文字コードに起因する脆弱性とその対策

2010年9月25日

HASHコンサルティング株式会社
徳丸　浩

本日お話しする内容

- 文字コード超入門
- 文字コードの扱いに起因する脆弱性デモ6連発
- 文字コードの扱いに関する原則
- 現実的な設計・開発指針
- まとめ

前提とする内容

- 文字コードに起因する脆弱性とは
  - 正しいセキュリティ対策をしているかに見えるコードにおいて、文字コードの取り扱いが原因で生じる脆弱性
- 以下の脆弱性に関する一般的な知識は既知のものとします
  - SQLインジェクション脆弱性
  - クロスサイト･スクリプティング(XSS)脆弱性
  - パストラバーサル脆弱性

徳丸浩の自己紹介

- 経歴
  - 1985年 京セラ株式会社入社
  - 1995年 京セラコミュニケーションシステム株式会社(KCCS)に出向・転籍
  - 2008年 KCCS退職、HASHコンサルティング株式会社設立
- 経験したこと
  - 京セラ入社当時はCAD、計算幾何学、数値シミュレーションなどを担当
  - その後、企業向けパッケージソフトの企画・開発・事業化を担当
  - 1999年から、携帯電話向けインフラ、プラットフォームの企画・開発を担当
    Webアプリケーションのセキュリティ問題に直面、研究、社内展開、寄稿などを開始
  - 2004年にKCCS社内ベンチャーとしてWebアプリケーションセキュリティ事業を立ち上げ
- その他
  - 1990年にPascalコンパイラをCabezonを開発、オープンソースで公開
    「大学時代のPascal演習がCabezonでした」という方にお目にかかることも
- 現在
  - HASHコンサルティング株式会社　代表　http://www.hash-c.co.jp/
  - 京セラコミュニケーションシステム株式会社　技術顧問　http://www.kccs.co.jp/security/
  - 独立行政法人情報処理推進機構　非常勤研究員　http://www.ipa.go.jp/security/

本を書いています

文字コード超入門

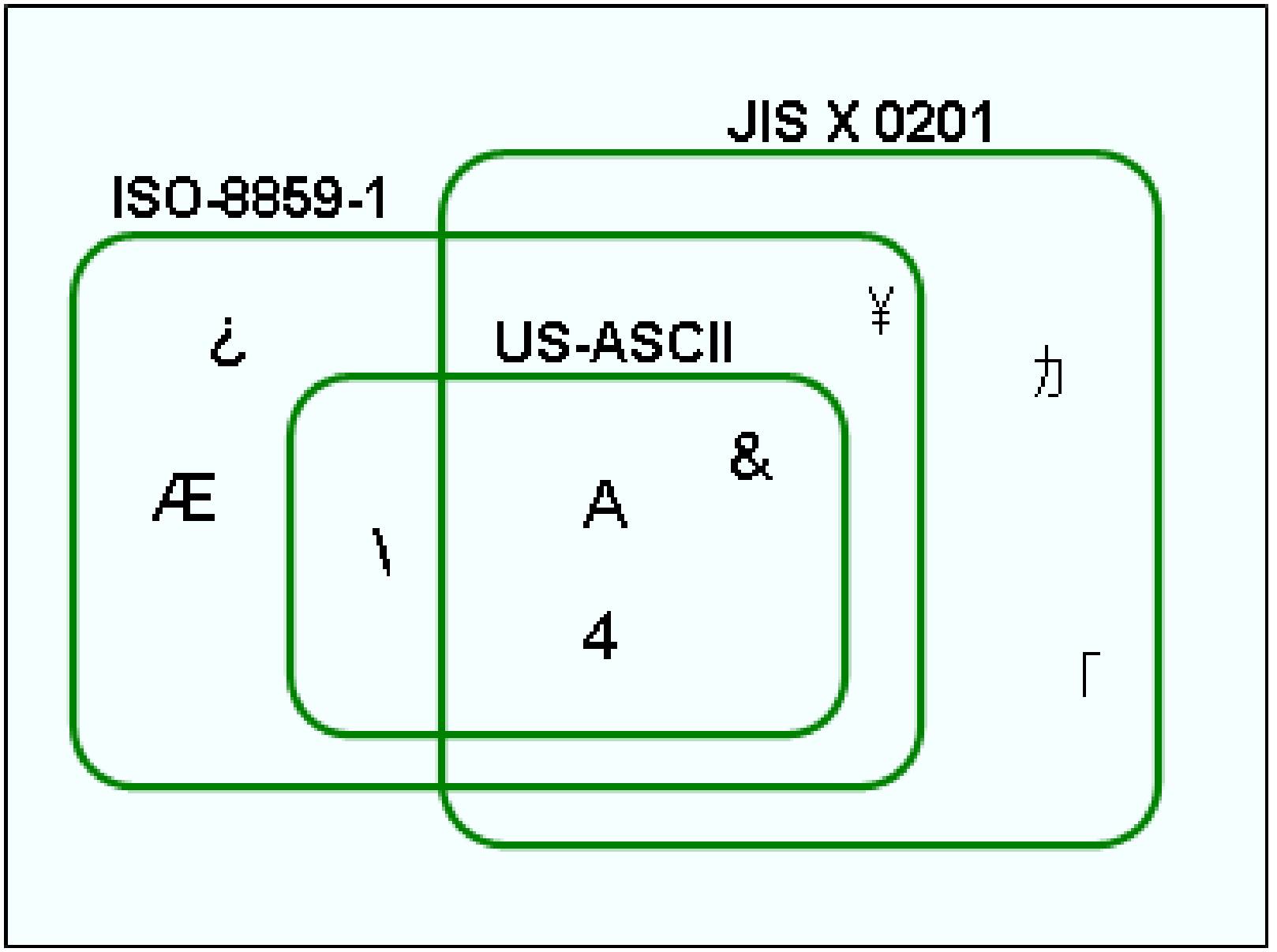
文字コードとはなにか

- 用語集的な説明は割愛
- 文字コード = 文字集合 + 文字エンコーディング
- 文字集合
  - 文字集合とは、文字を集めたもの
  - 符号化文字集合とは、集めた文字に符号（番号）をつけたもの
  - 代表的な（符号化）文字集合
    ASCII、ISO-8859-1、JIS X 0201、JIS X 0208、JIS X 0213
- 文字エンコーディング（文字符号化方式）
  - 文字集合を使ってコンピュータ処理するために、文字をバイト列として表現する方式
  - 文字集合は複数でもよい（ASCII + JIS X 0208など）
  - 代表的な文字エンコーディング
    Shift_JIS、EUC-JP、UTF-8、UTF-16

