Securing AngularJS Applications

Sebastian Lekies (@slekies)

Who Am I?

Sebastian Lekies (@slekies)

Senior Software Engineer at Google

➔ Tech Lead of the Web application security scanning team
➔ Google Internal Security Scanner & Cloud Security Scanner

PhD Student at the University of Bochum RUB

➔ Thesis topic: "Client-Side Web Application security"
➔ Interested in client-side attacks: XSS, ClickJacking, CSRF, etc.

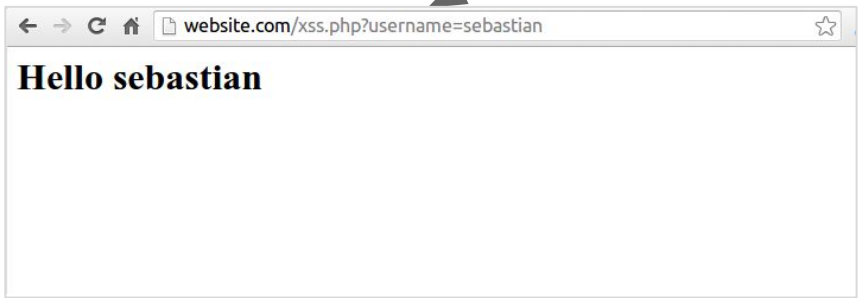
Agenda

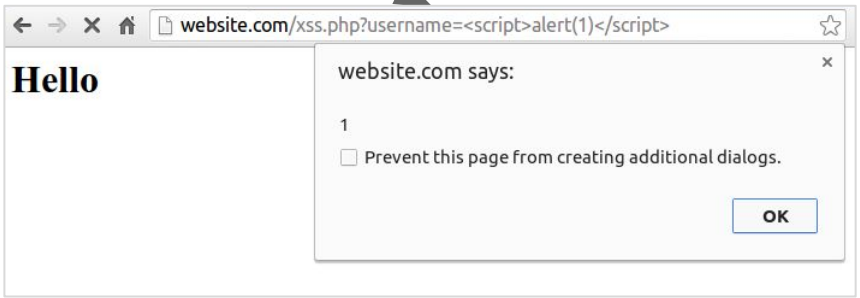
1. **Introduction**
   a. What is Cross-Site Scripting?
   b. What is AngularJS?
2. **Basic Angular Security Concepts**
   a. Strict Contextual Auto Escaping
   b. The HTML Sanitizer
3. **Common Security pitfalls**
   a. Server-Side Template Injection
   b. Client-Side Template Injection
   c. Converting strings to HTML
   d. White- and Blacklisting URLs
4. **Conclusion**

A quick introduction to Cross-Site Scripting (XSS)...

- XSS is a code injection problem:

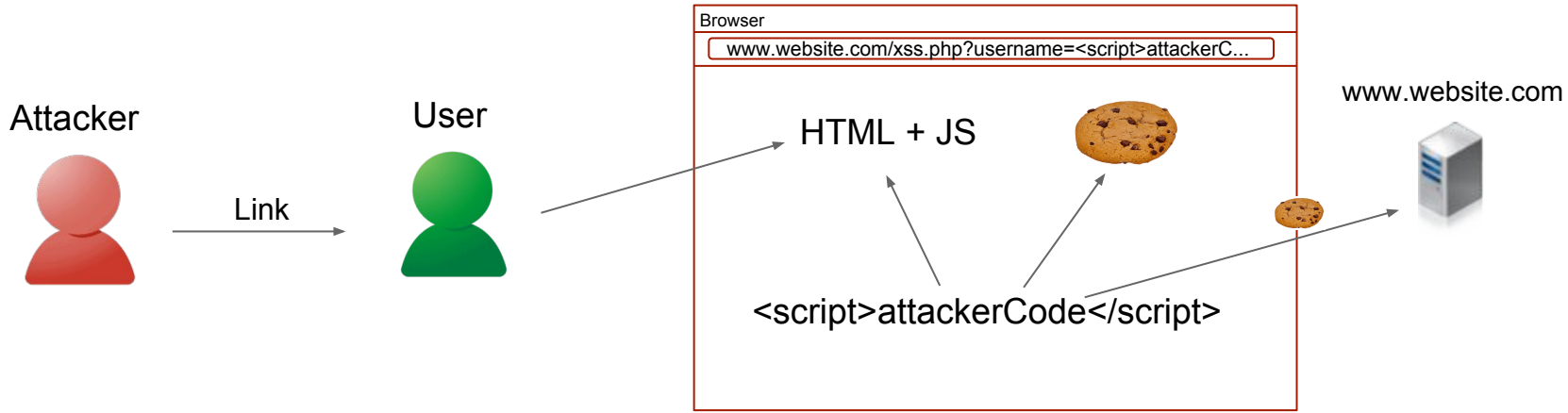
```php
<?php
  echo "<h1>Hello ".$_GET['username']."</h1>";
?>
```





