Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets

Sebastian Lekies
Google
slekies@google.com

Krzysztof Kotowicz
Google
koto@google.com

Samuel Groß
SAP
mail@samuel-gross.com

Eduardo A. Vela Nava
Google
evn@google.com

Martin Johns
SAP
martin.johns@sap.com

ABSTRACT

Cross-Site Scripting (XSS) is an unremitting problem for the Web. Since its initial public documentation in 2000 until now, XSS has been continuously on top of the vulnerability statistics. Even though there has been a considerable amount of research [15, 18, 21] and developer education to address XSS on the source code level, the overall number of discovered XSS problems remains high. Because of this, various approaches to mitigate XSS [14, 19, 24, 28, 30] have been proposed as a second line of defense, with HTML sanitizers, Web Application Firewalls, browser-based XSS filters, and the Content Security Policy being some prominent examples. Most of these mechanisms focus on script tags and event handlers, either by removing them from user-provided content or by preventing their script code from executing.

In this paper, we demonstrate that this approach is no longer sufficient for modern applications: We describe a novel Web attack that can circumvent all of theses currently existing XSS mitigation techniques. In this attack, the attacker abuses so called *script gadgets* (legitimate JavaScript fragments within an application's legitimate code base) to execute JavaScript. In most cases, these gadgets utilize DOM selectors to interact with elements in the Web document. Through an initial injection point, the attacker can inject benign-looking HTML elements which are ignored by these mitigation techniques but match the selector of the gadget. This way, the attacker can hijack the input of a gadget and cause processing of his input, which in turn leads to code execution of attacker-controlled values. We demonstrate that these gadgets are omnipresent in almost all modern JavaScript frameworks and present an empirical study showing the prevalence of script gadgets in productive code. As a result, we assume most mitigation techniques in web applications written today can be bypassed.

1 INTRODUCTION

Web technology is moving forward at a rapid pace. Everyday new frameworks and APIs are pushed to production. This constant development also leads to a change in attack surface and vulnerabilities. In this process Cross-Site Scripting (XSS) vulnerabilities have evolved significantly in the recent years. The traditional reflected XSS issue is very different from modern DOM-based XSS vulnerabilities such as mXSS [12], or expression-language-based XSS [10]. While the topic of XSS becomes increasingly more complex, many mitigation techniques only focus on the traditional and well-understood reflected XSS variant.

In this paper, we present a novel Web attack which demonstrates that many mitigation techniques are inefficient when confronted with modern JavaScript libraries. At the core of the presented attack are so-called *script gadgets*, small fragments of JavaScript contained in the vulnerable site's legitimate code. Generally speaking, a script gadget is piece of JavaScript code which reacts to the presence of specifically formed DOM content in the Web document. In a gadget-based attack, the adversary injects apparently harmless HTML markup into the vulnerable Web page. Since the injected content does not carry directly executable script code, it is ignored by the current generation of XSS mitigations. However, during the web application lifetime, the site's script gadgets pick up the injected content and involuntarily transform its payload into executable code. *Thus, script gadgets introduce the practice of code-reuse attacks [27], comparable to return-to-libc , to the Web.*

To explore the severity and prevalence of the underlying vulnerability pattern, we conduct a qualitative and quantitative study of script gadgets. For this, we first identify the various gadget types, considering their functionality and their potential to undermine existing XSS mitigations. Furthermore, we examine 16 popular JavaScript frameworks and libraries, focusing on contained script gadgets and mapping the found gadget instances to the affected XSS mitigations. For instance, in 13 out of the 16 examined code-bases we found gadgets capable to circumvent the emerging `strict-dynamic` variant of the Content Security Policy [34]. Finally, we report on a large-scale empirical study on the prevalence of script gadgets in popular web sites.

By crawling the Alexa top 5000 Web sites and their first-level links, we measured gadget-related data flows for approximately 650,000 individual crawled URLs. In total, we measured 4,352,491 sink executions with data retrieved from the DOM. Using our fully-automated exploit generation framework, we generated exploits and verified gadgets on 19.88% of all domains in the data set. As we applied a very conservative, but false-positive-free verification approach, we believe that this number is just a lower bound and that the numbers of gadgets are considerably higher in practice.

CCS '17, Dallas, TX, USA
 978-1-4503-4946-8/17/10…$15.00
DOI: 10.1145/3133956.3134091

In particular, this paper makes the following contributions:

- To the best of our knowledge, we are the first researchers to systematically explore this new Web attack that allows to circumvent popular XSS mitigation techniques by abusing script gadgets. We describe the attack in detail and give a categorization of different types of gadgets.
- In order to explore script gadgets in detail, we present the results of a manual study on 16 modern JavaScript libraries. Based on proof-of-concept exploits we demonstrate that almost all of these libraries contain gadgets. Furthermore, we demonstrate how these different script gadgets can be used to circumvent all 4 popular classes of mitigation techniques: The Content Security Policy, HTML sanitizers, Browser-based XSS filters and Web Application Firewalls.
- Based on the results of the manual study, we built a tool chain capable of automatically detecting and verifying gadgets at scale. Based on this tool, we conducted an empirical study of the Alexa top 5000 Web sites including more than 650k Web pages. The results of this study suggests that script gadgets are omnipresent in modern JavaScript-heavy applications. While our study is very conservative when measuring gadgets, we managed to detect and verify gadgets in 19.88% of all domains. This number just represents a lower bound and is likely much higher in practice.

2 TECHNICAL BACKGROUND

2.1 JavaScript, HTML and the DOM

Since its development, JavaScript has been used to interact with the DOM to make HTML documents more interactive. To do this, JavaScript working in the browser uses many different ways to read data from the DOM. Most of the corresponding functions such as `document.getElementById` or `document.getElementsByClassName` are based on *DOM selectors*[33] by providing convenient wrappers around `document.querySelectorAll`.

DOM selectors are a powerful pattern language that can be used to query the DOM for certain elements, and therefore are the basis for all modern JavaScript frameworks. For example, one of the most famous JavaScript functions - jQuery's $ function - enhances the browser-based selector language with a lot of syntactic sugar. In the following table, we describe some selector features in detail:

Selector	E.g.	Matches...
Tag-based	`div`	`div` elements
Id-based	`#foo`	elements with id 'foo'
Class-based	`.foo`	elements with class 'foo'
Attr.-based	`[foo]`	elements with an attribute named 'foo'

2.2 Cross-site Scripting (XSS)

The term Cross-site Scripting (XSS) [29] describes a class of string-based code injection vulnerabilities that let adversaries inject HTML and/or JavaScript into Web content that is not legitimately under their control. XSS vulnerabilities are generally categorized based on the location of the vulnerable source code, i.e., *server-* or *client-side* XSS, and the persistence of the injected attack code, i.e., *reflected* or *stored* XSS.

XSS can be avoided through secure coding practices, which mainly rely on the careful handling of attacker controlled input and context-aware sanitization/encoding of untrusted data before processing it in a security sensitive context. For brevity, we'll omit further details on the basic vulnerability class and refer to the vast body of existing work on the topic [7, 8, 17, 18, 21, 31].

2.3 XSS Mitigation Techniques

The basic XSS problem has been recognized since the beginning of the decade [5], the root cause is understood, and a significant amount of work has been done to design approaches to detect and prevent XSS issues in source code. XSS is statistically still the most common vulnerability class however, and there seems to be no overall decline in its prevalence. It therefore seems safe to assume that XSS problems will not be solved completely with secure coding practices alone.

For this reason various XSS mitigations have been introduced as an important second line of defense. Instead of removing the underlying vulnerability, XSS mitigations aim to prevent the *exploitation* of the vulnerability by stopping the execution of the injected script code. XSS mitigations are widely implemented in four different forms:

(1) **HTML Sanitizers.** These are libraries used by developers to clean untrusted HTML into HTML that is safe to use within the application. This category contains examples such as DOMPurify[1] and Google Closure[2] HTML sanitizer.
(2) **Browser XSS Filters.** These filters are implemented as part of the browser navigation and rendering, and they attempt to detect an XSS attack and neuter it. Internet Explorer, Edge, and Chrome implement XSS filters as part of their default configuration. Firefox does not have one, but the popular NoScript[3] AddOn implements one.
(3) **Web Application Firewalls.** This is software that runs on the server, and attempts to allow benign requests from web traffic, while detecting and blocking malicious requests. An example of an open-source Web Application Firewall is ModSecurity[4] with OWASP Common Rule Set[5].
(4) **Content Security Policy [34].** This is a browser feature that a web developer can configure to define a policy that allows the browser to whitelist the JavaScript code that belongs to the application.

