SQLインジェクション”ゼロ”のPostgreSQLの利用方

今更聞けないSQLインジェクションの現実と対策

~~Definitive~~ Guide to SQL Injections for PostgreSQL

自己紹介

- 氏名：大垣靖男
- ユーザ会：日本PostgreSQLユーザ会、他
- 会社：エレクトロニック・サービス・イニシアチブ
- メール：yohgaki@ohgaki.net
- ID: **yohgaki** (Twitter、Facebook、mixi他)

SQLインジェクションとは

- プログラマが意図していない、不正なSQL文を実行させる攻撃

何故、不正なSQL文が実行されるのか？

- SQL文を実行させるコードの不備が”主な”原因

SQLインジェクション対策

- 出力時に対策が完全であることを補償できる

SQLインジェクション対策

- SQL文を実行させる場合の不備が”主な”原因
- 正しくSQL文を実行すればインジェクションは発生しない
- 出力時に対策が完全であることを補償できる

SQLインジェクションは100%防げる脆弱性

完全な
SQLインジェクション対策

- プリペアードクエリだけを使う
- ORMだけ使う

ご清聴

ありがとうございました

不完全な
SQLインジェクション対策

- プリペアードクエリだけを使う
- ORMだけ使う

SQLインジェクション

の復習

SQLインジェクション

- SQL文の構造を壊し、本来実行すべきでないSQL文を実行させる攻撃
  - **直接SQLインジェクション**
  - **間接SQLインジェクション**
- よく見るSQLインジェクション攻撃例
  - **http://example.jp/login?user=admin&pass='+or+'a'='a**

SQLインジェクション の常識

- データベース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**
- ツールは色々出回っている

本物のブラインドSQLインジェクション攻撃例

- 本物の攻撃者は手作業などで攻撃しない

```
#!/usr/bin/perl

#////////////////////////////////\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\#
#\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\////////////////////////////////#
#                                                                        #
# [o] TPDugg Joomla Component 1.1 Blind SQL Injection Exploit       #
#                                                                        #
#      Software : com_tpdugg version 1.1                                 #
#      Vendor   : http://www.templateplazza.com/                         #
#      Author   : NoGe                                                   #
#      Contact  : noge[dot]code[at]gmail[dot]com                         #
#      Blog     : http://evilc0de.blogspot.com - http://pacenoge.org #
#                                                                        #
# [o] Usage                                                          #
#                                                                        #
#      root@noge:~# perl tpdugg.pl                                       #
#                                                                        #
#                                                                        #
#      [+] URL Path : www.target.com/[path]                              #
#      [+] Valid ID : 1                                                  #
#      [+] Column   : username                                           #
#                                                                        #
```

本物のブラインドSQLインジェクション攻撃例

- 本物の攻撃者は手作業などで攻撃しない

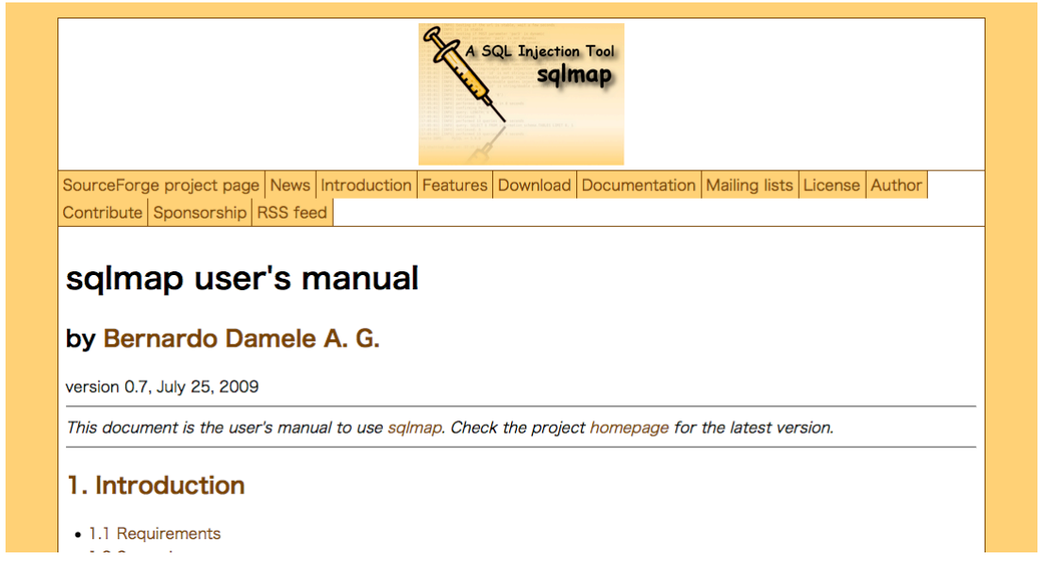
```
#!/usr/bin/perl

#////////////////////////////////\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\#
#\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\////////////////////////////////#
#                                                                        #
# [o] TPDugg Joomla Component 1.1 Blind SQL Injection Exploit      #
#                                                                        #
#    Software : com_tpdugg version 1.1                                   #
sub exploit() {
    my $limit = $_[0];
    my $chars = $_[1];
    my $blind = '+AND+ASCII(SUBSTRING((SELECT+'.$column.'+FROM+'.$table.'+LIMIT+0,1),'.$limit.',1))='.$chars;
    my $inject = $target.$vuln.$id.$blind;
    my $content = get_content($inject);
    return $content;
}
#                                                                        #
#    root@noge:~# perl tpdugg.pl                                         #
#                                                                        #
#                                                                        #
#    [+] URL Path : www.target.com/[path]                                #
#    [+] Valid ID : 1                                                    #
#    [+] Column   : username                                             #
#                                                                        #
```