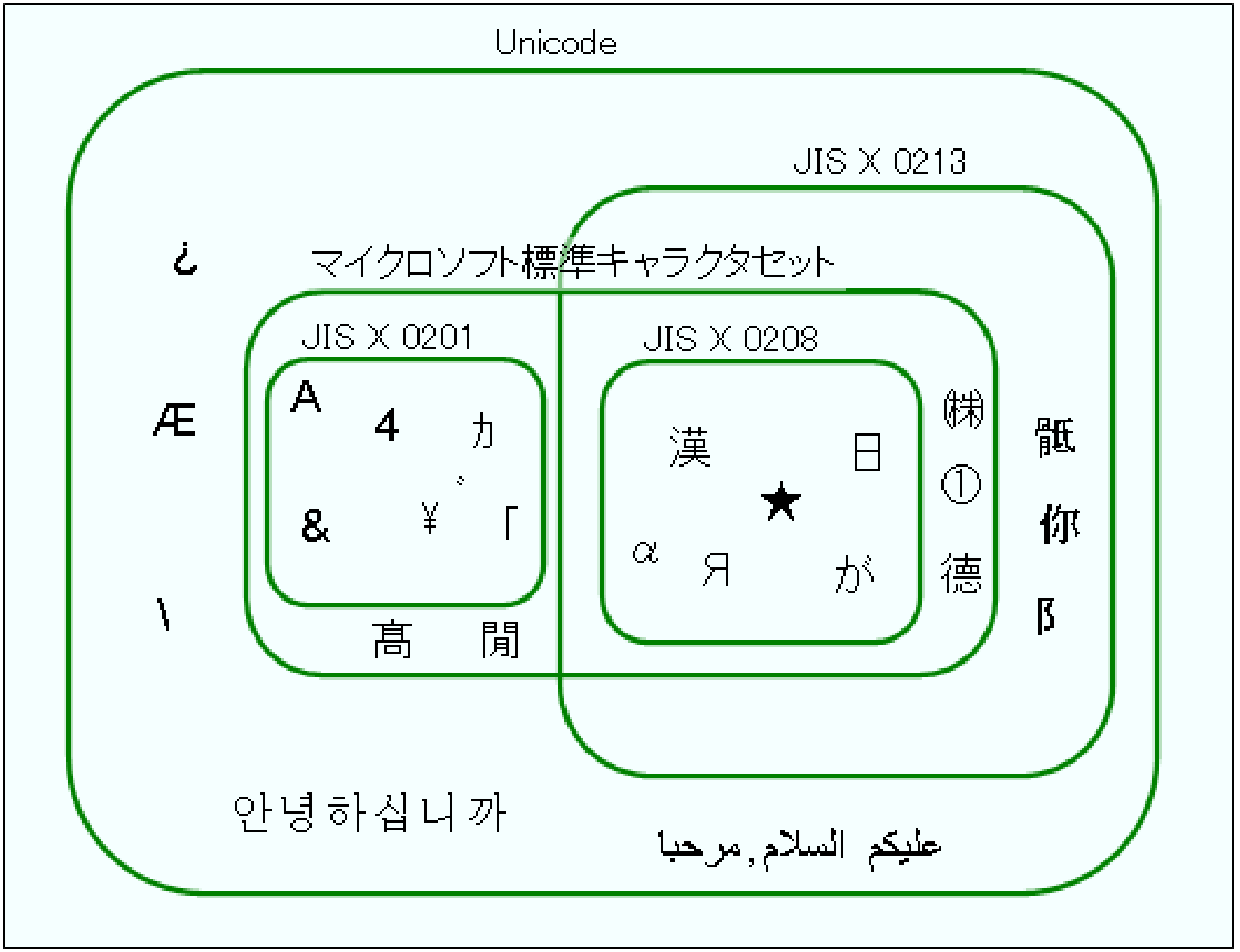
代表的な文字集合

文字集合名	ビット長	対応言語	説明
ASCII	7ビット	英語	規格化された最古の文字集合
ISO-8859-1	8ビット	西ヨーロッパ諸語	ASCII にフランス語やドイツ語などアクセントつき文字などを追加
JIS X 0201	8ビット	英字・カタカナ	ASCII とカタカナ
JIS X 0208	16ビット	日本語	第二水準までの漢字
マイクロソフト標準キャラクタセット	16ビット	日本語	JIS X 0201 と JIS X 0208 に、NEC と日本 IBM の機種依存文字を追加
JIS X 0213	16ビット	日本語	第四水準までの漢字
Unicode	21ビット	多言語	世界共通の文字集合

