



# PostgreSQL セキュリティ入門

安全なアプリケーション構築の必須知識

日本PHPユーザ会 / JPUG四国支部  
大垣 靖男 / yohgaki@ohgaki.net

---

# 接続と認証

- pg\_hba.conf
  - データベースにアクセス可能なホストを設定
- UNIXドメインとTCP接続
  - ローカルホストの場合、UNIXドメインの方が高速
- 書式

```
local  DATABASE USER METHOD [OPTION]  
host   DATABASE USER CIDR-ADDRESS METHOD [OPTION]  
hostssl DATABASE USER CIDR-ADDRESS METHOD [OPTION]  
hostnossl DATABASE USER CIDR-ADDRESS METHOD [OPTION]
```



# pg\_hba.confの書式

- USER
  - 個別、カンマ区切り、グループ(+プレフィックス)
  - @プレフィックスで別ファイルから読み込み
- METHOD
  - trust, reject, md5, crypt, password, krb5, ident, pam
  - crypt, passwordは使わずmd5を利用
  - Webアプリケーションからのアクセスにtrustは絶対に利用しない
    - 例外: DBMS自体を管理するPhpPgAdminのようなアプリ

# 権限

- 権限管理は8.0からロールベース

CREATE USER name

は

CREATE ROLE name LOGIN

と同等

- DBオブジェクトに対して権限を設定可能

SELECT, INSERT, UPDATE, DELETE, REFERENCES, TRIGGER, CREATE, CONNECT,  
TEMPORARY, EXECUTE, USAGE

- Webアプリケーションには不必要な権限を与えない
  - 公開サイトで参照のみ可能なWebアプリケーションならDBユーザはSELECT権限のみ与える
  - 削除が不必要ならDELETE権限は与えない

# ロールの切り替え

- ロール権限の継承は制御可能

```
CREATE ROLE joe LOGIN INHERIT;  
CREATE ROLE normal NOINHERIT;  
CREATE ROLE admin NOINHERIT;  
GRANT normal TO joe; GRANT admin TO normal;
```

- ログイン直後はjoeの権限とnormal権限を利用可能
  - 間接的に利用可能なwheel権限は利用不可
- wheelの権限で動作させる

```
SET ROLE wheel;
```

- アプリケーションでより大きな権限が必要な場合にのみ利用することも可能

# スキーマ

- PostgreSQLは接続後にデータベースの切り替えはできない
  - 異なる権限でデータベースを共有する場合に schemaの利用が効果的

```
CREATE SCHEMA schemaname AUTHORIZATION username;
```

- スキーマの利用

```
SELECT * FROM schemaname.tablename;
```

```
SET search_path schemaname;  
SELECT * FROM tablename;
```

# リソース

- データベースクエリの結果はすべてクライアントに送信される
  - 100万件のレコードがSELECTされた場合
    - 100万件のレコードを保持するメモリが必要
  - サーバへの負荷も大きい
- LIMIT句

```
SELECT * FROM table LIMIT 1000 OFFSET 0;
```



# SQLインジェクション

- SQLインジェクションは比較的簡単に防御可能なアプリケーションの脆弱性
  - 直接SQLインジェクション
  - 間接SQLインジェクション
  - ブラインドSQLインジェクション
- SQLインジェクション対策の基本
  - ユーザ入力をSQL文として実行させない

# SQLインジェクション

- アプリケーションの動作の変化で検出
  - SOME\_VALUE or 1=1– など
- データベースの構造を知らなくても攻撃可能
  - ブラインドSQLインジェクション
- ユーザ入力以外のデータからも攻撃可能
  - 間接SQLインジェクション(セカンドオーダーSQLインジェクション)

# PostgreSQLのSQL文実行

- libpqのPQexec関数は複数のSQL文を同時に実行可能
- 例えば、以下のPHPスクリプトはtableを削除可能

```
<?php
$sql = "SELECT * FROM table; DROP TABLE table;";
pg_query($sql);
?>
```

- Javaの場合、複数クエリは実行されず、最初のクエリのみ実行される

# PostgreSQLのSQL文実行

- IF文をサポートしていないのでIF文を利用したブラインドSQLインジェクションは不可能

– SQL Serverの例

```
if ((select user) = 'sa') select 1 else select 1/0
```

– ユーザがsa (SQL Serverのシステム管理アカウント) 以外なら例外エラーが発生

- ASCII()関数を利用したブラインドSQLインジェクションは可能

```
ascii(lower(substring((SELECT tablename FROM pg_tables WHERE  
schemaname='public' LIMIT 1), 1, 1))) > 109
```



# SQLインジェクション対策

- 文字エンコーディングのチェック
- 全ての文字列のエスケープ
- プリペアドクエリの利用

# 直接SQLインジェクション

- 例えば

- <http://example.com/search?id=123>

- idが「123%3BDROP%20TABLE%20product」の場合

```
$sql = "SELECT * FROM product WHERE id = ".$_GET['id'].>";  
pq_query($sql);
```

