Waiting for CSP – Securing Legacy Web Applications with JSAgents

Mario Heiderich, Marcus Niemietz(✉), and Jörg Schwenk

Horst Görtz Institute for IT-Security, Ruhr-University Bochum, Bochum, Germany
{mario.heiderich,marcus.niemietz,joerg.schwenk}@rub.de

Abstract. Markup Injection (MI) attacks, ranging from classical Cross-Site Scripting (XSS) and DOMXSS to Scriptless Attacks, pose a major threat for web applications, browser extensions, and mobile apps. To mitigate MI attacks, we propose JSAgents, a novel and flexible approach to defeat MI attacks using DOM meta-programming. Specifically, we enforce a security policy on the DOM of the browser at a place in the markup processing chain "just before" the rendering of the markup. This approach has many advantages: Obfuscation has already been removed from the markup when it enters the DOM, mXSS attack vectors are visible, and, last but not least, the (client-side) protection can be individually tailored to fit the needs of web applications.

JSAgents policies look similar to CSP policies, and indeed large parts of CSP can be implemented with JSAgents. However, there are three main differences: (1) Contrary to CSP, the source code of legacy web applications needs not be modified; instead, the policy is adapted to the application. (2) Whereas CSP can only apply one policy to a complete HTML document, JSAgents is able, through a novel cascading enforcement, to apply different policies to each element in the DOM; this property is essential in dealing with JavaScript event handlers and URIs. (3) JSAgents enables novel features like coarse-grained access control: e.g. we may block read/write access to HTML form elements for all scripts, but human users can still insert data (which may be interesting for password and PIN fields).

1 Introduction

Cross-Site Scripting. XSS attacks are one of the major threats to web application security. The goal of an attacker is to execute a (malicious) JavaScript function of his own choice in the context of the target web page. If he succeeds, the Same Origin Policy (SOP) of the browser will grant them full acess to all elements and variables of the target web page (including stored passwords, session cookies, and security tokens), and the script may trigger other potentially harmful actions (drive-by-downloads and alike).

In the literature, three main classes of XSS are described: (1) Reflected XSS, where the attack vector is sent to the target web server in a HTTP request (e.g., a search request), this input is integrated by the server into a dynamically

© Springer International Publishing Switzerland 2015
G. Pernul et al. (Eds.): ESORICS 2015, Part I, LNCS 9326, pp. 23–42, 2015.
DOI: 10.1007/978-3-319-24174-6_2

generated web page, and the attack is executed when this page is rendered. (2) Stored XSS, where the attack vector is stored in a subpage of the target web application (e.g., discussion forum), and the attack is executed each time a victim visits this subpage. (3) DOMXSS [1], where the attack vector is inserted by a (legal) client-side script into the web page. Mutation-based XSS (mXSS) is a recent new variant [2], which has the potential to circumvent several known mitigation techniques, including advanced XSS filters.

HTML5 and Scriptless Attacks. Since the advent of HTML5 (HTML, HTML 5.1 Nightly and HTML.next), new attack techniques are continuously being discovered. Different browser strategies for processing XML, XHTML[1] and HTML content which may all be mixed together can be exploited through SVG images or MathML markup [3]. Even deactivating JavaScript completely, a method repeatedly proposed by security authorities, does not protect against HTML5-based attacks as certain publications [4,5] have shown that *Scriptless Attacks* are possible. Scriptless Attacks additionally complicate the task of server- and client-side filters: It is nearly impossible to decide which HTML5 tags may be dangerous before an attack vector has been published.

Markup Injection Attacks. We will use the term *Markup Injection (MI)* to denote the superclass of attacks formed by Scriptless Attacks and the different kinds of XSS vectors. For a MI attack on a web application to be successful, all three conditions listed below must be fulfilled.

1. *Injectability.* It must be possible to inject potentially malicious markup into a web page.
2. *Executability.* It must be possible for the browser to parse and execute the markup.
3. *Extractability.* It must be possible for the attacker to exfiltrate sensitive information (e.g., session cookies) from the browser and to transfer them to a device where he can retrieve it.

Classical Defense Approaches. As a first line of defense, server- and client-side filters, which have different restrictions, are deployed. Server-side filters must be able to detect JavaScript snippets even if they are obfuscated, a task that becomes harder with every new markup functionality introduced with HTML5. Client-side filters are embedded into the browser (MSIE/ WebKit/ Blink) or can be installed as a plugin (NoScript). In both cases, they have to apply the same policy to all visited web applications, which in many cases proves too weak or too strong.