A quick introduction to Cross-Site Scripting (XSS)...

- Attacker model
  - *Exploit: http://website.com/xss.php?username=<script>attackerCode</script>*



Defending against Cross-Site Scripting (XSS)...

Defending against XSS: Context-aware escaping and validation

- HTML Context

```
<?php
  echo "<h1>Hello " .htmlentities($_GET['username']). "</h1>";
?>
```

- Mixed Context: HTML + URI Context

```
<?php
  echo "<a href = '" .encodeForHTML(validateUri($_GET['uri'])). "'>link</a>";
?>
```

(A brief) Introduction to AngularJS

What is AngularJS?

AngularJS is a client-side MVC/MVVM Web application framework…

- ...redefining the way client-side-heavy single page apps are written

"Angular is what HTML would have been,
had it been designed for applications" *

- **Problem**: HTML is great for static pages, but not so great for dynamic UIs
- **Solution**: Angular's declarative templating system with two-way data bindings

* https://docs.angularjs.org/guide/introduction

Introduction to Angular - Example

Include the newest version of Angular...

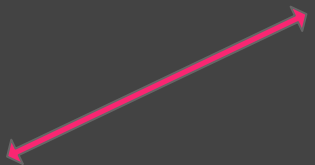
```
<script src="./angularjs/1.5.7/angular.min.js"></script>
```

Introduction to Angular - Example

Create a module...

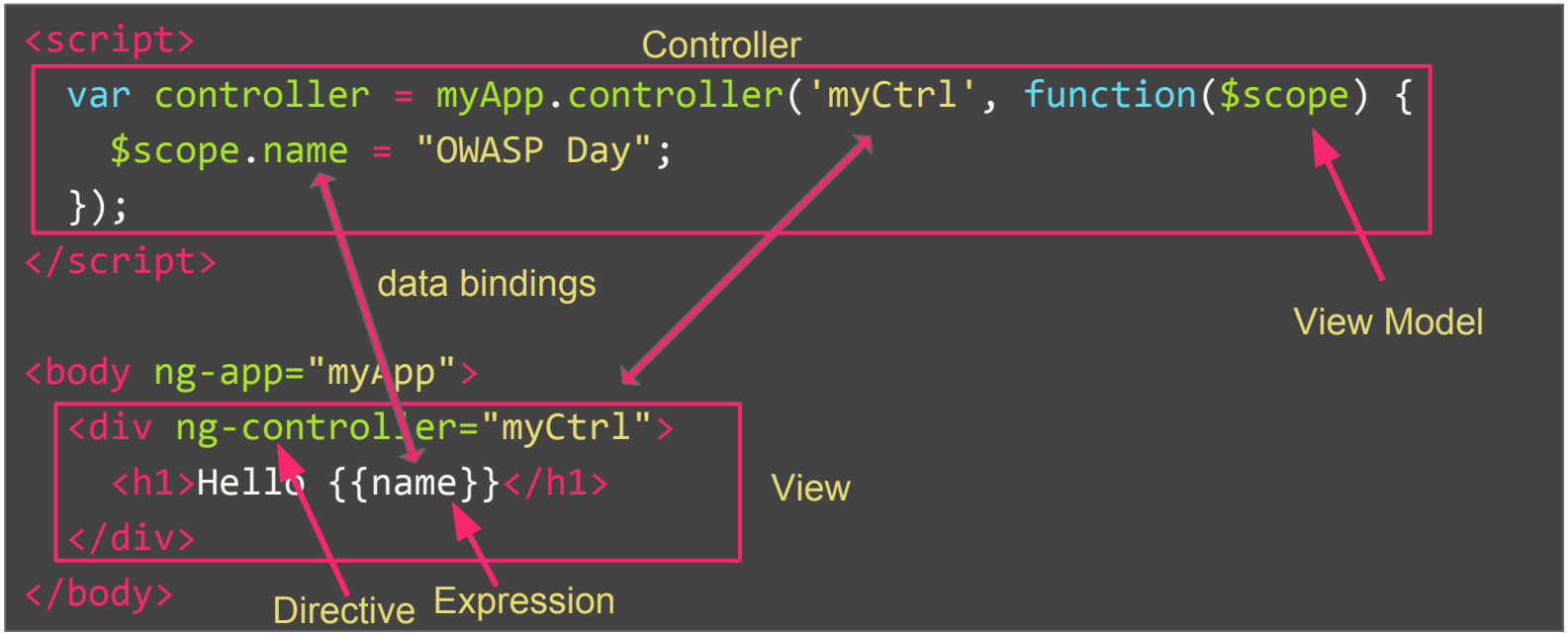
```
<script>
  var myApp = angular.module('myApp', []);
</script>

<body ng-app="myApp">
  ...
</body>
```