- \$sqlの内容

```
SELECT * FROM product WHERE id = 123;DROP TABLE product
```

# 直接SQLインジェクション: 文字エンコーディングを利用した攻撃

- 攻撃対象SQL

```
SELECT * FROM table WHERE name = 'STR1' and type = 'STR2';
```

- 'STR1', 'STR2'の部分がユーザ入力、文字エンコーディングがSJISな場合

```
http://example.com/search.php?name=%95&type=%3BDROP%20TABLE%20table%3B%20--
```

- 実行されるSQL文

```
SELECT * FROM table WHERE name = '表 and type = ' ; DROP TABLE table; --  
STR2';
```

# 直接SQLインジェクション： 文字エンコーディングを利用した攻撃

- マルチバイト文字は不正な入力でシステムを誤作動させる
  - SJISの「表」は0x95と0x5C。SQL文が生成された時点で「SELECT \* FROM table WHERE name = '表 and type = ' ; DROP TABLE table; --STR2';」は正しい文字エンコーディングかつ正しいSQL文
- SJIS以外もマルチバイト文字は潜在的な危険性をもっている
  - 不正な文字エンコーディングはアプリケーションによって誤って解釈される可能性がある

# 間接SQLインジェクション

- DBMS等に保存された値を利用した攻撃
  - ユーザ登録SQL

```
INSERT INTO user (username , password) VALUES ( 'admin'--, 'pass');
```

## – パスワード変更SQL

```
UPDATE user SET password = 'cracked' WHERE username = 'admin'-- AND password = 'pass';
```

## – 任意のパスワードに設定....

# セキュアコーディングプラクティス

- ユーザ入力に限らず全ての変数を文字列としてエスケープ

```
$sql = "SELECT * FROM product WHERE id > ".escape($_GET['min'])." AND id < ".escape($_GET['max']).";";
```

- テーブル、フィールド名など、DBオブジェクト名をユーザ入力で設定する場合、ホワイトリストを使用する

# セキュアコーディングプラクティス

- 全てのSQLにプリペアドクエリを使用する

```
pg_query_params('SELECT * FROM table WHERE param1 = $1 OR param2 = $2 OR param3 = $3', $_POST);
```

- 動的クエリ生成と同じく、テーブル・フィールド名などにユーザ入力を利用する場合は必ずホワイトリストを利用する

# セキュアコーディングプラクティス

- 絶対にブラックリスト方式で安全性を確保しない

```
DR/**/OP TABLE table;
```

- PostgreSQLでは動作しないが、SQL Server、MySQLではtableがDROPされる
- 許可しないのであれば権限を調整。ユーザを変えてSELECT権限のみ許可するなど
  - 注意: PHPの永続的接続は接続パラメータの違いをみる。接続数不足に注意が必要



# セキュアコーディングプラクティス

- ORMを利用していてもプログラマにはDBMSとSQLの基礎知識が必須
  - ORMでもSQLの一部を設定するケースはある
  - SQLを全く使わないとDoSやパフォーマンスに問題が発生する

# セキュアコーディングプラクティス

- エスケープ関数の例

```
function sql_escape($val)
{
    return ""'.pg_escape_string($val)."";
}
```

少なくともPostgreSQLは‘(シングルクォート)で数値を囲っても数値型の場合でも数値として扱われ、形式が不正な場合はエラーとなる

```
function sql_escape_bytea($val)
{
    return ""'.pg_escape_bytea($val)."";
}
```

# セキュアコーディングプラクティス

- プリペアドクエリの利用
  - プリペアドクエリのパラメータは全てクエリの変数として扱われる
    - すべての変数をエスケープするのと同じ効果

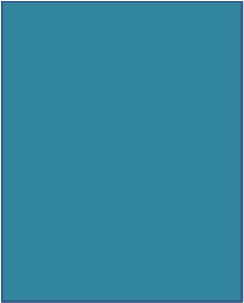
```
$sql = "SELECT * FROM table WHERE pram1 = $1 AND $param2 = $2";  
pg_query_params($sql , $_POST); // バリデーション無しの$_POSTでも安全
```

# セキュアコーディングプラクティス

- 「全て」の入力のバリデーションを行う
  - 文字列長さ
  - 値の範囲
  - 文字列の形式(正規表現で確認しなくてもよい場合、正規表現は使わない)
- バリデーションを行った値でも「全て」の変数をエスケープ処理
  - セカンドオーダー攻撃(間接攻撃)
  - 複数階層のセキュリティ
- SQLインジェクション対策のみでなく、XSS対策も同じ

# セキュアコーディングプラクティス

- PHPで正規表現によるバリデーションを行う場合、`mbstring`の正規表現を利用する
  - バイナリセーフ
  - デフォルトで“^”、“\$”が文字列全体の先頭、末尾にマッチ
    - 内部的に^ -> ¥A(文字列先頭), \$ -> ¥z(文字列末尾)の変換が行われる
  - POSIX正規表現(`ereg`関数など)は非バイナリセーフ
  - PCRE正規表現はデフォルトでは“\$”は行末にマッチ
    - PHP独自のDモディファイアを利用すると文字列末尾にマッチ
      - PerlにはないPHP独自の拡張
    - “^”を強制的に文字列全体の先頭にマッチさせるモディファイアはない
      - 正規表現によっては文字列先頭にマッチしないことも



ご静聴ありがとうございました