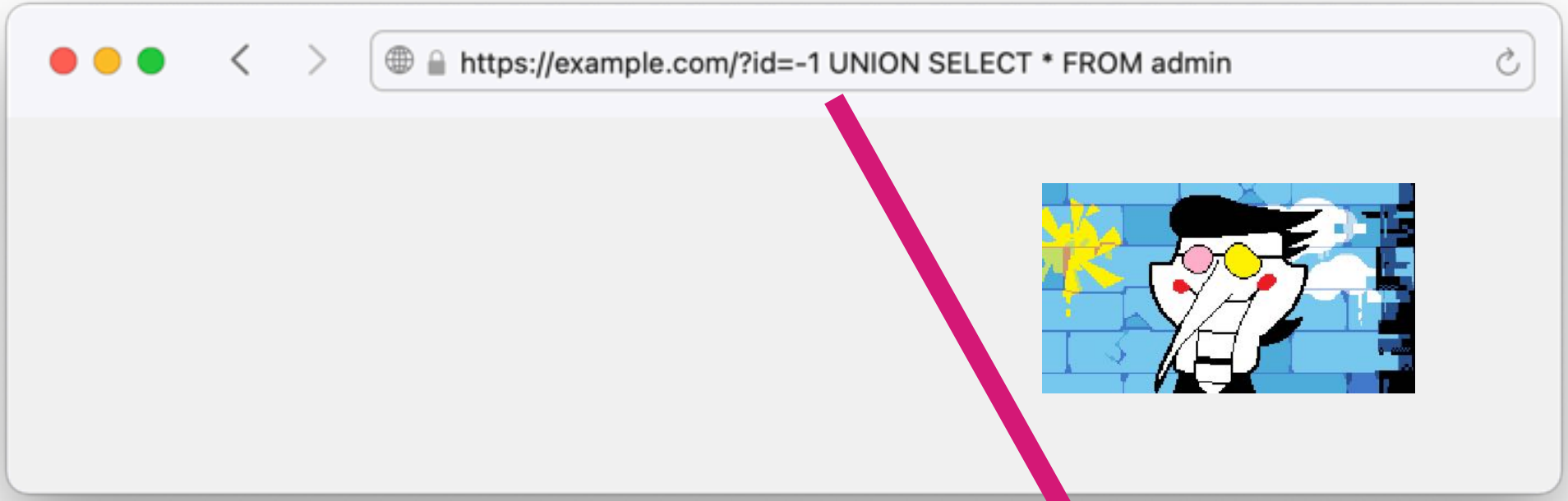
PRESENTED BY: Craig Francis

The `literal-string` Type


```
$sql = 'SELECT * FROM user WHERE id = ' . $_GET['id'];
```


```
$sql = 'SELECT * FROM user WHERE id = ' . $_GET['id'];
```

SELECT * FROM user WHERE id = 123




```
$sql = 'SELECT * FROM user WHERE id = $_GET['id'];
```

SELECT * FROM user WHERE id = -1 UNION SELECT * FROM admin


```
$sql = 'SELECT * FROM user WHERE id = ?';

$db->query($sql, $_GET['id']);
```


```
$sql = 'SELECT * FROM user WHERE id = ' . $_GET['id'];

$db->query($sql);
```

Database Abstractions:

Doctrine
Laravel DB
Propel ORM
RedBean
CakePHP

Mistakes with Database Abstractions:

https://eiv.dev/

Doctrine

```php
<?php

$qb->select(array('u'))
    ->from('User', 'u')
    ->where($qb->expr()->orX(
        $qb->expr()->eq('u.id', '?1'),
        $qb->expr()->like('u.nickname', '?2')
    ))
    ->orderBy('u.surname', 'ASC');

```

```
$qb->select('u')
    ->from('User', 'u')
    ->where('u.id = :identifier')
    ->setParameter('identifier', $_GET['id']);
```

```
$qb->select('u')
    ->from('User', 'u')
    ->where('u.id = ' . $_GET['id']); // INSECURE
```

Propel / RedBean

```
$articles->where('author_id', $id);

$articles->where('author_id IS NULL');

$articles->where('DATE(published)', $date);
```

Propel / RedBean

```
$articles->where('author_id', $id);
```

```
$articles->where('author_id IS NULL');
```



```
$articles->where('DATE(published)', $date);
```

```
$articles->where('word_count > 1000');
```

```
$articles->where('word_count > ', $count);
```

```
$articles->where('word_count > ' . $count);
```

'word_count > word_count UNION SELECT * FROM admin'

```
$html = '<p>Hi {{ name }}</p>';

$template->render($html, ['name' => $name]);
```

```
$html = '<p>Hi ' . $name . '</p>';

$template->render($html);
```

' Hi <script>alert();</script> '

Context aware?

```
$html = '<a href="{{ url }}">Link</a>';

$template->render($html, ['url' => $url]);
```

```
'<a href="javascript:alert()">Link</a>'
```

```
$exec = 'grep "' . $search . '" /path/to/file';
```

```
grep "" /path/to/secrets; # " /path/to/file
```

"Distinguishing strings from a trusted developer, from strings that may be attacker controlled"