Novel Defense Approaches. Whereas classical MI countermeasures concentrate on condition (1.) injectability (by detecting and removing markup injections with client- or server-side filters), modern approaches take into account the other two conditions. For example, Content Security Policy (CSP) [6,7] and

[1] For example, the different XHTML treatment of self-closing tags.

HTTPonly cookies [8] mitigate extractablility (CSP by allowing HTTP connections only to a small number of white-listed URLs, HTTPonly cookies by making themselves inaccessible from the DOM), and sandboxed Iframes [9] try to prevent executability by restricting script execution.

Content Security Policy. CSP 1.0 is fully supported by all current webbrowser versions. Its main feature is domain whitelisting for `<script>`, `<object>`, `<style>`, `<img>`, `<media>` and `<iframe>` elements, and for fonts and websockets, to mitigate condition 3 (Extractability) for a successful MI attack. For some of these elements (e.g. scripts) this strict whitelisting policy can be relaxed by allowing inline sources through `unsafe-inline`.

CSP 1.1 has a much broader scope: The whitelisting can be applied to a broader set of DOM elements (e.g. through `form-action`), the use of inline scripts can be protected through script nonces, only whitelisted plugins will be activated, client-side XSS filters can be activated, and many more. Thus CSP can be seen as a specification where most research on web application security has been condensed; subsequently CSP is often cited as a comparison for new approaches. However, this comparison is a little weak since CSP 1.1 has to be fully implemented yet. We will nevertheless compare our approach to CSP 1.1 below.

JSAgents Library. Our JSAgents library specifically targets conditions (2.) and (3.) through DOM meta-programming. It is a client-side solution which, in contrast to the client-side XSS filters, can be tailored to a given web application. In addition, it does not have to care about code obfuscation since this has already been removed by the browser. We can restrict execution of any markup, not only JavaScript, thus mitigating XSS and in part Scriptless Attacks (Executability); we can restrict HTTP leakage for elements in the browser's DOM, and we can read-protect certain DOM elements (Extractability). Furthermore, we can write-protect DOM elements; a feature that mitigates complex cross-domain attacks (e.g., through `document.location`). We can enforce different policies for each part of the DOM tree through the use and enforcement of a *cascading* policy language (comparable to CSS).

JSAgents vs. CSP 1.1. With CSP 1.1, stronger security guarantees can be enforced, because CSP is implemented directly in each browser core, it runs with 'browser root privileges'. Through the DOM metaprogramming approach, we can never achieve more than 'page level privileges'. A different CSP 1.1 policy can be applied to each Iframe, but within each Iframe only a one-level, non-cascading security policy is applied. Through Cascading Style Sheets (CSS), web programmers are experienced in cascading style declarations. JSAgents makes use of the CSS syntax to define cascading security policies, where one, two or more iterations over the document's DOM can be made to enforce security rules. This is especially useful for legacy web applications running in a single document context (see below). JSAgents additionally allows to block read and write access

to DOM elements and their attributes, a feature that is not part of CSP 1.1 but is currently in discussion for input values on the WHATWG mailing list[2].

Architectural Overview JSAgents. JSAgents uses a static JavaScript library (`jsa.js`) and a customizable configuration file to achieve its application-specific goals: `jsa.js` must be inserted into the web page as the *first* JavaScript function to be executed. Insertion points may include the web application itself, a HTTP proxy, or a browser extension (webapp, proxy, and extension mode). As soon as `jsa.js` is executed, it uses a FrozenDOM approach to stop other active markup from being executed and reads the (cascaded) configuration file. The directives contained in this file are used to set different flags on the elements of the frozen DOM. After all flags are set, the frozen DOM is parsed and all restrictions expressed by the flags are enforced: Elements may be deleted, read- or write-protected, or their actions may be limited to white-listed URLs.

