is_literal()

A proposed function for PHP

Craig Francis

is_literal()

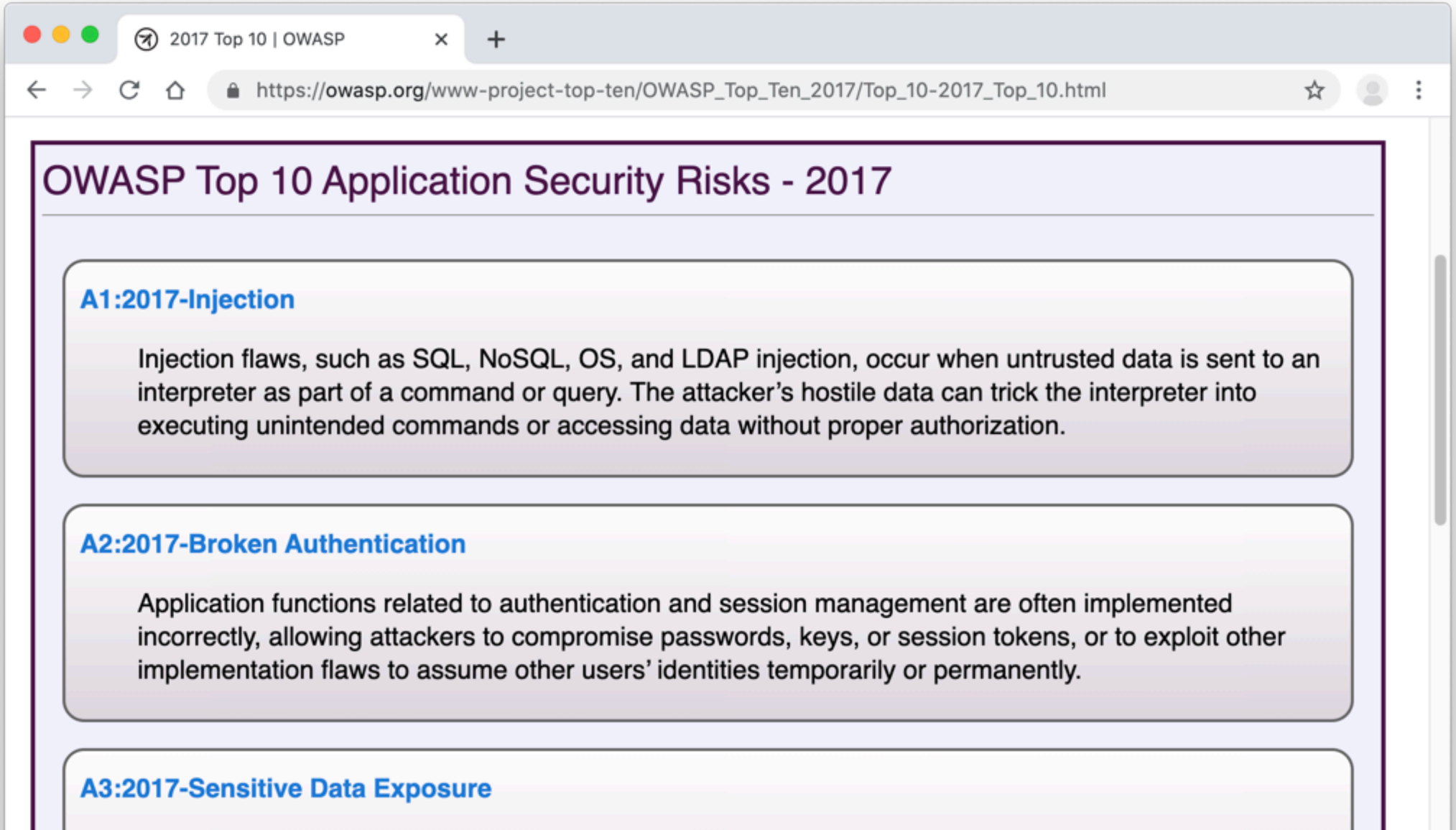
To distinguish between safe strings,
which have been defined in your PHP script.

is_literal()

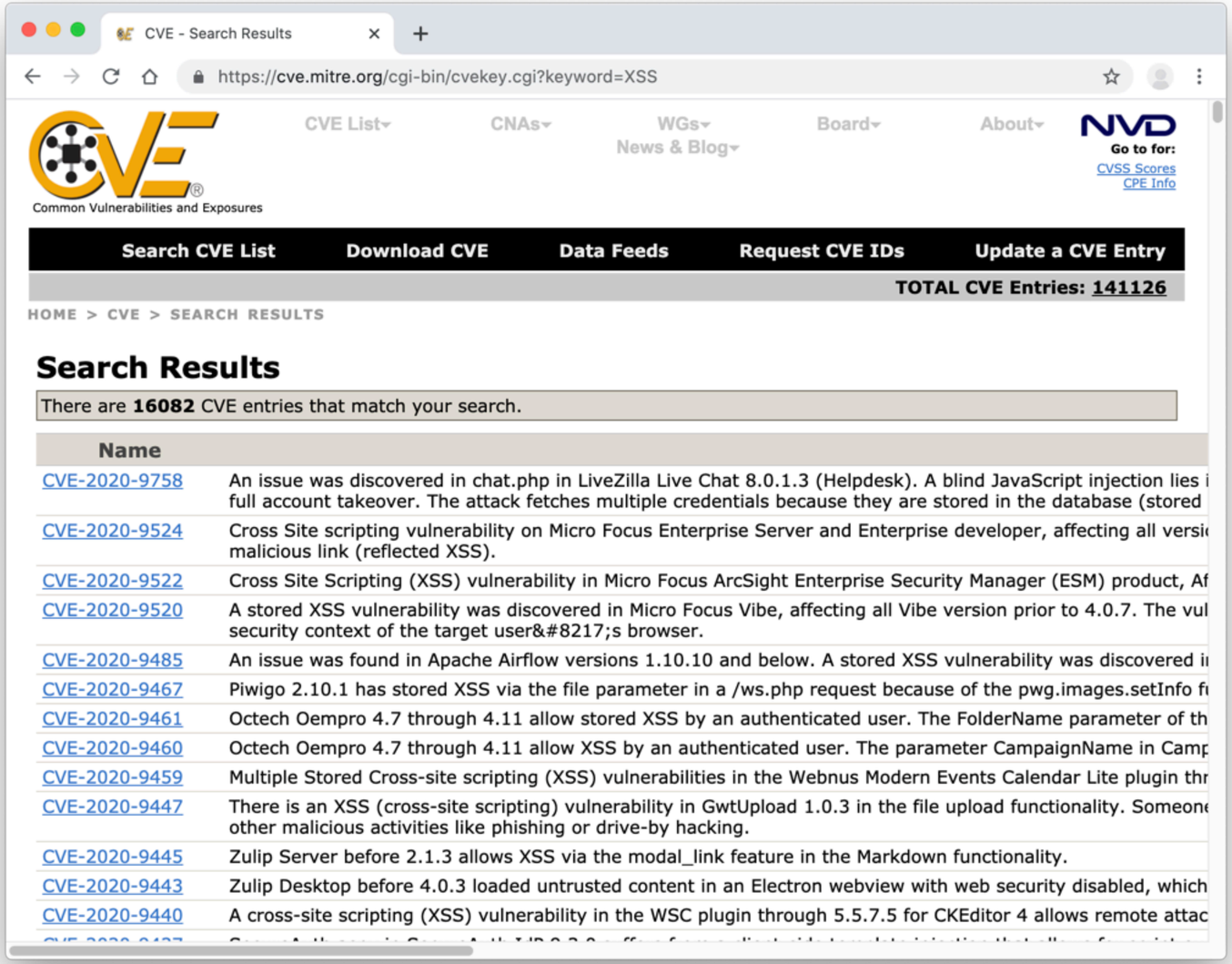
To distinguish between safe strings,
which have been defined in your PHP script.