Mike Samuel - 27th March 2019

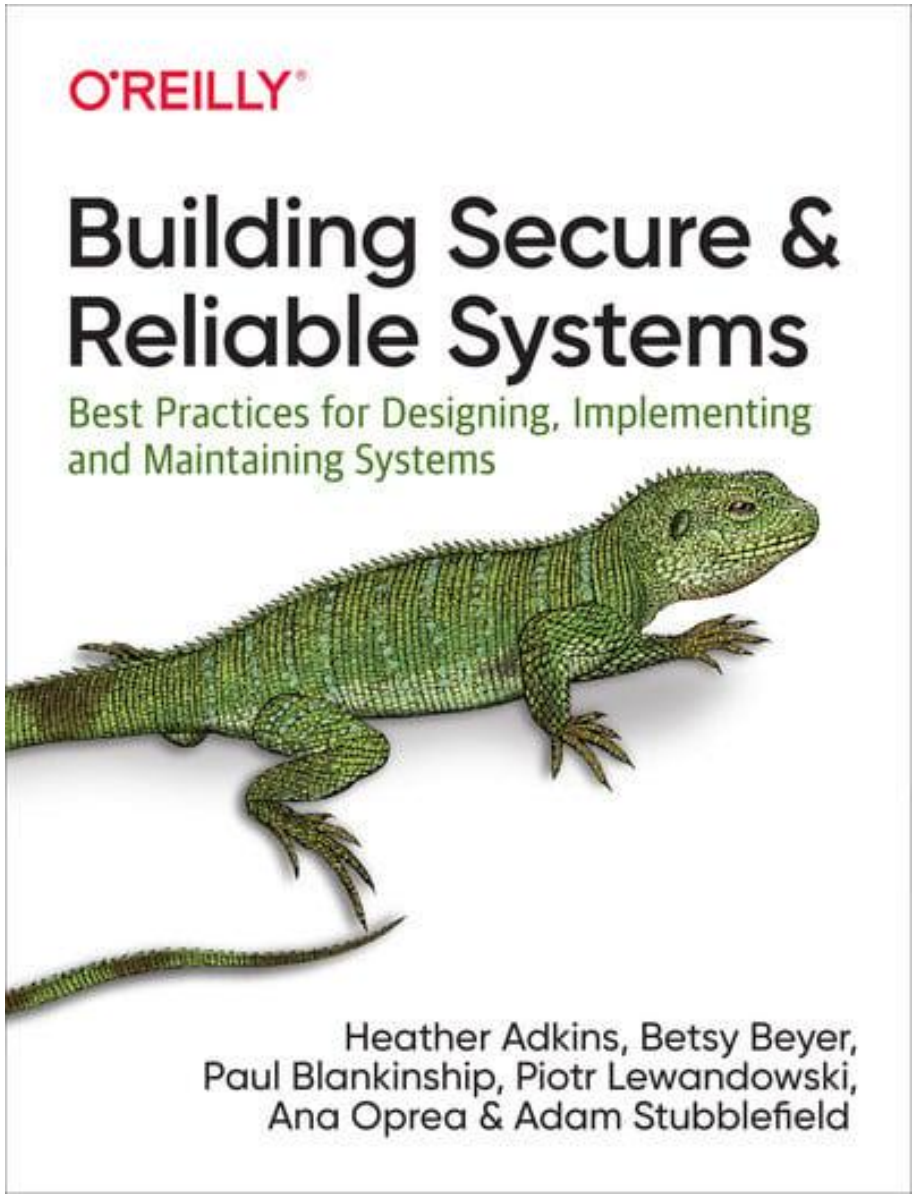
Christoph Kern

Preventing Security Bugs through Software Design

USENIX Security 2015

AppSec California 2016

https://youtu.be/ccfEu-Jj0as



Building Secure and Reliable Systems

March 2020

ISBN 9781492083078

Common Security Vulnerabilities

Page 266

Taint Checking?

Where variables note if they are Tainted, or Untainted.

Untainted

```
$html = '<p>Hello Everybody,</p>';
```

Untainted

Untainted Tainted Untainted

```
$html = '<p>Hi ' . $name . '</p>';
```

Tainted

Untainted

Tainted

Untainted

```
$html = '<p>Hi ' . htmlspecialchars($name) . '</p>';
```

Untainted

Escaped, to make it Safe :-)

Hello <script>alert()</script>

Unfortunately Taint Checking incorrectly assumes escaping makes a value “safe” for *any* context.

Untainted

Tainted

Untainted

```
$html = "<a href='" . htmlspecialchars($url) . "'>Link</a>";
```

Escapes & ' " < > ... Is this Safe?

Link

Untainted

Tainted

Untainted

```
$html = "<img src=" . htmlspecialchars($url) . " />";
```

Missing Quotes

Escapes & ' " < > ... Is this Safe?

Untainted

Tainted

Untainted

```
$html = "<img src='" . htmlspecialchars($url) . "' />";
```

Before PHP 8.1, single quotes were not encoded by default :-)

Is this Safe?

```
<img src='/' onerror='alert()' />
```

Untainted

Tainted

```
$sql = "WHERE id = " . mysqli_real_escape_string($link, $id);
```

Missing Quotes

Escaped ... Is this Safe?