Introduction to Angular - Example

Create controllers, views and viewmodels...



Important Terms: Directives

Directives are markers for enriching HTML with custom functionality:

```
// Directive as a tag
<person name="expression"></person>

// Directive as an attribute
<div person name="expression"></div>
```

AngularJS comes with a lot of built-in directives: e.g. **ngBind**, **ngIf**, **ngInclude**, etc.

More about directives: https://docs.angularjs.org/guide/directive

Important Terms: Expressions

Angular Expressions are JavaScript-like code snippets…

- ...evaluated against the corresponding scope
- ...sandboxed to prevent access to global JS properties (not for security!!)

```
// Used for string interpolation
<div>{{1+2}}</div> → <div>3</div>
<div>Hello {{getName()}}</div>
<div id="{{id}}"></div>

// Used within directives
<div ng-click="greet()">greet</div>
```

More about expressions: https://docs.angularjs.org/guide/expression

Angular's Security Concepts

Strict Contextual Auto Escaping

Recap: XSS can be prevented by proper output encoding and validation

```php
<?php
        echo "<iframe src='".$_GET['url']."'></iframe>"; // XSS vulnerability
?>
```

Output encoding required:

- Encode all HTML control characters
- E.g. htmlentities in php

URL Validation required:

- No JavaScript, data or about URI
- Only same-domain URLs

Manual output encoding in a complex project is doomed to fail!

Strict Contextual Auto Escaping

Let Angular do the encoding and validation for you:

- Within the controller

```
$scope.url = <user-controlled>;
```

- Within the view

```
<!-- url gets auto-encoded and validated -->
<iframe ng-src="{{url}}"></iframe>
```

Angular templates are XSS free...

- ...by automatically encoding output
- ...and validating URLs
- ...if you do not tamper with security

Behind the Scenes: Output Encoding and URL validation

When parsing an expression Angular determines the context:

1. HTML
2. URL
3. RESOURCE_URL
4. CSS (currently unused)
5. JS (currently unused, interpolation inside scripts is not supported)

...and applies the correct output encoding or validation function

Behind the Scenes: Output Encoding and URL validation

HTML Context

```
1. <div>Hello {{name}}!</div>
2. <div attribute="{{name}}"></div>
```

Managed by the ***$sceProvider***

- `enabled(boolean);`
- Enabled by Default

If enabled all values are encoded with a secure encoding function

Never disable Strict Contextual Auto Escaping!!

Behind the Scenes: Output Encoding and URL validation

URL Context (for passive content)

```
1. <a ngHref="url">
2. <img ngSrc="url">
```

Managed by the ***$compileProvider***

- `aHrefSanitizationWhitelist([regexp]);`
- `imgSrcSanitizationWhitelist([regexp]);`
- By default: **http**, **https**, **mailto** and **ftp**

If a given URL matches the regular expression

- … the URL gets written into the DOM
- If not, the string "**unsafe:**" is prepended to the URL

Behind the Scenes: Output Encoding and URL validation

RESOURCE_URL Context (for active content)

```
1. <iframe ngSrc="url">
2. <script ngSrc="url">
3. <div ngInclude="url"></div>
```

Managed by the ***$sceDelegateProvider***

- `resourceUrlWhitelist([whitelist]);`
- `resourceUrlBlacklist([blacklist]);`

Allowed list values: 'self', RegExp, String (with * and ** wildcards)
By Default: Only same-domain URLs are supported

The HTML Sanitizer

Use Case: Angular escapes output. What if I want to render HTML?