VS unsafe strings,
which have been tainted by external sources.

The Problem


2017 Top 10 | OWASP
https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_Top_10.html
OWASP Top 10 Application Security Risks - 2017
A1:2017-Injection
Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
A2:2017-Broken Authentication
Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently.
A3:2017-Sensitive Data Exposure

The Problem



CVE - Search Results

https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=XSS

CVE List CNAs WGs News & Blog Board About NVD

Go to for: CVSS Scores CPE Info

Common Vulnerabilities and Exposures

Search CVE List | Download CVE | Data Feeds | Request CVE IDs | Update a CVE Entry

TOTAL CVE Entries: 141126

HOME > CVE > SEARCH RESULTS

Search Results

There are **16082** CVE entries that match your search.

Name	
CVE-2020-9758	An issue was discovered in chat.php in LiveZilla Live Chat 8.0.1.3 (Helpdesk). A blind JavaScript injection lies i full account takeover. The attack fetches multiple credentials because they are stored in the database (stored
CVE-2020-9524	Cross Site scripting vulnerability on Micro Focus Enterprise Server and Enterprise developer, affecting all versi malicious link (reflected XSS).
CVE-2020-9522	Cross Site Scripting (XSS) vulnerability in Micro Focus ArcSight Enterprise Security Manager (ESM) product, Af
CVE-2020-9520	A stored XSS vulnerability was discovered in Micro Focus Vibe, affecting all Vibe version prior to 4.0.7. The vul security context of the target user’s browser.
CVE-2020-9485	An issue was found in Apache Airflow versions 1.10.10 and below. A stored XSS vulnerability was discovered i
CVE-2020-9467	Piwigo 2.10.1 has stored XSS via the file parameter in a /ws.php request because of the pwg.images.setInfo f
CVE-2020-9461	Octech Oempro 4.7 through 4.11 allow stored XSS by an authenticated user. The FolderName parameter of th
CVE-2020-9460	Octech Oempro 4.7 through 4.11 allow XSS by an authenticated user. The parameter CampaignName in Camp
CVE-2020-9459	Multiple Stored Cross-site scripting (XSS) vulnerabilities in the Webnus Modern Events Calendar Lite plugin thr
CVE-2020-9447	There is an XSS (cross-site scripting) vulnerability in GwtUpload 1.0.3 in the file upload functionality. Someone other malicious activities like phishing or drive-by hacking.
CVE-2020-9445	Zulip Server before 2.1.3 allows XSS via the modal_link feature in the Markdown functionality.
CVE-2020-9443	Zulip Desktop before 4.0.3 loaded untrusted content in an Electron webview with web security disabled, which
CVE-2020-9440	A cross-site scripting (XSS) vulnerability in the WSC plugin through 5.5.7.5 for CKEditor 4 allows remote attac

The Problem

CVE - Search Results

https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=SQL%20injection

CVE List CNAs WGs News & Blog Board About NVD

Go to for: CVSS Scores CPE Info

Common Vulnerabilities and Exposures

Search CVE List | Download CVE | Data Feeds | Request CVE IDs | Update a CVE Entry

TOTAL CVE Entries: 141126

HOME > CVE > SEARCH RESULTS

Search Results

There are **8419** CVE entries that match your search.

Name	Description
CVE-2020-9521	An SQL injection vulnerability was discovered in Micro Focus Service Manager Automation (SMA), affecting versions 2019.08, 2019.05, 2019.02, 2018.08, 2018.05, 2018.02. The vulnerability could allow for the improper neutralization of special elements in SQL commands and may lead to the product being vulnerable to SQL injection.
CVE-2020-9483	**Resolved** When use H2/MySQL/TiDB as Apache SkyWalking storage, the metadata query through GraphQL protocol, there is a SQL injection vulnerability, which allows to access unpexcted data. Apache SkyWalking 6.0.0 to 6.6.0, 7.0.0 H2/MySQL/TiDB storage implementations don't use the appropriate way to set SQL parameters.
CVE-2020-9465	An issue was discovered in EyesOfNetwork eonweb 5.1 through 5.3 before 5.3-3. The eonweb web interface is prone to a SQL injection, allowing an unauthenticated attacker to perform various tasks such as authentication bypass via the user_id field in a cookie.
CVE-2020-9402	Django 1.11 before 1.11.29, 2.2 before 2.2.11, and 3.0 before 3.0.4 allows SQL Injection if untrusted data is used as a tolerance parameter in GIS functions and aggregates on Oracle. By passing a suitably crafted tolerance to GIS functions and aggregates on Oracle, it was possible to break escaping and inject malicious SQL.
CVE-2020-9398	ISPConfig before 3.1.15p3, when the undocumented reverse_proxy_panel_allowed=sites option is manually enabled, allows SQL Injection.
CVE-2020-9340	fauzantrif eLection 2.0 has SQL Injection via the admin/ajax/op_kandidat.php id parameter.
CVE-2020-9318	Red Gate SQL Monitor 9.0.13 through 9.2.14 allows an administrative user to perform a SQL injection attack by configuring the SNMP alert settings in the UI. This is fixed in 9.2.15.
CVE-2020-9269	SQPlanning 1.45 is vulnerable to authenticated SQL Injection that leads to command execution via the

The Problem

Version 5.4.1 | WordPress.org

https://wordpress.org/support/wordpress-version/version-5-4-1/#security-updates

Register Log In

Security updates

Six security issues affect WordPress versions 5.4 and earlier; version 5.4.1 fixes them, so you'll want to upgrade. If you haven't yet updated to 5.4, there are also updated versions of 5.3 and earlier that fix the security issues.