These mitigations all fundamentally rely one of three basic strategies:

(1) **Request filtering** blocks HTTP requests before they reach the application, working either at the browser level (like NoScript), or at the network or application level (like WAFs).

[1] https://github.com/cure53/DOMPurify
[2] https://github.com/google/closure-library
[3] https://noscript.net/
[4] https://modsecurity.org/
[5] https://github.com/SpiderLabs/owasp-modsecurity-crs

(2) **Response sanitization** focuses on detecting malicious code and sanitizing it out of the response. Examples of these are HTML sanitizers, as well as Internet Explorer's and Edge's XSS filter.
(3) **Code filtering** detects malicious JavaScript just before it is executed and tries to detect whether it is benign or not. Examples of this strategy include CSP and Chrome's XSS filter.

We will go into more details about the implementation of such strategies and the ways to bypass them in Section 4.

3 SCRIPT GADGETS

In this section, we introduce the concept of script gadgets, explaining how injecting a benign HTML markup may result in arbitrary JavaScript execution by reusing parts of legitimate application code and how this can be used to negate the effects of XSS mitigations.

3.1 Benign HTML markup

XSS mitigation techniques described in Section 2.3 aim to stop XSS attacks by blocking execution of illegitimate, injected JavaScript code. Mitigations detect the injected code, present in inline event handlers or in separate `script` elements and prevent its execution, while legitimate JavaScript code, carrying appropriate trust information, is left as-is and is allowed to execute.

Those XSS mitigations ignore injected HTML markup that would not result in JavaScript execution - we'll call such markup benign HTML. Benign HTML does not contain `<script>` tags, inline event handlers, `src` or `href` attributes with `javascript:` or `data:` URLs, or other tags capable of JavaScript execution (`<link rel=import>`, `<meta>`, `<style>`). The following snippet is an example of benign HTML:

```
<div class="greeting">
  <b>Hello</b> world!
</div>
```

Listing 1: Benign HTML markup ignored by the mitigation

3.2 DOM selectors

The presence of benign HTML in a document does not directly trigger code execution. However, in virtually all web applications JavaScript code already present in the page interacts with the DOM, reading data from the document by using various DOM selectors (2.1). For example, a web application might take all elements with a `tootltip` attribute to decorate them by showing a given text when the user selects these elements. JavaScript code reading data from the DOM based on a selector is a common pattern in both user-land and library code - example code snippets might look like this:

```
// Userland code
var button = document.getElementById("button");
button.getAttribute("data-text");

var links = $("a[href]").children();

// Reading 'ref' attributes in Aurelia framework
if (attrName === 'ref') {
      info.attrName = attrName;
      info.attrValue = attrValue;
      info.expression = new NameExpression(
        this.parser.parse(attrValue), 'element',
        resources.lookupFunctions);=
}
// Vue.js reading from v-html attribute
if ((binding = el.attrsMap['v-html'])) {
  return [{ type: EXPRESSION, value: binding }]
}
```

Listing 2: Reading data from the DOM

By injecting benign HTML markup matching DOM selectors used in the application we are able to trigger the execution of specific pieces of legitimate application code [6] - script gadgets.

3.3 Script Gadgets - Introduction

Script gadgets are fragments of legitimate JavaScript code belonging to the web application that execute as a result of benign HTML markup present in the web page. Script gadgets are not injected by the attacker - they are already present either in the user-land web application code, or one of the libraries/frameworks used by the web application.

Our research explores using script gadgets to bypass XSS mitigations. In order to do that, gadgets must both result in arbitrary script execution, and be triggered from benign HTML injection. For example, a web application might assign a value read from the DOM to the `innerHTML` property of an element:

```
var button = getElementById("my-button");
button.innerHTML = button.getAttribute("data-text");
```

Listing 3: Simple innerHTML gadget

Simple gadgets like these are often explored in the context of DOM XSS vulnerabilities [16], but for the purpose of this research we propose a new classification of gadgets of varying complexity. But first we'll explain how to use script gadgets in attacks against XSS mitigations.

[6] An alternative way of triggering specific code paths in a web application from benign markup is DOM clobbering. DOM clobbering allows markup to override variables in JavaScript execution environment, making it possible to trigger specific script behavior. While we have identified working bypasses of some XSS mitigations via DOM clobbering, for clarity we focus only on DOM selector-based code triggers.

3.4 Attack Outline

In this paper, we introduce a novel XSS attack that relies on script gadgets to cause the execution of the adversary's JavaScript code.

Attacker model: The applicable attacker is the classic XSS attacker [29], who is able to inject arbitrary HTML code into the content of the attacked web document. In the context of this paper whether the injection technique used is reflected or stored XSS is irrelevant.

As discussed above, existing XSS mitigations rely on the basic assumption that malicious code is being directly injected into the affected page in the course of an XSS attack. All non-script carrying, injected HTML content is therefore assumed to be benign and remains untouched by the mitigation. This assumption is exploited by the proposed attack method. The HTML code injected by the attacker exposes two characteristics:

(1) The actual attack payload, for example the attack's JavaScript, is contained in the benign HTML in a non-executable form.
(2) The HTML is specifically crafted so that its presence in the web document triggers a script gadget already contained in the web page's legitimate JavaScript code. In other words, the injected HTML payload triggers a code-reuse attack, similar to ret2libc techniques used in exploitation of memory-corruption vulnerabilities.

In the course of an attack, a script gadget accesses the injected DOM content and uses the contained information in an insecure manner, ultimately leading to the execution of the adversary's code, which was hidden in the benign HTML code. In summary, the class of attacks described in this paper follows this basic pattern:

(1) **Injection into the raw HTML.** The attacker controls the DOM of the webpage and injects a payload that triggers script gadgets in the application code. This payload contains only benign HTML markup and matches the DOM selectors used by the web application.
(2) **Mitigation attempt.** An XSS mitigation inspects the injected content, trying to detect script insertion. The benign HTML markup is left as-is.
(3) **Gadgets transforms the markup.** Gadgets present in the legitimate JavaScript code take the injected payload from the DOM using the DOM selectors and transform it into JavaScript statements.
(4) **Script executes**. The transformed JavaScript statements are executed, resulting in XSS.

The precise ways to abuse gadgets to bypass XSS mitigations depend on the type of mitigation and implemented mitigation strategy, as we described in Section 2.3

3.5 Gadget Types

We identified several types of script gadgets useful in bypassing XSS mitigations. Some of them may result in indirect script execution on their own; others need to be combined in chains to be useful in an attack.

3.5.1 String manipulation gadgets. These gadgets transform their string input by using regular expressions, character replacement and other types of string manipulation. When present, they can be used to bypass mitigations based on pattern matching. For example, the following gadget can be used to bypass some mitigations by using the `inner-h-t-m-l` attribute name that will later on be used by Polymer framework to assign to element's `innerHTML` property.

```
dash.replace(/-[a-z]/g, (m) => m[1].toUpperCase())}
```

Listing 4: Camel-casing the input in Polymer

Similar features are present in AngularJS frameworks, which allows the attackers to use benign `data` attributes in place of `ng-` attributes that would be blocked by HTML sanitizers:

```
var PREFIX_REGEXP = /^((?:x|data)[:\-_])/i;
var SPECIAL_CHARS_REGEXP = /[:\-_]+(.)/g;
function directiveNormalize(name) {
  return name.replace(PREFIX_REGEXP, '')
    .replace(SPECIAL_CHARS_REGEXP, fnCamelCaseReplace);
}
```

Listing 5: Directive name normalization in AngularJS

3.5.2 Element construction gadgets. These gadgets create new DOM elements. For XSS mitigation bypass purposes, we're mostly focused on identifying gadgets that may programmatically create new `script` elements.

```
document.createElement(input)
document.createElement("script")
jQuery("<" + tag + ">")
jQuery.html(input) // if input contains <script>
```

Listing 6: Example element creation gadgets

One notable element construction gadget is present in jQuery's `$.globalEval` function. This function creates a new `script` element, sets its `text` property and appends the element to the DOM, executing the code. `$.globalEval` combines an element creation gadget with a JavaScript execution gadget (3.5.4). As `$.globalEval` is called in various common jQuery methods (e.g. `$.html`), a controlled input to those may create new `script` elements, which is a useful property for bypassing strict-dynamic CSP (see 4.4)