Solution: ~~ng-bind-html-unsafe~~ (< Angular 1.2), ng-bind-html & the sanitizer

```
// Within the Controller
$scope.html = "<script>alert(1)</script><h1 onclick="alert(1)">Hello World!</h1>";
<!--  Within the view -->
<div ng-bind-html="html"></div>
```

```
<!--  Result -->
<div>
  <h1>Hello World!</h1> <!--  The script tag and the event handler get sanitized -->
</div>
```

Common Security Pitfalls

(based on real-world bugs)

Server-Side Template Injection

Server-side template injection

Angular is a client-side framework…

- The logic is implemented in JavaScript
- The server is a mechanism to store data and code.
- The server **must not** generate templates based on user input

Any template received from the server is considered trusted

Templates vs. Prepared Statements

Prepared statements for SQL Injection prevention

```
// The statement itself is considered trusted.
stmt = db.prepareStatement("SELECT * FROM users WHERE username = ?' )
// Untrusted data is inserted separately.
stmt.setValue(1, userInput);
```

Auto-escaping templates for XSS prevention

```
// The template itself is considered trusted.
<div>{{username}}</div>
// Untrusted data is inserted via data bindings.
$scope.username = userInput ;
```

Server-side template injection - The wrong way

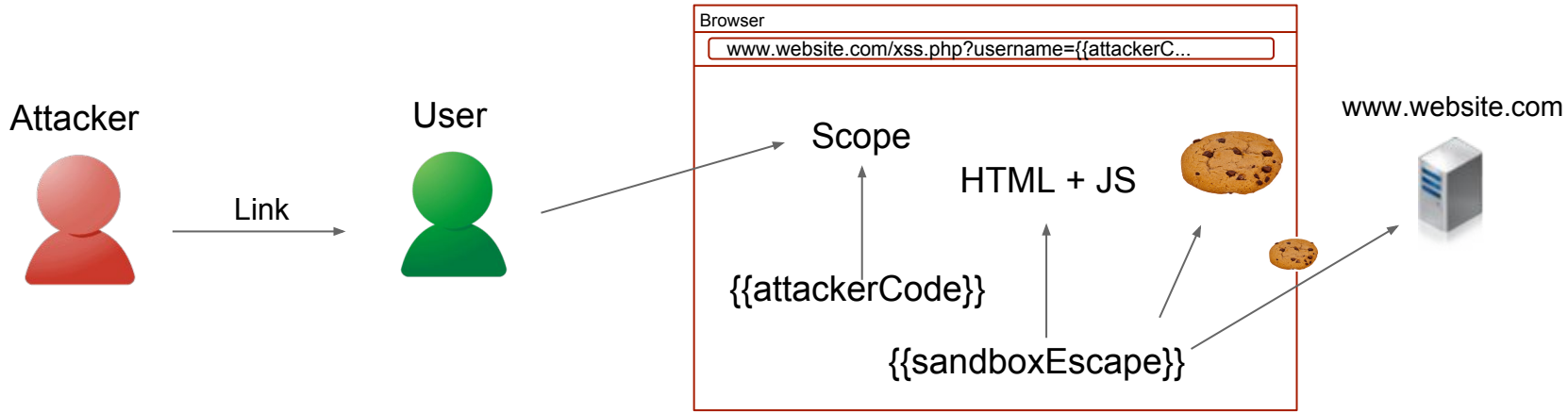
Unfortunately, people mix traditional applications with angular

```
<script src="./angularjs/1.5.7/angular.min.js"></script>
<div ng-app="exampleApp" ng-controller="exampleCtrl">
<?php
  echo "<h1>Hello ".htmlentities($_GET['username'])."</h1>"; # This is a vulnerability.
?>
</div>
```

Including Angular into this server-generated page, creates a vulnerability

Consequences of an expression injection

- *Exploit: http://website.com/xss.php?username={{attackerCode}}*



Server-side template injection

Do not dynamically generate Angular templates on the server-side.

