Experience Report:
An Empirical Study of PHP Security Mechanism Usage

Johannes Dahse and Thorsten Holz
Ruhr-University Bochum, Germany

ISSTA 2015, July 13-17, Baltimore, Maryland, USA
Experience Report:
An Empirical Study of PHP Security Mechanism Usage

1. Introduction
2. Security Mechanisms
3. Static Enumeration
4. Empirical Study
1.1 Web Application State

- 82% of all websites run PHP as server-side language
- Weakly-typed language; requires more checks, introduces oddities
- 25% of all found vulnerabilities (CVE database) are related to PHP
- XSS and SQLi still one of the most commonly found security flaws
1.2 Security Mechanisms

- Best-practice security guidelines evolved, but no standards
- Developers apply their own favorite security mechanism
- Different programming patterns for input validation or input sanitization emerged, with advantages and drawbacks
- Some work generically, others work only for a certain markup context
- Programming mistakes and misplacing due to common pitfalls can still lead to vulnerabilities
1.3 Research Questions

Essential for developers, code auditors, and static analysis engineers

**RQ1.** Which security mechanisms are available?

**RQ2.** Which pitfalls might these mechanisms imply?

**RQ3.** Which security mechanisms are used how often in modern (web) applications?

**RQ4.** Which security mechanism is used to prevent which vulnerability type in which markup context?

**RQ5.** Which pitfalls occur in practice?
Experience Report:
An Empirical Study of PHP Security Mechanism Usage

2. Security Mechanisms
2.1 Taint-style Vulnerabilities
2.1 Types of Security Mechanisms

- Source
- Sanitization
- Sensitive sink
2.1 Types of Security Mechanisms

- Source
- Sanitization
- Validation
2.1 Types of Security Mechanisms
2.2 Generic Input Sanitization

- Explicit Type Casting

```
1 $id = $_GET['id'];
2 echo intval($id);
3 mysql_query('SELECT * FROM t WHERE id='.(int)$id);
```
2.3 Context-Sensitive Input Sanitization

- **Converting**

```php
$url = htmlentities($_GET['id']);

$e <a href="">$url</a>;

$e "<a href='$url'>click</a>";

$e '<a href="'>$url</a>' >click</a>;' ;
```

- **Escaping**

```php
$id = mysql_real_escape_string($_GET['id']);

mysql_query("SELECT * FROM t WHERE id = '$id'");

mysql_query('SELECT * FROM t WHERE id = ' . $id);
```
2.3 Context-Sensitive Input Sanitization

- Converting

```
$u = htmlentities($_GET['id']);  " → &quot;
```

- Escaping

```
$id = mysql_real_escape_string($_GET['id']);
$sql_query = "SELECT * FROM t WHERE id = '$id';
```

```
2.3 Context-Sensitive Input Sanitization

• Converting

```
1 $url = htmlentities($_GET['id']); " → &quot;
2 echo '<a href="">'. $url .'</a>'; < → &lt;
3 echo "<a href='$url'>click</a>"; 'onclick='alert(1)
4 echo '<a href="">'. $url .'">click</a>';
```

• Escaping

```
5 $id = mysql_real_escape_string($_GET['id']);
6 mysql_query("SELECT * FROM t WHERE id = '$id'"));
7 mysql_query('SELECT * FROM t WHERE id = ' . $id);```

Pitfalls

source → sanitization → sensitive sink
2.3 Context-Sensitive Input Sanitization

- **Converting**

  1. \$url = htmlentities($_GET['id']);  " → &quot;
  2. echo '<a href="">'.\$url . '</a>';  &lt; → &lt;
  3. echo "<a href="\$url">click</a>"; 'onclick='alert(1)
  4. echo '<a href="\$url">click</a>';

- **Escaping**

  5. \$id = mysql_real_escape_string($_GET['id']);
  6. mysql_query("SELECT * FROM t WHERE id = '\$id'");
  7. mysql_query('SELECT * FROM t WHERE id = ' . \$id);
2.3 Context-Sensitive Input Sanitization

- **Converting**

```php
$url = htmlentities($_GET['id']);
echo '<a href="">'. $url . '</a>';
echo "<a href='$url'>click</a>";
echo '<a href="'. $url . '">click</a>';  
```

- **Escaping**

```php
$id = mysql_real_escape_string($_GET['id']);
mysql_query("SELECT * FROM t WHERE id = '$id'"));
```

**source** → **sanitization** → **sensitive sink**
2.3 Context-Sensitive Input Sanitization

• Converting

```php
1 $url = htmlentities($_GET['id']);
2 echo '<a href="">' . $url . '</a>';
3 echo "<a href='$url'>click</a>";
4 echo '<a href="">click</a>';
```

• Escaping

```php
5 $id = mysql_real_escape_string($_GET['id']); ' → \'
6 mysql_query("SELECT * FROM t WHERE id = '$id'");
7 mysql_query('SELECT * FROM t WHERE id = $.id');
```

1 or 1=1
2.4 Generic Input Validation

- **Type Validation**

```php
$id = $_GET['id'];
if(is_numeric($id)) {
    echo $id;
}
if(is_int($id) === true) {
    echo $id;
}
if((int)$id) {
    echo $id;
}
if($id = (int)$id) {
    echo $id;
}
```