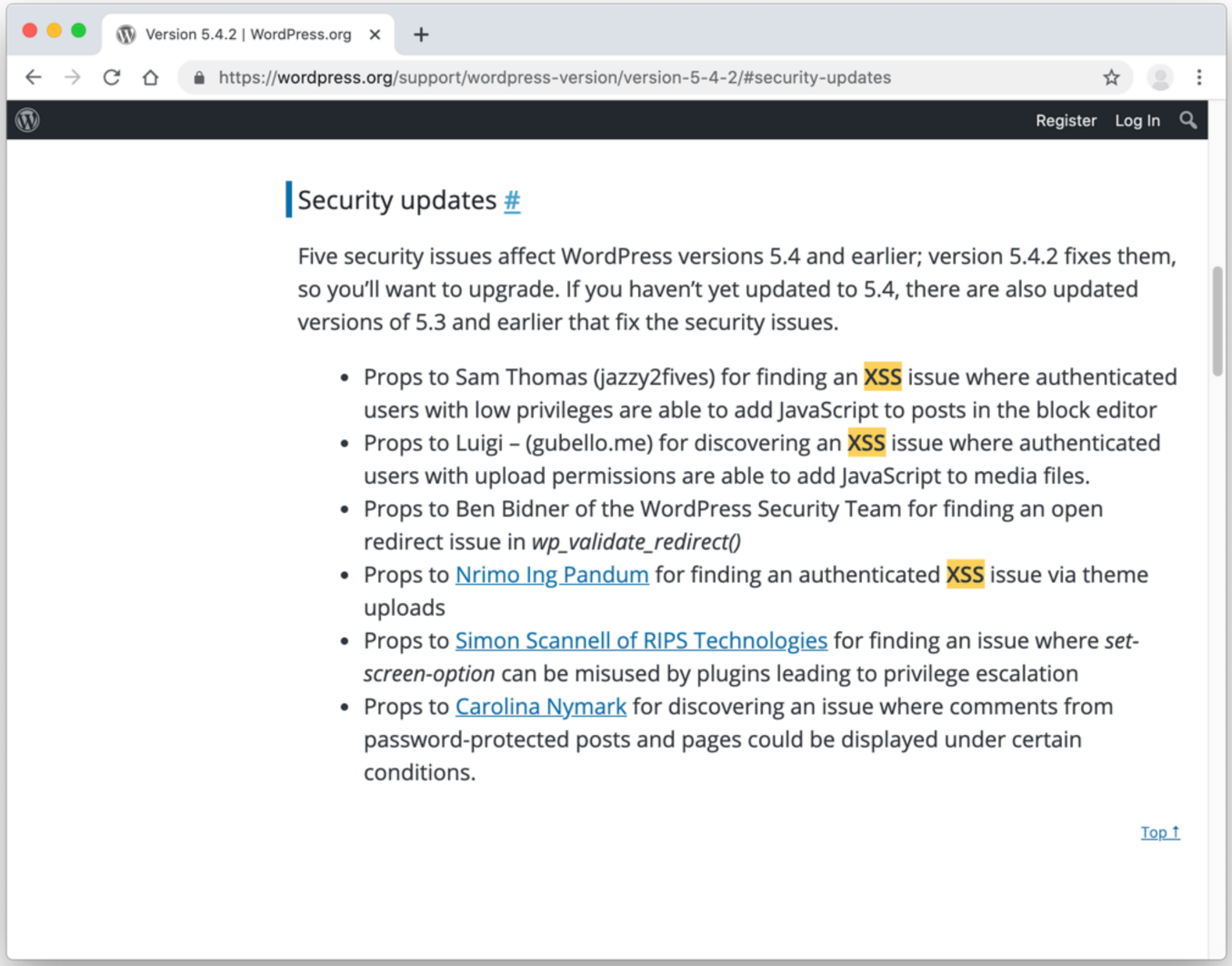
- Props to Muaz Bin Abdus Sattar and Jannes who both independently reported an issue where password reset tokens were not properly invalidated
- Props to ka1n4t for finding an issue where certain private posts can be viewed unauthenticated
- Props to Evan Ricafort for discovering an XSS issue in the Customizer
- Props to Ben Bidner from the WordPress Security Team who discovered an XSS issue in the search block
- Props to Nick Daugherty from WPVIP.com / WordPress Security Team who discovered an XSS issue in wp-object-cache
- Props to Ronnie Goodrich (Kahoots) and Jason Medeiros who independently reported an XSS issue in file uploads.
- Additionally, an authenticated XSS issue in the block editor was discovered by Nguyen the Duc in WordPress 5.4 RC1 and RC2. It was fixed in 5.4 RC5. We wanted to be sure to give credit and thank them for all of their work in making WordPress more secure.

Top ↑

Version 5.4.1

April 29, 2020

The Problem



Version 5.4.2

June 10, 2020

Background

Google engineer Christoph Kern, 2015:

https://www.usenix.org/conference/usenixsecurity15/symposium-program/presentation/kern

Background

Google engineer Christoph Kern, 2015:

"Developer education doesn't solve the problem"

Background

Google engineer Christoph Kern, 2015:

"Bugs are hard to find after the fact"

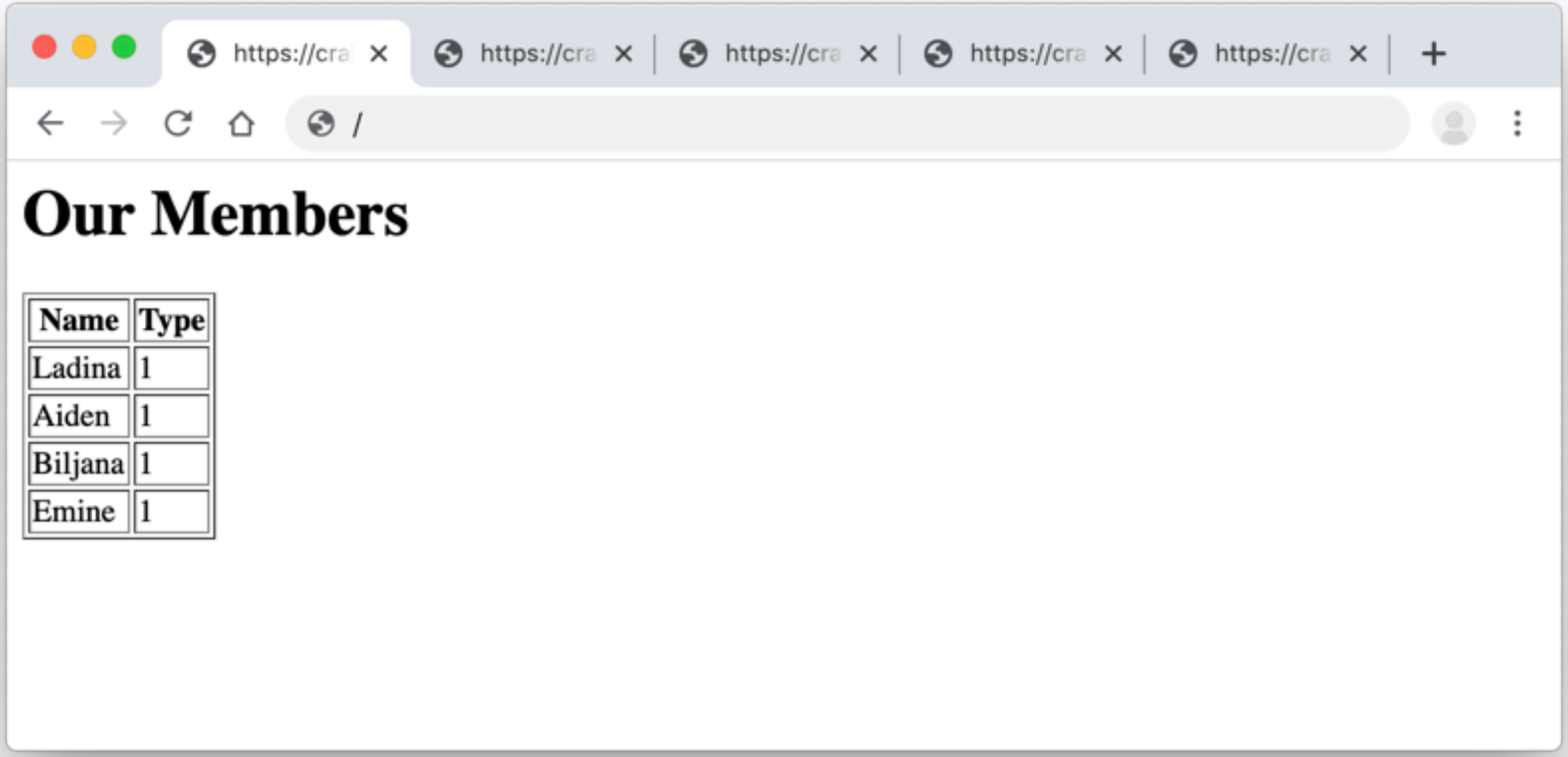
Background

Google engineer Christoph Kern, 2015:

"Bugs are hard to find after the fact"