WHERE id = -1 UNION SELECT * FROM admin

Taint Checking is close,
but escaping should be done by a Library.

We can simplify this, by checking for:

"strings from a trusted developer"

Safe* vs Unsafe

When talking about
Injection Vulnerabilities

Safe*

A developer defined string.
(in the source code)

Unsafe

Everything else.

Safe*

```
$sql = 'SELECT * FROM user WHERE id = ?';
```

```
$db->query($sql, $id);
```

Unsafe

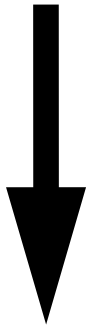
Safe*

Unsafe

```
$sql = 'SELECT * FROM users WHERE id = ' . $id;

$db->query($sql);
```

???



Safe*

Unsafe

```
$sql = 'SELECT * FROM users WHERE id = ' . $id;
```

```
$db->query($sql);
```

Unsafe

```
$sql = 'SELECT * FROM users WHERE id = ' . $id;
```

```
$db->query($sql);
```

Safe*

```
$html = '<p>Hi {{ name }}</p>';
```

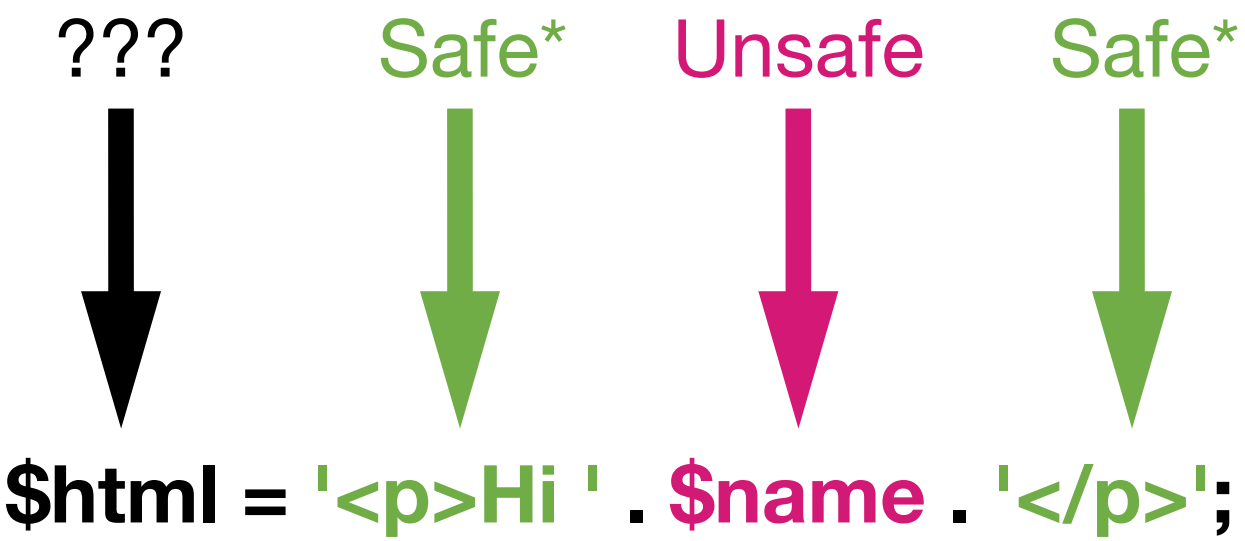
```
$template->render($html, ['name' => $name]);
```

Unsafe

Safe* Unsafe Safe*

```
$html = '<p>Hi ' . $name . '</p>';
```

```
$template->render($html);
```




```
$template->render($html);
```

Unsafe

```
$html = '<p>Hi ' . $name . '</p>';
```

```
$template->render($html);
```

Safe*

```
$command = 'grep ? /path/to/file';

shell_exec($command, [$search]);
```

Unsafe

Safe* Unsafe Safe*

```
$command = 'grep "' . $search . '" /path/to/file';

shell_exec($command);
```

???

Safe*

Unsafe

Safe*

```
$command = 'grep "' . $search . '" /path/to/file';
```

```
shell_exec($command);
```

Unsafe

```
$command = 'grep "' . $search . '" /path/to/file';

shell_exec($command);
```

Remember, only "Safe" when talking about Injection Vulnerabilities.

Safe*

$path = "/";

rm -rf /

```
$command = 'rm -rf ?';

shell_exec($command, [$path]);
```

Unsafe

Special Cases

Did you remember to
ensure all were integers?

```
$sql = 'WHERE id IN (' . implode(',', $ids) . ')';

$db->query($sql);
```

'WHERE id IN (1, 7, 9)'

WHERE id = (-1) UNION SELECT * FROM admin WHERE id IN (2)

```
$sql = 'WHERE id IN (' . in_parameters(count($ids)) . ')';

$db->query($sql, $ids);
```

'WHERE id IN (?, ?, ?)'

```
$sql = 'WHERE id IN (' . in_parameters(count($ids)) . ')';
```

```
function in_parameters($count) {
    $sql = '?';
    for ($k = 1; $k < $count; $k++) {
        $sql .= ',?';
    }
    return $sql;
}
```

Could try to escape the field...
But should *any* field be allowed?

```
$sql = 'ORDER BY ' . $order_field;
```

```
$fields = [
    'name',
    'email',
    'created',
  ];

$order_id = array_search($order_field, $fields);

$sql = 'ORDER BY ' . $fields[$order_id];
```

List of Allowed fields

Array of "developer defined strings"

```
$fields = [
    'name' => 'u.full_name',
    'email' => 'u.email_address',
    'created' => 'DATE(u.created)',
  ];
```

List of Allowed fields

```
$sql = 'ORDER BY ' . ($fields[$order_field] ?? 'u.full_name');
```

Array of "developer defined strings"

What about config values?

e.g. table names, set in an INI/JSON/YAML file.