- **Comparing**

```php
$name = $_GET['name'];
if($name == 'issta') {
    echo $name;
}
if($name === 'issta') {
    echo $name;
}
if($name == 15) {
    echo $name;
}
if($name === 15) {
    echo $name;
}
```
2.4 Generic Input Validation

- Type Validation

```php
$id = $_GET['id'];
if(is_numeric($id)) {
    echo $id;
}
if(is_int($id) === true) {
    echo $id;
}
if((int)$id) {
    echo $id;
}
if($id = (int)$id) {
    echo $id;
}
```

- Comparing

```php
$name = $_GET['name'];
if($name == 'issta') {
    echo $name;
}
if($name === 'issta') {
    echo $name;
}
if($name == 15) {
    echo $name;
}
if($name === 15) {
    echo $name;
}
```
2.4 Generic Input Validation

- Type Validation

```
1 $id = $_GET['id'];
2 if(is_numeric($id)) {
3     echo $id;
4 }
5 if((int)$id) {
6     echo $id;
7 }
8 if($id = (int)$id) {
9     echo $id;
10 }
```

- Comparing

```
6 $name = $_GET['name'];
7 if($name == 'issta') {
8     echo $name;
9 }
10 if($name === 'issta') {
11    echo $name;
12 }
13 if($name == 15) {
14    echo $name;
15 }
16 if($name === 15) {
17    echo $name;
18 }
```
2.4 Generic Input Validation

- Type Validation

```php
$id = $_GET['id'];
if(is_numeric($id)) { echo $id; }
if(is_int($id) === true) { echo $id; }
if((int)$id) { echo $id; }
if($id = (int)$id) { echo $id; }
```

- Comparing

```php
$name = $_GET['name'];
if($name == 'issta') { echo $name; }
if($name === 'issta') { echo $name; }
if($name === 15) { echo $name; }
if($name !== 15) { echo $name; }
```

15<svg onload=alert(1)>
Experience Report:
An Empirical Study of PHP Security Mechanism Usage

3. Static Enumeration
3.1 Approach

- Erroneous approach: Count occurrences of security related features
  - Over-approximation when features are used for other purposes
  - Under-approximation when features are used in reusable code
- Our approach: Use modified version of our static analysis prototype
- Leverages backwards-directed context-sensitive taint analysis
- Count security mechanism only when data reaches a sensitive sink that was previously sanitized/validated and was previously tainted
Experience Report:
An Empirical Study of PHP Security Mechanism Usage

1. Introduction
2. Security Mechanisms
3. Static Enumeration
4. Empirical Study
4.1 Selected Applications

• 25 PHP applications
  • Open source, active and popular according to W3Tech's usage statistic
  • Size of at least 20 KLOC
    • Works standalone, does not extensively use reflection or framework
• 2.5 million lines of code (LOC) in total
• 26,006 unique data flows analyzed
Experience Report:
An Empirical Study of PHP Security Mechanism Usage

4.2 LOC and Markup Contexts

[Graph showing LOC, HTML, SQL, and JS contexts with bars for each context type across different LOC values.]
4.3 Markup Contexts and Security Mechanisms

- **Markup Contexts**: 64% HTML, 31% SQL, 5% Other

- **Mechanism Types**: 53% Sanitization, 47% Validation

- **Top Mechanisms**: 65% Explicit Typecast, 19% Type Validation, 16% Other
4.3.1 HTML Markup Security

Mechanisms correctly applied
- Replace
- Regex Validate
- Explicit Typecast
- Comparing
- Type Validation
- Converting
- Other

Mechanisms wrongly applied
- Converting
- Comparing
- Regex Replace
- Escaping
- Regex Validate
- Replace
- Other
4.3.2 JavaScript Markup Security

Mechanisms correctly applied

- Regex Replace
- Null Validation
- Replace
- Comparing
- Explicit Typecast
- Type Validation
- Other

Mechanisms wrongly applied

- Truncate
- Regex Replace
- Regex Validate
- Comparing
- Replace
- Converting

94% correctly applied
6% wrongly applied
Experience Report:
An Empirical Study of PHP Security Mechanism Usage

4.3.3 SQL Markup Security

Mechanisms correctly applied

- Prepare
- Replace
- Regex Validate
- Comparing
- Type Validation
- Escape
- Explicit Typecast
- Other

Mechanisms wrongly applied

- Regex Replace
- Prepare
- Truncate
- Comparing
- Replace
- Regex Validate
- Escaping
- Other

31% 67% 2%

1. Introduction
2. Security Mechanisms
3. Static Enumeration
4. Empirical Study
Experience Report:
An Empirical Study of PHP Security Mechanism Usage

4.4 Lessons Learned

Pitfall density (bars) versus markup frequency (line)
4.5 Threads to Validity

- Only 25 popular applications
- Static code analysis is limited (FP, FN, mistakes)
- Misinterpretation of developer intention
- Caution to draw strong conclusions and to generalize
Experience Report:
An Empirical Study of PHP Security Mechanism Usage

Questions

johannes.dahse@hgi.rub.de
Thank you!
Enjoy the conference.