3.5.3 Function creation gadgets. These gadgets create new `Function` objects. The function body is usually composed of a mix of the input and constant strings. Note that the created function object needs to be executed by a different gadget.

```
// Knockout Function creation gadget.
var body = "with($context){with($data||{}){return{" +
  rewrittenBindings + "}}}";
return new Function("$context", "$element", body);

// Underscore.js Function creation gadget.
source = "var __t,__p='',__j=Array.prototype.join," +
  "print=function(){__p+=__j.call(arguments,'');};\n" +
  source + 'return __p;\n';
var render = new Function(
  settings.variable || 'obj', '_', source);
```

Listing 7: Example function creation gadgets

3.5.4 JavaScript execution sink gadgets. These gadgets are usually standalone, or are the last in the constructed gadget chain, taking the input from the previous gadgets and putting it into a DOM XSS[16] JavaScript execution sink.

```
eval(input);
inputFunction.apply();
node.innerHTML = "prefix" + input + "suffix";
jQuery.html(input);
scriptElement.src = input;
node.appendChild(input);
```

Listing 8: Example execution sink gadgets

3.5.5 Gadgets in expression parsers. Some modern JavaScript frameworks (for example, Aurelia[7], AngularJS[8], Polymer[9], Ractive.js[10], Vue.js[11]) interpret parts of the DOM tree as templates for the application UI components. Those templates contain expressions written in framework-specific expression languages to bind a result of expression evaluation to a given position in the rendered UI. For example, the following expression displays a capitalized customer name:

```
<td>${customer.name.capitalize()}</td>
```

Listing 9: Sample expression in Aurelia

The framework extracts the template definition from the DOM, identifies embedded expressions by searching for appropriate code delimiters (here: ${ and }), parses the expressions into an AST, and evaluates them when the UI is rendered.

If the expression language syntax is expressive enough, attackers can create expressions resulting in arbitrary JavaScript code execution - for example by traversing a `prototype` chain or accessing object constructors [9] [10]. We found that various script gadgets can be typically identified in the framework expression parsing and evaluation engine which can lead to arbitrary code execution. For example, the following gadgets can be found in Aurelia's expression parser:

```
if (this.optional('.')) { // Property access
 result = new AccessMember(result, name);}
}
AccessMember.prototype.evaluate = function(...) {
 return instance[this.name];
};
if (this.optional('(')) { // Function call
 result = new CallMember(result, name, args);
}
CallMember.prototype.evaluate = function(...) {
 return func.apply(instance, args);
};
```

Listing 10: Script gadgets in Aurelia expression parser (simplified code)

It's possible to link the above script gadgets into chains that execute arbitrary functions such as `window.alert` - all by using only benign HTML markup injection. (Aurelia looks for `ref` and `*.bind` attributes in the document - that triggers our gadgets).

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

Listing 11: HTML Markup triggering gadget chain in Aurelia

In a similar fashion, the following benign HTML markup may trigger a gadget chain calling `alert` in Polymer 1.x:

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

Listing 12: HTML Markup triggering gadget chain in Polymer 1.x

3.6 Expressiveness of Gadget-based Exploits

In this section we discuss the expressiveness of gadget-based mitigation bypasses. Via gadgets, an attacker is able to execute arbitrary, Turing-complete code. In general, we identified three ways of doing so:

- **Eval-like functions:** If a gadget is able to trigger a call to `eval` or another eval-like function, executing arbitrary code is straightforward. In our examples, we usually demonstrate how the gadget is able to call a single function

[7] http://aurelia.io/
[8] https://angularjs.org/
[9] https://www.polymer-project.org/
[10] http://www.ractivejs.org/
[11] https://vuejs.org/

inside the `window` object with a single attacker-controlled parameter (e.g. `alert(1)`). As the `eval` function is also located inside the `window` object and accepts one or more parameters, all of these examples are capable of executing arbitrary, Turing-complete JavaScript code.

- **Appending a script element:** Another class of gadgets aims at appending a script element with either an attacker-controlled `src` attribute or an attacker-controlled script body. Similar to eval-based gadgets, this allows an attacker to execute arbitrary code.
- **Abusing the expressiveness of an expression language:** Most gadget-based mitigation bypasses leverage eval-like functions or new script elements. However, in Web applications employing some variants of CSP (see Section 4.1.1), it is not possible to use these bypass methods. In these cases, we can leverage expression languages to gain arbitrary code execution. All expression languages that we investigated are Turing-complete. If an exploit is able to execute the expression interpreter, the exploit is as expressive as the expression language itself. However, even if the expression language itself is not Turing-complete, we can still gain Turing-complete code execution in some cases. Listing 17, for example, shows a very simple expression-based attack to steal and reuse a CSP nonce in order to add a seemingly trusted script, that allows us to achieve arbitrary JavaScript code execution.

3.7 Finding Script Gadgets

Script gadgets (3.3) on their own are legitimate, trusted JavaScript statements or code blocks. While some of them (3.5.4) are also DOM XSS [16] sinks, others are as benign as property assignment, or property traversal statements. This fact makes it particularly difficult to identify such gadgets in the web application codebase.

We found the following two techniques are useful to identify script gadgets:

3.7.1 Manual code inspection. First of all, gadgets can be found manually or with the assistance of static-analysis tools. Finding some of the simpler gadget types (for example, JS execution sinks or Function creation gadgets) is straightforward. We found that more complex gadgets, especially the ones present in expression parsers, require significant effort to locate and evaluate for usefulness. A gadget may only be used if it's reachable from a benign HTML markup injection. For example, any property access, property setter, or function call may potentially be useful in a chain, but only if the property name or function object may be directly controlled from the markup.

We found that manual code inspection makes it possible to find gadgets that would not otherwise be triggered in the usual application code flow. For example, in Polymer 1.x (see Listing 12) we were able to determine that overriding a `_rootDataHost` property lets us execute JavaScript statements in a different scope, which lets us trigger subsequent gadgets in the chain. This "private" `_rootDataHost` property was never meant to be accessible from Polymer expressions.

In this research, we used manual code inspection to identify gadgets in modern JavaScript frameworks (4.1).

3.7.2 Taint tracking. A subset of gadgets may be identified by rendering the web application in a browser enriched with a taint-tracking engine [17]. By marking the entirety of DOM tree as tainted (i.e. simulating that the attacker has a reflected HTML injection capability), and checking whether tainted values reach specific JavaScript execution sinks, we were able to identify flows linking certain DOM selectors with JavaScript execution. While this approach is effective at scale, it has the limitation of only discovering gadgets that are already used in a given web application (albeit not neccesarily for script execution).

In this research, we used the taint tracking approach to evaluate script gadget prevalence in user-land code (5.4).

4 CONCRETE XSS MITIGATION BYPASSES USING SCRIPT GADGETS

In this section, we provide detailed information on how script gadgets can be leveraged to circumvent concrete state-of-the-art XSS mitigations. We'll follow the countermeasure classifications, based on their underlying mechanisms, that we introduced in Section 2.3.

4.1 Gadgets in Popular JavaScript Libraries

In order to measure the effectiveness of gadgets in bypassing XSS mitigations, we needed to collect:

(1) A list of XSS mitigation implementations with different strategies
(2) A list of as many gadgets as possible in popular frameworks and libraries

4.1.1 Collecting a list of popular XSS mitigations. We selected XSS mitigations that were either open-source, or widely distributed. We also wanted a cross-section different mitigation implementation strategies. The mitigations we decided to test were:

- **Content Security Policy** using different types of code filtering:
  - **Whitelist-based** where code is trusted based on where it originates.
  - **Nonce-based** where code is trusted only if it's accompanied by a secret cryptographic `nonce`.
  - **Unsafe-eval** source expression is usually used together with other policies, but looking at it separately allows us to investigate `eval`-based gadgets.
  - **Strict-dynamic** source expression is usually used together with a nonce-based CSP to automatically propagate the trust of a nonced script to all script elements generated by it.
- **Client-side HTML sanitizers** using different approaches of sanitization:
  - **DOMPurify** is a JavaScript-based HTML sanitizer that supports HTML, SVG, MathML, among others.
  - **Google's Closure** library contains another JavaScript-based HTML sanitizer that only supports HTML.
- **Web Application Firewalls** are request filtering mitigations deployed as hardware in front of web servers, as well as as software next to the web server itself.