This will be covered after the next section :-)

But in short, the library needs to handle these (safely).

Checking in PHP, with Static Analysis

Using Psalm

Thanks to Matthew Brown

```
composer require --dev vimeo/psalm

./vendor/bin/psalm --init
```

Check Psalm is at level 3 or stricter.

(level 1 is the most strict)

```php
<?php

$id = (string) ($_GET['id'] ?? '');

class db {

    /**
     * @psalm-param literal-string $sql
     */

    public function query(string $sql, array $parameters = []) : void {

        // Send $sql and $parameters to the database.

    }

}

$db = new db();

$db->query('SELECT * FROM user WHERE id = ?', [$id]);

$db->query('SELECT * FROM user WHERE id = ' . $id);
```

Use 'literal-string' type for $sql

```php
<?php

$id = (string) ($_GET['id'] ?? '');
```

Terminal

```
craig$ ./vendor/bin/psalm
Scanning files...
Analyzing files...



ERROR: ArgumentTypeCoercion - public/index.php:23:12 - Argument 1 of db::query expects literal-string, parent type non-empty-string provided
(see https://psalm.dev/193)
$db->query('SELECT * FROM user WHERE id = ' . $id);


------------------------------
1 errors found
------------------------------

Checks took 0.00 seconds and used 4.375MB of memory
No files analyzed
Psalm was able to infer types for 100% of the codebase
craig$
```

```php
$db = new db();

$db->query('SELECT * FROM user WHERE id = ?', [$id]);

$db->query('SELECT * FROM user WHERE id = ' . $id);
```

Using PHPStan

Thanks to Ondřej Mirtes

```
composer require --dev phpstan/phpstan
```

Check PHPStan is at level:

5 or stricter when an argument uses a single type.
7 or stricter when an argument uses multiple types.

(level 9 is the most strict)

```php
<?php

$id = (string) ($_GET['id'] ?? '');

class db {

    /**
     * @phpstan-param literal-string $sql
     * @phpstan-param array<int, string> $parameters
     */

    public function query(string $sql, array $parameters = []) : void {

        // Send $sql and $parameters to the database.

    }

}

$db = new db();

$db->query('SELECT * FROM user WHERE id = ?', [$id]);

$db->query('SELECT * FROM user WHERE id = ' . $id);
```

Use 'literal-string' type for $sql

```php
<?php

$id = (string) ($_GET['id'] ?? '');

class db {
```

Terminal

```
craig$ vendor/bin/phpstan analyse --level 9 public
 1/1 [▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓] 100%

 ------ ------------------------------------------------------------------------------------
  Line   index.php
 ------ ------------------------------------------------------------------------------------
  24     Parameter #1 $sql of method db::query() expects literal-string, non-empty-string given.
 ------ ------------------------------------------------------------------------------------

 [ERROR] Found 1 error

craig$
```

```php
$db = new db();

$db->query('SELECT * FROM user WHERE id = ?', [$id]);

$db->query('SELECT * FROM user WHERE id = ' . $id);
```

How about Identifiers in SQL?

If you cannot use an "allow list" of developer defined strings...

Using Identifiers

```php
<?php

    $db = new db();

    $db->query('SELECT name FROM {table}', [], ['table' => $table]);

    $db->query('SELECT name FROM ' . $table); // REJECTED

?>
```