SQLインジェクション
脆弱性は致命的

- データベースを丸ごと盗める
- 書き込み可能なデータベースを全て改ざんできる
- ファイルも読み書き、コマンドも実行できる
- 攻撃者にはXSSなどと比べられない
  「おいしい脆弱性」

sqlmapによる解析デモ

- 教科書通りのログイン画面に脆弱性が
  あるアプリを**sqlmap**で解析
  - DB：**PostgreSQL**
  - Web：**Apache+PHP**



sqlmapで可能な攻撃

- データベース種別の検出（PostgreSQL, MySQL, Oracle, SQL Server)
- Webシステムの検出（OS, Webサーバ、言語環境など）
- 列挙（詳細なDBサーバ情報のダンプ）
- アクセスしているユーザ名と権限
- DBユーザのパスワードハッシュのダンプ

sqlmapで可能な攻撃

- DBユーザ権限の一覧
- 利用可能なDBの一覧
- テーブルの一覧
- テーブル定義の一覧
- 任意のSQL文の実行
- ファイルシステムへの読み書きアクセス
- OSコマンドの実行

sqlmapで可能な攻撃

- コマンドラインの取得
- リモートホストのシェル取得
- 権限の昇格
- バッファーオーバーフローの攻撃
- セッションの保存と再開

sqlmapデモ

SQLインジェクション

脆弱性は**致命的**

- **データベースを丸ごと盗める**
- **書き込み可能なデータベースを全て改ざんできる**
- **ファイルも読み書き、コマンドも実行できる**
- **攻撃者にはXSSなどと比べられない**

「おいしい脆弱性」

なぜSQLインジェクション

脆弱性が無くならないのか？

原因

- 「セキュリティ基礎知識の理解不足」が”主な”原因
- 「危険性の理解不足」も原因の一つ？

- アプリケーションプログラマの責任（原因）である事が多いが、アプリケーションプログラマの責任でない場合もある

セキュリティ基礎知識

- SQLインジェクション対策に必要なセキュリティの基礎知識

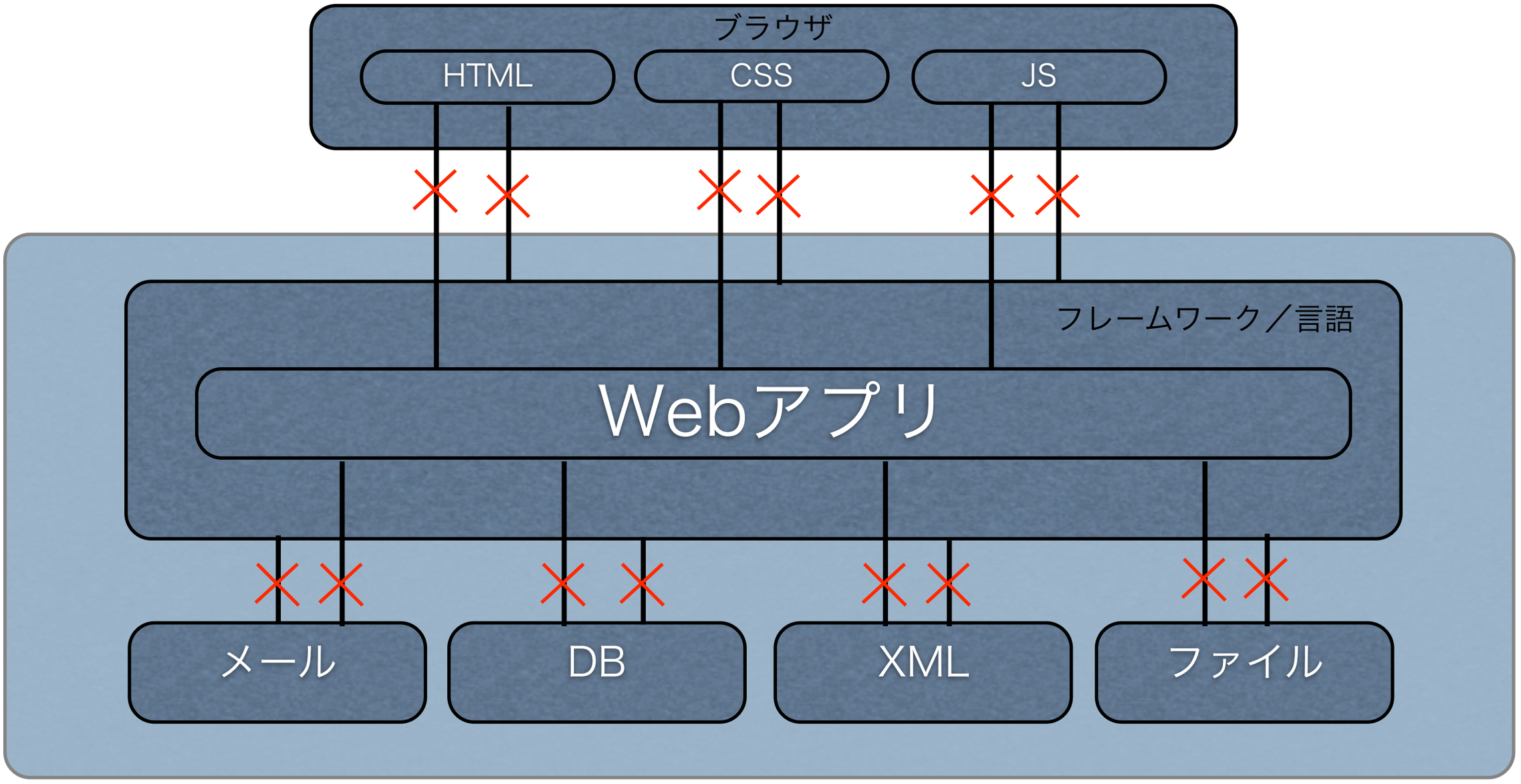
- **徹底したトラストバウンダリの管理**
- **多重のセキュリティ対策**
- **厳格な文字エンコーディング処理**