1バイト文字集合の包含関係

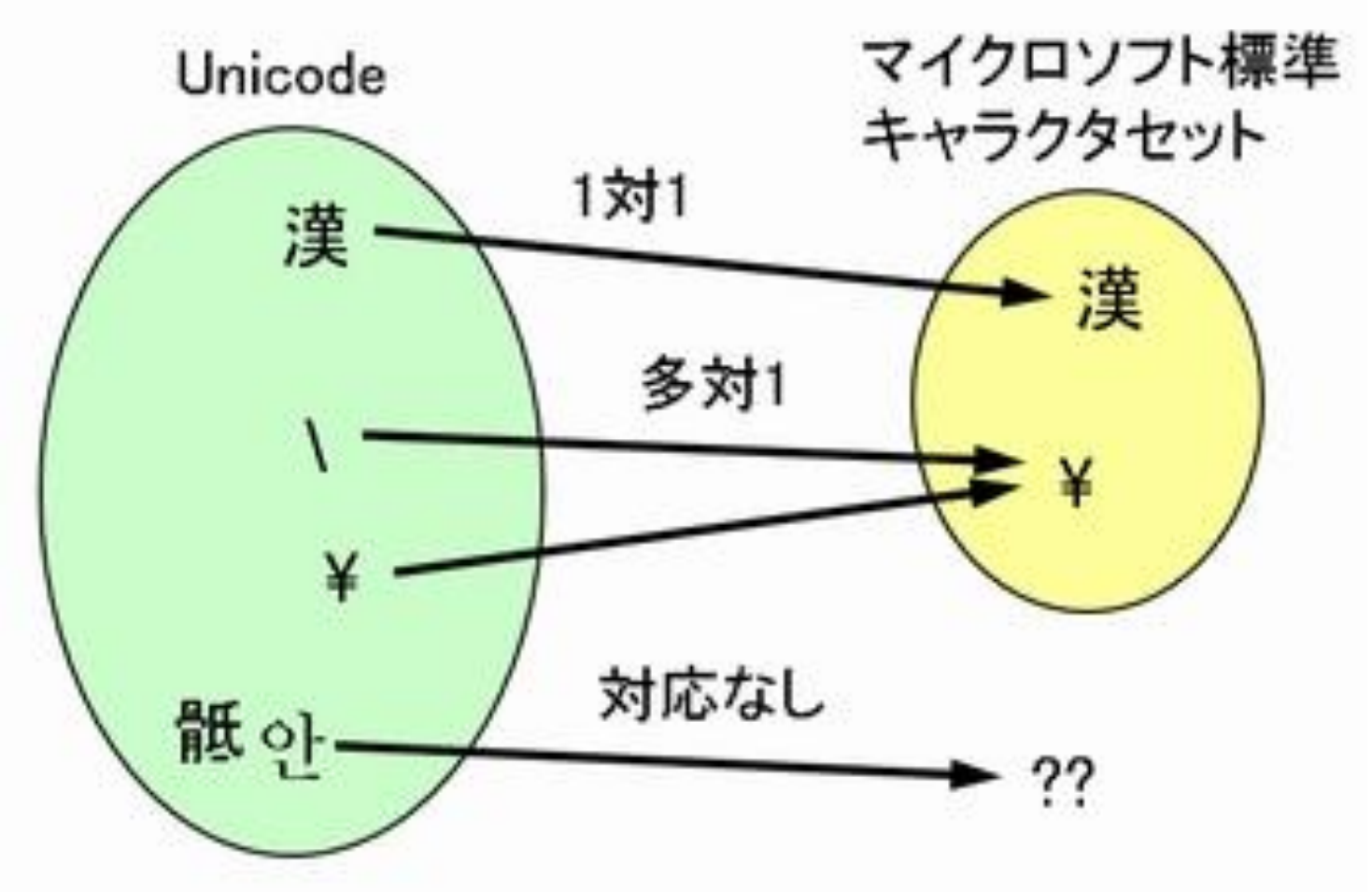


多バイト文字集合の包含関係



文字集合の変更に注意

- 文字集合を変更すると、一対一対応でない箇所で文字化け等の原因になる



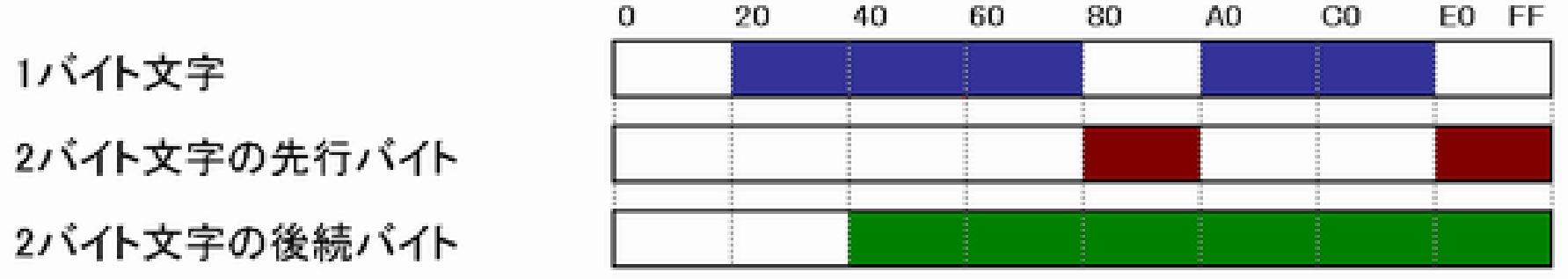
5CとA5には特に注意

- 文字集合によって、円記号の位置が異なるので、文字集合が変更された場合に事故の元になる可能性がある

文字集合	0x5C	0xA5
US-ASCII	\	%
JIS X 0201	¥	・
ISO-8859-1	\	¥
Unicode	\	¥

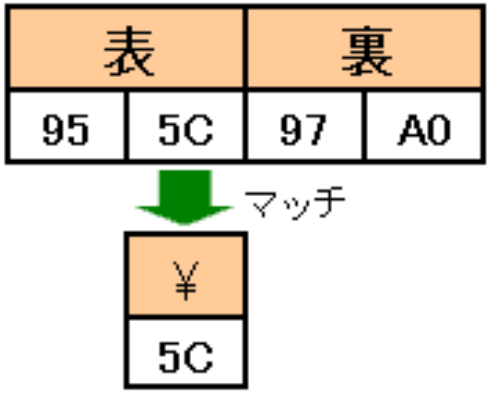
Shift_JIS

- JIS X 0201と英数片仮名とJIS X 0208の漢字（第一、第2水準）を混在できる文字エンコーディング
- 1バイトor 2バイトで1文字を表現。ビット利用効率が高い



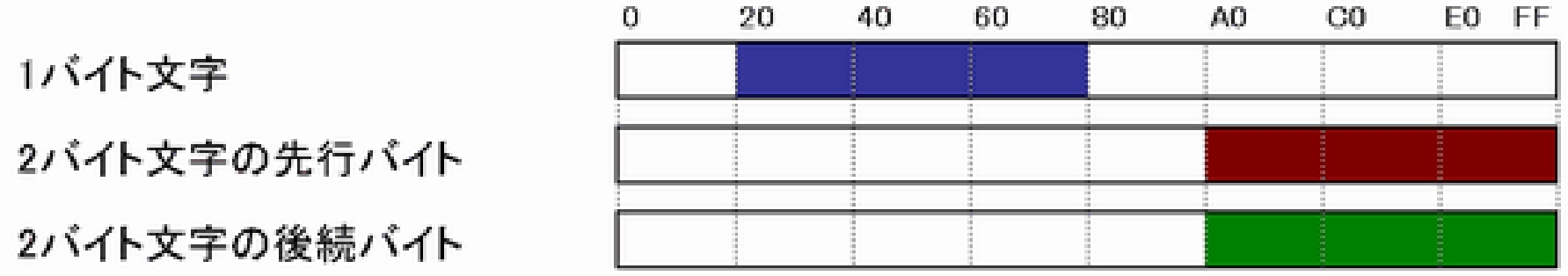
- 1バイト文字と後続バイト文字の領域が重なっているので、いわゆる「5C問題」が発生する

例えば「表」という文字の2バイト目が「¥」にマッチする

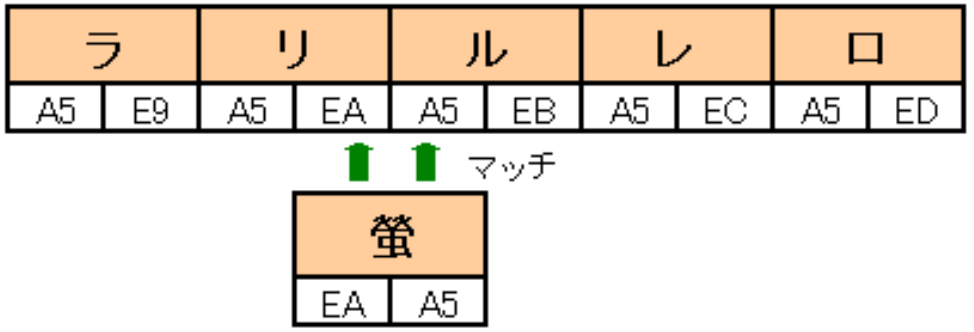


EUC-JP

- UNIX上で日本語を扱うために開発された文字エンコーディング
- ASCIIとJIS X 0208を混在して利用可能。JIS X 0201の片仮名（いわゆる半角片仮名）も、2バイトで表現可能
- 1バイト文字と後続バイトは領域が重ならないので、5C問題は発生しない