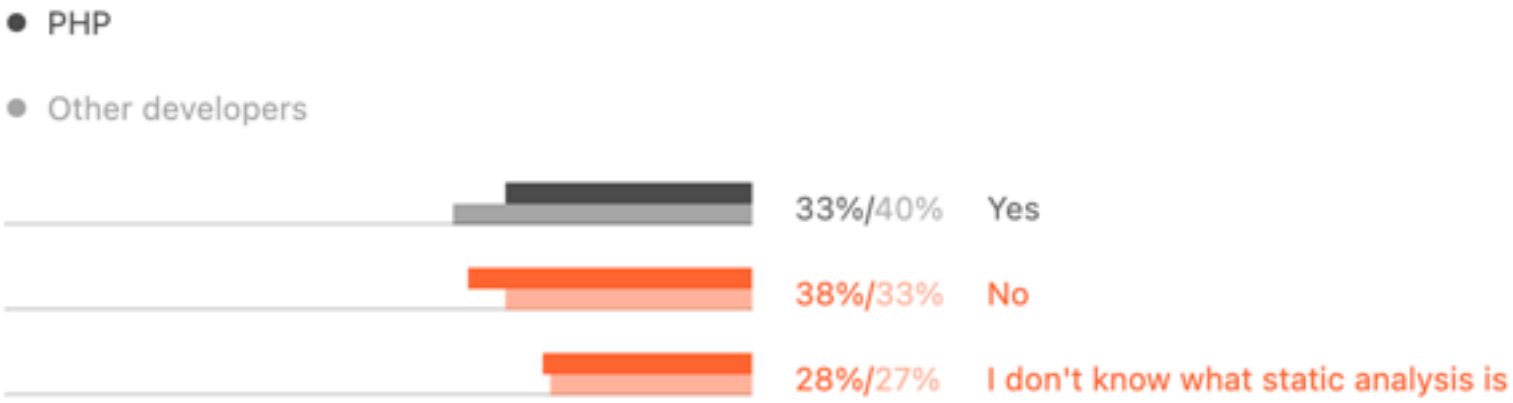
Line: 1 PHP Tab Size: 4 Symbols

The Future

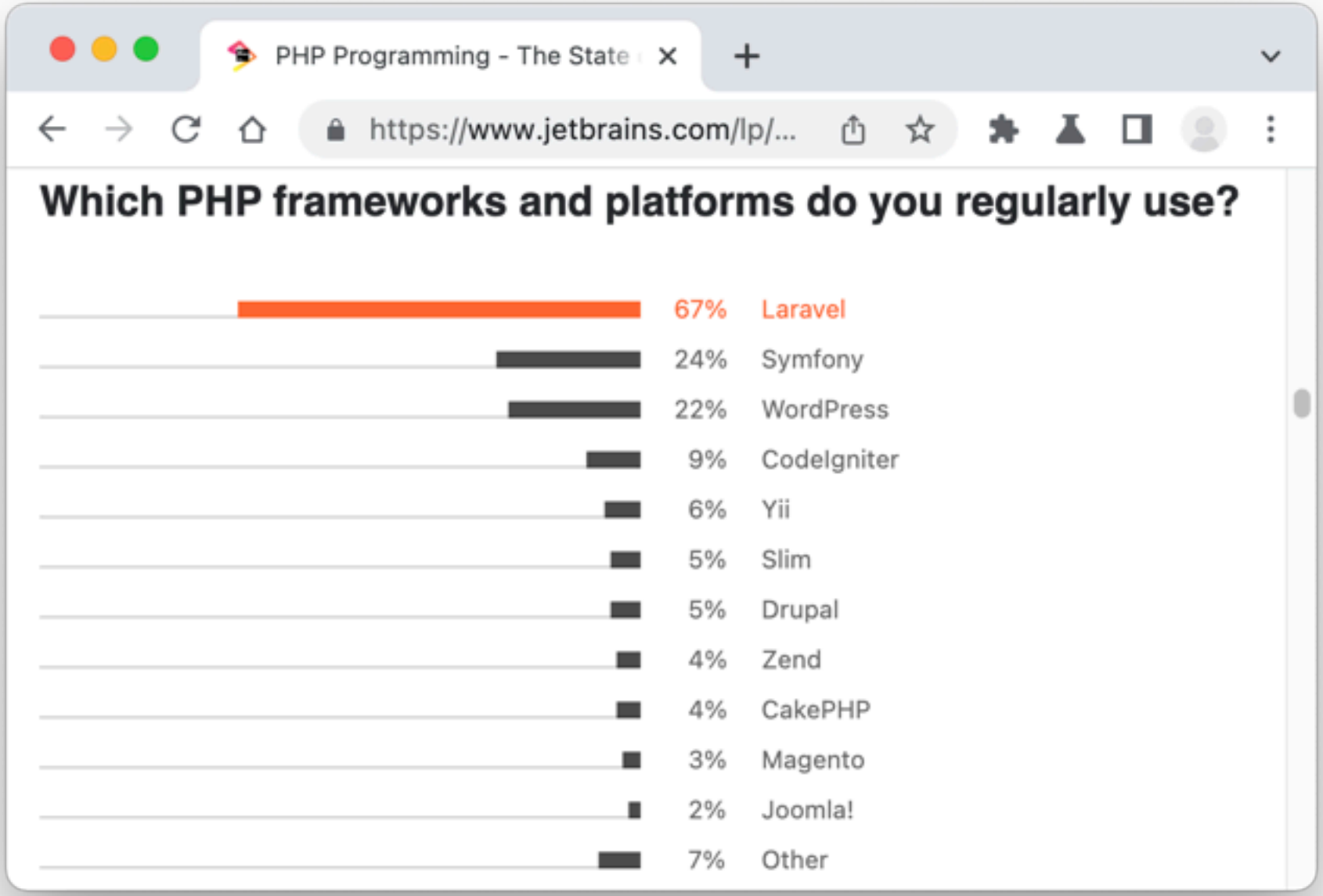
PHP Programming - The State

https://www.jetbrains.com/lp/...

Do you use static analysis?



The popularity of static analysis tools in the PHP ecosystem continues to grow. Although, when compared to other languages, PHP's static analysis adoption is still below average.