トラストバウンダリ

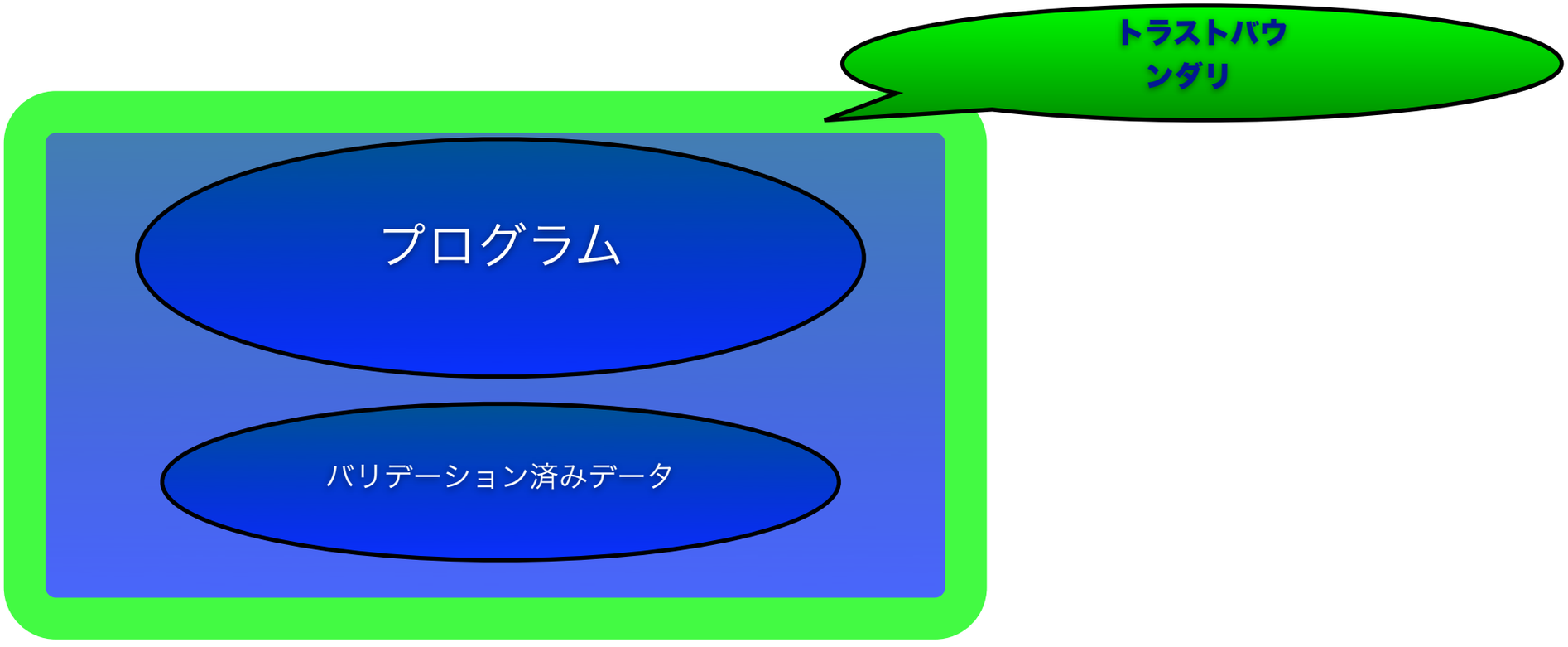
トラストバウンダリ

- **プログラムが信用できるデータやプログラムの境界**
- 最も重要なセキュリティの基礎知識。
  なぜか広く理解されていない?!
- 当たり前過ぎて自分のセキュリティ本（Webアプリセキュリティ対策入門 - 技術評論社）でも詳しく解説していない事が問題?!

