DOMPurify: Client-Side Protection Against XSS and Markup Injection

Mario Heiderich(✉), Christopher Späth, and Jörg Schwenk

Ruhr-University Bochum, Bochum, Germany
mario.heiderich@rub.de

Abstract. To prevent Cross-Site Scripting (XSS) and related attacks, sanitation of untrusted content is usually performed either on the server side, or by client-side filters like XSS Auditor or NoScript. However, modern web applications (including mobile apps) may not be able to rely on these mechanisms any more since untrusted content may pass these filters as ciphertext or may completely be processed within the DOM of the browser/app.

To cope with this problem, XSS sanitation *within* the Document Object Model (DOM) is required. This poses a novel technical challenge: A DOM-based sanitizer must rely on native JavaScript functions. However, in the DOM, any function or property can be overwritten, through a class of attacks called *DOM Clobbering*.

We present a two-part solution: First we show how to embed *any* server or client side filtering technology securely into the DOM. Second, we give an example instantiation of an XSS filter which is highly efficient when implemented in Javascript. Both parts are combined into a working and battle-tested proof-of-concept implementation called DOMPurify.

Keywords: Cross-Site Scripting · JavaScript · DOM Clobbering · Expression injection · Sanitization · Webmail encryption

1 Introduction

Since their introduction to the broader debate in the year 1999[1], XSS and Markup Injection attacks have been recognized as major threats to web applications. New attack classes and sub-classes of XSS are discovered and documented regularily [1–3]. To prevent these attacks and ensure that no malicious scripts are executed, any untrusted input must be sanitized thoroughly (removal of scripts, event handlers, certain styles, expression syntax, and other contextually-risky elements) prior to being rendered in the browser. At the same time, it is crucial that a good sanitizer does not remove too much markup, keeping False Positives at bay.

[1] http://blogs.msdn.com/b/dross/archive/2009/12/15/happy-10th-birthday-cross-site-scripting.aspx.

S.N. Foley et al. (Eds.): ESORICS 2017, Part II, LNCS 10493, pp. 116–134, 2017.
DOI: 10.1007/978-3-319-66399-9_7

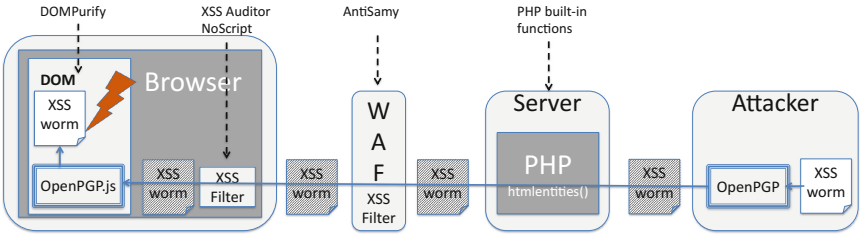


Fig. 1. Standard sanitation points, the position of DOMPurify and a possible attack scenario for an encrypted XSS worm

The sanitization can, for example, take place at the web server, at a dedicated web application firewall (WAF) module, or at a browser-embedded XSS detection module like MSIE's XSS filter, the Webkit/Blink XSS Auditor or NoScript (cf. Fig. 1). Regardless of whether server-side or client-side filters are used, the sanitization takes place *outside* a browser's Document Object Model (DOM), which means before the HTML page is rendered and the JavaScript DOM API is activated for this particular web page. However, sanitization may cease to be possible, given the modern and increasingly complex web application scenarios. For example, if untrusted content is encrypted when passing these filters (see the encrypted webmail example in Sect. 2), or when untrusted content is directly processed in the DOM of a browser/app (see Sect. A for examples).

Thus there is an urgent need for input sanitization *within the DOM*. Such sanitization may range from filtering to completely rewriting the input. However, one new challenge must be faced.

Introducing DOM Clobbering. Any sanitization function must be invoked from JavaScript, e.g. by calling `sanitize()`. An adversary may now overwrite this function by injecting an image with an appropriate `name` attribute, e.g. `<img name="sanitize">`. Calls to the sanitize function will now result in a error. This "overwriting" of important DOM methods is called *DOM Clobbering*, and is an attack technique that uses global variables created by the legacy DOM Version 0 still present in all relevant browsers.

Adversarial Model. We strictly adhere to the well-established *web attacker model* [4], and all our attacks are working in this model. Our adversary is thus able to set up malicious web applications, lure victims to visit and use these applications, may send emails, and may access any open web application on the Internet.

DOM Sanitation Challenges. Each XSS sanitizer that shall operate within the DOM should solve the following (sub-)challenges: **(1) Security Against XSS Attacks.** We used continuous automated testing against all *known* attack

vectors, and by community challenges aimed at discovering *novel* attack vectors. **(2) Robustness Against DOM Clobbering.** No web attacker should be able to deactivate the sanitizer, or, at the very least, such deactivation should be detectable. **(3) Low False Positive Rate.** It must be tolerant towards rich markup (e.g. novel legal HTML5 features like SVG). **(4) High Performance.** This challenge may seem trivial but please note that the best server-side filters, e.g. HTMLPurifier, rely on heavy code-rewriting, which is very performance-intensive and can be used to cause denial-of-service attacks by using deeply nested HTML lists or tables. **(5) Easy Deployment.** It must be easy to deploy and use. A safe setting should be default but configuration options should allow for an easy customization. **(6) Broad Deployment.** If the sanitizer is written purely in JavaScript, no changes to the browser engine are required for the sanitizer to work in any browser. The use of browser extensions, Java, Flash or similar technologies should be avoided.

Proposed Solution. The proposed solution mainly consists of two parts: The *DOMPurify HTML sanitizer* (which could be replaced by other filtering solutions) to address challenge 1, and the *DOMPurify DOM Clobbering Detection*, which addresses challenge 2.