- しかし、文字境界をまたがってマッチングする場合はある
  =螢問題（オレ呼称につき注意）

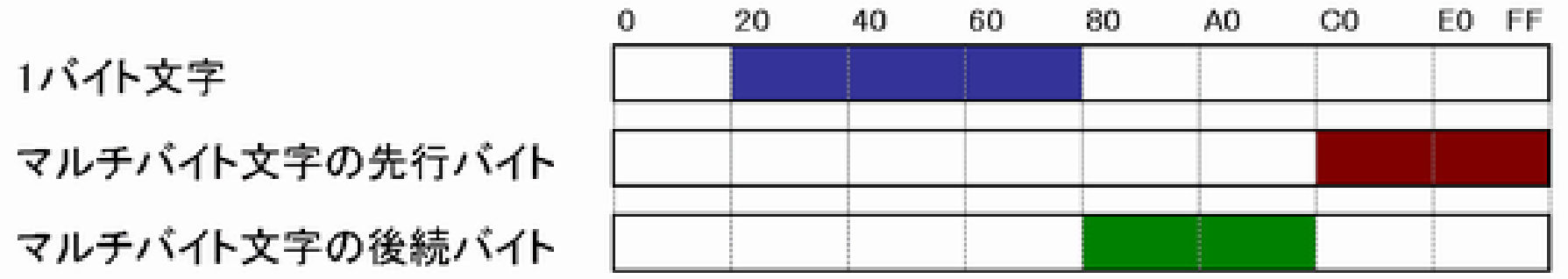


UTF-8

- Unicodeの文字エンコーディング。ASCIIと互換性を持たせるため、1～4バイトで1文字を表現。

コードポイントの範囲	UTF-8 ビットパターン	ビット長
0 ～ 7F	0xxxxxxx	7ビット
80 ～ 7FF	110xxxxx 10xxxxxx	11ビット
800 ～ FFFF	1110xxxx 10xxxxxx 10xxxxxx	16ビット
10000 ～ 10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	21ビット

- 1バイト文字、先行バイト、後続バイトの領域はまったく重ならないので、5C問題や「螢問題」は発生しない



- しかし、「非最短形式問題」には注意

UTF-8の非最短形式

- UTF-8で、Nバイトで表現できる文字は、N+1バイト以上の形式でも表現できる（計算上は）

スラッシュ「/」の非最短形式

1バイト	0x2F	最短形式
2バイト	0xC0 0xAF	非最短形式
3バイト	0xE0 0x80 0xAF	
4バイト	0xF0 0x80 0x80 0xAF	

- 同じ文字がバイト列上は「別の文字」と判定されるため、脆弱性の原因になる。現在は非最短形式は禁止されている
- PHPのmbstringは、非最短形式のUTF-8をエラーにするが、htmlspecialcharsは最近(5.3.1)までエラーにしなかった
- JREは比較的最近(Java SE6 Update10)までUTF-8の非最短形式を許容していた

UTF-7

- 7ビットでUnicodeを符号化する方式
- 元々電子メールでの利用を想定していたが、現在規格上は廃止されている
- 後方互換のため（ホントか?）主要ブラウザで現在でも利用できる
- 英数字と一部の記号はASCIIそのまま、それ以外の文字は以下の方式で変換

  UTF-16で符号化 → Base64エンコーディング →
  先頭に「+」、末尾に「-」を追加
- <script>alert(1)</script>はこうなる

  +ADw-script+AD4-alert(1)+ADw-/script+AD4-

文字コードの扱いに起因する脆弱性デモ6連発

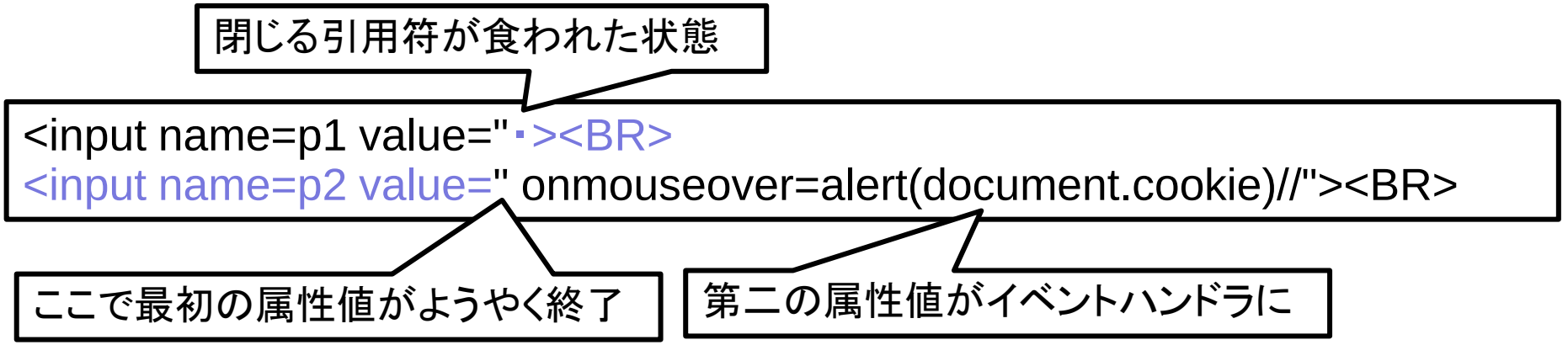
デモ1:半端な先行バイトによるXSS

- 半端な先行バイトとは
  - Shift_JIS、EUC-JP、UTF-8などマルチバイト文字の1バイト目だけが独立して存在する状態
  - 次の文字が、マルチバイト文字の2バイト目以降の文字として「食われる」状況になる
  - input要素などの引用符「”」を食わせて、イベントハンドラを注入する攻撃

デモ1：PHPソース

```
<?php
  session_start();
  header('Content-Type: text/html; charset=Shift_JIS');
  $p1 = @$_GET['p1'];
  $p2 = @$_GET['p2'];
?>
<body>
<form>
PHP Version:<?php
  echo htmlspecialchars(phpversion(), ENT_NOQUOTES,
  'Shift_JIS'); ?><BR>
<input name=p1 value="<?php echo
  htmlspecialchars($p1, ENT_QUOTES, 'Shift_JIS'); ?>"><BR>
<input name=p2 value="<?php echo
  htmlspecialchars($p2, ENT_QUOTES, 'Shift_JIS'); ?>"><BR>
<input type="submit" value="更新">
</form>
</body>
```

デモ1:半端な先行バイトによるXSS



- 半端な先行バイトによるXSSが発生する条件は、以下のいずれかを満たす場合
  - htmlspecialcharsの第3引数を指定していない
  - PHPの5.2.11以前あるいはPHP5.3.1以前を使用
- すなわち対策としては
  - PHPの最新版を使う
  - htmlspecialcharsの第3引数を指定する

デモ2:UTF-8非最短形式によるパストラバーサル

```
String msg = "";
String pathname = "";
try {
  String path = request.getParameter("path");
  File f = new File(path);
  // パス名からファイル名を取り出す(PHPのbasename())
  String filebody = f.getName();
  // UTF-8としてデコード
  filebody = new String(
          filebody.getBytes("ISO-8859-1"), "UTF-8");
  // ディレクトリを連結
  pathname = "c:/home/data/" + filebody;
  // 以下ファイル読み出し
  FileReader fr = new FileReader(pathname);
  BufferedReader br = new BufferedReader(fr);
  ...
```

文字コード	2E	2E	C0	AF	2E	2E	C0	AF	62	6F
文字	.	.	À	¯	.	.	À	¯	b	o

元の文字列
(ISO-8859-1と解釈)