Webアプリの攻撃箇所

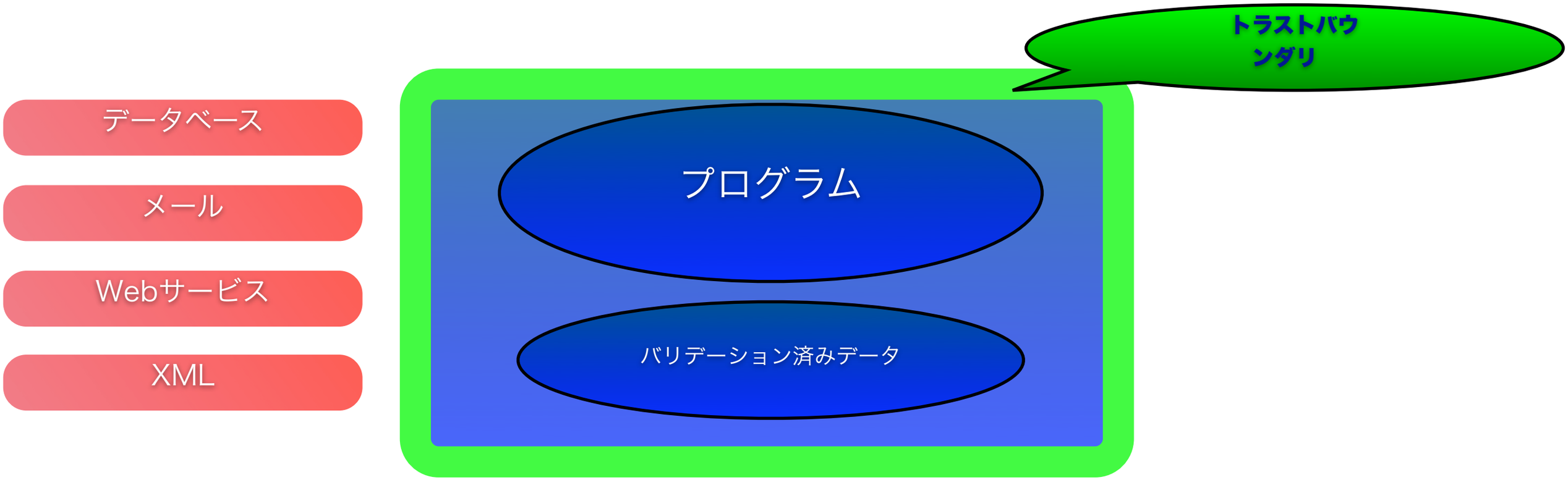


トラストバウンダリとは？



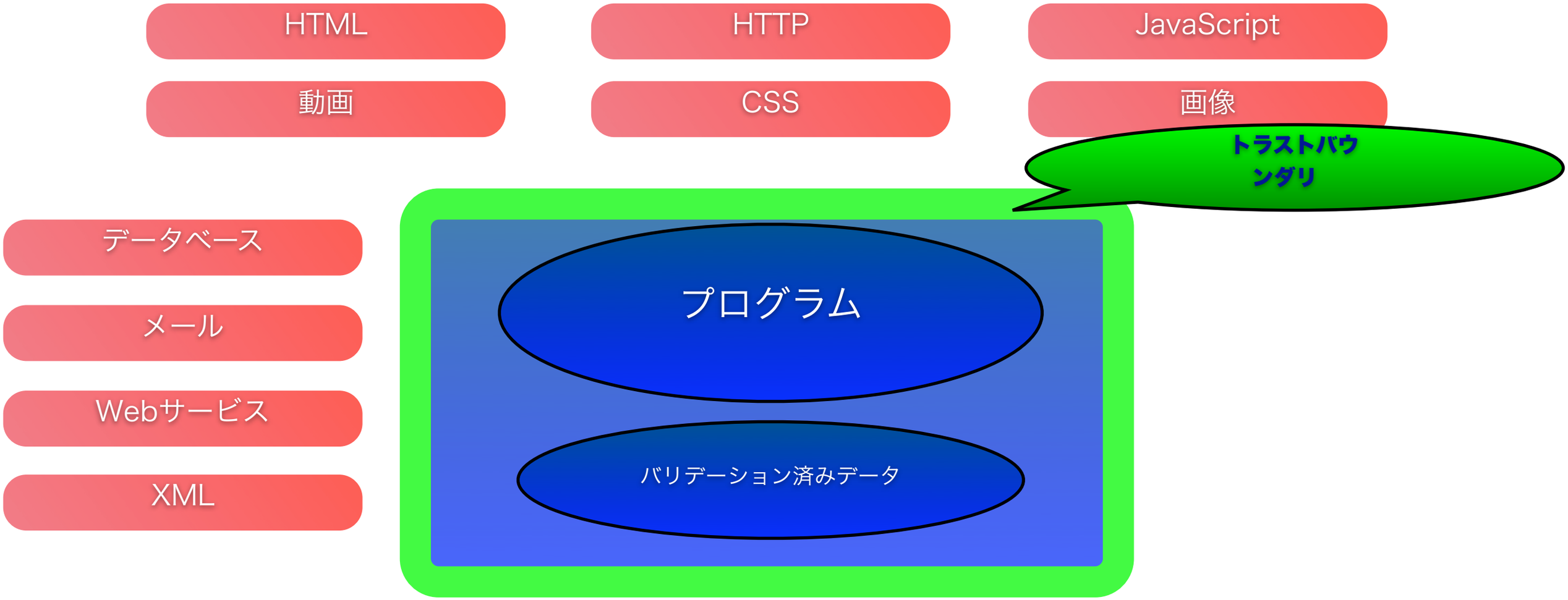
バリデーションされていないデータとプログラムは、「**全て信用できない**」データとプログラム