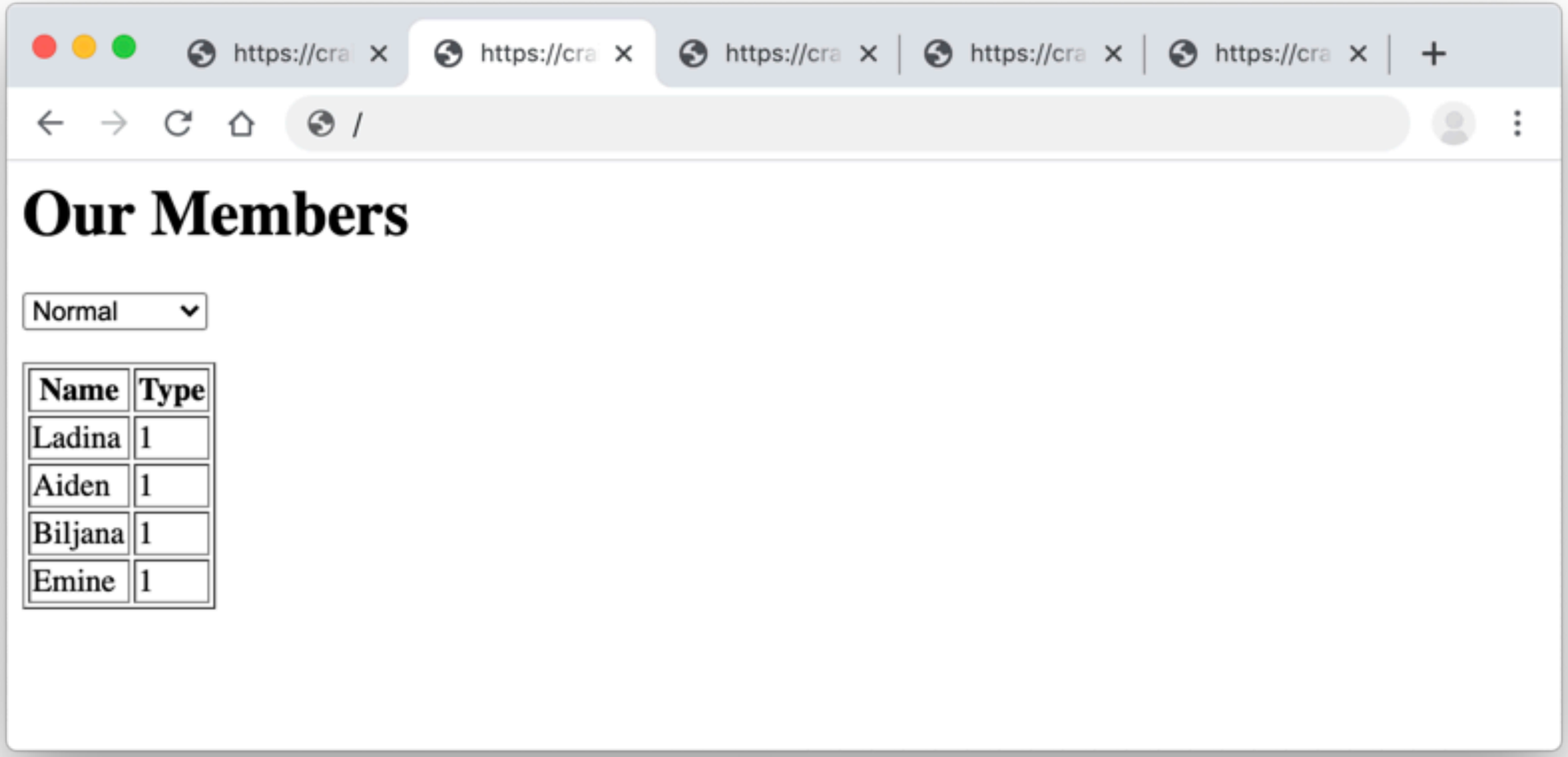
"Manual Testing, Automated Testing, Static Analysis, Human Code Reviews;
... will find *some* of these bugs..." @04:02

Examples

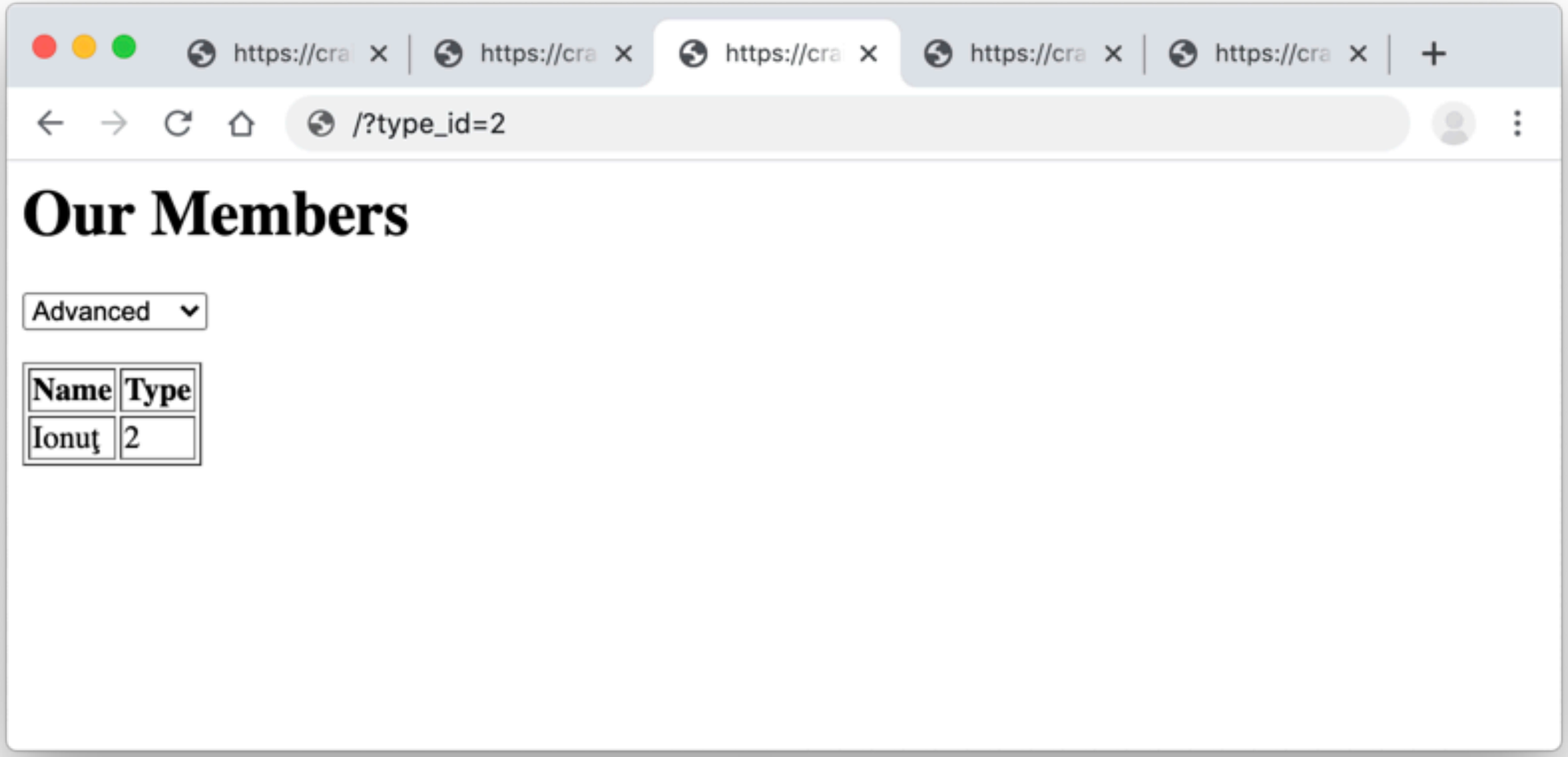


Name	Type
Ladina	1
Aiden	1
Biljana	1
Emine	1

Examples


https://cra
https://cra
https://cra
https://cra
https://cra
/
Our Members
Normal
Name
Type
Ladina
1
Aiden
1
Biljana
1
Emine
1

Examples


https://cra
https://cra
https://cra
https://cra
https://cra
/?type_id=2
Our Members
Advanced
Name
Type
Ionuţ
2

Examples

Doctrine QueryBuilder

```
$users = $queryBuilder
        ->select('u')
        ->from('User', 'u')
        ->where('u.type_id = 1')
        ->getQuery()
        ->getResult();
```

Examples

```
$users = $queryBuilder
        ->select('u')
        ->from('User', 'u')
        ->where('u.type_id = ?1')
        ->setParameter(1, $_GET['type_id'])
        ->getQuery()
        ->getResult();
```