PHP Programming - The State
https://www.jetbrains.com/lp/...
Which PHP frameworks and platforms do you regularly use?
67% Laravel
24% Symfony
22% WordPress
9% CodeIgniter
6% Yii
5% Slim
5% Drupal
4% Zend
4% CakePHP
3% Magento
2% Joomla!
7% Other

Make it a key part of the Programming Language

In Go, you can use an un-exported string type

This is how "go-safe-html" works.

In Rust, you can use a Procedural Macro

Thanks Geoffroy Couprie

In C++, you can use a consteval annotation

Thanks Jonathan Müller

In Java, you can use @CompileTimeConstant annotation

Thanks to ErrorProne from Google

In Node, you can use goog.string.Const

Thanks to Google's Closure Library

In Node, you can use isTemplateObject

From the polyfill package "is-template-object"
Thanks Mike Samuel

In JavaScript, hopefully one day you can use:

isTemplateObject

or

TrustedHTML.fromLiteral

Thanks to Krzysztof Kotowicz

PHP, and the is_literal() RFC

Thanks to Joe Watkins and Máté Kocsis

PHP: rfc:is_literal

https://wiki.php.net/rfc/is_literal

php Edit this page Admin Logout Craig Francis (craigfrancis) Search

start › rfc › is_literal

PHP RFC: Is_Literal

- Version: 1.1
- Voting Start: 2021-07-05 19:30 BST / 18:30 UTC
- Voting End: 2021-07-19 19:30 BST / 18:30 UTC
- RFC Started: 2020-03-21
- RFC Updated: 2021-07-04
- Author: Craig Francis, craig#at#craigfrancis.co.uk
- Contributors: Joe Watkins, Máté Kocsis
- Status: Voting
- First Published at: https://wiki.php.net/rfc/is_literal
- GitHub Repo: https://github.com/craigfrancis/php-is-literal-rfc
- Implementation: https://github.com/php/php-src/compare/master...krakjoe:literals

Introduction

Add the function *is_literal()*, a lightweight and effective way to identify if a string was written by a developer, removing the risk of a variable containing an Injection Vulnerability.

– Table of Contents

No dependencies.

Easy to use.

No *need* to use Static Analysis,

But works very well with it.

Works with existing code / libraries.

e.g. No need to re-write everything to use a query builder.

You can choose how to handle mistakes.

Log to a file/db/api, throw an exception, do nothing, etc.

Libraries won't need everyone to read/understand *all* of their documentation.

Libraries won't rely on developers never making a mistake.

```php
<?php

class db {

    private function literal_check($var) {

        if (function_exists('is_literal') && !is_literal($var)) {
            throw new Exception('Non-literal value detected!');
        }

    }

    public function query($sql, $parameters = []) {

        $this->literal_check($sql);

        // Send $sql and $parameters to the database.

    }

```

Backwards Compatibility

```php
<?php

class db {

    private function literal_check($var) {

        if (function_exists('is_literal') && !is_literal($var)) {
            throw new Exception('Non-literal value detected!');
        }

    }

    public function query($sql, $parameters = []) {

        $this->literal_check($sql);

        // Send $sql and $parameters to the database.

    }

```

Run the check

```php
<?php

class db {

    private function literal_check($var) {

        if (function_exists('is_literal') && !is_literal($var)) {
            throw new Exception('Non-literal value detected!');
        }

    }

    public function query($sql, $parameters = []) {

        $this->literal_check($sql);

        // Send $sql and $parameters to the database.

    }

```

Too Strict?

```php
<?php

class db {

    private $protection_level = 1;
        // 0 = No checks, could be useful on the production server.
        // 1 = Just warnings, the default.
        // 2 = Exceptions, for anyone who wants to be absolutely sure.

    public function enforce_injection_protection() {
        $this->protection_level = 2;
    }

    public function unsafe_disable_injection_protection() {
        $this->protection_level = 0; // Not recommended, try `new unsafe_value('XXX')`
    }

    private function literal_check($var) {
        if (!function_exists('is_literal') || is_literal($var)) {
            // Fine - This is a programmer defined string (bingo), or not using PHP 8.X
        } else if ($var instanceof unsafe_value) {
            // Fine - Not ideal, but at least they know this one is unsafe.
        } else if ($this->protection_level === 0) {
            // Fine - Programmer aware, and is choosing to disable this check everywhere.
        } else if ($this->protection_level === 1) {
            trigger_error('Non-literal value detected!', E_USER_WARNING);
        } else {
            throw new Exception('Non-literal value detected!');
        }
    }
```

Protection Level

```php
<?php

class db {

    private $protection_level = 1;
        // 0 = No checks, could be useful on the production server.
        // 1 = Just warnings, the default.
        // 2 = Exceptions, for anyone who wants to be absolutely sure.

    public function enforce_injection_protection() {
        $this->protection_level = 2;
    }

    public function unsafe_disable_injection_protection() {
        $this->protection_level = 0; // Not recommended, try `new unsafe_value('XXX')`
    }

    private function literal_check($var) {
        if (!function_exists('is_literal') || is_literal($var)) {
            // Fine - This is a programmer defined string (bingo), or not using PHP 8.X
        } else if ($var instanceof unsafe_value) {
            // Fine - Not ideal, but at least they know this one is unsafe.
        } else if ($this->protection_level === 0) {
            // Fine - Programmer aware, and is choosing to disable this check everywhere.
        } else if ($this->protection_level === 1) {
            trigger_error('Non-literal value detected!', E_USER_WARNING);
        } else {
            throw new Exception('Non-literal value detected!');
        }
    }
```