To address Challenge 1 (Security against XSS), we answer the following question: which element-attribute combinations can be considered safe and should be white-listed?

For Challenge 2 (Robustness against DOM Clobbering) we have to investigate which attack vectors may influence the functionality of DOMPurify and similar DOM-based sanitizers. We describe how we protect against these attacks. Here we concentrate on DOM Clobbering, because this has only been investigated for complete XSS attack vectors up till now. Again please note that *any* DOM-based sanitizer may be deactivated by a DOM Clobbering vector that does not trigger an XSS attack and thus passes all known filters even if unencrypted.

Challenges 3 and 4 are in scope of an exhaustive investigation using over 1400 emails as discussed in Sect. 4.

Challenge 5 is solved by inclusion as an external script: `<script type="text/javascript" src="purify.js"></script>`. Strings are sanitized by executing the following code: `var clean = DOMPurify.sanitize(dirty);`.

DOMPurify is a DOM-only XSS sanitizer for HTML and SVG. In addition, DOMPurify only makes use of properties and methods available in the XHTML namespace, hence it can also be deployed in scenarios where MIME types such as `text/html` or `application/xml` are being used.

It's written in JavaScript and works in all modern browsers (Safari, Opera (15+), Edge, Internet Explorer (10+, `toStaticHTML()` fallback for older IE), Firefox and Chrome - as well as almost anything else using Blink or WebKit). DOMPurify doesn't break on IE6 or other legacy browsers but rather does nothing there. DOMPurify sanitizes HTML and prevents XSS attacks.

One can feed DOMPurify a string full of dirty HTML and receive a string with clean HTML in return. DOMPurify will remove everything that contains

dangerous HTML and thereby prevent XSS attacks and alike. We primarily use the technologies the browser provides and turn them into an XSS filter. The faster a browser DOM engine, the faster DOMPurify.

Advantages of DOM-Based XSS Filters Compared to Existing Solutions. DOM-based XSS filters offer plenty of advantages, especially when compared to their classic pendants on the server-side.

Immunity Against Obfuscation. One major advantage of placing the XSS filter in the DOM is the absence of code obfuscation. Importantly, such obfuscation has already been removed by the browser when loading the markup `dirty.html`. The problem of, for example, Charset XSS does not exist for a DOM-based filter because the browser is already operating in the correct charset setting[2].

Knowledge Advantage. A client-side, DOM-based XSS filter knows exactly the DOM of the browser it runs in. In essence, there is no knowledge gap between a constructed DOM on the server-side (as used by HTMLPurifier or AntiSamy) and the real DOM of the browser. This eliminates situations where a server assumes an element to be harmless but the browser uses it to execute (unwanted) JavaScript – this is for instance possible with mXSS or expression injections. Browser peculiarities that a server-side filter may not be aware of exist in a very limited and marginal capacity.

Performance. A Denial-of-Service attack can be conducted against server-side XSS filters with a use of very longs strings, deeply nested DOM nodes, XML attacks and other similar attack vectors. Once executed, such attacks can affect many users at the same time. If XSS filtering is however performed in the client, the effect of a DoS attack doesn't really extend to the server at all, but exclusively impacts the browsers of the targeted users.

Contributions. This paper presents original work. One of the authors published the first description of DOM Clobbering as a blog post, and is the core maintainer of the software described here. This paper makes the following contributions:

- We propose a framework for the novel problem of markup sanitation within the DOM, which will gain importance with the increasing usage of end-to-end encryption libraries like `OpenPGP.js`.
- As a proof-of-concept, we developed DOMPurify DOM Clobbering Detection, the first constructive solution to this issue which takes into account all of the research challenges. DOMPurify uses novel techniques to detect DOM Clobbering attacks: E.g. it verifies the integrity of a given function during runtime.

[2] http://zaynar.co.uk/docs/charset-encoding-xss.html.

– We propose a client-side XSS filter that is as tolerant as possible and doesn't remove benign user-input like forms, ID attributes, SVG and many other elements that are removed by other sanitizers for often no reason.
– We performed an extensive security evaluation of DOMPurify, by using automated tests and challenges to the research community. We also evaluated the performance and usability.

2 Example: An OpenPGP.js Worm

To exemplify the necessity of XSS mitigation within the DOM, let's consider the following case of an encrypted XSS worm as depicted in Fig. 1.

End-to-End Email Encryption. End-to-end encryption (E2E) has always been a desirable goal for IT security, even though we only became aware of its full practical importance in the post-Snowden era. Email is still one of the most important messaging services on the Internet, and the standards (PGP, S/MIME) and implementations (Thunderbird, Enigmail, Outlook, iOS Mail, K9-Mail, etc.) of email encryption are freely available. Nevertheless, the proportion of encrypted email communication is still negligibly low. One of the reasons behind this situation is that the large community of webmail users simply lacked the ability to decrypt or sign mails: browsers supported neither PGP nor S/MIME, and webmail users were thus forced to receive unencrypted mails. Since the publication of the JavaScript library OpenPGP.js, the landscape has changed rapidly: more and more projects are using this library to implement mail encryption for webmail applications. Additionally, Google have started their own end-to-end experiments, introducing yet another open-source E2E library to the market[3].