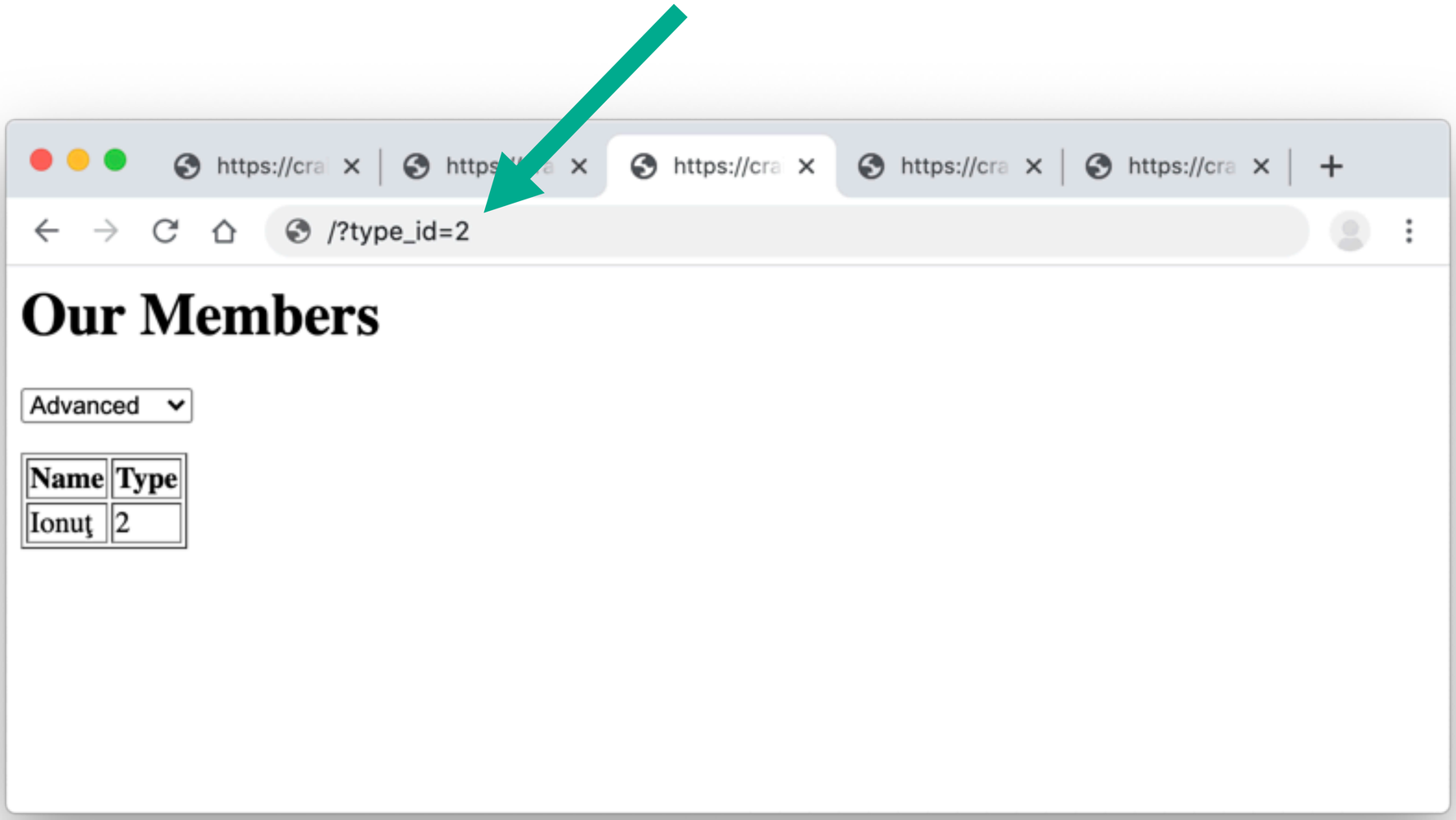
Examples

```
$users = $queryBuilder
            ->select('u')
            ->from('User', 'u')
            ->where('u.type_id = 1')
            ->getQuery()
            ->getResult();
```

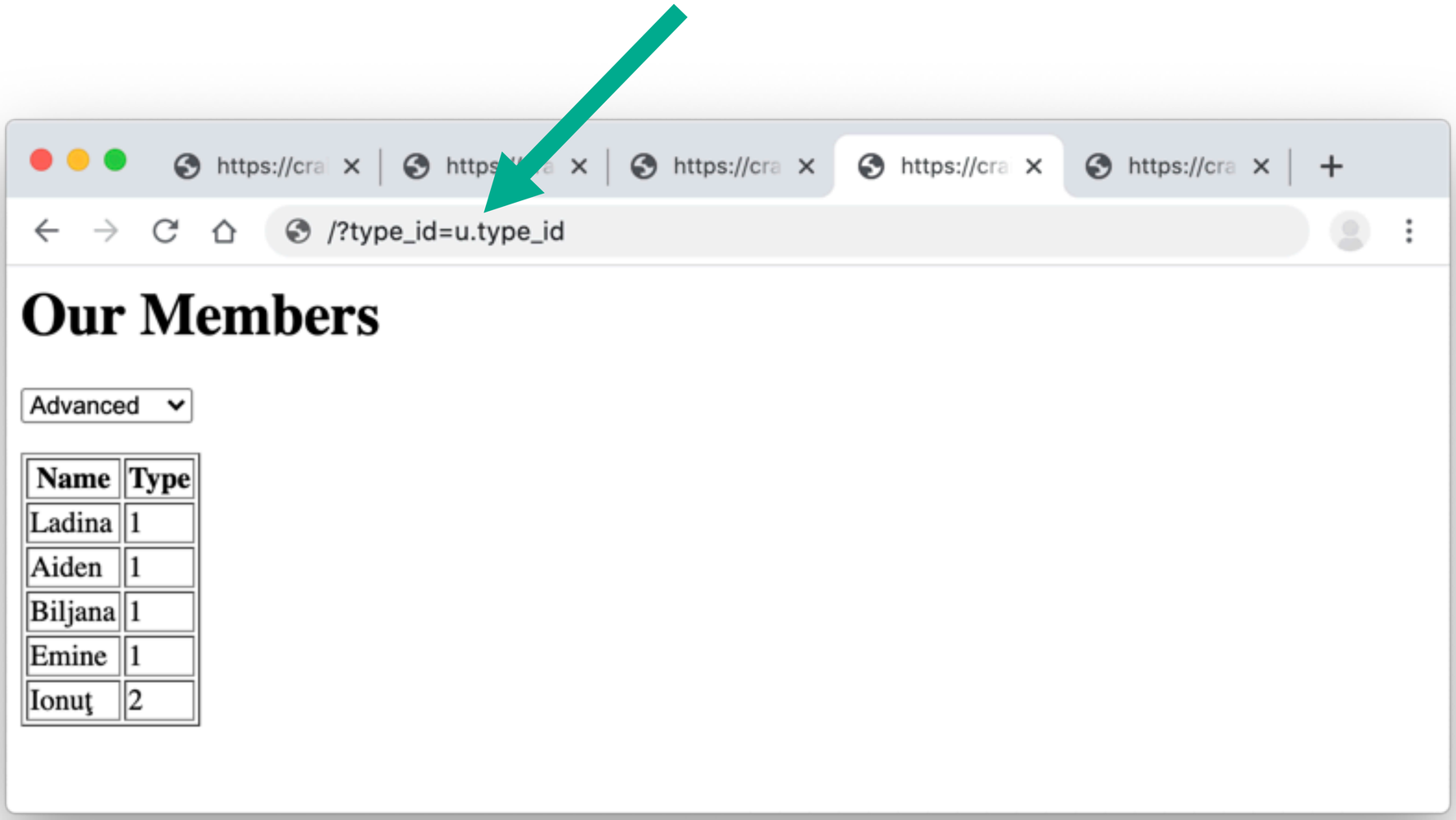
Examples

```
$users = $queryBuilder
            ->select('u')
            ->from('User', 'u')
            ->where('u.type_id = ' . $_GET['type_id'])
            ->getQuery()
            ->getResult();
```