CSP				XSS Filters			HTML Sanitizers		WAFs
Whitelists	Nonces	Unsafe-eval	Strict-dynamic	Chrome	Edge	NoScript	DomPurify	Closure	ModSecurity
3	4	10	13	13	9	9	9	6	9

Table 1: Mitigation-bypasses via gadgets in 16 Popular Libraries

  - **ModSecurity** is an open-source Web Application Firewall, commonly used with the OWASP Core Rule Set.
- **XSS filters** employ either request filter, response sanitization or code filtering approaches.
  - **Chrome / Safari** employs a code filtering approach, blacklisting scripts that appear in the request.
  - **Internet Explorer / Edge** employs a response sanitization approach, rewriting potentially dangerous responses with something safe.
  - **NoScript** employs a request filtering approach, blocking requests that look suspicious or potentially malicious.

4.1.2 Collecting a list of popular JavaScript libraries. In order to find as many different gadgets as possible to test against mitigations, we decided to search for gadgets in different popular JavaScript frameworks and libraries. We obtained the lists of popular frameworks and libraries from various online resources[12] [13] [14] [15] [16]. From those lists, we focused on searching for gadgets in the following frameworks (selected based on popularity and code familiarity by the authors):

- **Trending JavaScript frameworks** (Vue.js, Aurelia, Polymer)
- **Widely popular frameworks** (AngularJS, React, EmberJS)
- **Older still popular frameworks** (Backbone, Knockout, Ractive, Dojo)
- **Libraries and compilers** (Bootstrap, Closure, RequireJS)
- **jQuery-based libraries** (jQuery, jQuery UI, jQuery Mobile)

The process we used for manually identifying gadgets is described in Section 3.7.1, but generally it was done by identifying `HTML` and `eval`-based sinks, as well as any documented feature that seemed like an expression language. In cases when no sinks of that form were reachable, we then looked for any mechanism exposed by the framework or library that touched the DOM in any way, and manually audited the code.

In Table 1 we summarize how many frameworks had gadgets that could bypass each of the mitigations. Complete bypass collection found during this analysis is available in the GitHub repository[17].

[12] **Mustache Security** is a list of frameworks with gadgets. https://github.com/cure53/mustache-security/tree/master/wiki
[13] **GitHub** contains a list of trending front-end JavaScript frameworks. https://github.com/showcases/front-end-javascript-frameworks
[14] **TodoMVC** is a list of a sample application written in many different JavaScript frameworks. http://todomvc.com/
[15] **JS.org Rising Stars 2016** is based on the activity on different GitHub projects related to JavaScript frameworks in 2016. https://risingstars2016.js.org/
[16] **State of JS 2016** is based on a survey to web developers. http://stateofjs.com/2016/frontend/
[17] https://github.com/google/security-research-pocs

Table 2 within the Appendix also summarizes our research findings. For clarity, in the following sections we present and discuss only a chosen selection of those bypasses.

4.2 Bypassing Request Filtering Mitigations

Request filtering mitigations attempt to identify malicious or untrusted HTML patterns, and stop them before they reach the application. To accomplish this, these mitigations generally employ the following approaches:

- **Enumerate known strings used in attacks.** For example, HTML tags like `<script>` or attributes such as `onerror` allow the user to execute JavaScript with a single HTML injection. The ModSecurity Core Rule Set version 3.0 is, at the time of writing, one of the most comprehensive lists of attack vectors.
- **Detect characters used to escape from the contexts where XSS vulnerabilities usually occur.** For example, if an XSS vulnerability existed by directly injecting HTML where the application expected to just output text, a request filtering mitigation will attempt to detect the injection of < or >. If the vulnerability is present when injecting inside an HTML attribute, escaping from the attribute would be detected as the vulnerability.
- **Detect patterns and sequences frequently used in exploits.** For example, when an XSS attack is succesful, the user will often attempt to steal credentials, or issue HTTP requests. Therefore, some mitigations attempt to detect access to `document.cookie`, or access to `XMLHTTPRequest`. They also attempt to detect usual mechanisms to obfuscate code execution, like references to `eval` or `innerHTML`, even after doing several layers of agressive decoding.

Examples of XSS mitigations that adopt these approaches are:

- NoScript XSS Filter
- Web Application Firewalls

Request filtering mitigations detect only specific, XSS-related HTML tags and attributes. Gadgets use HTML tags and attributes that are considered benign, and that makes them capable of bypassing such mitigations. For example, if a library takes the value of the `data-html` attribute and executes it as HTML, mitigations in this group would not be able to detect that as malicious. An example of HTML markup triggering such gadget chain was shown in Listing 11.

In addition, detection of context-breaking characters suddenly becomes ineffective because some gadgets change the meaning of otherwise-safe text sequences, and make them dangerous. For example, in AngularJS the use of two curly braces `{{` is a way to define the beginning of an AngularJS expression. Aurelia, in turn, uses a different delimiter: `${`. An example of such seemingly-benign markup was shown in Listing 9.

```
<iframe src="//knockout.example.com/?xss=
      <div data-bind=value:a=location></div>
      <div data-bind=value:a.href=name></div>"
  name="javascript:alert(1)"></iframe>
```

Listing 13: Example of bypassing NoScript with Knockout gadget

A good example of how to bypass request filtering mitigations like NoScript with gadgets is presented in Listing 13. In this example the expressiveness of the framework is used to split an exploit such as `location.href=name` (which is detected as an attack by NoScript as the global name property can generally be set by an attacker to arbitrary content), into two components. `a=location` followed by `a.href=name`. Individually, these expressions are harmless, but together they allow the user to redirect the user to a JavaScript URL specified in the name attribute. NoScript is not able to parse the markup to figure out that they are both meant to be executed together.

4.3 Bypassing Response Sanitization Mitigations

Response sanitization mitigations are designed to reduce the number of false positive results that are potentially generated by request filtering. Instead of blocking potentially malicious requests, response sanitization mitigations aim to detect whether a suspicious payload actually gets injected into the response.

Response sanitization mitigations usually follow one of two different techniques:

- **Remove or neuter the malicious attack.** One possible way to tackle the potential injection of code is to neuter it, or remove it from the HTTP response. In this approach, the rest of the response is left as-is, but the suspicious code is removed or made inert.
- **Block the response completely.** Another possible way to react to an injection attempt is to completely block the response, and display an error to the user. This approach avoids cases in which an attacker tricks a mitigation technique into blocking a legitimate script (e.g. a frame buster).

Examples of implementations of XSS mitigations that adopt these types of approaches are:

- **HTML sanitizers.** Most HTML sanitizers work by taking a piece of HTML code and cleaning it of any malicious input, and returning otherwise safe HTML. Most HTML sanitizers, however, are based on whitelists that try to enumerate safe HTML tags and attributes across all browsers.
- **Internet Explorer / Edge XSS filter.** The XSS filter in Microsoft Internet Explorer and Edge also sanitizes HTML by replacing parts of HTML attributes and tag names with a pound # symbol. Note that while HTML sanitizers use whitelists, XSS filters on the other hand work on a blacklisting approach, enumerating dangerous HTML tags and attributes known by the browser.

Bypassing HTML sanitizers usually requires a slightly different approach than bypassing XSS filters. For HTML sanitizers, the gadgets must reuse an otherwise safe and whitelisted attribute, such as `class` or `id`. Gadgets that bypass XSS filters can also use custom HTML tags and attributes such as `ng-click` in Angular or `v-html` in Vue.

Given that mitigations based on response sanitization only block vulnerabilities, but make no attempts at detecting artifacts of exploits, this makes them easier to bypass, since gadgets are by definition "safe" code that becomes unsafe when it interacts with other JavaScript code that is otherwise safe. Aiming to lower the false positive rate by using response sanitization has the downside of not being able to detect attacks that exploit features that are normally safe when the JavaScript library is not used.

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

Listing 14: Example of bypassing DOMPurify with jQuery Mobile gadget

An example on how to use gadgets to bypass response sanitization mitigations is presented in listing 14. As far as DOMPurify is aware, the HTML it sanitized is completely safe. However, jQuery Mobile, upon encountering an element with the attribute `data-role=popup`, will automatically try to inject an HTML comment with its `id`. In the code above, we can escape from that comment and execute our code. Note that the same attack works against Internet Explorer's XSS filter.

4.4 Bypassing Code Filtering Mitigations

Code filtering mitigations are an evolution on top of response sanitization. They attempt to leave the potentially malicious markup untouched, and instead focus on preventing the execution of malicious code. This approach has even lower false positive rate than sanitization, since the code is filtered out only if it's actually about to be executed.

However, one side-effect of such an approach is that since gadgets do not directly execute any malicious code, but do so indirectly through trusted code, it is a lot harder for XSS mitigations based on code filtering to detect injections using gadgets.