Possibility of an OpenPGP.js Worm. However, webmail security cannot be reduced to encryption. Since a computer worm can simply copy itself into emails sent to all recipients in the victim's address book, email remains an ideal basis for Internet worm propagation, together with social networking tools. The Samy worm (also known as "JS.Spacehero") is certainly the most famous XSS worm to date. Just eight months later the Yamanner worm used Yahoo! Webmail to spread itself by sending copies of its XSS code to all recipients in the address books of subsequent victims. Webmail XSS worms have already been described back in 2002[4] and a proof-of-concept implementation of a webmail worm which runs in different webmailers has been published in 2007[5]. XSS worms are still a problem[6], but large webmailers today mostly know how to sanitize *unencrypted*

[3] https://github.com/google/end-to-end.
[4] http://seclists.org/bugtraq/2002/Oct/119.
[5] http://www.xssed.com/article/9/Paper_A_PoC_of_a_cross_webmail_worm_XWW_called_Nduja_connection/.
[6] http://blog.gdssecurity.com/labs/2013/5/8/writing-an-xss-worm.html.

email traffic (on the server-side) to prevent XSS attacks. However, without utilizing a novel sanitation approach, we may soon face the threat of an "OpenPGP.js Worm" (cf. Fig. 1). This worm would simply contain a script that executes as soon as the decrypted mail is rendered, read all entries in the address book containing a PGP public key, copy its XSS payload to an email, and encrypt its content to avoid sanitation. Precursors of this likely occurrences can already be noted in attacks against ProtonMail and Tutanota, in which un-obfuscated XSS attacks were smuggled into the browser to abuse the mail encryption[7].

Reliability of Browser-Side XSS Filters. Existing browser-side XSS detectors and filters like MSIE's XSS filter, WebKits's XSS Auditor or even NoScript cannot mitigate this problem. Given their position in the markup processing chain, they only see the encrypted content. Content sanitation can solely be done *after* decryption of the message by OpenPGP.js is completed, i.e. only within the DOM of the browser. Please note that the inability of current browser-side XSS filters to mitigate this novel scenario is solely based on the fact that they cannot be called from the DOM: If the DOM API would be extended by a function e.g. called `sanitize()`, which would accept HTML Markup and returned a sanitized version of this markup, then this function could be used as an XSS filter for end-to-end encrypted content, too. *So in principle we can adapt any known filtering solution (e.g. all solutions from* Sect. 5*) to the novel scenario, provided we make them accessible from the DOM by a DOM API call.*

Insecurity of Straightforward Solutions. Regrettably, without additional protection mechanisms, this solution may be easily switched off by an attacker: it is here where techniques like DOM Clobbering come into play! If a client-side XSS filter can be called via a JavaScript function, it must either rely on standard interfaces offered by the DOM, e.g. functions to select certain elements for inspection and traversal, or it must be offered as an extension to the DOM API (e.g. as a function called `sanitize()`). If an attacker manages to deactivate one of these basic functions (or the new function `sanitize()`), and does so by using an attack technique that doesn't count as an XSS attack, then the filtering logic will let it pass. DOM Clobbering is such an attack. The attack works as follows: (1) An attacker sends an initial email which contains no XSS attack vector, but only a DOM Clobbering attack that deactivates a basic DOM API function. If this email is decrypted and rendered, it will simply switch the XSS sanitizer off. (2) Now a full XSS attack vector can be sent with a second mail, which will not be sanitized at all.

DOM Clobbering. These two-stage attacks may seem strange at first, but are actually quite easy to perform: if the first mail contains an element `<img src=""name="getElementsByTagName">`, the rendering of this mail will overwrite one of the most important selector methods available to a

[7] http://www.theregister.co.uk/2014/07/11/tutanota/.

JavaScript function, the method `document.getElementsByTagName()`. When this function/method is called afterwards, an exception will be raised stating that `document.getElementsByTagName()` is not a function. This would break most JavaScript based XSS filters, so we have to protect against such attacks. This "overwriting" of important DOM methods is called *DOM Clobbering*, and is an attack techniques that uses global variables created by the legacy DOM Version 0 still present in all browsers.

How DOMPurify Mitigates the OpenPGP.js Worm. In an end-to-end encrypted webmail application, DOMPurify will be applied directly to the result of the OpenPGP.js decryption (cf. Listing 1.1).

```
sanitized_message =
DOMPurify.sanitize(openpgp.decryptMessage(mykey.key,
    openpgp_encrypted_message));
```

Listing 1.1. Filtering of decrypted webmails

DOMPurify has already been implemented in the Mailvelope software that is currently being used to deliver end-to-end encryption features to several large mail providers including government-backed programmes such as de-mail. DOMPurify is in addition being used as the client-side security filter for FastMail and other web-mail providers.

3 DOMPurify

This section is dedicated to a presentation of deployment and basic functionality of our DOM-based XSS filter DOMPurify. This should aid an understanding of the novel security challenges. These newly outlined and challenging issues apply to *all* DOM-based sanitizers. The ways we chose for tackling and ultimately solving these challenges are discussed in the next section.

3.1 Novel Mitigation Paradigms

The following section lists and discusses the abstract mitigation concepts and derives general rules applicable to other client-side validation tools running in similar contexts.

Reviewed List of Element-Attribute Combinations. For an XSS filter and HTML sanitizer a capacity to tell apart "the good" and "the bad" is of paramount importance. This is usually done in two ways. The first option is that a filter employs a black-list of known-bad elements and attributes as well as attribute values. The alternative depends on the creation and subsequent use of a list of benign items that is maintained and enforced as a classic white-list. While this second option may appear tempting, our studies strongly suggest that the majority of XSS filters actually behave in a too strict manner and remove too

many benign elements. Therefore, they tend to cripple a user-submitted HTML unnecessarily and negatively impact on the usage experience. Further, only few of the inspected tools allow using SVG and provide a subset of considerably safe HTML and SVG elements and attributes. DOMPurify aims to be as tolerant as possible, which is highlighted in the fact that it supports HTML5+, SVG 1.2 Full and MathML 3.

We thoroughly studied the behavior of different HTML, SVG and MathML elements in all supported browsers. We concluded that the list of permitted elements can be larger than usually perceived and normally implemented by other tools. DOMPurify currently considers 206 different elements safe and permits them for user-submitted HTML, SVG and MathML. We similarly studied the behavior of element-attribute combinations and arrived at the result that deemed allowing 295 different attributes possible, seeing as they were considered safe for usage and incapable of leading to a JavaScript execution. Furthermore, we examined what was necessary for a client-side filter to successfully sanitize Shadow DOM elements. As a result, we implemented code to permit DOMPurify to perform that operation as well. To illustrate this in Sect. 4 we demonstrate that the concept of maximum tolerance is useful for sanitizing the entirety of SVG images used by Wikipedia without overwhelming amounts of false positives.