Define your Angular templates statically and populate your templates via data bindings on the client-side.

Client-Side Template Injection

Client-side template injection

New trend: Mixing Angular, with other third-party libraries

```
<script>
  // A non angular-related library. Secure without Angular. Insecure with Angular.
  document.write(escapeForHTML(userInput));
</script>
<script src="./angularjs/1.5.7/angular.min.js"></script>
```

Do not write user input to the DOM before angular runs.

Inserting HTML into the DOM.

Use Case: Enrich user-provided values with HTML

Use case: "Enrich user input with HTML!"

- User input: "OWASP Day"

```
// Within the controller
$scope.html = "Hello <b>" + userInput + "</b>!";
<!--  Within the view -->
<div>{{html}}</div>
```

- Result:

```
<div>Hello &lt;b&gt;OWASP Day&lt;/b&gt;!</div>
```

Mhhh, the results are auto-encoded!

Wrong way 1: Disable the escaping

Wrong Solution 1: Let's disable the escaping!

- User input: "OWASP Day"

```
// Within the controller
$sce.enabled(false); // Disables strict auto escaping
$scope.html = "Hello <b>" + userInput + "</b>!";
<!--  Within the view -->
<div>{{html}}</div>
```

- Result:

```
<div>Hello <b>OWASP Day</b>!</div>
```

This works, but security is _completely_ disabled!

Wrong way 2: Use jqLite APIs

Wrong Solution 2: Use element.html() to insert HTML

- User input: "OWASP Day"

```
// Within the controller
angular.element(someElem).html("Hello <b>" + userInput + "</b>" )
```

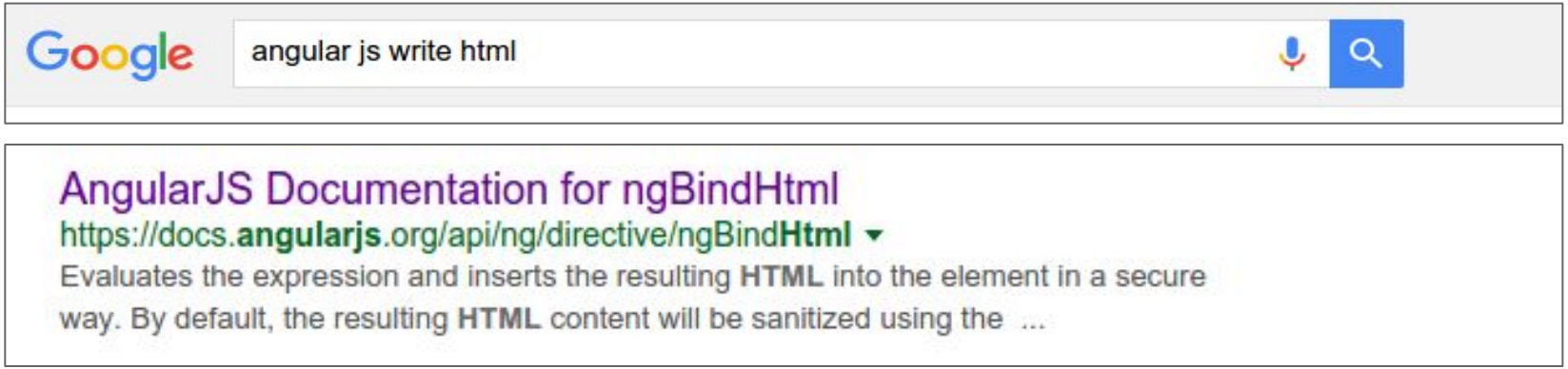
- Result:

```
<div>Hello <b>OWASP Day</b>!</div>
```

This works, but value is not auto-escaped!