The approaches taken by XSS mitigations based on code filtering are:

- **Detect malicious code.** To detect whether a specific piece of code is malicious, it is checked against the HTTP request. If the code to be executed is also present in the request, it is blocked as not trustworthy and potentially attacker-controlled.
- **Detect benign code.** Benign code passes various policy checks based on code provenance, content, or generation method. Code violating the policy requirements is considered malicious and its execution is blocked.

Examples of implementations of XSS mitigations that adopt this approach are:

- **Chrome and Safari's XSS Auditor.** The latest XSS filter to be implemented in a major browser was Chrome and Safari's XSS Auditor. The XSS Auditor hooks into JavaScript runtime in the browser. XSS Auditor uses the 'detect malicious code' approach - before Auditor permits code execution, it validates that the code was not included in the HTTP request, and blocks it if it was.
- **Content Security Policy.** Content Security Policy [34] is the most popular example of code-filtering mitigation. Web applications using this mitigation define a policy that specifies which scripts are benign and should be allowed to execute. Scripts violating the policy are blocked by the supporting browser. Existing policies usually adopt one the filtering variants described in Section 4.1.1. A typical policy is either URL whitelist-based or nonce/hash-based. A policy may also use `strict-dynamic` and/or `unsafe-eval` source expressions. These keywords propagate trust to additional code created by already trusted scripts, making CSP easier to adopt on existing websites.

Code filtering mitigations hook on code execution and aim to assure only legitimate code gets executed. Since script gadgets are already part of a legitimate code base they are extremely useful in bypassing this mitigation group. In the analysis performed against popular frameworks and libraries in section 4.1, we found that code filtering mitigations are the ones most vulnerable to gadgets. We used element construction gadgets (3.5.2), JavaScript execution sink gadgets (3.5.4) and gadgets in expression parsers (3.5.5) to bypass code filtering mitigations. While we found that expression-parser-based gadgets were the most universally applicable, some bypass methods employed were mitigation-variant specific:

Bypassing XSS Auditor. We bypassed XSS Auditor in 13 out of 16 frameworks, as many gadgets use traditional DOM XSS [16] sinks, DOM XSS protection being a known shortcoming of XSS Auditor [32]. For example, a gadget in the Dojo framework calls an `eval` function, with the value extracted from the `data-dojo-props` attribute. This allowed us to create the following bypass:

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

Listing 15: Example of bypassing XSS Auditor with Dojo gadget

Bypassing unsafe-eval CSP. In order to bypass CSP with an `unsafe-eval` keyword we either used gadgets in expression parsers or gadgets calling an `eval`-like function. Listing 15 demonstrates a bypass using such gadget. We were able to circumvent policies using `unsafe-eval` in 10 out of 16 frameworks.

Bypassing strict-dynamic CSP. Adding a `strict-dynamic` keyword to the CSP enables already trusted code to programmatically create new script elements. When such scripts are introduced into the DOM, they are implicitly trusted and allowed to execute. We found that most analyzed JavaScript frameworks contain gadgets capable of creating and inserting script elements with controlled body or `src` attribute. Such gadgets can be used to bypass `strict-dynamic` CSP. As an example, we present the bypass found in RequireJS:

```
<script data-main='data:1,alert(1)'></script>
```

Listing 16: Example of bypassing strict-dynamic with RequireJS gadget

Since the `<script>` tag has a `data-main` attribute, a gadget in RequireJS will generate a new `script` element, with its source pointing to `data:,alert(1)`. As RequireJS is already trusted, `strict-dynamic` propagates this trust to the new element, and the code will execute, bypassing the page's Content Security Policy.

We found `strict-dynamic` bypasses in 13 out of 16 tested frameworks (two of the bypasses relied on co-presence of `unsafe-eval`). The prevalence of script gadgets in the tested JavaScript frameworks suggests that using the `strict-dynamic` variant of CSP to mitigate XSS vulnerabilities in modern web applications is less effective than previously thought [35].

Bypassing other CSP variants. Both aforementioned CSP keywords relax the restrictions of the policy in order to facilitate its adoption. Some websites opt to use a stronger version of CSP, e.g. relying solely on nonces, or using a whitelist of script source URLs, with no known bypasses in the list of allowed origins [35]. We found that even such variants of Content Security Policy can be bypassed using script gadgets in expression parsers (3.5.5). In some frameworks, expression parsers themselves create a runtime environment that allows the attacker to obtain a `window` object reference and call arbitrary JavaScript functions. Such vectors do not use `eval` and do not create new script elements, so Content Security Policy cannot detect and block them. Listings 11 and 12 present examples for this type of bypasses. Such gadgets were found in Aurelia, Vue.js and Polymer 1.x. Additionally, in Ractive we found a gadget capable of exfiltrating the CSP nonce into a newly created script, allowing for its execution, despite a strong, only nonce-based policy:

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

Listing 17: Bypass exfiltrating CSP nonce in Ractive

It's worth noting that the success of CSP mitigation depends on the used variant. If the policy is configured to use whitelists, hashes, or nonces alone, then only gadgets in expression parsers (3.5.5) are useful, as the code passed to JavaScript execution sinks (3.5.4) would not be trusted. A notable exception is `strict-dynamic`, which

propagates trust to `<script>` tags generated programmatically. Attackers may bypass such CSP with gadgets generating arbitrary HTML elements, or importing nodes from foreign DOM documents. Such gadgets are common in templating libraries.

As we have presented above, the gadgets used to bypass different mitigations vary significantly from mitigation to mitigation. Some abuse the expression language in libraries, others inject markup in a text attribute, while others abuse trust propagation in DOM element creation. This indicates which type of gadgets to search for to bypass different types of mitigations.

5 PREVALENCE OF SCRIPT GADGETS

In this section we present the results of an empirical study on the prevalence of script gadgets in real-world applications. We first present our research questions and methodology, then discuss the results.

5.1 Research Statement

As shown above, script gadgets have the potential to undermine the protections provided by XSS mitigations. While we manually discovered many of these gadgets in popular libraries, it is important to understand the prevalence of these code patterns at scale. If gadgets are rare in real-world code, we can address the problem by taking special care when building generic libraries. If script gadgets are wide-spread in real-world applications however, addressing this problem might be as hard as fixing XSS itself. Therefore, the goal of this study is to measure the prevalence of gadgets in real-world applications.

After measuring gadget pervasiveness, we aim to find out more about the impact of script gadgets on specific XSS mitigations. Specifically, we would like to focus on the Content Security Policy and HTML sanitizers as these mitigation techniques seem to be the most robust and relevant ones.

A previous study [35] has already demonstrated that the domain whitelisting and the `'unsafe-inline'` CSP source expression harm the protection capabilities of CSP. In this study, we'd like to investigate the `'unsafe-eval'` and `'strict-dynamic'` source expressions. Specifically, we want to investigate how prevalent script gadgets are that can potentially bypass these expressions.

Many sanitizers, by default, allow seemingly benign attributes such as `data-*`, `id` or `class`. Furthermore, sanitizers usually allow non-malicious tags such as `div` or `span` tags. Hence, we'd like to understand how many real-world gadget chains can be triggered from such tags and attributes.

5.2 Methodology

In order to detect gadgets in real-world applications, we built a toolchain to automatically detect and verify them at scale. Based on this toolchain, we crawled the Alexa Top 5000 Web sites.

Detecting Gadgets at Scale. As we did not expect to see many expression parsers (see 3.5.5) present in user-land code (assuming that expression parsers are mostly present in JavaScript frameworks), we decided to focus on gadgets that end in HTML, JavaScript or URL execution sinks (see 3.5.4). In order to detect such potential gadgets, we built a browser-based, dynamic taint tracking engine. The engine is capable of reporting data flows from DOM nodes into security sensitive functions such as `eval`, `innerHTML`, `document.write`, or `XMLHttpRequest.open()`[18]. We used this engine to crawl our data set and identify all data flows. Each of these flows represents a potentially exploitable gadget chain.