トラストバウンダリとは？



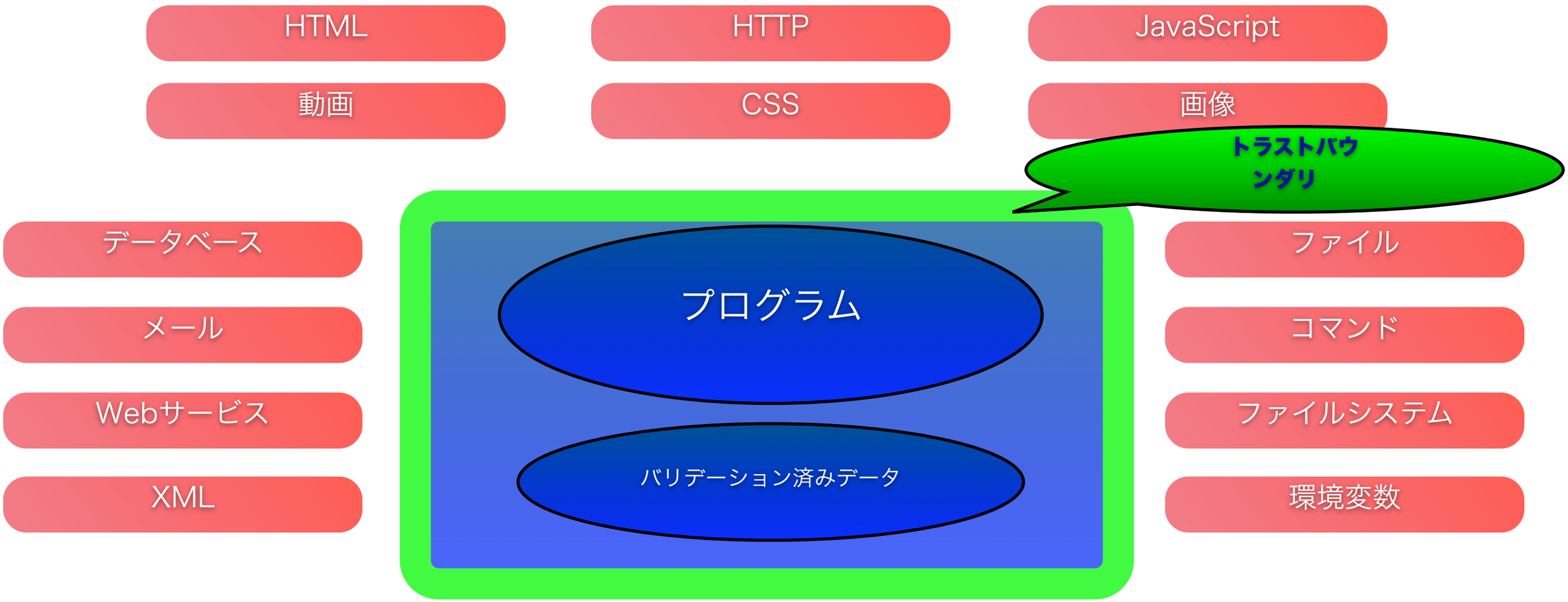
バリデーションされていないデータとプログラムは、「**全て信用できない**」データとプログラム