Legacy Web Applications. CSP imposes restrictions on JavaScript event handlers and JavaScript URIs (cf. Section A.2) that makes adoption of CSP nearly impossible for legacy web applications: A complete redesign of each application is necessary to be able to use CSP without the'unsafe-inline' option for scripts. With JSAgents, we can use the fact that policy files can be cascaded to sketch a generic solution for this problem: First we disallow all inline scripts, event handlers, and URIs. Then we can allow those JavaScript embeddings which are essential for a correct functionality of a web page, based on a whitelist extracted from the legacy application. Thus we can achieve the same effect as CSP 1.1 script nonces for inline scripts, but in contrast to CSP 1.1 we can extend this approach to JavaScript URIs and event handlers. We were able to deploy JSAgents successfully for two large classes of legacy applications: (a) Web-mailers and (b) Identity Providers in Single-Sign-On Systems. In both cases, JSAgents could successfully be deployed to enhance security, without affecting functionality. We are confident that, due to the flexibility of our approach, JSAgents can be deployed with nearly all legacy applications. In some cases (1 out of 13 IdPs) we have detected incompatibilities with other large JavaScript libraries, which indicates that we may not be able to achieve 100 % coverage.

Project Evaluation. We evaluated three different aspects of JSAgents: Security, usability, and performance. In Sect. 4, we describe the results of a public challenge to break JSAgents. The goal of our usability evaluation was to show that JSAgents policy files can indeed be adapted to the two classes of web applications mentioned above. During this usability evaluation, we also investigated compatibility with other popular JavaScript libraries: JSAgents is compatible with jQuery, Prototype, and Underscore, but has compatibility issues with RequireJS. Finally, we measured the performance of our solution based on randomly generated HTML code of different sizes. The results can be found in Sect. 5.

[2] Write-only Form Elements, http://mikewest.github.io/credentialmanagement/write only/.

Contributions. This paper makes the following contributions:

- *Novelty.* We give a novel, comprehensive, DOM-meta-programming-based approach to defend against MI attacks. We demonstrate the large potential of novel DOM meta-programming features like Object.defineProperty and DOM Mutation Observers.
- *Impact.* We are able to mitigate most attack classes, including mXSS and HTTP request leaks. We describe a flexible and powerful policy language such that JSAgents can be adapted to numerous (legacy) applications scenarios.
- *Usability.* In contrast to CSP, JSAgents can easily be deployed with legacy web applications, since no changes to the source code are necessary.
- *Public Availability.* We present a free open-source project from the JSAgents core that can be used as a universal client-side HTML filter ("DOMPurify" project on GitHub).

2 Related Work

From the large body of research on XSS and beyond, we provide a brief overview of the relevant literature, detailing both scholarly work and research-driven sources pertaining to this subject area.

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 on the client-side. The IE8 XSS Filter was the first fully integrated solution [10], timely followed by the Chrome XSS Auditor in 2009 [11]. For Firefox, client-side XSS filtering is implemented through the NoScript extension. Unsurprisingly, XSS attacks' mitigation strategies have been covered in numerous publications [12–17]. Noncespaces [18] use randomized XML namespace prefixes as an XSS mitigation technique, which would make detection of the injected content reliable. DSI [19] tries to achieve the same goal based on a process of clasifying HTML content into trusted and untrusted variety on the server side, subsequently changing browser parsing behavior so that the said distinction is taken into account. Blueprint [20] 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.

Mutation-Based (mXSS) and Scriptless Attacks. Weinberger et al. [21] 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. Vela Nava et al. [22] and Bates et al. [11] have shown 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* [23]. Additionally, he later

dedicates a section to backtick mutation in his volume "The Tangled Web" [24]. Other mutation-based attacks have been reported by Barth et al. [25] and Heiderich [26]. In the latter, mutation may occur *after* client-side filtering (WebKit corrected a self-closing script tag before rendering, thus activating the XSS vector) or *during* XSS filtering (XSS Auditor strips the `code` attribute value from an applet tag, thus activating a second malicious code source). Hooimeijer et al. describe the dangers associated with the sanitization of content [27] 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 [28]. 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 [4]) a real threat. For example, SVG images and their active elements can be used to steal passwords even if JavaScript is deactivated [5].