Verifying Gadgets. In order to verify whether a found flow is exploitable from benign HTML markup, we built a generator that is capable of creating a real-world exploit based on each flow. The generator is similar to the one presented in [17]. Subsequently, we simulate a reflected XSS vulnerability in the page, into which we inject the generated exploit. The goal of the exploit is to indirectly execute a JavaScript function from a source that would not usually execute such code (e.g. from a `data-` attribute). Listing 18 shows an exemplary gadget that might exist in a legitimate JavaScript file.

```
<!-- source element -->
<div id="button" data-text="I am a button"></div>

<script>
  // Script gadget reading from #button element.
  var button = document.getElementById("button");
  button.innerHTML = button.getAttribute("data-text");
</script>
```

Listing 18: An exemplary gadget

For this sample, the engine detects a data flow originating from `button.getAttribute('data-text')` that ends up in the HTML execution sink `innerHTML`. Based on the context of the sink (HTML, JavaScript, URL), the exploit generator generates an exploit that triggers JavaScript execution within this context:

```
<svg onload=verify()>
```

Listing 19: XSS payload

Subsequently, we use the source element to generate the final exploit as shown in Listing 20. The actual XSS payload can thereby be disguised via the use of different encoding schemes (depending on the injection context).

```
<div id="button"
    data-text="&lt;svg onload=verify()&gt;">
</div>
```

Listing 20: Final Exploit

This lets us build the exploits in a way that our verifier function does not trigger by default. This function is called *only* if a script gadget reads the payload from benign markup and executes it. Therefore, if the function gets called, we have verified the gadget in a false-positive-free way.

[18] In total the engine supports 60+ sinks, which we cannot easily list due to space constraints

Crawling The Data Set. Our initial seed data set consists of the Alexa Top 5000 Web sites. We crawled these pages and also visited all the `http:` and `https:` links from these pages that point to the same domain or a subdomain. This approach might bias the data set, since Web pages with more links on the start pages will be over-represented in the final data set. The same is true for subdomains: Some Web sites make excessive use of subdomains, while others are not using them at all. Because of this, we decided to deduplicate our final results based on the first domain before the top level domain (subsequently called "second level domains"). E.g. we merge results from `sub.example.co.uk`, `example.co.uk` and `foo.example.co.uk` and just regard all of these domains as belonging to `example.co.uk`. We are aware that this approach has a significant impact on the final results, but we think that this provides the most realistic view on the data.

5.3 Limitations

Our testing and verification approach has the following limitations:

Only first level links: We only followed the first-level of links, so our data set does not cover all the pages of a site.

No user interaction: Our crawlers do not interact with the page. This means that we are only able to find gadgets in code that get executed at page load by default.

No authentication: Our crawlers do not authenticate to the pages under test. Consequently, we might have missed results in authenticated parts of an application, significantly reducing the potential coverage of crawled web applications.

Verification does not focus on mitigation bypasses: In the study, we do not artificially add, modify or remove any specific XSS mitigation to crawled websites. We only verify that a data flow from a non-executing source is capable of executing arbitrary code in a page via a gadget, even in the presence of a given mitigation. The reason for this is that some mitigations cannot be easily applied to Web sites. For example, applying a Web Application Firewall or Content Security Policy (see 2.3) to a page requires a non-trivial amount of configuration, and is likely to break the functionality when done automatically. Furthermore, exploits need to be adopted to the specific mitigation techniques. Hence, by focusing on the mere code execution aspect, we can verify gadgets more efficiently.

Our XSS simulation approach is false-negative-prone: In a real-world mitigation setting, the initial XSS attack should be blocked by stopping the execution of the injected code. However, even when the original injection was stopped, a gadget can still potentially execute the injected content, effectively bypassing the mitigation. For example, while `script` elements are initially blocked by CSP, they remain in the DOM and gadgets may reintroduce them, triggering them again. While this would be a valid mitigation-specific bypass, this payload would execute directly without triggering any gadget when a CSP is not present. In order to avoid such false-positive findings, we only generate exploits that do not trigger JavaScript execution by default. For example, we did **not** inject gadgets in the following form:

```
<div id="foo"><script>verify()</script></div>
```

Listing 21: Invalid Exploit

Instead, we transform the payload into a form that cannot execute by default, by using the `xmp` plaintext tag, for example:

```
<xmp id="foo"><script>verify()</script></xmp>
```

Listing 22: Non-executing Exploit

While this approach completely removes false positives from our results, it might cause a considerable number of false negatives. For example, often the name of a tag is part of the DOM selector trigerring the gadget. Hence, by changing the tag name (in the example: from `div` to `xmp`), the exploit might not be able to trigger the gadget correctly. Effectively we lowered our verification rate and in turn significantly increased the quality of our results.

Limitation Summary. All these limitations should be taken into account when reading the following sections. Most importantly, we want to point out that the presented results are lower bounds. If deep crawling, user interaction and a less restrictive verification are applied, the resulting numbers will likely be higher.

5.4 Results

This section is divided into several subsections. After reporting on general crawling results, we present numbers and statistics about the detected data flows. Then we report on the results of our automatic gadget verification, and finally we discuss the results in the context of XSS mitigation techniques.

5.4.1 Crawling Results. As mentioned above, our initial data set consisted of the Alexa top 5000 Web sites. By following the first-level links, we crawled 647,085 Web pages on the same domains or subdomains of this set, which finally contained 37,232 different sub domains and 4,557 second-level-domains. The number of second-level domains is lower than 5000, because some entries in the Alexa Top Sites file redirect to the same domain based on geo location. For example, google.it, google.de, google.fr all redirect to google.com. Furthermore, some Web sites were not reachable or timed out while crawling. In some cases, this is due to sites that only use regional CDNs. For example, a site from Asia might be fast in Asia but very slow when requested from the US or Europe. For all the remaining pages, we collected data flows using our taint engine.

5.4.2 Taint Results. On average we measured 7.67 sink calls per crawled URL and around 450 sink calls aggregated per second-level domain. In total, we counted 4,352,491 sink calls with data resulting from 4,889,568 unique sources within the DOM. Grouped by second-level domain, sink and source, we measured 22,379 unique combinations.

5.4.3 Mitigation results. In the following, we want to relate these results to the XSS mitigations, especially CSP 'unsafe-eval', CSP 'strict-dynamic' and HTML sanitizers.

Content Security Policy - 'unsafe-eval': As opposed to the 'unsafe-inline' keyword, `unsafe-eval` in the past seemed to be more secure in general. While `unsafe-inline` almost completely removes the protection capabilities of a CSP policy, `unsafe-eval` by default does not make the policy bypass-able. In order to bypass the policy with `unsafe-eval` an attacker needs to find an injection into a JavaScript execution function (`eval`, `new Function`, `setTimeout`, `setInterval`, etc.). Finding a direct injection is often hard and time consuming, because the use of such function is limited and can be easily audited by the application owner. Hence 'unsafe-eval' was seen as an acceptable trade-off between security and usability of CSP. However, the results of our study imply that this long-held belief should be changed. Gadgets can be used as an indirect way of reaching an execution sink. If DOM content gets evaluated by default, the attacker can inject the code as a DOM node in order to abuse the eval-gadget to execute arbitrary code. In our data set 47.76% of all second-level domains contained a data flow that ended within a JavaScript execution function. During our crawl, for example, we unintentionally automatically bypassed Tumblr's CSP policy with a gadget bypassing its `unsafe-eval` source expression.

Content Security Policy - 'strict-dynamic': The `strict-dynamic` source expression was added to CSP to increase the usability of nonce-based policies. As described in 4.1.1, `strict-dynamic` enables automatic trust propagation to child scripts. If a nonced, and thus legitimate, script appends a child script element to the DOM, the child script would be blocked unless the parent script propagates the nonce to the script as well. As many libraries are not aware of CSP, these libraries do not propagate the nonce and thus CSP would block the child script and break the library's functionality. When `strict-dynamic` is enabled trust is automatically propagated to non-parser-inserted script elements. Consequently, under `strict-dynamic`, child script elements are automatically executed even if they do not carry a nonce. In this situation, attackers may use gadgets to bypass CSP. If DOM content gets injected into a script element, or into a library function (e.g. `jQuery.html`) that creates and appends new `script` elements, `strict-dynamic` CSP can be bypassed. In order to measure potentially affected Web sites, we counted the following data flows:

- The data flows ending within `text`, `textContent` or `innerHTML` of a `script` tag
- The data flow ending within `text`, `textContent` or `innerHTML` of a tag, where the tag name is DOM-controlled (tainted)
- The data flow ending within `script.src`
- The data flow ending in a API which is known for creating and appending `script` tags to the DOM.

In total, 73.03% of all second-level domains contained at least one data flow with the described characteristics. For example, we detected a gadget capable of bypassing `strict-dynamic` in Facebook's fbevents.js library[19].

Content Security Policy - Summary. Given the numbers and examples provided above, we believe that `unsafe-eval` and `strict-dynamic` considerably weaken a CSP policy. Great care should be taken when using these source expressions.