Private function,
Used by the library

```php
<?php

class db {

    private $protection_level = 1;
        // 0 = No checks, could be useful on the production server.
        // 1 = Just warnings, the default.
        // 2 = Exceptions, for anyone who wants to be absolutely sure.

    public function enforce_injection_protection() {
        $this->protection_level = 2;
    }

    public function unsafe_disable_injection_protection() {
        $this->protection_level = 0; // Not recommended, try `new unsafe_value('XXX')`
    }

    private function literal_check($var) {
        if (!function_exists('is_literal') || is_literal($var)) {
            // Fine - This is a programmer defined string (bingo), or not using PHP 8.X
        } else if ($var instanceof unsafe_value) {
            // Fine - Not ideal, but at least they know this one is unsafe.
        } else if ($this->protection_level === 0) {
            // Fine - Programmer aware, and is choosing to disable this check everywhere.
        } else if ($this->protection_level === 1) {
            trigger_error('Non-literal value detected!', E_USER_WARNING);
        } else {
            throw new Exception('Non-literal value detected!');
        }
    }
```

Special Cases?

```php
<?php

class unsafe_value {

    private $value = '';

    function __construct($unsafe_value) {
        $this->value = $unsafe_value;
    }

    function __toString() {
        return $this->value;
    }

}

?>
```

Line: 1 PHP Tab Size: 4 Symbols

A Stringable Value Object.

Should not be needed.

```
$unsafe = new unsafe_value('Something ' . $weird);
```

Easy for Auditor to find :-)

WTF???

```
$unsafe = new unsafe_value('WHERE id ' . $comparison . ' ?');

$db->query($unsafe, [$id]);
```

How about Identifiers in SQL?

If you cannot use an "allow list" of developer defined strings...

Using Identifiers

```php
<?php

    $db = new db();

    $db->query('SELECT name FROM {table}', [], ['table' => $table]);

    $db->query('SELECT name FROM ' . $table); // REJECTED

?>
```

Line: 1 PHP Tab Size: 4 Symbols

```php
<?php

class db {

    // ...

    function query($sql, $parameters = [], $identifiers = []) {

        $this->literal_check($sql);

        foreach ($identifiers as $name => $value) {
            if (!preg_match('/^[a-z0-9_]+$/', $name)) {
                throw new Exception('Invalid identifier name "' . $name . '"');
            } else if (!preg_match('/^[a-z0-9_]+$/', $value)) {
                throw new Exception('Invalid identifier value "' . $value . '"');
            } else {
                $sql = str_replace('{' . $name . '}', '`' . $value . '`', $sql);
            }
        }

        // Send $sql and $parameters to the database.

    }

```

Variable Identifiers can still be risky (see the ORDER BY example).

But this shows how special values (e.g. from INI/JSON/YAML files) can be used safely *after* checking the SQL has been written by the programmer.

```php
<?php

class db {

    // ...

    function query($sql, $parameters = [], $identifiers = []) {

        $this->literal_check($sql);

        foreach ($identifiers as $name => $value) {
            if (!preg_match('/^[a-z0-9_]+$/', $name)) {
                throw new Exception('Invalid identifier name "' . $name . '"');
            } else if (!preg_match('/^[a-z0-9_]+$/', $value)) {
                throw new Exception('Invalid identifier value "' . $value . '"');
            } else {
                $sql = str_replace('{' . $name . '}', '`' . $value . '`', $sql);
            }
        }

        // Send $sql and $parameters to the database.

    }

```

```php
<?php

class db {

    // ...

    function query($sql, $parameters = [], $identifiers = []) {

        $this->literal_check($sql);

        foreach ($identifiers as $name => $value) {
            if (!preg_match('/^[a-z0-9_]+$/', $name)) {
                throw new Exception('Invalid identifier name "' . $name . '"');
            } else if (!preg_match('/^[a-z0-9_]+$/', $value)) {
                throw new Exception('Invalid identifier value "' . $value . '"');
            } else {
                $sql = str_replace('{' . $name . '}', '`' . $value . '`', $sql);
            }
        }

        // Send $sql and $parameters to the database.

    }

```

Over to you :-)

```
$articles->where('field', $value);
```

1.php — Downloads

```php
<?php

class orm {

    // ...

    function where($field, $value = NULL) {

        $this->literal_check($field);

        // ...

    }

}

?>
```

Line: 1 PHP Tab Size: 4 Symbols

Run the check

$users->order($order);

```php
<?php

class orm {

    // ...

    function order($sql) {

        $this->literal_check($sql);

        // ...

    }

}

?>
```

Line: 1 PHP Tab Size: 4 Symbols

Run the check

CLI

```
parameterised_exec('grep ? /path/to/file', [$search]);
```

First argument is checked

```
parameterised_exec('grep "' . $search . '" /path/to/file');
```