トラストバウンダリとは？



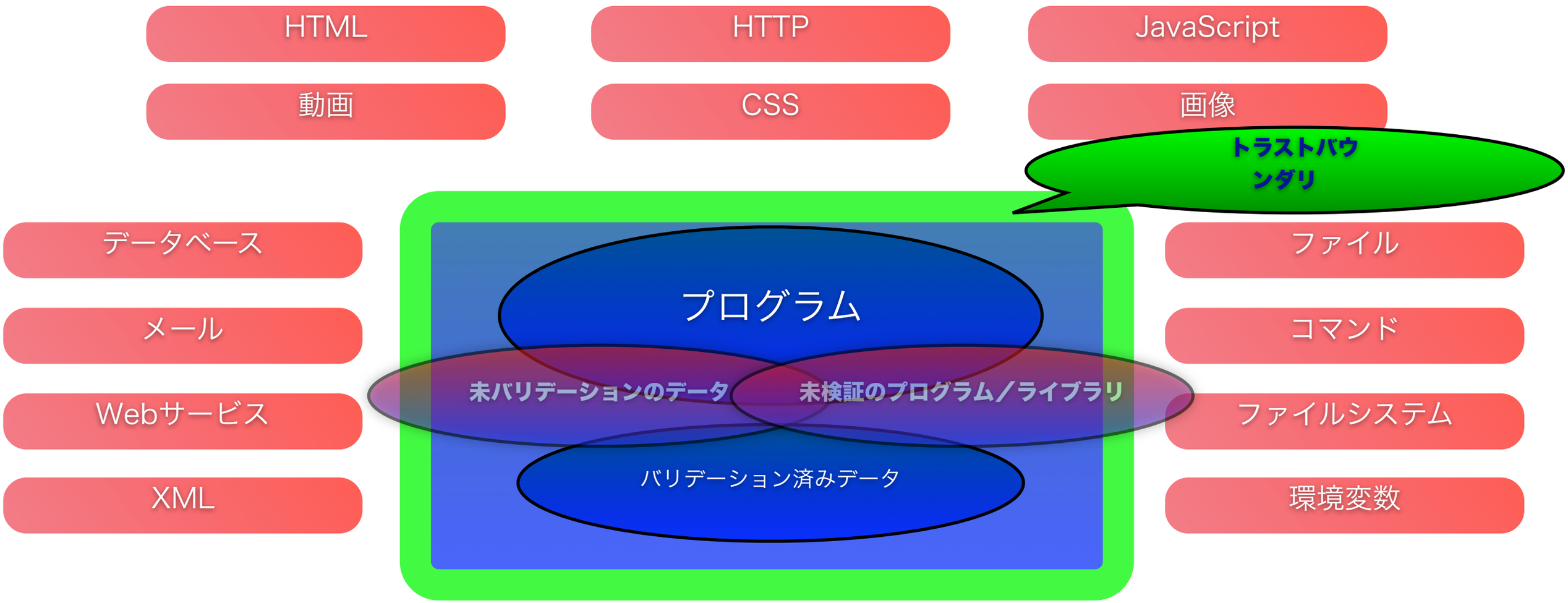
バリデーションされていないデータとプログラムは、「**全て信用できない**」データとプログラム

トラストバウンダリとは？



バリデーションされていないデータとプログラムは、「**全て信用できない**」データとプログラム

トラストバウンダリとは？



バリデーションされていないデータとプログラムは、「**全て信用できない**」データとプログラム

トラストバウンダリ

- **入力**：自分のプログラムが誤作動しないように、
  バリデーション
- **出力**：出力先のプログラムが誤作動しないように、
  正しく出力
- **SQLインジェクションは出力対策だけで**
  100%防御可能　＝　SQLインジェクション対策は簡単

多重のセキュリティ対策

多重のセキュリティ対策

- セキュリティ対策を繰り返し行う事
- “**このパラメータは数値だからエスケープなし?!**”

- “SELECT name FROM user WHERE **id = $id**”

- **セキュリティ対策の基本は”無駄”**
  → 効率ばかりを求めない

類似例

- **”このパラーメータはテーブル/フィールド名だからエスケープなし？！”**
  - “SELECT **$field** FROM **$table** WHERE type = 1”
- **“これはSQL文の一部だからエスケープなし？！”**
  - “SELECT id FROM table WHERE type=1 ORDER BY **$order**”

文字エンコーディング

文字エンコーディング

- 文字エンコーディングは「**セキュリティ対策の”要”**」の一つ
  - 不正な文字エンコーディングや間違った文字列操作や設定
    - SQLインジェクション以外にもJavaScriptインジェクションなどの攻撃に利用可能

なぜ文字エンコーディング攻撃が可能なのか？

- 最も解り易いSJISの例
  - PostgreSQL 7.4まで文字列エスケープに”**¥**”を利用
  - **表** は**SJIS**では**0x95,0x5C**
  - **0x5C = “¥”**
  - **“表”をエスケープ → 0x95, “¥”, “¥” ＝ “表¥”**
  - ‘安全対策一覧表’ → ‘安全対策一覧　**¥’**

文字エンコーディングを利用したSQLインジェクション

- **現在のPostgreSQLではほぼ解決済み**
  - ただし、自分の足を打つ事が可能な場合も！

- **他のシステムの攻撃（XSS、DoSなど）に悪用されるのでエンコーディングは厳格に！**
  - 悪い例：RHELのデフォルト文字エンコーディングは **ASCII**

libpqから学ぶ

SQLインジェクション対策

libpqとは？

- PostgreSQLのクライアントライブラリ
  - データベースへの接続や**クエリの実行**、**エスケープ**を行うライブラリ
- 記述言語：　C

libpqを使った安全なクエリの実行

- プリペアードクエリ
  - **Step1 PQprepareでSQL文をプリペア**
  - **Step2 パラメータとプリペア済みSQLをPQexecuteで実行**
- パラメータのエスケープ
  - **Step1 パラメータをPQescapeString(Bytea)Connでエスケープしてsql文を作成**
  - **Step2 PQexecでSQL文を実行**

libpqを使ったI/F

PHP	pgsqlモジュール、PDO pgsql (OO)
Ruby	pgモジュール、DBD::pg
Perl	pgモジュール、DBD::pg
Python	PyGreSQL - pg, pgdb(OO)
Java	JDBCドライバはpure Java(TypeIV)
.NET	nmake一発、 Npgsqlはpure C#