Wrong way 3: Make the value trusted

Wrong Solution 3: Use ngBindHtml & trustAsHtml



Wrong way 3: Make the value trusted

Wrong Solution 3: Use ngBindHtml & trustAsHtml

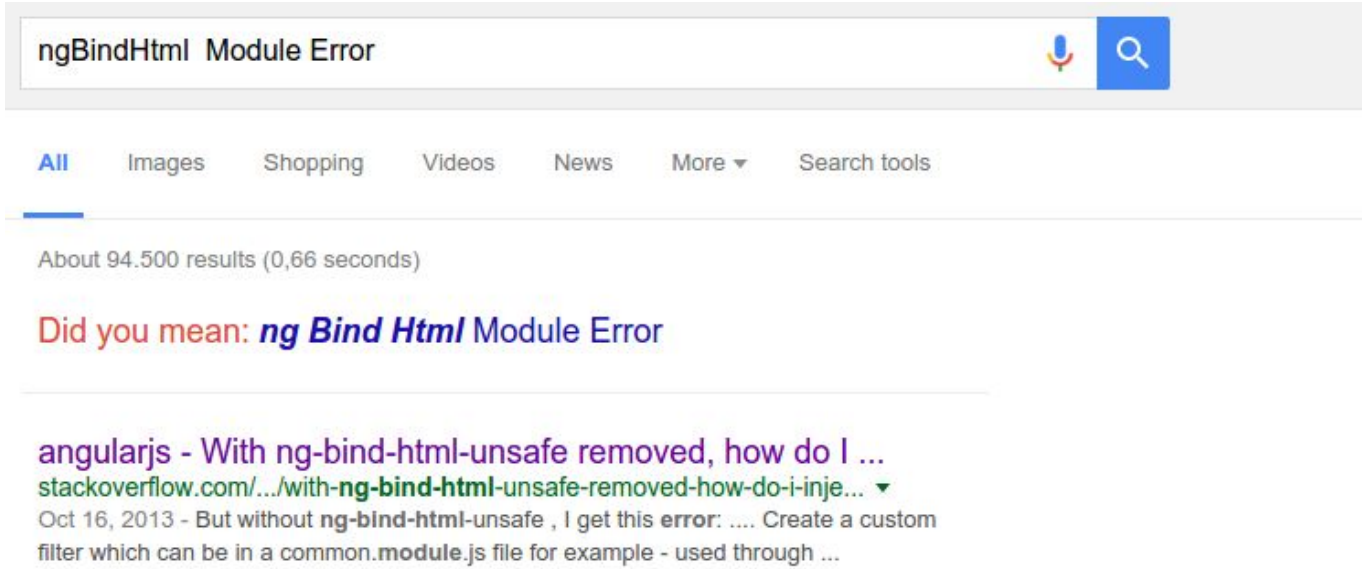
```
// Within the Controller
$scope.html = "Hello <b>World</b>!";
<!--  Within the view -->
<div ng-bind-html="html"></div>
```

```
Uncaught Error: [$injector:modulerr] http://errors.angularjs.org/1.4.7/$injector/modulerr?p0=myApp&p1=Error%3A%2…
ogleapis.com%2Fajax%2Flibs%2Fangularjs%2F1.4.7%2Fangular.min.js%3A19%3A463)
>
```

Mhhh, a "Module Error" exception? What is this about?

Wrong way 3: Make the value trusted

Wrong Solution 3: Use ngBindHtml & trustAsHtml



Wrong way 3: Make the value trusted

Wrong Solution 3: Use ngBindHtml & trustAsHtml

You indicated that you're using Angular 1.2.0... as one of the other comments indicated, `ng-bind-html-unsafe` has been deprecated.

Instead, you'll want to do something like this:

```
<div ng-bind-html="preview_data.preview.embed.htmlSafe"></div>
```

In your controller, inject the `$sce` service, and mark the HTML as "trusted":

```
myApp.controller('myCtrl', ['$scope', '$sce', function($scope, $sce) {
  // ...
  $scope.preview_data.preview.embed.htmlSafe =
     $sce.trustAsHtml(preview_data.preview.embed.html);
}
```

Note that you'll want to be using 1.2.0-rc3 or newer. (They fixed a bug in rc3 that prevented "watchers" from working properly on trusted HTML.)

Wrong way 3: Make the value trusted

Wrong Solution 3: Use ngBindHtml & trustAsHtml

- User input: "OWASP Day"

```
// Within the controller
$scope.html = $sce.trustAsHtml("Hello <b>" + userInput + "</b>!");
<!--  Within the view -->
<div>{{html}}</div>
```

- Result:

```
<div>Hello <b>OWASP Day</b>!</div>
```

This works, but security is disabled!