First argument is checked

```php
<?php

function parameterised_exec($cmd, $parameters = []) {

  if (!is_literal($cmd)) {
    throw new Exception('The first argument must be a literal');
  }

  $offset = 0;
  $k = 0;
  while (($pos = strpos($cmd, '?', $offset)) !== false) {
    if (!isset($parameters[$k])) {
      throw new Exception('Missing parameter "' . ($k + 1) . '"');
      exit();
    }
    $par = escapeshellarg($parameters[$k]);
    $cmd = substr($cmd, 0, $pos) . $par . substr($cmd, ($pos + 1));
    $offset = ($pos + strlen($par));
    $k++;
  }
  if (isset($parameters[$k])) {
    throw new Exception('Unused parameter "' . ($k + 1) . '"');
    exit();
  }

  return exec($cmd);

}
```

Strict Check

```php
<?php

function parameterised_exec($cmd, $parameters = []) {

  if (!is_literal($cmd)) {
    throw new Exception('The first argument must be a literal');
  }

  $offset = 0;
  $k = 0;
  while (($pos = strpos($cmd, '?', $offset)) !== false) {
    if (!isset($parameters[$k])) {
      throw new Exception('Missing parameter "' . ($k + 1) . '"');
      exit();
    }
    $par = escapeshellarg($parameters[$k]);
    $cmd = substr($cmd, 0, $pos) . $par . substr($cmd, ($pos + 1));
    $offset = ($pos + strlen($par));
    $k++;
  }
  if (isset($parameters[$k])) {
    throw new Exception('Unused parameter "' . ($k + 1) . '"');
    exit();
  }

  return exec($cmd);

}
```

Escaping all values

Apply $parameters

```php
<?php

function parameterised_exec($cmd, $parameters = []) {

  if (!is_literal($cmd)) {
    throw new Exception('The first argument must be a literal');
  }

  $offset = 0;
  $k = 0;
  while (($pos = strpos($cmd, '?', $offset)) !== false) {
    if (!isset($parameters[$k])) {
      throw new Exception('Missing parameter "' . ($k + 1) . '"');
      exit();
    }
    $par = escapeshellarg($parameters[$k]);
    $cmd = substr($cmd, 0, $pos) . $par . substr($cmd, ($pos + 1));
    $offset = ($pos + strlen($par));
    $k++;
  }
  if (isset($parameters[$k])) {
    throw new Exception('Unused parameter "' . ($k + 1) . '"');
    exit();
  }

  return exec($cmd);

}
```

Run $cmd

HTML

```
$template->render('<p>Hi {{ name }}</p>', ['name' => $name]);
```

First argument is checked

```
$template->render('<p>Hi ' . $name . '</p>');
```

First argument is checked

About 300 Lines :-)

Protection Level

Allowed tags, attributes, and attribute values

HTML Parsing... in XML mode :-)

Node walking, to find "?" for parameters

Return HTML with (checked) parameters

The `unsafe_value` object (should not be needed)

```
$html = ht('<p>Hi <span>?</span></p>', [$name]);
```

```
$template = ht('<p>Hi <span>?</span></p>');


$html_1 = $template->html([$name_1]);
$html_2 = $template->html([$name_2]);
$html_3 = $template->html([$name_3]);
```

```
$html = ht('<p>Hi ' . $name . '</p>');
```

```
$html = ht('<p>Hi ' . $name . '</p>');
```

```
$url = 'https://example.com';

$html = ht('<a href="?">Link</p>', [$url]);
```

```
$url = 'javascript:alert()';

$html = ht('<a href="?">Link</p>', [$url]);
```

And in ~10 years time...

Native functions,

like mysqli::prepare() and PDO::prepare()...

Accept everything,

but warn if not given a "developer defined string"...

But there would need to be a way for special cases to be trusted...

e.g. strings created by a library.

```php
<?php

class unsafe_value {

    private $value = '';

    function __construct($unsafe_value) {
        $this->value = $unsafe_value;
    }

    function __toString() {
        return $this->value;
    }

}

?>
```

Line: 1 PHP Tab Size: 4 Symbols

Maybe a "stringable value-object"?

```php
<?php

class db_trusted_sql {

    private $sql = '';

    function __construct($sql) {
        $this->sql = $sql;
    }

    function __toString() {
        return $this->sql;
    }

}

?>
```

Line: 1 PHP Tab Size: 4 Symbols

Some way to trust the stringable value object

What it can be trusted for.

```php
class db {

    //...

    function __construct() {

        // ...

        spl_trust_object('db_trusted_sql', 'sql');

    }

    function query($sql, $parameters = []) {

        // ...

        $trusted_sql = new db_trusted_sql($sql);

        mysqli_query($this->link, $trusted_sql);

    }

```

Accepted by the native function

There would be a way to disable the check.

At least the developer is then aware their code is unsafe.

There would be a way to enforce the check.

For developers, confident in their system, to enforce this protection.

"Distinguishing strings from a trusted developer, from strings that may be attacker controlled"

Mike Samuel - 27th March 2019

Thank You

Questions?

https://eiv.dev/

@craigfrancis