しかしすべてのデータベースエンジン，データベースライブラリでバインドメカニズムが利用できるとは限らない。もしバインドメカニズムを利用できるシステムであれば，積極的にバインドメカニズムを活用していただきたい。利用できない場合は前節までで紹介した手法を駆使してほしい。

LIKE 句では % と _ に気をつけよう

最後に LIKE 句について触れておく。SQL 文の LIKE 句では「%」と「_」は次のようなワイルドカードを表す特殊文字である。この 2 つの文字はバインドメカニズムを使っても，ただの文字としてではなくワイルドカードとして扱われる。

- 「%」　任意数の任意の文字にマッチ
- 「_」　1 文字の任意の文字にマッチ

LIKE 句において，この 2 つの文字をワイルドカードとしてではなくただの「%」や「_」の文字として扱いたい場合は，次のようにエスケープする必要がある。

- 「%」　→　「¥%」
- 「_」　→　「¥_」

バインドメカニズムを利用した場合でも，プレースホルダが LIKE 句に含まれる場合は，バインド変数を予め上述のようにエスケープする必要があることに注意していただきたい。

まとめ

SQL 文の組み立てにおいてユーザ入力値をそのまま使用してしまうと，任意の SQL 文を混入され自在にデータベースに干渉されてしまう。バインドメカニズムを活用することで任意の SQL 文を混入される危険を回避できる。もしバインドメカニズムが利用できない場合は，入力値チェックの徹底および文字列定数エスケープの徹底によって，バインドメカニズム同様に危険を回避できる。

関連記事

『2-2. スクリプトに埋め込まれた DB パスワード』

『9-2. 入力値チェックの手法』

参考文献

『Programming the Perl DBI』（英文），Chapter 4: Programming with DBI
http://www.oreilly.com/catalog/perldbi/chapter/ch04.html

『MySQL』（英文）
http://www.mysql.com/

『Direct SQL Command Injection』（英文），Open Web Application Security Project
http://www.owasp.org/projects/asac/iv-sqlinjection.shtml