補足

- ruby-postgresqlは古い
- DBD::PgPPはPure Perl

libpqを理解すれば

- **PHP, Ruby, Perl, Pythonの対策は完璧**

SQLインジェクションを防ぐlibpqの使い方

- 正しいエスケープ処理を行う
  - 必ず**PQescapeStringConn**, **PQescapeByteaConn**を利用
  - DB接続構造体がクライアント**文字エンコーディング設定を保持**
  - 接続前にエスケープしない

PQescapeStringConn

- size_t **PQescapeStringConn** (
  PGconn *conn,
  char *to,
  const char *from,
  size_t length,
  int *error);

PQescapeByteaConn

- unsigned char
  *PQescapeByteaConn(
  PGconn *conn,
  const unsigned char
  *from,
  size_t from_length,
  size_t *to_length);

使ってはならない古いAPI

- **PQescapeString**
- **PQescapeBytea**

- 言語レベルからは見分けが付け辛い場合も...

SQLインジェクションを防ぐlibpqの使い方

- クライアント文字エンコーディング
  - 必ずAPIで変更
    - **PQsetClientEncoding**を使用する
- 絶対にSQL文で変更しない
  - **SET NAMES, set**

SQLインジェクションを防ぐlibpqの使い方

- プリペアードクエリを利用する
  - **PQprepare**, **PQexecPrepared**
- プリペアードクエリのような実行関数を利用する
  - **PQexecParams** - パラメータを別引数として渡す

libpq関数対応表

libpq	PHP pgsql	Ruby pg	Perl pg	Python pg
PQescapeStringConn **正しいエスケープ**	pg_escape_string	conn.escape_string	**N/A**	pg_escape_string
PQescapeByteaConn **正しいエスケープ**	pg_escape_bytea	conn.escape_bytea	**N/A**	pg_escape_bytea
PQsetClientEncoding **エンコーディング設定**	pg_set_client_encoding	conn.set_client_encoding	**N/A**	**N/A**
PQprepare **クエリの準備**	pg_prepare	conn.prepare	$dbh->prepare	**N/A**
PQexecPrepared **準備したクエリの実行**	pg_excecute	conn.exec_prepared	$dbh->execute	**N/A**
PQsendPrepare **非同期のクエリの準備**	pg_send_prepare	conn.send_prepare	N/A	N/A
PQsendQueryPrepared **非同期の準備したクエリの実行**	pg_send_execute	conn.send_query_prepared	$dbh->execute	N/A
PQexecParams **SQLとパラメータの実行**	pg_query_params	conn.exec	$dbh->execute	N/A
PQsendQueryParams **非同期のSQLとパラメータの実行**	pg_send_query_params	conn.send_query	$dbh->execute	N/A
PQescapeString **使用禁止**	**pg_escape_string**	**PGconn.escape_string**	N/A	**escape_string**
PQescapeBytea **使用禁止**	**pg_escape_bytea**	**PGconn.escape_bytea**	N/A	**escape_bytea**

DB抽象化ライブラリ問題

- DB抽象化ライブラリはDB特有のAPIをサポートしていない（場合が多い）
  - PQsetClientEncoding
  - PQescapeByteaConn
- プリペアードクエリが「なんちゃって」プリペアードクエリの場合もある

独自実装のライブラリ

Perl - PgPP

- Pure Perl実装
- プリペアード文の利用が前提
  - bind_param, execute

Perl - PgPP

```
sub bind_param
{
    my $sth = shift;
    my ($index, $value, $attr) = @_;
    my $type = (ref $attr) ? $attr->{TYPE} : $attr;
    if ($type) {
        my $dbh = $sth->{Database};
        $value = $dbh->quote($sth, $type);
    }
    my $params = $sth->FETCH('pgpp_param');
    $params->[$index - 1] = $value;
}
```

(PgPPのソースより)

文字列型はクオート（エスケープ）している、と思われる。

安全？

Java - JDBCドライバ

- Pure Java実装
  - JDBC 3

- 普通はプリペアードクエリを使う（でしょう）
  - prepareStatementでプリペア
    文、setInt、setLong, setStringなどでパラメータを設定

Java - JDBCドライバ

```
PreparedStatementps=conn.prepareStatement(“INSERT INTO BOOKLIST "+
 "(AUTHOR, TITLE, ISBN) VALUES (?, ?, ?)”);
ps.setString(1, “Zamiatin, Evgenii”);
ps.setString(2, “We”);
ps.setLong(3, 0140185852);
```

- プログラマが正しいデータ型を指定する事を期待したインターフェース
- 最低限でもps.setLong(3, “; DELETE FROM user; --”)などと出来ないように**バリデーションはしていると思われる**

The **JDBC driver is responsible for mapping** this to the corresponding JDBC type (one of the SQL types defined in java.sql.Types) so that **it is the appropriate type to be sent to the data source**.
(JDBC API 3.0 Specificationより)