Exploring the other side of the continuum as well, we managed to identify attributes that are potentially harmful but considered safe by WHATWG and therefore make AngularJS's sanitizer[8] prone to XSS attacks (an attack we reported is currently being fixed):

The WHATWG organization maintains their own list of considerably safe elements and attributes[9]. This list is being used by the sanitizer functionality offered by the AngularJS library. We identified a problem with this list as the WHATWG did not fully test all element-attribute combinations. Thus, once implemented, the presented collection, despite being formerly assumed benign, in fact allows for dangerous XSS attacks. The problematic attributes reside in the SVG namespace and the sample attack vector below shows a full bypass which leads to XSS whenever WHATWG's unadapted list is unreflexively used. A change request was filed to update the WHATWG's list to a safer level. The page is now displaying a deprecation warning.

```
<svg><a xmlns:xlink="http://www.w3.org/1999/xlink" xlink:href
    ="?"><circle r="4000"></circle><animate attributeName="
    xlink:href" begin="0" from="javascript:alert(document.
    domain)" to="&amp;"></animate></a></svg>
```

Listing 1.2. Using xlink:href-animation to cause XSS via SVG

Internal DOM Clobbering Protection. As a library that runs entirely in a browser's DOM, DOMPurify is of course prone to DOM Clobbering attacks. In brief, an attacker could try to craft a HTML string that is to be sanitized by

[8] https://docs.angularjs.org/api/ngSanitize/service/%24sanitize.
[9] Sanitization rules, https://wiki.whatwg.org/wiki/Sanitization_rules.

DOMPurify. Upon parsing, the HTML encapsulated by the string could attempt to clobber the DOM and overwrite important functionality needed by DOMPurify to sanitize successfully.

To mitigate DOM Clobbering attacks against DOMPurify and its core functionality, we use the boot-strapping phase between mapping the markup for sanitization into a document implementation, and the initialization of the node iteration. This means a walk over all generated HTML elements after checking if all methods utilized by DOMPurify are really the methods we believe them to be. The same applies to certain DOM properties used by our library - here we use constructor checks as well as instances of operators to verify their identity. In case it turns out that a method is not what we expect it to be, the library simply returns an empty string and no potentially dangerous markup can enter the DOM of the protected website.

External DOM Clobbering Protection. DOMPurify also assures that the markup resulting from a sanitation process cannot clobber the already existing data on the website that the HTML is being used on. Overall, this relies on a generous allowing of ID and NAME attributes, as well as upfront checks against the hosting DOM (to verify if it already uses references of the same name). It is important to note that for those checks an "in" operator is used. We cannot make use of "typeof" since most modern browsers return "undefined" for `typeof document.all`, for instance. Further note that the clobbering protection checks both "window" and "document" in case "ID" attributes are found and only verifies the document if "name" attributes are discovered. This is because Gecko-based browsers (Firefox, etc.) create two rather than just one reference for HTML elements applied with ID attributes – one in the global object and one in "document".

Specifics in DOM Parsing and Traversal. To avoid being vulnerable to certain re-indexing - and mXSS attacks, DOMPurify makes sure that attributes and elements are exclusively parsed and processed in reverse order of appearance in the parent element or container. Note that this feature was also tested extensively in MSIE where the DOM-engine doesn't maintain the original attribute order but also orders attributes of elements in an alphabetic order after applying them to the DOM. This behavior can lead to a race-condition-based bypass of client-side XSS filters and was mitigated in DOMPurify.

3.2 Description

DOMPurify uses a combination of both element and attributes white-lists, which are paired with a very scarce reliance on regular expressions for detecting potentially dangerous values. It comprises two main components: the DOM Clobbering Detection *(DCD)* mainly consists of the function `_isClobbered`, which checks if references to the actual element being (XSS-)sanitized have been overwritten. Additionally, for each `id` or `name` attribute it is checked if its value could be

used to overwrite functions that are essential to DOMPurify. *This DCD can be adapted to protect any XSS mitigation solution proposed in the literature* (Fig. 2).

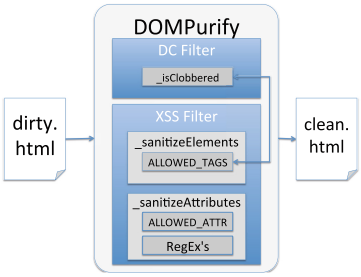


Fig. 2. Block diagram DOMPurify.

The *HTML Sanitizer* is a novel design to implement XSS mitigation in the DOM. Since no obfuscation may be present within the DOM, it uses a combination of whitelists (`ALLOWED_TAGS, ALLOWED_ATTR`) to skip secure tags/attributes, and carefully crafted regular expressions to delete dangerous attributes.

The function `_sanitizeElements()` uses the white-list `ALLOWED_TAGS` to skip secure tags, and the function `_isClobbered` to check if the markup contains a DOM Clobbering vector. It iterates over all elements in `dirty.html`. The function `_sanitizeAttributes()` uses the white-list `ALLOWED_ATTR` to skip secure attributes and employs some specially crafted regular expressions to ensure the detection of dangerous attributes. Before describing the novel mitigation paradigms - the key foundations of DOMPurify – we have to explain the novel security challenges that a DOM-based XSS sanitizer faces. More importantly, we outline how we coped with these challenges.