Examples


https://cra
https://cra
https://cra
https://cra
https://cra
/?type_id=2
Our Members
Advanced
Name
Type
Ionuţ
2

Examples


https://cra
https://cra
https://cra
https://cra
https://cra
/?type_id=u.type_id
Our Members
Advanced
Name
Type
Ladina
1
Aiden
1
Biljana
1
Emine
1
Ionuţ
2

Examples

```
$users = $queryBuilder
            ->select('u')
            ->from('User', 'u')
            ->where('u.type_id = 1')
            ->getQuery()
            ->getResult();
```

Safe String, a Literal

Examples

Safe String

Unsafe String

```
$users = $queryBuilder
        ->select('u')
        ->from('User', 'u')
        ->where('u.type_id = ' . $_GET['type_id'])
        ->getQuery()
        ->getResult();
```

Examples

```
$users = $queryBuilder
        ->select('u')
        ->from('User', 'u')
        ->where('u.type_id = ' . $_GET['type_id'])
        ->getQuery()
        ->getResult();
```

'WHERE u.type_id = **u.type_id**'

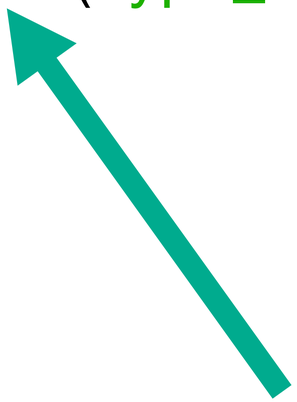
Examples

```
$sql = 'SELECT u FROM User u WHERE u.type_id = ' . $_GET['type_id'];

$query = $entityManager->createQuery($sql);
```

Doctrine - CreateQuery, DQL

Examples

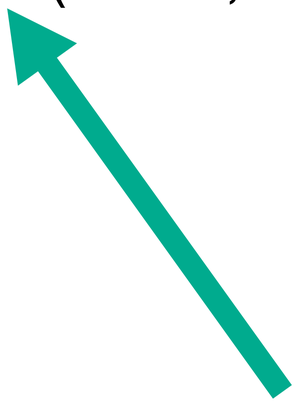
```
$users = UserQuery::create()->where('type_id = ' . $_GET['type_id'])->find();
```



Propel ORM - Where

Examples

```
$result = R::find('user', 'type_id = ' . $_GET['type_id']);
```



RedBeanPHP - Find

Examples

Safe String, a Literal

```
$template = $twig->createTemplate('<p>Hello {{ name }}</p>');
```

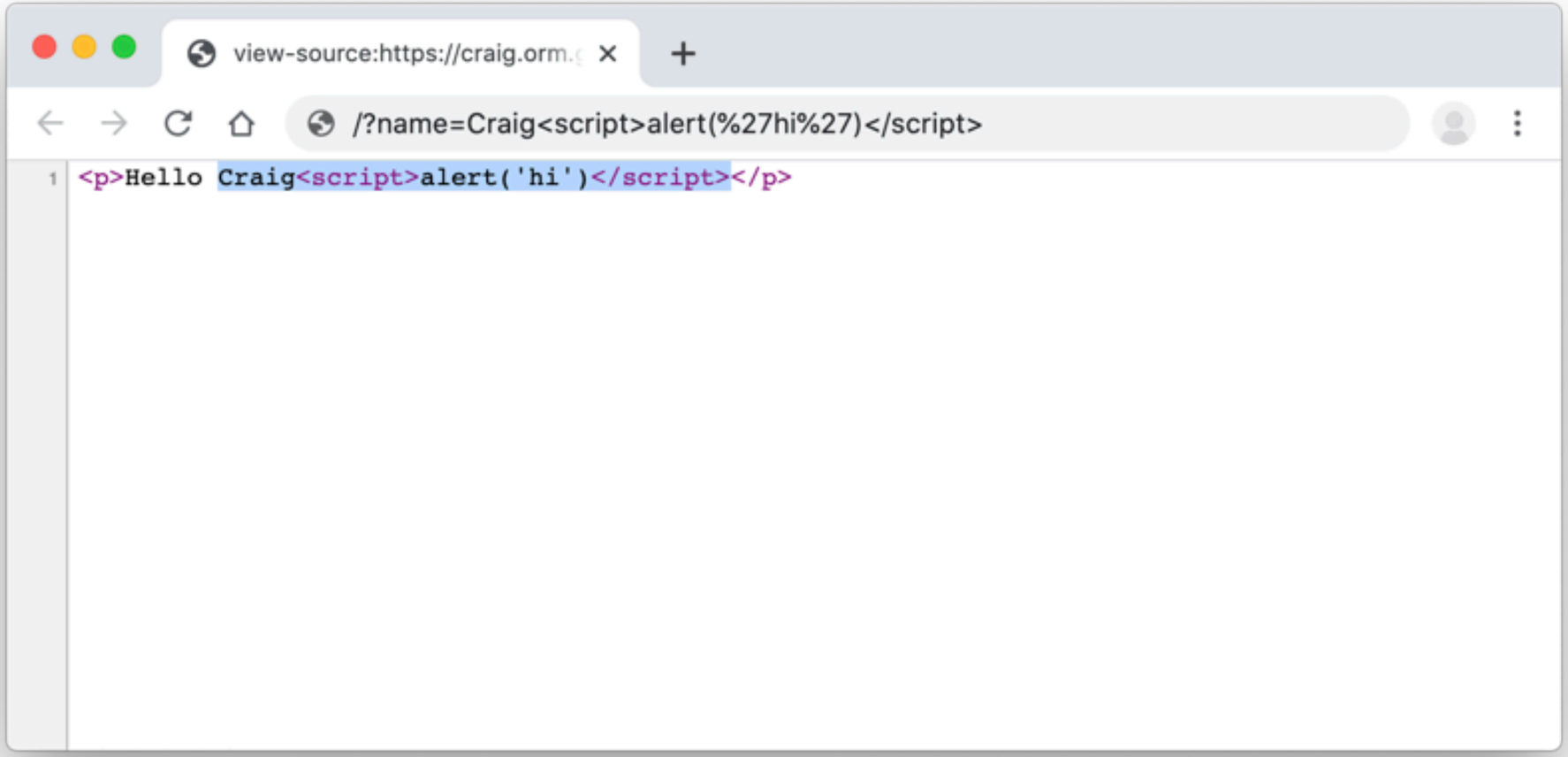
```
echo $template->render(['name' => $_GET['name']]);
```

Examples

Unsafe String

```
$template = $twig->createTemplate('<p>Hello ' . $_GET['name'] . '</p>');

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

Examples


view-source:https://craig.orm.
/?name=Craig<script>alert(%27hi%27)</script>
1 Hello Craig<script>alert('hi')</script>

Examples

Safe String, a Literal

```
$output = shell_exec('ls /');
```

Examples

Unsafe String

```
$output = shell_exec('ls ' . $_GET['path']);
```

Taint Checking

https://pecl.php.net/package/taint

Taint Checking

```
$prefix = 'Hi ';
```

```
$welcome_html = $prefix . 'Name';
```

Taint Checking

Untainted

```
$prefix = 'Hi ';
```

```
$welcome_html = $prefix . 'Name';
```

Taint Checking

Untainted

```
$prefix = 'Hi ';
```

Untainted

Untainted

```
$welcome_html = $prefix . 'Name';
```

Taint Checking

```
$prefix = 'Hi ';
```

Untainted, Safe

```
$welcome_html = $prefix . 'Name';
```

Taint Checking

```
$name = $_GET['name'];

$prefix = 'Hi ';

$welcome_html = $prefix . $name;
```

Taint Checking

Tainted

```
$name = $_GET['name'];

$prefix = 'Hi ';

$welcome_html = $prefix . $name;
```

Taint Checking

```
$name = $_GET['name'];

$prefix = 'Hi ';

$welcome_html = $prefix . $name;
```

Taint Checking

```
$name = $_GET['name'];
```

Untainted

```
$prefix = 'Hi ';
```

```
$welcome_html = $prefix . $name;
```

Taint Checking

```
$name = $_GET['name'];

$prefix = 'Hi ';

$welcome_html = $prefix . $name;
```

Untainted

Tainted

Taint Checking

```
$name = $_GET['name'];

$prefix = 'Hi ';
```

Tainted, not safe... complain when printed.