[19]https://developers.facebook.com/docs/ads-for-websites/pixel-events/v2.9

HTML Sanitizers: Sanitizers aim at removing potentially malicious content. Most sanitizers do this by defining a known-good list of tags and attributes and removing anything else from a provided string. This list varies from sanitizer to sanitizer. The Closure sanitizer for example, removes `data-` attributes, while DOMPurify allows them in its default configuration. Furthermore, all sanitizers we looked at allow `id` and `class` attributes. Hence, we investigated whether this behavior is secure. In our data set 78.30% of all second-level domains had at least one data flow from an HTML attribute into a security-sensitive sink, whereas 59.51% of the sites exhibited such flows from `data-` attributes. Furthermore, 15.67% executed data from `id` attributes and 10% from `class` attributes. Based on these numbers, we recommend to revisit at least the sanitization approach towards blocking `data-` attributes.

5.4.4 Gadget Results. Based on the identified data flows, we generated 1,762,823 gadget-based exploit candidates, based on which we validated 285,894 gadgets on 906 (19.88%) of all second-level domains.

6 SUMMARY & DISCUSSION

Our study has demonstrated that data flows from the DOM into security-sensitive functions are very frequent in modern applications and frameworks. In fact, 81.85% of all second-level domains exhibited at least one relevant data flow. Furthermore, we have shown that we can detect these flows and generate exploits that are capable of bypassing all modern XSS mitigations. In a fully automated fashion, we detected and verified gadgets on 19.88% of all second-level domains. However, due to our methodology, we believe that this is just a lower bound for the real extent of this problem. By applying deeper crawling, authentication, user interaction and less conservative testing approach the numbers would doubtlessly increase considerably. We specifically removed or changed all exploits that would result in an immediate execution at the initial injection.

Given these results, we believe that XSS mitigations in their current form are not well aligned with modern applications, frameworks and vulnerabilities. In general, we see three different ways to address the issue of script gadgets:

6.1 Fix the Mitigation Techniques

Making mitigation techniques gadget-aware in general is hard. Today there are so many expression languages, frameworks, libraries and instances of user-land code that it will be very difficult to address all of the different types of gadgets. For example, request filtering mitigations (4.2) will have a hard time in detecting all the various forms that script gadgets can take, especially when the gadget chain makes use of string transformation functions. However, we believe that a few of the vectors can be addressed by specific mitigations. HTML sanitizers, for example, could start to filter `data-`, `id` or `class` attributes.

6.2 Fix the Applications

Another approach to address the identified problems is to try to fix the applications. Popular libraries and frameworks, for example, could aim at removing gadgets in order to safeguard their users.

Given the extent of the problem however, we will likely not be able to address this problem at scale.

As some gadgets and gadget chains are part of the feature set of a framework, it is unlikely that developers of such frameworks are willing to remove or restrict these features for preventing XSS mitigation bypasses. Furthermore, we found a number of unintentional gadgets; code paths that were triggered through gadgets that were not intended by their developers. These unintended code paths are hard to find, sometimes even harder than a simple XSS vulnerability. As a result, we believe that fixing XSS mitigations and script gadgets might be as hard and time consuming as fixing the XSS problem itself.

6.3 Shift from Mitigation to Isolation and Prevention techniques

Due to the results of our study, we believe that the focus of Web Security engineers should shift from mitigation techniques towards isolation and prevention techniques. Sandboxed Iframes [13], Suborigins [36] or Isolated Scripts [22] are promising proposals for Isolation techniques. Furthermore, the Web needs to focus on XSS prevention techniques: The Web platform is inherently insecure. A novice programmer without much security knowledge is hardly able to create a secure Web application. The Web platform should let a developer easily create a secure app by providing secure-by-default APIs. Language-based security concepts, for example, could be added to the Web platform, so that it is impossible to introduce security vulnerabilities without malicious intent.

7 RELATED WORK

Client-side XSS:. While the source of the initial content injection can be caused by all classes of XSS, gadget-based attacks are rooted in insecure client-side data flows caused by JavaScript. Thus, the closest related class of vulnerabilities is client-side XSS, also known as *DOM-based XSS*. The first public documentation of this vulnerability class was done by Amit Klein in 2005 [16]. In 2013 Lekies et al. [17] conducted a large scale study that demonstrated the prevalence of this XSS type, showing that approximately 10% of the examined web sites exposed at least one client-side XSS problem. To address this problem, Stock et al. [32] proposed a taint tracking-based protection mechanism to stop insecure data-flows within the web browser. While taint tracking could potentially detect or stop gadget-based attacks, this paper only covers client-side data flows. Most of our exploits, however, have hybrid data flows that span across the client and the server. Hence, in its current version Stock et al.'s approach cannot stop our attacks. More recently, Parameshwaran et al. [26] advanced this defense via server-side instrumentation of the JavaScript code, thus eliminating the need of browser modifications. It is unclear to which degree these taint-based techniques can be adapted to address script gadget attacks, as the initial payload does not come from a untrusted source, and thus, are not easily distinguishable from the legitimate targets of the gadget code.

The potential security problems of insecure JavaScript transforming DOM content was initially documented by Heiderich et al. in two distinct variations. In the first, they showed how JavaScript frameworks like AngularJS create insecure injection vulnerabilities which are out-of-scope for classic server-side XSS sanitization techniques, due to custom client-side markup conventions [10]. Furthermore, they uncovered how specific, non-standard browser behavior potentially transformed initially secure DOM content into executable code, if read and rewritten via JavaScript [12]. Athanasopoulos et al. [2] described return-to-JavaScript, a similar attack scenario circumventing mitigations based on script whitelists. In their attack, the attacker executes already whitelisted scripts in an unwanted fashion. The basic assumption of their attack is that an XSS exists in the application and the attacker is only able to execute already whitelisted scripts. Under these assumptions the attacker could try to repurpose whitelisted scripts. For example, if there is a button with a whitelisted event handler that logs out the user, the attacker could reuse the whitelisted event handler and attach it to an `onload` event via the XSS vulnerability. In this way users would be logged out immediately once they visit the application. While the mitigation prevents general exploitation, the attacker could still harm the user experience considerably by abusing the existing scripts.

Circumventing XSS mitigations: The topic of undermining the protective capabilities of XSS mitigations has been explored previously as well. Zalewski [37] outlined potential future direction of mitigation combating in his influential essay "Postcards from the post-XSS world", touching many emerging techniques, such as content infiltration, whitelist abuse, or potential possibilities for Web code reuse attacks.

On the topic of browser-based XSS mitigations, Nava and Lindsay [23] and Bates et al. [3] exposed inherent weaknesses in XSS mitigation approaches that rely on regular expression based detection mechanism. These results directly motivated the design of the XSSAuditor [3]. In turn, Stock et al. [32] demonstrated the weakness of all string-based XSS filters in non-trivial vulnerability scenarios, such as partial or double injections.

In addition to research on client-side XSS filters, Content Security Policy was subject of several research endeavors. For one, in concurrent work Weichselbaum et al [35] and Calzavara et al. [4] examined the quality and effectiveness of currently deployed CSP policies with sobering results. In addition, Weichselbaum et al. [35] demonstrated how whitelist-based policies can be easily evaded using overly permissive whitelisted script providers. In complementary work, Chen et al. [6] and Van Acker et al.[1] presented various techniques to evade CSP's information flow restrictions. Furthermore, Pan et al [25] investigated how to automatically generate secure CSP policies (without the unsafe-inline or unsafe-eval keywords). While these policies could resist simple gadgets, such strong policies are still vulnerable to expression-based gadgets as outlined in section 4.4. Finally, Heiderich et al. [11] demonstrated how injected HTML and CSS code alone is sufficient to conduct a wide range of attacks, even when a comprehensive CSP for script execution prevention is in place.

8 CONCLUSION

In this paper, we comprehensively explored code-reuse attacks in Web pages using script gadgets. Script gadgets come in many variations and, as our empirical study uncovered, are omnipresent in modern Web code.

As we have demonstrated, the current generation of XSS mitigations is unable to handle XSS attacks that leverage script gadgets to execute their payloads. And, unfortunately, there is no linear upgrade path to adapt the current mitigation approaches to robustly handle the uncovered vulnerability pattern. While specific mitigation techniques can be modified to handle selected gadget types, the high variance of script gadget form and functionality, due to the vastly growing amount of custom client-side code and the constant flow of new client-side frameworks, prevents a comprehensive adaption to accommodate the problem.