One of the additional core features of DOMPurify is the detection and mitigation of DOM Clobbering attacks – both against the library itself and against the surrounding website. This is achieved through the use of strict type checking of all DOM features that the library uses. If one DOM feature appears to have been tampered with, DOMPurify will abort immediately and return an empty string instead of the potentially unsanitized content. Further, DOMPurify allows sanitizing inactive elements inside a Shadow DOM. This works even for recursive Shadow DOM implementations where one Shadow DOM hosts several other, nested Shadow DOM instances.

The clobbering detection happens in two different locations of the library and covers both element clobbering and clobbering of global properties inside window or document. The clobbering tests are being performed for each HTML element or node that is being iterated over. When DOMPurify sanitizes an HTML string containing form nodes, it checks if the DOM it is working in could be clobbered by those form elements. The same takes place for any element applied with an ID or NAME attribute, aiming at avoiding global and document clobbering. If an element in the untrusted HTML string may have clobbering effects on the surrounding document, then it will be removed. In any other case it will be left intact. DOMPurify is therefore the only HTML filter that safely allows the use of ID and NAME attributes in untrusted HTML. This enables DOMPurify to also sanitize full forms and preserves ID attributes that are often important for site navigation via `location.hash`/anchors. To our knowledge, no other analyzed HTML and XSS filter allows that to occur in a safe manner.

To summarize the description of how DOMPurify works internally to maintain a high level of compatibility to benign markup and at the same time make

sure that no form of malicious markup is allowed to pass the filtering mechanisms, a list of interla processing steps has been created:

(1) DOMPurify, upon being started, first verifies that all necessary parameters are set and valid and then checks if the browser is compatible with all required features. If DOMPurify is not fully supported, it will attempt to call a fall-back method such as `toStaticHTML` or simply return the same string it was receiving for sanitation. This will make sure that DOMPurify exposes maximal compatibility paired with good protection on older browsers such as MSIE8, and have no noticeable impact on browsers incompatible with DOMPurify such as MSIE6 (it will simply do nothing here).
(2) DOMPurify will then perform a check against the DOM to verify that all needed features are indeed trustable and free from tampering (i.e. is the `removeChild` method really what it claims to be?). DOMPurify will create and store safe references for all verified safe functions and methods to make sure, that an attacker cannot interfere with the library at runtime and exchange important objects and methods in mid flight.
(3) DOMPurify will then determine, how to best create a safe, reliable and isolated document object given the browser it is running on (inert DOM). DOMPurify will preferably chose the DOMParser API and fall back to `document.implementation` where necessary. The created document object is then being populated with the string or node to be sanitized. Note that depending on not only the browser but also the browser version, different methods need to be chosen to produce a safe document. This is especially for Chrome browsers in versions 12 to 16 and Firefox browsers starting around version 34, as they implement slightly different behaviors, leading to insecure isolated documents if used improperly.
(4) Once the inert DOM has been created and populated, DOMPurify will start iterating over each of the elements in that DOM by using the safely stored and reliable NodeIterator API. Before the first element is being inspected, DOMPurify will call an optionally present hook function. By adding hooks, developers can customize the behavior and extend the feature list. The library offers hooks at all relevant joint-points between unsanitized markup and the sanitization process itself. Any hook method is given the current execution context as parameter to avoid the risk of developers accidentally getting access to a malicious context.
(5) DOMPurify will then inspect the first element, match it to the existing whitelist and either remove it or keep it in the DOM. Two additional hooks can be called during this process for extended customization. If the inspected element is a standard DOM element (such as a DIV or an anchor), DOMPurify will next iterate over all attributes of that element. If the element is however a template element, DOMPurify will invoke a different internal function that allows to recursively sanitize a Shadow DOM before continuing with the next elements. DOMPurify will, if enabled via configuration, in this step also check the element's text nodes for strings that indicate presence of a templating expression and, if instructed to do so, remove it. This was

implemented to protect against XSS via template expressions, popular in AngularJS applications.

(6) Once finished with the basic sanitization of an element itself, DOMPurify will as mentioned iterate over and initially remove all existing attributes in reverse order to respect the internal indexing browsers perform. The attribute name and value will both be matched against the mentioned white-lists, and the library will, if instructed so, also inspect the attribute value for template expressions. In case the attribute is classified to be used in combination with URLs (such as `href` or `action`), DOMPurify will also sanitize this value to prevent XSS and mXSS via URL, especially respecting the risks on XSS via Unicode whitespace and HTML5 character references in Chrome, Opera and Safari. During the process of attribute sanitization, DOMPurify will check for three additional hooks to be present to enable customized behavior. DOMPurify will further check, if the element is applied with attributes that cause DOM Clobbering and check the existing DOM for collisions.

(7) Once DOMPurify checked all attributes, it re-adds the safe ones and returns the sanitized element so it can be added to the safe document and proceeds to the next element selected by the NodeIterator. If no additional element is present, DOMPurify will take the existing DOM tree, serialize it into a string and return it, or, if instructed via configuration, return a DOM fragment or a DOM node. If more elements are present instead, DOMPurify will continue sanitizing the document until the final element has been reached. Note that the attacker cannot inject new elements into the document to sanitize during the process of sanitization, mitigating denial of service risks.

DOMPurify does not store any internal states after a sanitization process, so an attacker cannot bypass the library using multiple sanitization runs in a sequence. This was possible in very early versions of the library thanks to an attack using "Double-Clobbering", namely first changing the library core and then bending the sanitization functionality to produce harmful HTML in a second sanitization run. In the currently deployed version of the library, no bypasses are known.

4 Evaluation

This section discusses the evaluation of DOMPurify's security, performance and false positives.

4.1 Security Evaluation

Methodology. The security of DOMPurify was evaluated in two parts. The first part was a strict empirical analysis where we used automated tests to check the security of DOMPurify against all *known attack vectors*. In the second part we went one step beyond this evaluation, by challenging the security community to test DOMPurify in a white-box test against yet *unknown attack vectors*.