Wrong way 4: Encode the value and then trust it

Wrong Solution 4: Use ngBindHtml & trustAsHtml & custom encoding

- User input: "OWASP Day"

```
// Within the controller
var escapedUserInput = escapeForHtml(userinput);
$scope.html = $sce.trustAsHtml("Hello <b>" + escapedUserInput + "</b>!");
<!--  Within the view -->
<div>{{html}}</div>
```

- Result:

```
<div>Hello <b>OWASP Day</b>!</div>
```

This works, but managing security on your own is dangerous!

The right way: Use ngBindHtml and the sanitizer

Correct Solution: use ngBindHtml and the sanitizer

```
// Within the Controller
$scope.html = "Hello <b>" + userInput + "</b>!";
<!--  Within the view -->
<div ng-bind-html="html"></div>
```

```
Uncaught Error: [$injector:modulerr] http://errors.angularjs.org/1.4.7/$injector/modulerr?p0=myApp&p1=Error%3A%2…
ogleapis.com%2Fajax%2Flibs%2Fangularjs%2F1.4.7%2Fangular.min.js%3A19%3A463)
>
```

The sanitizer module dependency is missing

The right way: Use ngBindHtml and the sanitizer

Correct Solution: use ngBindHtml and the sanitizer

```
<script src="//code.angularjs.org/1.5.7/angular-sanitize.js"></script>
<script>
 var myApp = angular.module('myApp', ["ngSanitize"]);
 var controller = myApp.controller('myCtrl', function($scope) {
  $scope.html = "Hello <b>" + userInput + "</b>!"
 });
</script>
<!--  Within the view -->
<div ng-bind-html="html"></div>
```

Inserting HTML into the DOM: Summary

White- and Blacklisting URLs

White- and Blacklisting Resource URLs

Angular supports many URL-based directives:

- *ngSrc*, *ngInclude*, *ngHref*
- These directives should never contain user-provided data

Angular validates URLs based on predefined white- and blacklists.

- ***$sceDelegateProvider.****`resourceUrl(White|Black)list([list]);`*
- By default only same domain URLs are allowed
- `String, RegExes and 'Self' are allowed`
- `Strings support two wildcards`
  - `*: Matches all but URL control characters (:, /, ?, &, ., ;)`
  - `**: Matches all characters`

White- and Blacklisting Resource URLs

Wrong way 1: Wildcards in the scheme

```
// Whitelist all possible schemes
"**://example.org/*"
```

- Exploit 1: http://evil.com/?ignore=://example.org/a
- Exploit 2: javascript:alert(1);//example.org/a

```
// Less permissive, but still bad
"*://example.org/*"
```

Linebreak to end single line comment

- Exploit 1: javascript://example.org/a%0A%0Dalert(1)

White- and Blacklisting Resource URLs

Wrong way 2: ** Wildcards in the domain

```
// Whitelist all possible subdomains
"https://**.example.org/*"
```

- Exploit 1: https://evil.com/?ignore=://example.org/a

```
// Whitelist all possible top level domains
"https://example.**"
```

- Exploit 1: https://example.evil.com
- Exploit 2: https://example.:foo@evil.com

White- and Blacklisting Resource URLs

Wrong way 3: Use Regular Expressions

```
// Use a RegEx to whitelist a domain
/http:\/\/www.example.org/g
```

- Exploit 1: http://wwwaexample.org // (dots are not escaped)
- Exploit X: All the wildcard-based exploits can be applied as well

White- and Blacklisting Resource URLs

Do's and Dont's

- Never use regular expressions!
- Do not use wildcards in the scheme!
- Do not use ** in the host!
- Only use * for subdomain and or the path!
- Optimal: Whitelist only specific URLs!

Conclusion

Conclusion

AngularJS offers strong security guarantees…

- ...if you follow the Angular philosophy

Templates are considered trusted

- Do not generate them dynamically at runtime
- Do not mix angular with other libraries
- Do not switch off strict contextual auto escaping

If you need to add HTML...

- ...use ng-bind-html and the sanitizer
- ...avoid using trustAsHTML
- ...**never** use DOM or jqLite APIs

If you need to whitelist URLs, stay away from regular expressions and wildcards.

Thank you!