.NET - Npgsql

- Pure C#実装
- 普通はプリペアードクエリを使う（でしょう）
- prepareStatementでプリペア文、NpgsqlParameterでパラメータを設定

```
// Declare the parameter in the query string
        using(NpgsqlCommand command = new NpgsqlCommand("select * from tablea where column1
= :column1", conn))
        {
            // Now add the parameter to the parameter collection of the command specifying its type.
            command.Parameters.Add(new NpgsqlParameter("column1", DbType.Int32);

            // Now, prepare the statement.
            command.Prepare();

            // Now, add a value to it and later execute the command as usual.
            command.Parameters[0].Value = 4;
```

- データ型を指定するタイプ。PgPP, JDBCと同じ。

間違った

SQLインジェクション対策

間違った
SQLインジェクション対策

- **プリペアードクエリを利用すればOK**
  - テーブル、フィールド名、SQL語句がパラメータな場合も！
  - 「なんちゃって」プリペアードクエリ実装も！
- **やはりエスケープとバリデーションが基本！**
- プリペアードクエリはlibpqのラッパーを使うと一番効率的で確実

間違った
SQLインジェクション対策

- **自前のエスケープ関数でも大丈夫**
  - APIのエスケープ関数を使うべき。
  - Byteaのエスケープも大丈夫？
  - 文字エンコーディングは考慮している？

間違った
SQLインジェクション対策

- **ORMだけ利用すればOK**
  - ORMだけではパフォーマンスが問題
    - ORMから取得したデータをWebアプリでJOINなど問題外
  - SQL文の一部やSQLを直接記述可能なORMも多い
    - HibernateのHSQLやNative SQL

間違った
SQLインジェクション対策

- **キャストは有害**
  - “SELECT f FROM t WHERE id =” . **(int)$id**
  - 一般的にデータベースのID型はINT8（64bit）
  - さらに任意精度型の数値も利用可能
  - SQLインジェクションは防げるがデータを破壊
- **広い範囲の値をINT4, INT8に丸めるのは間違い**

間違った
SQLインジェクション対策

- **テーブル名の推測が難しいようにランダムなプレフィックスをテーブル名に付ける**

- ブラインドSQLインジェクションには無力

- **複雑なDBユーザパスワードは有効**

SQLインジェクション対策

基本を知る

- SQL構文を破壊するパラメータを完全に排除

完全な SQLインジェクション対策

- 全てのパラメータをプリペアードクエリのパラメータとして渡す
  - libpqラッパーやライブラリが安全である事を確認
- 全てのパラメータをAPIを使ってエスケープする
- エスケープ出来ないパラメータ（フィールド名、テーブル名、SQL語句など）はホワイトリスト方式でバリデーションする（特殊文字攻撃に注意！）
- 文字エンコーディングを厳格に取り扱う

ブラインドSQLインジェクション対策

- データベースエラーの場合にアプリケーションの振る舞いが変らないように実装
  - DBエラーメッセージを表示しない
  - DBエラーと入力エラーのメッセージを同じにする

SQLインジェクションが無くならない最大の原因?!

- プリペアードクエリを使えば大丈夫！
- ORMだけ使っていれば大丈夫！
- エスケープしなくても良い物をエスケープするのは無駄！

libpqのプリペアードクエリが一番使い易い

- **速い**　余計なチェックがない
- **簡単**　データ型の考慮必要なし
- **安全**　プリペアードクエリの実装は確実

プリペアードクエリの利用は超簡単！

```php
<?php
error_reporting(E_ALL);

$msg = 'ユーザ名とパスワードを送信してください';
if (!empty($_REQUEST['username'])) {
    // check login
    $conn = pg_connect('host=localhost dbname=vulndb user=postgres password=postgres');
    $sql = "SELECT id FROM users WHERE username= '".$_REQUEST['username']."' AND passwor
    $res = pg_query($conn, $sql);
    if (pg_num_rows($res)) {
        $msg = 'ログインしました';
    } else {
        $msg = 'ユーザ名とパスワードを確認してください';
    }
}
?>
```

- 単純なラッパーを書くだけ！

```php
function do_query($conn, $sql, $params) {
    pg_prepare($conn, md5($sql), $sql);
    return pg_execute($conn, md5($sql), $params);
}
```

```php
<?php
error_reporting(E_ALL);

function do_query($conn, $sql, $params) {
    pg_prepare($conn, md5($sql), $sql);
    return pg_execute($conn, md5($sql), $params);
}

$msg = 'ユーザ名とパスワードを送信してください';
if (!empty($_REQUEST['username'])) {
    // check login
    $conn = pg_connect('host=localhost dbname=vulndb user=postgres password=postgre
    $sql = 'SELECT id FROM users WHERE username= $1 AND password = $2;';
    $res = do_query($conn, $sql, array($_REQUEST['username'], $_REQUEST['password']
    if (pg_num_rows($res)) {
        $msg = 'ログインしました';
```

プリペアードクエリの
の注意点

- 本物のプリペアードクエリはデータベースサーバにパース済みのSQL文をキャッシュ
  - 使い捨てSQL文の場合はゴミが溜まる
- 使い捨てSQL文の場合PQexecPrams関数を使用
  - 時々データベース接続を切断

今度こそ、本当にご清聴ありがとうございました。