This leads to a conundrum for the future of client-side Web security: The last 15 years of difficulty in addressing XSS have shown that XSS apparently cannot be thoroughly addressed in practice through secure coding practices alone. And the subject of this paper, especially in combination with complementary results [9, 32], suggest that the current approaches in XSS mitigation are insufficient to compensate the deficits of code-based XSS prevention.

The question then arises: how do we handle XSS on the road ahead? As discussed above, sophisticated isolation techniques could offer a third way of dealing with the potential consequences of attacker controlled JavaScript. Alternatively, safe code abstractions [15] and secure-by-default browser APIs [20] might also be an option to overcome today's inherent problems of ad-hoc, insecure Web content generation.

However, regardless of which paradigm the next generation of XSS countermeasures will be build upon, it is essential that they have to be capable to handle the unexpected client-side execution- and data-flows which may be caused by legitimate script gadgets.

REFERENCES

[1] Acker, S. V., Hausknecht, D., and Sabelfeld, A. Data Exfiltration in the Face of CSP. In *AsiaCCS* (2016).

[2] Athanasopoulos, E., Pappas, V., Krithinakis, A., Ligouras, S., Markatos, E. P., and Karagiannis, T. xjs: practical xss prevention for web application development. In *Proceedings of the 2010 USENIX conference on Web application development* (2010), USENIX Association, pp. 13–13.

[3] Bates, D., Barth, A., and Jackson, C. Regular expressions considered harmful in client-side XSS filters. In *WWW '10: Proceedings of the 19th international conference on World wide web* (New York, NY, USA, 2010), ACM, pp. 91–100.

[4] Calzavara, S., Rabitti, A., and Bugliesi, M. Content security problems?: Evaluating the effectiveness of content security policy in the wild. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 1365–1375.

[5] CERT/CC. CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests. [online], http://www.cert.org/advisories/CA-2000-02.html (01/30/06), February 2000.

[6] Chen, E. Y., Gorbaty, S., Singhal, A., and Jackson, C. Self-exfiltration: The dangers of browser-enforced information flow control. In *Proceedings of the Workshop of Web* (2012), vol. 2, Citeseer.

[7] Gundy, M. V., and Chen, H. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-site Scripting Attacks. In *16th Annual Network and Distributed System Security Symposium (NDSS 2009)* (2009).

[8] Heiderich, M. *Towards Elimination of XSS Attacks with a Trusted and Capability Controlled DOM.* PhD thesis, Ruhr-University Bochum, 2012.

[9] Heiderich, M. Jsmvcomfg - to sternly look at javascript mvc and templating frameworks. [online], https://www.slideshare.net/x00mario/jsmvcomfg-to-sternly-look-at-javascript-mvc-and-templating-frameworks, 2013.

[10] Heiderich, M. Mustache security wiki. [online], https://github.com/cure53/mustache-security, 2014.

[11] Heiderich, M., Niemietz, M., Schuster, F., Holz, T., and Schwenk, J. Scriptless attacks: stealing the pie without touching the sill. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 760–771.

[12] Heiderich, M., Schwenk, J., Frosch, T., Magazinius, J., and Yang, E. Z. mxss attacks: Attacking well-secured web-applications by using innerhtml mutations. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 777–788.

[13] Hickson, I. The iframe element, November 2013.

[14] Jim, T., Swamy, N., and Hicks, M. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web* (2007), ACM, pp. 601–610.

[15] Kern, C. Securing the tangled web. *Communications of the ACM 57*, 9 (2014), 38–47.

[16] Klein, A. Dom based cross site scripting or xss of the third kind. *Web Application Security Consortium, Articles 4* (2005), 365–372.

[17] Lekies, S., Stock, B., and Johns, M. 25 Million Flows Later - Large-scale Detection of DOM-based XSS. In *Proceedings of the 20th ACM Conference on Computer and Communication Security (CCS '13)* (2013).

[18] Louw, M. T., and Venkatakrishnan, V. BluePrint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *IEEE Symposium on Security and Privacy (Oakland'09)* (May 2009).

[19] Maone, G. Noscript, 2009.

[20] MSDN. toStaticHTML method. [API], https://msdn.microsoft.com/library/Cc848922.

[21] Nadji, Y., Saxena, P., and Song, D. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Network & Distributed System Security Symposium (NDSS 2009)* (2009).

[22] Nava, E. A. V. Fighting XSS with Isolated Scripts. [online], http://sirdarckcat.blogspot.de/2017/01/fighting-xss-with-isolated-scripts.html, January 2017.

[23] Nava, E. V., and Lindsay, D. Our favorite XSS filters/IDS and how to attack them. Presentation at the BlackHat US conference, 2009.

[24] Oda, T., Wurster, G., van Oorschot, P. C., and Somayaji, A. Soma: Mutual approval for included content in web pages. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 89–98.

[25] Pan, X., Cao, Y., Liu, S., Zhou, Y., Chen, Y., and Zhou, T. Cspautogen: Black-box enforcement of content security policy upon real-world websites. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 653–665.

[26] Parameshwaran, I., Budianto, E., Shinde, S., Dang, H., Sadhu, A., and Saxena, P. Auto-patching dom-based xss at scale. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2015), ACM, pp. 272–283.

[27] Roemer, R., Buchanan, E., Shacham, H., and Savage, S. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Info. & System Security 15*, 1 (Mar. 2012).

[28] Ross, D. Ie 8 xss filter architecture/implementation. *Blog: http://blogs. technet. com/srd/archive/2008/08/18/ie-8-xss-filter-architecture-implementation. aspx* (2008).

[29] Ross, D. Happy 10th birthday cross-site scripting! [online], https://blogs.msdn.microsoft.com/dross/2009/12/15/happy-10th-birthday-cross-site-scripting/, 2009.

[30] Stamm, S., Sterne, B., and Markham, G. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web* (2010), ACM, pp. 921–930.

[31] Stamm, S., Sterne, B., and Markham, G. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web* (New York, NY, USA, 2010), WWW '10, ACM, pp. 921–930.

[32] Stock, B., Lekies, S., Mueller, T., Spiegel, P., and Johns, M. Precise Client-side Protection against DOM-based Cross-Site Scripting. In *23rd USENIX Security Symposium (USENIX Security '14)* (2014).

[33] Tantek Celik, Daniel Glazman, I. H. P. L. J. W. Selectors level 4. *W3C Editor's Draft* (2017).

[34] W3C. Content Content Security Policy Level 3. W3C Editor's Draft, 10 May 2017, https://w3c.github.io/webappsec-csp/, May 2017.

[35] Weichselbaum, L., Spagnuolo, M., Lekies, S., and Janc, A. Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 1376–1387.

[36] Weinberger, J., Akhawe, D., and Eisinger, J. Suborigins. W3C Editor's Draft, 18 May 2017, https://w3c.github.io/webappsec-suborigins/, May 2017.

[37] Zalewski, M. Postcards from the post-xss world. *Online at http://lcamtuf. coredump. cx/postxss* (2011).

A XSS MITIGATION BYPASSES VIA SCRIPT GADGETS IN JS FRAMEWORKS

Framework / Library	CSP whitelists	CSP nonces	CSP unsafe-eval	CSP strict-dynamic	Chrome XSS Auditor	EDGE XSS filter	NoScript XSS Filter 5.0.2	DOMPurify 0.8.7	Google Closure HTML sanitizer (2017-05-01)	ModSecurity OWASP CRS 3.0.0
Vue.js 2.3.0			✓	✓	✓	✓	✓	✓	✓	✓
Aurelia (2017-03-21)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
AngularJS 1.6.1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Polymer 1.7.1	✓	✓	✓	✓	✓	✓	✓	- (<template)	- (<template)	✓
Underscore 1.8.3 / backbone			✓	-	✓	✓	✓	✓	✓	✓
Knockout 3.4.1			✓	✓	✓	✓	✓	✓	- (data- or comments)	✓
jQuery Mobile 1.4.5	-	-	✓	✓	✓	✓		✓	✓	✓
Ember.js 2.10.2	-	-	✓	✓						
React		-								
Closure				✓	✓	- (<a*)	✓			
Ractive 0.8.1	- ({{}} uses eval)	✓	✓	✓	✓	- (<script)	- (script node)	- (script)	- (script)	- (script)
Dojo 1.12.2			✓		✓	✓	✓	✓	- (data-)	✓
Requirejs 2.3.2				✓	✓	- (<script)				
jQuery 3.1.1	-	-		✓		- (<script)				
jQuery UI 1.12.1	-	-		✓	✓		✓	✓	✓	✓
Bootstrap 3.3.7				✓	✓	✓	- (HTML in HTML attr)	✓		