スラッシュ「/」やバックスラッシュ「\」がないので、ファイル名として妥当と判断される

コードポイント	002E	002E	002F	002E	002E	002F	62	006F	006F	0074
文字	.	.	/	.	.	/	b	o	o	t

UTF-8としてデコード
C0AFがスラッシュに

デコード後文字列

../../boot.ini

C:/home/data/ と連結

C:/home/data/../../boot.ini

正規化

C:/boot.ini

- 脆弱性が発生する条件(AND条件)
  - Java SE6 update10以前を使用
  - 文字エンコーディング変換前にファイル名をチェックしている

デモ3:5C問題によるSQLインジェクション

- 5C問題とは
  - Shift_JIS文字の2バイト目に0x5Cが来る文字に起因する問題
    ソ、表、能、欺、申、暴、十 … など出現頻度の高い文字が多い
  - 0x5CがASCIIではバックスラッシュであり、ISO-8859-1など1バイト文字と解釈された場合、日本語の1バイトがバックスラッシュとして取り扱われる
  - 一貫して1バイト文字として取り扱われれば脆弱性にならないが、1バイト文字として取り扱われる場合と、Shift_JISとして取り扱われる場合が混在すると脆弱性が発生する

ソースコード（要点のみ）

```
<?php
  header('Content-Type: text/html; charset=Shift_JIS');
  $key = @$_GET['name'];
  if (! mb_check_encoding($key, 'Shift_JIS')) {
    die('文字エンコーディングが不正です');
  }
  // MySQLに接続(PDO)
  $dbh = new PDO('mysql:host=localhost;dbname=books', 'phpcon', 'pass1');
  // Shift_JISを指定
  $dbh->query("SET NAMES sjis");
  // プレースホルダによるSQLインジェクション対策
  $sth = $dbh->prepare("SELECT * FROM books WHERE author=?");
  $sth->setFetchMode(PDO::FETCH_NUM);
  // バインドとクエリ実行
  $sth->execute(array($key));
?>
```

5C問題によるSQLインジェクションの説明

83	5C	27	20	4F	52	20	31	3D	31	23
ソ		'		O	R		1	=	1	#

元の文字列(Shift_JIS)

83	5C	27	20	4F	52	20	31	3D	31	23
NBH	¥	'		O	R		1	=	1	#

Latin1として解釈

PDO

27	83	5C	5C	5C	27	20	4F	52	20	31	3D	31	23	27
'	NBH	¥	¥	¥	'		O	R		1	=	1	#	'

クォートしてエスケープ

MySQL

27	83	5C	5C	5C	27	20	4F	52	20	31	3D	31	23	27
'	ソ		¥	¥	'		O	R		1	=	1	#	'

Shift_JISとして解釈

↑ ¥がエスケープされた物と解釈

↑ 文字列リテラルの終端

デモ4:UTF-7によるXSS

```
<?php
  session_start();
  header('Content-Type: text/html; charset=EUCJP');
  $p = @$_GET['p'];
  if (! mb_check_encoding($p, 'EUCJP')) {
    die('Invalid character encoding');
  }
?>
<body>
<?php echo
   htmlspecialchars($p, ENT_NOQUOTES, 'EUCJP'); ?>
</body>
```

UTF-7によるXSSの説明

HTTPレスポンス

```
HTTP/1.1 200 OK
Content-Length: 187
Content-Type: text/html; charset=EUCJP

<body>
+ADw-script+AD4-alert(document.cookie)+ADw-/script+AD4-
</body>
```

EUCJPという文字エンコーディングを
IEは認識できない(正しくはEUC-JP)

IEはレスポンスの内容から、このコンテンツはUTF-7と判定する

UTF-7として解釈されたコンテンツ

```
<body>
<script>alert(document.cookie)</script>
</body>
```

JavaScriptが
起動される

デモ5:U+00A5によるSQLインジェクション

```
Class.forName("com.mysql.jdbc.Driver");

Connection con = DriverManager.getConnection(
  "jdbc:mysql://localhost/books?user=phpcon&passwor
  d=pass1");

String sql = "SELECT * FROM books where author=?";

// プレースホルダ利用によるSQLインジェクション対策

PreparedStatement stmt = con.prepareStatement(sql);

// ? の場所に値を埋め込む(バインド)

stmt.setString(1, key);

ResultSet rs = stmt.executeQuery();  // クエリの実行
```

U+00A5によるSQLインジェクションの原理

【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

IPA:安全なSQLの呼び出し方（http://www.ipa.go.jp/security/vuln/websecurity.html）より引用

U+00A5によるSQLインジェクションの条件と対策

- 脆弱性が発生する条件
  - JDBCとしてMySQL Connector/J 5.1.7以前を使用
  - MySQLとの接続にShift_JISあるいはEUC-JPを使用
  - 静的プレースホルダを使わず、エスケープあるいは動的プレースホルダ（クライアントサイドのバインド機構）を利用している
- 対策（どれか一つで対策になるがすべて実施を推奨）
  - MySQL Connector/Jの最新版を利用する
  - 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）

デモ6:U+00A5によるXSS

```
// PHPでもU+00A5がバックスラッシュに変換されるパターンはないか
<?php
  header('Content-Type: text/html; charset=Shift_JIS');
  $p = @$_GET['p'];
  // JSエスケープ ¥→¥¥ '→¥' "→ ¥"
  $p1 = preg_replace('/(?=[¥¥¥¥¥¥'¥¥"])/u' , '¥¥', $p);
  $p2 = htmlspecialchars($p1, ENT_QUOTES, 'UTF-8');
?>
<html><head>
<script type="text/javascript">
 function foo($a) {
  document.getElementById("foo").innerText= $a;
 }
</script></head>
<body onload="foo('<?php echo $p2; ?>')">
p=<span id="foo"></span>
</body>
</html>
```

```
[mbstring]
mbstring.internal_encoding = UTF-8
mbstring.http_input = auto
mbstring.http_output = cp932
mbstring.encoding_translation = On
mbstring.detect_order = SJIS-win,UTF-8,eucJP-win
```

U+00A5によるXSSが発生する原理と条件

コードポイント	00A5	0027	0029	003B	0061	006C	0065	0072	0074	0028
文字	¥	'	)	;	a	l	e	r	t	(

元の文字列(Unicode)

コードポイント	00A5	005C	0027	0029	003B	0061	006C	0065	0072	0074	0028
文字	¥	¥	'	)	;	a	l	e	r	t	(

エスケープ後の文字列(Unicode)

文字コード	5C	5C	27	29	3B	61	6C	65	72	74	28
文字	¥	¥	'	)	;	a	l	e	r	t	(

Shift_JISに変換

```
onload="foo('¥¥');alert(document.cookie)//'"
```

- 脆弱性が発生する条件
  - PHP5.3.3(以降)を利用している
    - それより前のバージョンでは、U+00A5は全角の「￥」に変換されていた
  - 以下の文字エンコーディング
    - 入力:自動、あるいはなんらかの経路でU+00A5の文字が入る
    - 内部:UTF-8
    - 出力:**cp932** あるいは cp51932