For the first part we used automated testing using a unit test suite, with existing state-of-the-art collections of XSS vectors to automatically check each new version. This is the standard evaluation procedure for any XSS filter, and the collections are constantly being updated to cover each new attack class (e.g. Scriptless Attacks, mXSS, expression injection). We used the following collections: (1) The HTML5 Security Cheatsheet[10], which contains 149 XSS vectors; (2) the OWASP XSS Cheat Sheet, containing 108 XSS vectors.

Second, we received a large number of novel attack vectors by the security community. These consisted of bypasses of early versions of DOMPurify, including numerous DOM Clobbering attack vectors. Altogether we collected 400+ attack vectors. Any bypasses discovered in manual testing were dynamically added to this collection of attacks and will therefore be tracked from this point forward. The vast array of attack vectors used here is publicly available on Github[11].

A public browser-based smoke-test is made available, allowing anyone to test the software quickly and without a need to set up anything. In addition, the security of DOMPurify was tested and further enhanced through a third-party audit in February 2015[12].

Results. We designed DOMPurify to mitigate all known attack vectors from the collections mentioned above. Furthermore we adapted DOMPurify to also mitigate the novel attack vectors. In summary, in the current version of DOMPurify there are neither undetected XSS vectors nor bypasses to DOMPurify.

4.2 Performance and False Positive Evaluation

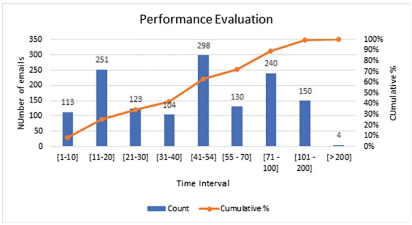
Methodology. For the evaluation of DOMPurify's performance and false positive rate we subscribed to more than 50 public email marketing lists[13,14] of a range of different topics (politics, sports, psychology, photography, daily digest). We decided to use this data set for several reasons. First, marketing newsletters contain a rich and diverse set of markup in order to attract the customer. Secondly, a wide field of topics and the way its content is presented is more representative for users of different ages, countries, sexes and interests - representing a variety of styles for composing emails. We provide a downloadable copy of the dataset for interested readers [5]. Over two months we accumulated a total of 1421 emails (136 MB of data). All emails were received using a google email account on a locally running Roundcube[15] instance on a Virtual Machine with Ubuntu. Roundcube is a popular web based email client, which we use to take care of the management of emails (load/save).

[10] https://html5sec.org/.
[11] https://github.com/cure53/DOMPurify/blob/master/test/expect.json.
[12] https://cure53.de/pentest-report_dompurify.pdf.
[13] https://www.getvero.com/resources/50-email-newsletters/.
[14] https://blog.bufferapp.com/best-newsletters.
[15] https://roundcube.net/.

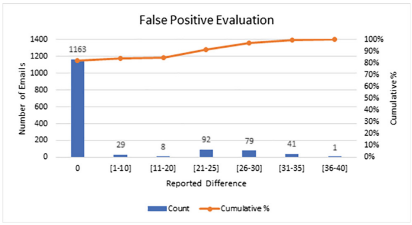
For the evaluation we proceeded as follows: we created a screenshot of the rendered DOM of both the clean and dirty email with html2canvas[16]. Then we computed the percentual difference between the two versions of the email using resembleJS[17] (e.g. 10%). The performance data constitutes the time taken (ms) by DOMPurify to process a given input file and output the result.

Of the 1421 emails we had to exclude 8 emails from the test set because they were not processed by html2canvas. This left us with a total of 1413 emails as the basis for our evaluation. All tests were executed on a Mac Book Pro Retina[18] on Firefox 52.0.2.

Results - Performance Evaluation. The results of our evaluation are depicted in Fig. 3a. DOMPurify's average processing time of the 1413 emails is 54.7 ms. Our evaluation shows that 62% of input data is processed below the average processing time. 80% of the emails are processed in ≤89 ms (Pareto principle). During our evaluation we observed that factors such as the loading and rendering of images and resources from remote hosts took several seconds up to minutes (for less responsive remote hosts). Summarizing our results, we observed no negative impact on user experience when using DOMPurify as a Sanitizer.



(a) Performance Evaluation: share of emails processed by DOMPurify within a given time frame



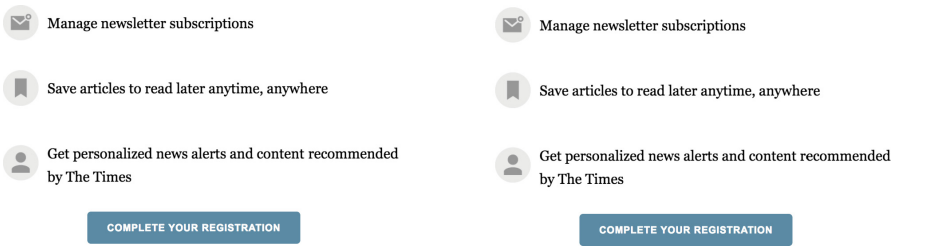
(b) False Positive Evaluation: share of emails with a reported difference according to resembleJS

Fig. 3. Performance and false positive evaluation results

Results - False Positives Evaluation. As shown in Fig. 3b, more than 80% of the testset show no visual difference (0.0%) between the original and processed email. For the remaining 20% we investigated the reported impact. We conclude that all reported differences can be attributed to the shifting of text.

To verify these findings we observed the visual display of the emails with the highest reported differences in the browser. For example, consider the rendering of the email with the highest reported difference of 40% in Fig. 4a and b. The astute reader will notice that when manually inspecting these two emails in the

[16] https://html2canvas.hertzen.com/.
[17] https://huddle.github.io/Resemble.js/.
[18] OSX 10.12.3 with a 3.1 GHz Intel Core i7 and 16 GB of 1867 MHz DDR RAM.