```
$welcome_html = $prefix . $name;
```

Taint Checking

```
$name = $_GET['name'];

$prefix = 'Hi ';

$welcome_html = $prefix . htmlentities($name);
```

Taint Checking

```
$name = $_GET['name'];

$prefix = 'Hi ';
```

Un-Taint

```
$welcome_html = $prefix . htmlentities($name);
```

Taint Checking

```
$name = $_GET['name'];
```

```
$prefix = 'Hi ';
```

Untainted, should be safe.

```
$welcome_html = $prefix . htmlentities($name);
```

Taint Checking

```
$url = $_GET['url'];

$name = $_GET['name'];

$link_html = '<a href="' . htmlentities($url) . '">' . htmlentities($name) . '</a>';
```

Taint Checking

Tainted

```
$url = $_GET['url'];
```

Tainted

```
$name = $_GET['name'];
```

```
$link_html = '<a href="' . htmlentities($url) . '">' . htmlentities($name) . '</a>';
```

Taint Checking

```
$url = $_GET['url'];

$name = $_GET['name'];
```

Tainted

Tainted

```
$link_html = '<a href="' . htmlentities($url) . '">' . htmlentities($name) . '</a>';
```

Taint Checking

```
$url = $_GET['url'];

$name = $_GET['name'];
```

Un-Taint

Un-Taint

```
$link_html = '<a href="' . htmlentities($url) . '">' . htmlentities($name) . '</a>';
```

Taint Checking

```
$url = $_GET['url'];
```

```
$name = $_GET['name'];
```

Untainted, surely this is safe?

```
$link_html = '<a href="' . htmlentities($url) . '">' . htmlentities($name) . '</a>';
```

Taint Checking

'javascript:alert("hi")'

```
$url = $_GET['url'];

$name = $_GET['name'];

$link_html = '<a href="' . htmlentities($url) . '">' . htmlentities($name) . '</a>';
```

Taint Checking

Tainted

```
$sql .= 'WHERE id = ' . mysqli_real_escape_string($link, $_GET['id']);
```

Taint Checking

Untainted

Un-Taint

```
$sql .= 'WHERE id = ' . mysqli_real_escape_string($link, $_GET['id']);
```

Taint Checking

Untainted, surely this is safe?

```
$sql .= 'WHERE id = ' . mysqli_real_escape_string($link, $_GET['id']);
```

Taint Checking

```
$sql .= 'WHERE id = ' . mysqli_real_escape_string($link, $_GET['id']);
```

Missing quotes

Taint Checking

"id"

```
$sql .= 'WHERE id = ' . mysqli_real_escape_string($link, $_GET['id']);
```

Taint Checking

"id"

```
$sql .= 'WHERE id = ' . mysqli_real_escape_string($link, $_GET['id']);
```

```
$sql .= 'WHERE id = id';
```

Taint Checking

Untainted Tainted Untainted

```
$img_html = '<img src=' . htmlentities($_GET['url']) . '>';
```

Taint Checking

Un-Taint

```
$img_html = '<img src=' . htmlentities($_GET['url']) . '>';
```

Taint Checking

Untainted, surely this is safe?

```
$img_html = '<img src=' . htmlentities($_GET['url']) . '>';
```

Taint Checking

```
$img_html = '<img src=' . htmlentities($_GET['url']) . '>';
```

Missing quotes

Taint Checking

"/ onerror=evil-js"

```
$img_html = '<img src=' . htmlentities($_GET['url']) . '>';
```

```
$img_html = '<img src=/ onerror=evil-js>';
```

Taint Checking

How about character encoding issues?

Taint Checking

The PDO Quote function clearly says it's *only* "theoretically safe".

https://www.php.net/pdo.quote

Static Analysis

Static Analysis

Injection problems are rarely solved by Static Analysis.

Static Analysis

Injection problems are rarely solved by Static Analysis.

Still do use this.

Static Analysis

Injection problems are rarely solved by Static Analysis.

But it's hard to follow *every* variable from Source to Sink.

Static Analysis

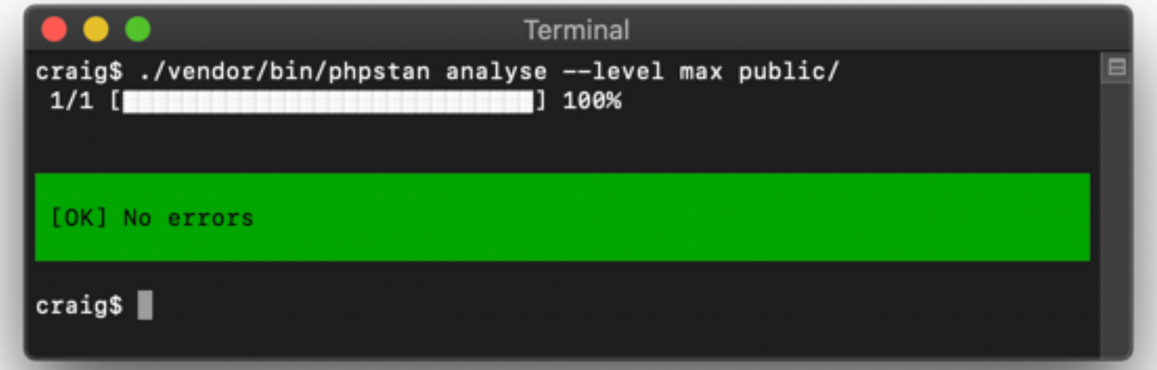
Injection problems are rarely solved by Static Analysis.

And don't expect every programmer to use.

Static Analysis

```
$mysqli = new mysqli('localhost', 'test', '???', 'test');

$result = $mysqli->query('SELECT * FROM user WHERE id = ' . $_GET['id']);

if ($result instanceof mysqli_result) {

    print_r($result->fetch_all());

}
```

PHPStan

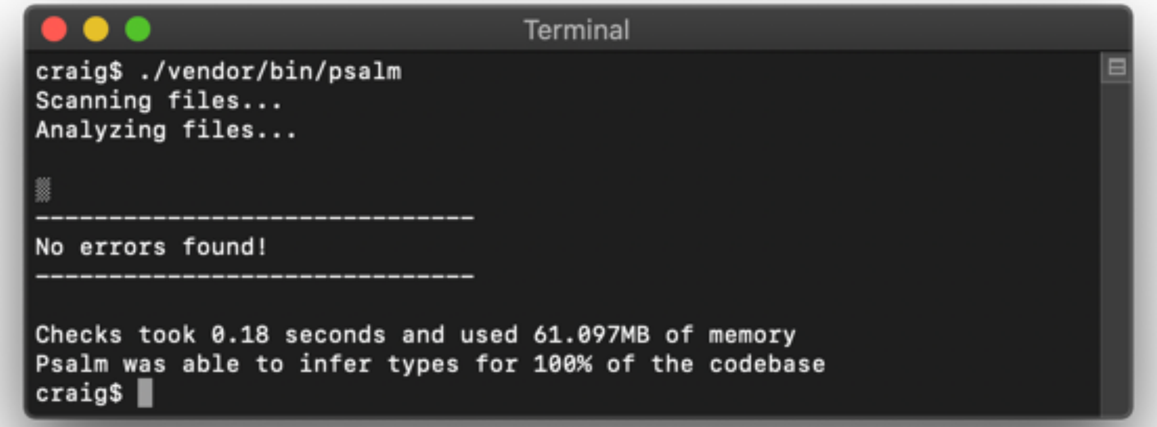


Static Analysis

```
$mysqli = new mysqli('localhost', 'test', '???', 'test');