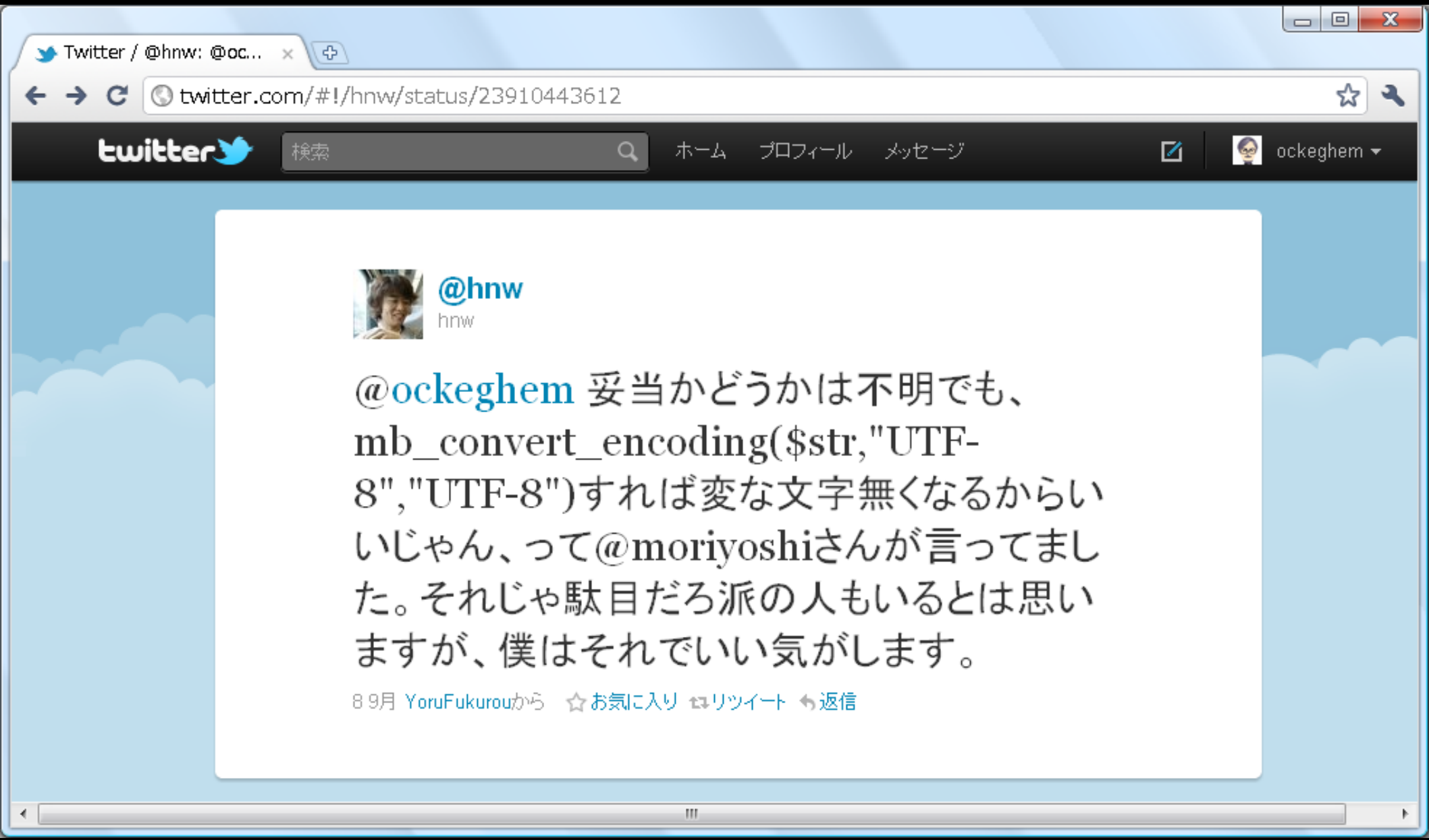
文字コードの扱いに関する原則

これまでのデモを原因別に分類すると…

- 文字エンコーディングとして不正なデータによる攻撃
  - デモ1:半端な先行バイトによるXSS
  - デモ2:UTF-8非最短形式によるパストラバーサル
- 文字エンコーディングとして不正でなく、マルチバイト文字対応が不十分なもの
  - デモ3:5C問題によるSQLインジェクション
  - デモ4:UTF-7によるXSS
- 文字集合の変更が原因となるもの
  - デモ5:U+00A5によるSQLインジェクション
  - デモ6:U+00A5によるXSS

不正な文字エンコーディング対策

- 入力時点でのアプローチ
  - 以下のいずれか、あるいは両方を実施する
    - すべての入力値の文字エンコーディングの妥当性をチェックする
    - 文字エンコーディングの変換を行う（不正な文字エンコーディングは削除）
  - Javaや.NETは内部UTF-16なので必ず文字エンコーディングが変換される。PHPはそのようなキマリがないので開発側で要注意
  - PHP6で内部UTF-16になるハズだったのだが…
- 出力（文字列を使う）時点でのアプローチ
  - 文字列を「使う」際の関数・APIが文字エンコーディングに対応していれば、脆弱性は発生しない
  - 最新のPHPを使う
  - 文字エンコーディング指定をさぼらない
- どっちをすればいいのか?
  - 両方やる

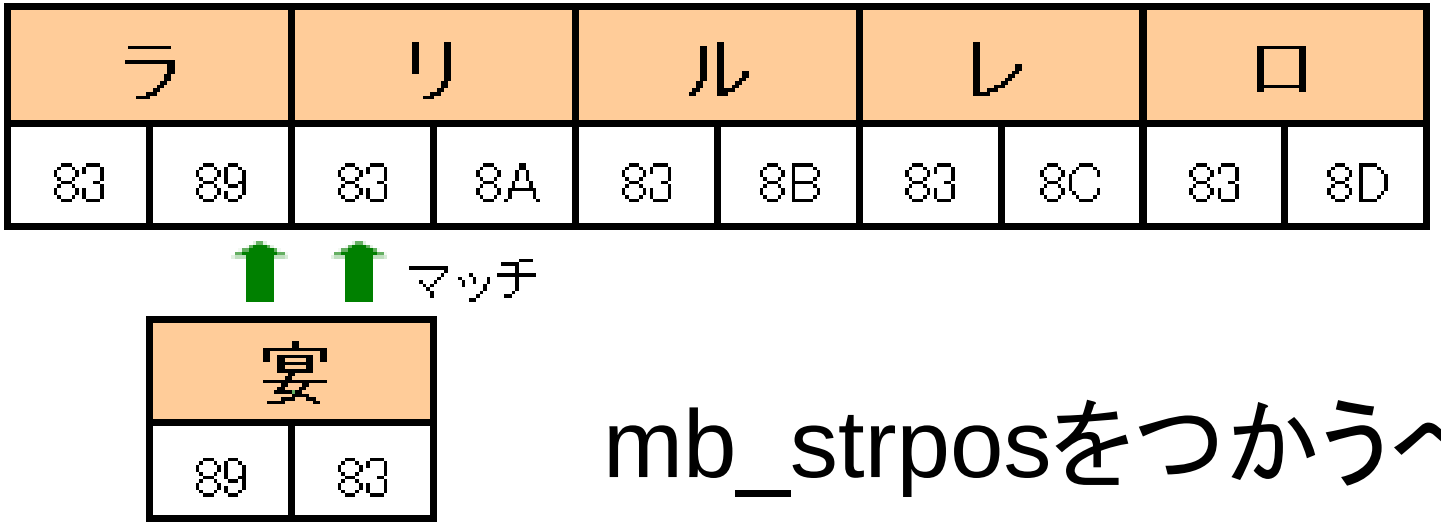

Twitter / @hnw: @oc...
twitter.com/#!/hnw/status/23910443612
twitter
検索
ホーム
プロフィール
メッセージ
ockeghem
@hnw
hnw
@ockeghem 妥当かどうかは不明でも、mb_convert_encoding($str,"UTF-8","UTF-8")すれば変な文字無くなるからいいじゃん、って@moriyoshiさんが言ってました。それじゃ駄目だろ派の人もいるとは思いますが、僕はそれでいい気がします。
8 9月 YoruFukurouから
お気に入り
リツイート
返信