(a) The display of the unmodified email. Screenshot taken manually

(b) The display of the cleaned email. Screenshot taken manually

Fig. 4. Example of a false positive evaluation

browser there is no visual difference between the original and the processed email - except for a shift of a few pixels. However, this shows that DOMPurify has little impact on the processed emails.

We provide a downloadable copy of the highest reported differences including an email with no reported differences for comparison [6].

5 Related Work

XSS Mitigation. Server-side mitigation techniques range from a simple character encoding or replacement, to a full rewrite of the HTML code. The advent of DOM XSS was one of the main reasons behind the introduction of XSS filters embedded into the browser. The IE8 XSS Filter was the first fully integrated solution [7], timely followed by the Webkit XSS Auditor in 2009 [8]. For Firefox, browser-embedded XSS filtering is implemented through the NoScript extension. XSS attacks' mitigation strategies have been covered in numerous publications [1,9–13]. Noncespaces [14] use randomized XML namespace prefixes as an XSS mitigation technique, which would make detection of the injected content reliable. DSI [15] tries to achieve the same goal based on a process of classifying HTML content into trusted and untrusted variety on the server side, subsequently changing browser parsing behavior so that the specified distinction is taken into account. Blueprint [16] generates a model of the user input on the server-side and transfers it, together with the user-contributed content, to the browser, making its behavior modified by an injection of a JavaScript library for processing the model along with the input. Content Security Policy (CSP) comes with a novel feature to block XSS attacks: source whitelisting. Thus even if an attacker may be able to execute a malicious script, he is not able to exfiltrate any security critical information, since he cannot establish a connection to his server. Unfortunately, this feature does not apply to `mailto:` URLs, so this would not block webmail XSS worms. CSP 2.0 and newer (http://www.w3.org/TR/CSP2/) provide a mechanism to distinguish trusted inline script from untrusted script: the `nonce-source` and `hash-source` directives. In both cases,

inline scripts are only executed if the value of their nonce-attribute matches the value given in the directive, or if the hash value of the script matches a given value. Note that Weichselbaum et al. discuss the practical value of CSP in great detail, shedding light on shortcomings and implementational problems [17].

Sandboxed iFrames. A straighforward solution to block webmail XSS worms seems to display HTML mails in sandboxed iFrames, a novel HTML5 feature. This works for simply reading emails, but as soon as any action is triggered (e.g. FORWARD or REPLY), the sandboxed iFrame must be opened, and the XSS vectors contained in the mail body will be executed. This is a problem with all of today's webmailers.

Mutation-Based (mXSS) and Scriptless Attacks. Weinberger et al. [18] give an example of the innerHTML being used to execute a DOM-based XSS. Comparable XSS attacks based on changes in the HTML markup have been initially described for client-side XSS filters. Nava and Lindsay [19] and Bates et al. [8] show that the IE8 XSS Filter could have once been used to "weaponize" harmless strings and turn them into valid XSS attack vectors. This relied on applying a mutation through the regular expressions used by the XSS Filter. Zalewski covers concatenation problems based on NUL strings in *innerHTML* assignments in the *Browser Security Handbook* [20]. Additionally, he later dedicates a section to backtick mutation in his volume "The Tangled Web" [21]. Other mutation-based attacks have been reported by Barth et al. [22]. Hooimeijer et al. describe the dangers associated with the sanitization of content [23] and claim that they were able to produce a string that would result in a valid XSS vector *after* sanitization for every single one of a large number of XSS vectors. The vulnerabilities described by Kolbitsch et al. may form the basis for an extremely targeted attack by web malware [24]. Those authors state that the attack vectors may be prepared for taking into account the mutation behavior of different browser engines. HTML5 introduces a script-like functionality in its different tags, making the so called "Scriptless Attacks" (a term coined in [25]) a real threat. For example, SVG images and their active elements can be used to steal passwords even if JavaScript is deactivated [26].

6 Conclusion and Outlook

Given the current trends in web application and app design, a client-side, DOM-based XSS filtering solution is urgently needed. This concept faces different threats when one compares it with server-side XSS filters. Those are unique to the browser's DOM and need to be discussed in depth. We present DOMPurify, an open-source library designed to reliably filter HTML strings and document objects from XSS attacks. It seeks to allow developers to safely use user-controlled and untrusted HTML in web applications, mobile apps and any other deployment that requires employing a browser(-like) DOM. Its filtering techniques to mitigate DOM Clobbering attacks can form the basis for a framework

to include any XSS mitigation technique into the DOM. DOMPurify accompanies and complements CSP 3.0, and closes the gaps that are not covered by the browser itself. Our DOMPurify library implements the current state of the art knowledge in the field of XSS defense. It draws on novel concepts which work surprisingly well in practice. However, as new threats are constantly emerging, the extendibility and configurability of DOMPurify is crucially important. It goes without saying that we encourage future research in this direction. Additional use cases for a security library that resides in the DOM may arise and can now be faced head on in their core rather than in an unrelated and distant layer on the web-server.

Acknowledgements. The research was supported by the German Ministry of research and Education (BMBF) as part of the OpenC3S research project.

A Deployment

Other Deployment Scenarios

JavaScript MVC (model-view-controller) frameworks are written to move application logic like view-generation, templating and site-interactivity from the server to the client. Example frameworks include AngularJS, EmberJS, KnockoutJS, React and others. They are often maintained by large corporations such as Google, Facebook or Yahoo. Using the frameworks properly requires a change in application design philosophy from the developers' side. While the server generated and delivered the HTML for classic applications, here only a minimal scaffold of HTML is server-generated and delivered. The majority of content is being created in the client and is based on raw JSON from the server containing the data, using static template files, as well as a complex event and widget logic residing almost entirely in the browsers, fuelled by the JavaScript MVC framework. That of course obsoletes server-side HTML and XSS filters as the server doesn't deliver any user-controlled HTML anymore. Now the JavaScript MVC framework must take care of that issue, and this is the point where DOMPurify can be applied as an additional level of mitigation. Note though that DOMPurify is also capable to run on *nodejs* in combination with *jsdom* – therefore it can also protect server-side web-frameworks and template-engines from XSS attacks. The automated tests running for every commit cover this deployment scenario in full.