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

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

if ($result instanceof mysqli_result) {

    print_r($result->fetch_all());

}
```

Avoid MixedAssignment

Psalm

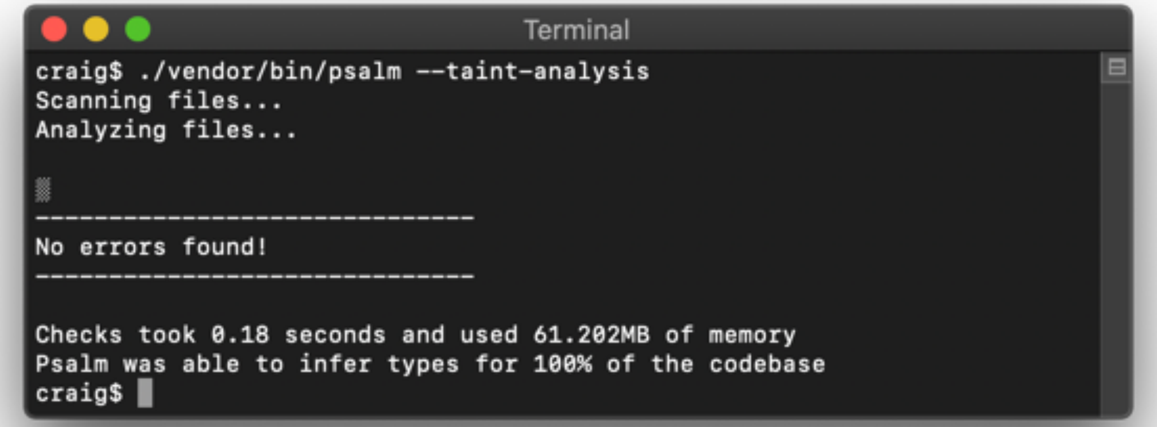


Static Analysis

```
$mysqli = new mysqli('localhost', 'test', '???', 'test');

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

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

if ($result instanceof mysqli_result) {

    print_r($result->fetch_all());

}
```

Avoid MixedAssignment

Psalm,
Taint Analysis



I'm not using "mysqli_query()"
https://github.com/vimeo/psalm/issues/4155

Summary

Do not mix safe strings (literals), with anything that may be attacker controlled.

Summary

We need a way to ensure this does not happen.

```
is_literal('This is a Literal');
```

```
is_literal('This is a Literal');
```

true

```
$a = 'This is a Literal';

is_literal($a);
```

true

```
$a = 'This is a Literal';

is_literal($a . 'And this');
```

true

```
$a = 'This is a Literal';

is_literal($a . $_GET['name']);
```

false

```
$a = 'This is a Literal';

is_literal($a . htmlentities($_GET['name']));
```

Still false

```
$a = 'This is a Literal';

is_literal($a . htmlentities($_GET['name']));
```

Still false

is_literal() is not the same as Taint Checking.

```
$a = 'This is a Literal';

is_literal($a . strtoupper('abc'));
```

Sorry, this is false - 'ABC' is no longer the string defined in the PHP script.

```
$sql .= ' AND id = "' . mysqli_real_escape_string($db, $_GET['id']) . '"';

is_literal($sql);
```

Still false

```
$sql .= ' AND id = ?';

$parameters[] = $_GET['id'];

is_literal($sql);
```

true

```
$users = $queryBuilder
        ->select('u')
        ->from('User', 'u')
        ->where('u.type_id = 1')
        ->getQuery()
        ->getResult();
```

Doctrine

```
$users = $queryBuilder
        ->select('u')
        ->from('User', 'u')
        ->where('u.type_id = 1')
        ->getQuery()
        ->getResult();
```

Is a Safe Literal?

```
public function where($predicates)
{
    if (!is_literal($predicates)) {
        throw new Exception('Can only accept a literal');
    }
    ...
}
```

```
$users = $queryBuilder
        ->select('u')
        ->from('User', 'u')
        ->where('u.type_id = ' . $_GET['type_id'])
        ->getQuery()
        ->getResult();
```

Not a Safe Literal

```
public function where($predicates)
{
    if (!is_literal($predicates)) {
        throw new Exception('Can only accept a literal');
    }
    ...
}
```

```
$template = $twig->createTemplate('<p>Hello {{ name }}</p>');

echo $template->render(['name' => $_GET['name']]);
```

Twig

```
$template = $twig->createTemplate('<p>Hello {{ name }}</p>');

echo $template->render(['name' => $_GET['name']])
```

Is a Safe Literal?

```
public function createTemplate(string $template, string $name = null): TemplateWrapper
{
    if (!is_literal($template)) {
        throw new Exception('Can only accept a literal');
    }
    ...
}
```

```
$template = $twig->createTemplate('<p>Hello ' . $_GET['name'] . '</p>');

echo $template->render()
```

Not a Safe Literal

```
public function createTemplate(string $template, string $name = null): TemplateWrapper
{
    if (!is_literal($template)) {
        throw new Exception('Can only accept a literal');
    }
    ...
}
```

Backwards compatibility

```
if (function_exists('is_literal') && !is_literal($a)) {
    trigger_error('Can only accept a literal', E_USER_NOTICE);
}
```

Notices might be safer for legacy projects.

Backwards compatibility

```
if (!function_exists('is_literal')) {
    function is_literal($variable) {
        return true;
    }
}
```

What about Table and Field names?

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

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

$field_id = array_search(($_GET['sort'] ?? 'created'), $fields);

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

What about WHERE IN?

```
$sql .= 'WHERE id IN (1, 2, 3)';
```

From an unknown source

```
$ids = [1, 2, 3];
```

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

```php
$ids = [1, 2, 3];

$sql .= 'WHERE id IN (' . implode(',', array_fill(0, count($ids), '?')) . ')';
```

```
$ids = [1, 2, 3];

$sql .= 'WHERE id IN (' . implode(',', array_fill(0, count($ids), '?')) . ')';
```

↓

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

Other Implementations

Other Implementations

Google have a similar solution in Go:

HTML Injection / XSS

https://github.com/google/go-safeweb/tree/master/safehttp

https://blogtitle.github.io/go-safe-html/

By Roberto Clapis

Other Implementations

Google have a similar solution in Go:

SQL Injection

https://github.com/google/go-safeweb/tree/master/safesql

A Query Builder, with an Append() method that only accepts **compile-time constant strings** (literal or const).

Other Implementations

JavaScript may get a similar solution:

https://github.com/tc39/proposal-array-is-template-object

"Distinguishing strings from a trusted developer,

from strings that may be attacker controlled"

Concerns

Concerns

- Performance

Concerns

- Performance

Might not be able to justify string concatenation.

Concerns

- Performance

Might not be able to justify string concatenation.

But could introduce a new function to combine an array of literals:

Concerns

- Performance

Might not be able to justify string concatenation.

But could introduce a new function to combine an array of literals:

```
$sql = ['SELECT u.name FROM user AS u'];

if ($id) {
  $sql[] = 'WHERE id = ?';
  $parameters[] = $id;
}

$sql = literal_implode(' ', $sql);
```

Concerns

- Performance
- Can array_fill() + implode() pass though the "is_literal" flag? (WHERE IN)
- Name it something else?
- Variables, like a table prefix, being stored in an INI / JSON / YAML file?

The Future

- Internal PHP functions, like mysqli_query(), preg_match(), exec(), etc; could all be setup to only accept safe literals.

- When printing strings (i.e. sending HTML to the browser), this could be blocked, with an exception for your trusted HTML templating engine. Maybe PHP could only accept a specific value object that has a __toString() method?

Thank You

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

https://github.com/craigfrancis/php-is-literal-rfc