マルチバイト文字対応を十分にするとは

- 一口で説明すると、「それぞれの処理を規格通りに正しく処理せよ」としか言えない
- HTTPレスポンスに「ブラウザが認識できる形式」で文字エンコーディングを指定する
  - ○ Shift_JIS / EUC-JP / UTF-8
  - × SJIS / cp932 / Shift-JIS / SJIS-win / EUCJP / UTF8
- すべての文字列処理でマルチバイト対応の関数を使用する
  - ここは日本語はない「はず」というのはやめた方がいい
- htmlspecialcharsの第3引数（文字エンコーディング）は必ず指定する
- データベース接続ライブラリには文字エンコーディングが指定できるものを選び、正しく指定する
- Shift_JISは避けた方がよい

クイズ:この関数はなにをするか分かりますか?

```
function isZenKana($ch) {
  $kanas = 'ァアィイゥウェエォオカガキギクグケゲコゴサザシジスズセゼソゾタダチヂッツヅテデトドナニヌネノハバパヒビピフブプヘベペホボポマミムメモャヤュユョヨラリルレロヮワヰヱヲンヴヵヶ';
  return strpos($kanas, $ch) !== false;
}
```

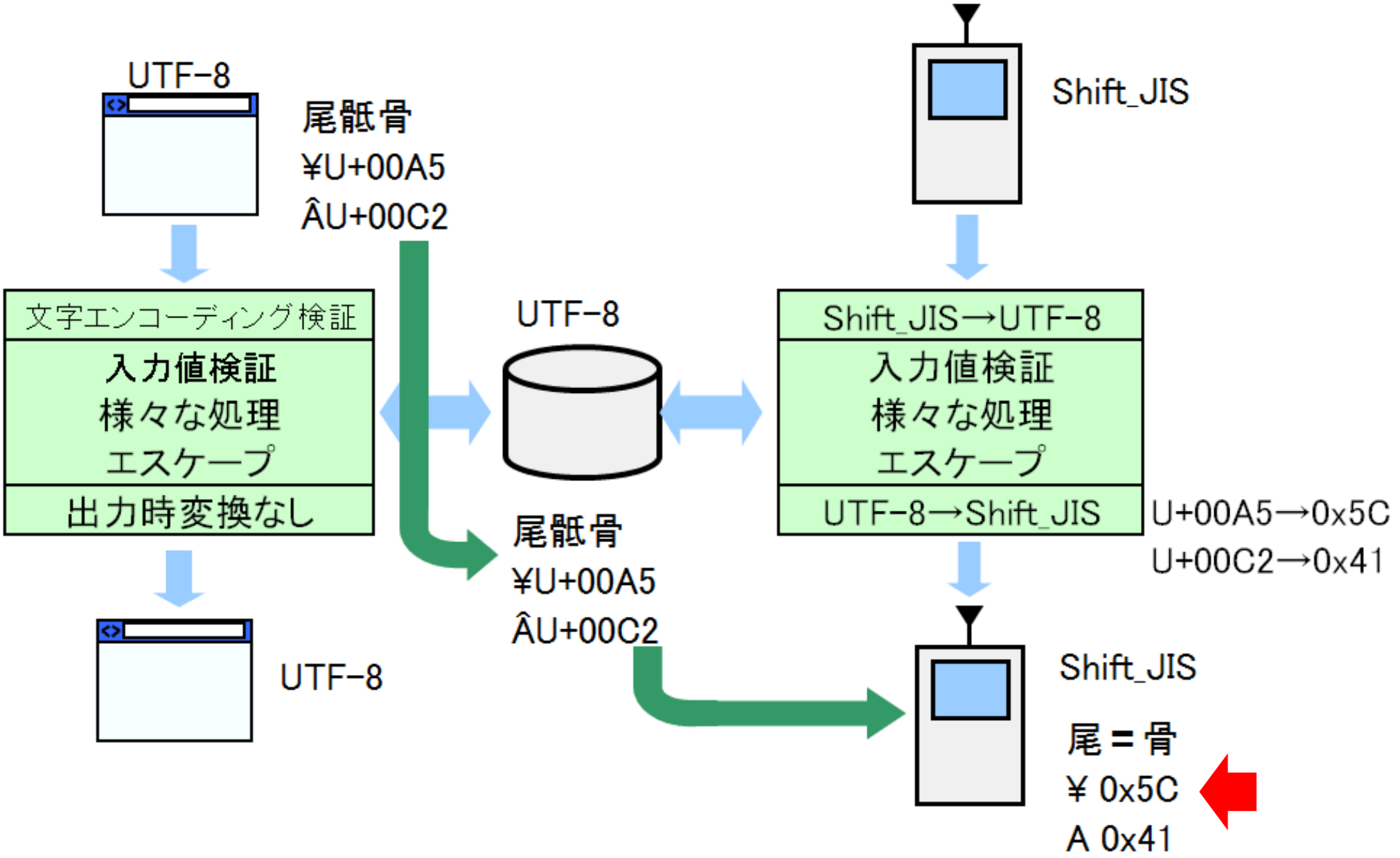


mb_strposをつかうべき

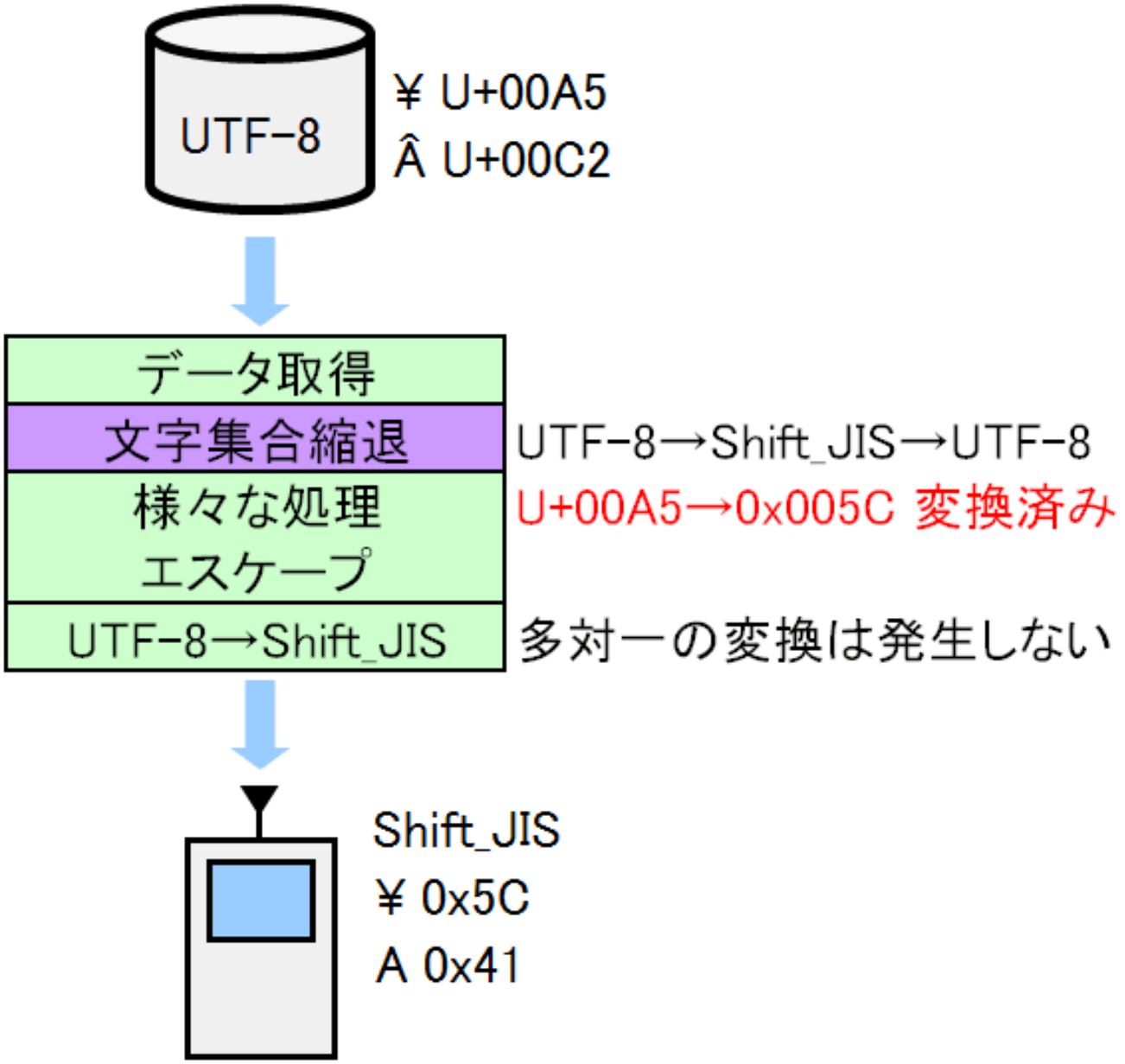
文字集合の変更が原因となるもの

- 文字集合の変更が脆弱性の原因になる場合がある
  - 多対一の変換 U+00A5 → \　など
- もっとも確実な解決策は、入り口から出口のすべての経路で、文字集合を変更しないこと
  - ○ UTF-8 → UTF-8 → UTF-8
  - ○ UTF-8 → UTF-16 → UTF-8
  - ○ Shift_JIS → EUC-JP → EUC-JP
  - × Shift_JIS → UTF-8 → Shift_JIS
- やむを得ず文字集合を変更する場合は、エスケープやチェック処理の前に、あらかじめ多対一変換が起こしておくとよい
  - 入り口で文字集合を絞り込む（Shift_JISに変換してからUTF-8に戻すなど）

PC・モバイル対応サイトでは文字集合の変更を伴う



文字集合の縮退を実施すれば安心・安全



現実的な設計・開発指針

文字コードの選定

- 以下の3パターンから選ぶ（Shift_JISは携帯電話向け）
  - UU:すべてUTF-8で通す（推奨）
  - JJ:外部はShift_JIS、内部はEUC-JP
  - JU:外部はShift_JIS、内部はUTF-8
- JUパターンは文字集合の変更を伴うので以下のいずれかを実施する
  - JIS文字集合にない文字が現れないことを保証する
  - JIS文字集合にない文字が現れる可能性がある場合は、「文字集合の縮退」処理を行う
  - 対症療法を実施する
    - JavaScript文字列リテラルを動的生成を避け、hiddenパラメータを生成して、DOMで読み込む（¥が問題になるコーディングをしない）
    - cp932とcp51932を避ける