Actual Deployment

The DOMPurify library is currently downloaded about 52000 times per month on the "npm" JavaScript package manager platform. It is being used by major web mail providers and several commonly known tools used in the context of web-mail end-to-end encryption. DOMPurify is further being utilized by browser extensions who need to sanitize user controlled HTML, giving developers a more fine grained control over what kind of rich text is supposed to be rendered and displayed – beyond what CSP is offering.

References

1. Johns, M.: Code injection vulnerabilities in web applications - exemplified at cross-site scripting. Ph.D. dissertation, University of Passau, Passau, July 2009
2. Heiderich, M., Frosch, T., Jensen, M., Holz, T.: Crouching tiger - hidden payload: security risks of scalable vector graphics. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 239–250. ACM (2011)
3. Heiderich, M., Schwenk, J., Frosch, T., Magazinius, J., Yang, E.Z.: mXSS attacks: attacking well-secured web-applications by using innerHTML mutations. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, pp. 777–788. ACM (2013)
4. Akhawe, D., Barth, A., Lam, P.E., Mitchell, J., Song, D.: Towards a formal foundation of web security. In: 23rd IEEE Computer Security Foundations Symposium (CSF) 2010, pp. 290–304. IEEE (2010)
5. Heiderich, M., Späth, C., Schwenk, J.: DOMPurify testset (2017). https://goo.gl/2g2BMz
6. Heiderich, M., Späth, C., Schwenk, J.: Output of ResembleJS (2017). https://goo.gl/9bdmZv
7. Ross, D.: IE8 security part IV: the XSS filter - IEBlog - site home - MSDN blogs (2008). http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-iv-the-xss-filter.aspx
8. Bates, D., Barth, A., Jackson, C.: Regular expressions considered harmful in client-side XSS filters. In: Proceedings of the 19th International Conference on World Wide Web, WWW 2010, pp. 91–100. ACM, New York (2010). http://doi.acm.org/10.1145/1772690.1772701
9. Zuchlinski, G.: The anatomy of cross site scripting. In: Hitchhiker's World, vol. 8, November 2003
10. Bisht, P., Venkatakrishnan, V.N.: XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In: Conference on Detection of Intrusions and Malware and Vulnerability Assessment (2008)
11. Gebre, M., Lhee, K., Hong, M.: A robust defense against content-sniffing XSS attacks. In: 2010 6th International Conference on Digital Content, Multimedia Technology and its Applications (IDC), pp. 315–320. IEEE (2010)
12. Saxena, P., Molnar, D., Livshits, B.: SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 601–614. ACM (2011)
13. Gourdin, B., Soman, C., Bojinov, H., Bursztein, E.: Toward secure embedded web interfaces. In: Proceedings of the USENIX Security Symposium (2011)
14. Gundy, M.V., Chen, H.: Noncespaces: using randomization to defeat cross-site scripting attacks. Comput. Secur. **31**(4), 612–628 (2012)
15. Nadji, Y., Saxena, P., Song, D.: Document structure integrity: a robust basis for cross-site scripting defense. In: NDSS. The Internet Society (2009)
16. Louw, M.T., Venkatakrishnan, V.N.: Blueprint: robust prevention of cross-site scripting attacks for existing browsers. In: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy, SP 2009, Washington, DC, USA, pp. 331–346. IEEE Computer Society (2009). http://dx.doi.org/10.1109/SP.2009.33
17. Weichselbaum, L., Spagnuolo, M., Lekies, S., Janc, A.: CSP is dead, long live CSP! On the insecurity of whitelists and the future of content security policy. In: Proceedings of the 23rd ACM Conference on Computer and Communications Security, Vienna, Austria (2016)

18. Weinberger, J., Saxena, P., Akhawe, D., Finifter, M., Shin, R., Song, D.: A systematic analysis of XSS sanitization in web application frameworks. In: Atluri, V., Diaz, C. (eds.) ESORICS 2011. LNCS, vol. 6879, pp. 150–171. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23822-2_9
19. Nava, E.V., Lindsay, D.: Abusing Internet Explorer 8's XSS Filters. http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf
20. Zalewski, M.: Browser Security Handbook, July 2010. http://code.google.com/p/browsersec/wiki/Main
21. Zalewski, M.: The Tangled Web: A Guide to Securing Modern Web Applications. No Starch Press (2011)
22. Bug 29278: XSSAuditor bypasses from sla.ckers.org. https://bugs.webkit.org/show_bug.cgi?id=29278
23. Hooimeijer, P., Livshits, B., Molnar, D., Saxena, P., Veanes, M.: Fast and precise sanitizer analysis with BEK. In: Proceedings of the 20th USENIX Conference on Security, SEC 2011, Berkeley, CA, USA, p. 1. USENIX Association (2011). http://dl.acm.org/citation.cfm?id=2028067.2028068
24. Kolbitsch, C., Livshits, B., Zorn, B., Seifert, C.: Rozzle: de-cloaking internet malware. In: Proceedings of IEEE Symposium on Security and Privacy (2012)
25. Heiderich, M., Niemietz, M., Schuster, F., Holz, T., Schwenk, J.: Scriptless attacks: stealing the pie without touching the sill. In: Proceedings of the 19th ACM Conference on Computer and Communications Security, pp. 760–771 (2012)
26. Stone, P.: Pixel perfect timing attacks with HTML5. http://contextis.co.uk/files/Browser_Timing_Attacks.pdf