簡単にできるテスト

- 以下の文字を入力・登録して、どのように表示されるかを調べる
  - ¥ (U+00A5)　　バックスラッシュに変換されないか
  - 骭 (U+9AB6)　　JIS X 0208にない文字
  - 𠮷 (U+20BB7)　　BMP外の文字 UTF-8では4バイトになる
- 尾骶骨テストや「つちよし」テストで、Unicodeがきちんと通るか確認しよう

処理に関する指針

- 入り口:以下のいずれかを行う
  - mb_check_encodingにより文字エンコーディングをチェックする
  - mb_convert_encodingにより文字エンコーディングを変換する
  - php.iniの設定により自動的に文字エンコーディングを変換する

  ※いずれの場合も、文字エンコーディングは明示する
- プログラム内の処理
  - mbstringの内部文字コード（mbstring.internal_encoding）を設定
  - すべての文字列処理でマルチバイト対応の関数を使用する
  - 文字エンコーディングのデフォルトが内部文字コートでない関数には文字エンコーディングを必ず指定する
- 出力（表示）
  - レスポンスヘッダに文字エンコーディングを正しく設定する
  - htmlspecialcharsの文字エンコーディングは必ず指定する

データベースに関する指針

- 文字エンコーディング指定のできるデータベース接続ライブラリを選定し、文字エンコーディングを正しく指定する

```
$dbh = new PDO('mysql:host=xxxx;dbname=xxxx;charset=cp932',
'user', 'pass', array(
 PDO::MYSQL_ATTR_READ_DEFAULT_FILE => '/etc/mysql/my.cnf',
 PDO::MYSQL_ATTR_READ_DEFAULT_GROUP => 'client', ));
# http://gist.github.com/459499 より引用(by id:nihen)
```

- 静的プレースホルダを使うよう指定する、あるいはプログラミングする

```
$dbh->setAttribute(PDO::ATTR_EMULATE_PREPARES,
   false);
```

- 詳しくは「安全なSQLの呼び出し方」を参照
  http://www.ipa.go.jp/security/vuln/websecurity.html

まとめ

- 正しい脆弱性対策をしていても、文字コードの扱いにより脆弱性が混入する場合がある
- 文字コードの選定（例えばShift_JISを避ける）
- 入力時に不正な文字エンコーディングを排除
- 処理毎の正しい文字エンコーディングの取り扱い
  - マルチバイト対応の関数を選定
  - 文字エンコーディングを指定
- 出力時の文字エンコーディングの扱い
  - レスポンスヘッダに文字エンコーディングを明示
  - htmlspecialcharsの文字エンコーディング指定を忘れずに
- データベースの文字エンコーディング指定と静的プレースホルダの利用

ご清聴ありがとうございました