3 JSAgents Architecture

3.1 Building Blocks

FrozenDOM. The current version of JSAgents uses a technique called FrozenDOM [26,29–31]. Upon execution of the JSAgents Core Library, the DOM is stopped from being rendered as a plain-text element (`<plaintext>`) and is being written right after the `<script>` element that contains JSAgents code. Note that using plain-text is employed in the sake of supporting legacy browsers; for modern browsers, JSAgents can make use of the Shadow DOM and the `<template>` element. The interrupted rendering flow allows the library to simultaneously quickly read the document markup and prevent race conditions introduced by the injected scripts or markup. In case the application uses a JavaScript templating engine/MVC framework, JSAgents can directly work on the HTML string that built before rendering and does not need to rely on `<plaintext>`. Adopting JSAgents reduces the performance when used on complex websites (see Sect. 5) but it must be underscored that the user experience on modern browsers (like Firefox 24+, MSIE9+, and Chrome 30+) is hardly affected at all.

DOM Mutation Observers. By using DOM Mutation Observers (DMO), JSAgents is able to monitor write-access to selected DOM nodes and trigger an execution of a callback function in cases where such access has taken place. This allows us to protect form elements from being overwritten, effectively making a commonly used technique of Web Injects against online banking portals void. JSAgents can detect scripted form element manipulation because actual keyboard input into form elements does *not* cause mutation events to be emitted, while, conversely, the scripted access does so. DMO are implemented in all modern browsers and can be emulated reliably in older versions without DMO support with the use of `onpropertychange`.

Object.defineProperty(). Almost arbitrary DOM objects can be set into an immutable state by using the ES5 functionality of `Object.defineProperty()`, and thereby be protected from external, potentially malicious, manipulations. By doing so, we can assure a certain level of integrity for the JSAgents library, essentially allowing to introduce tamper safety and detectability of manipulation attempts. Further, it is possible to manage attempts of potentially malicious scripts coveting to gain a write-access to form elements.

Document.querySelectorAll(). By using the `querySelectorAll` API, JSAgents is able to select all elements that match very specific criteria from a given document. Note that the API is similar to the CSS selector API and thereby helps existing front-end developers to precisely select these elements they want to impose the security restrictions and capability control onto. Creating JSAgents policy files is comparable to composing style sheets, as selectors are identical and property-value assignments use common terminology.

3.2 JSAgents Core Library

The Core Library includes the methods that ensure safe deployment and inner workings of JSAgents. The library must be executed as early as possible in the execution flow of the protected website in order to win the basic race condition. When the core library is executed, a sequence of events enumerated and discussed below is started (cf. Fig. 1).[3]

(1) Freezing and Sealing the Original DOM. Markup rendering is stopped by a plain-text element being written into the actual document. Consequently all HTML markup after this element is considered to be simple text and is

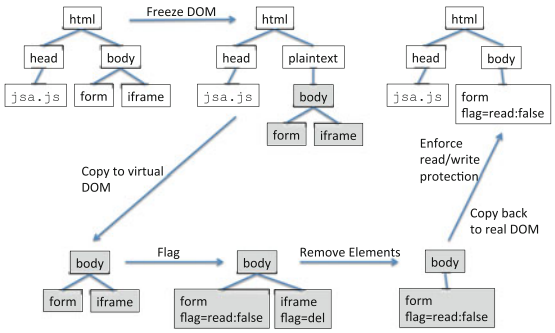

Fig. 1. Processing of a JSAgents protected HTML document.

[3] For review purposes, we have copied a password protected ZIP file of the JSAgents code to the following Dropbox URL: https://www.dropbox.com/s/17kjd8hrmjbzy6c/jsa.zip (password: *conference*).

not parsed into DOM objects. This is necessary to win possible attacker-caused race-conditions (for instance based on DOM-clobbering, see below).

```
this.freeze = function () {
    // seal existing document before freezing
    JSA.seal(document);
    // freeze and blind the whole document
    document.write('<plaintext id="' + JSA._random
      +'" style="display:none">');
    document.close();
}
```

Listing 1.1. The code to freeze a document by stopping its execution flow

By calling `this.seal()`, it is ensured that the attacker cannot tamper with the existing DOM properties that JS-Agents requires to work (for example DOM traversal, element and attribute manipulation). This technique effectively defeats "DOM clobbering"[4], a way of overwriting the native DOM methods by using HTML injections[5] in any tested browser.

```
this.seal = function (doc) {
  for(var item in doc){
    if(typeof doc[item] ==='function'){
      Object.defineProperty(
    doc, item, {value: doc[item], configurable:false}
      );}}
  return doc;}
```

Listing 1.2. Iterating over all methods to seal them reliably from external access

(2) Content Copying to a "fresh" DOM. We now extract the DOM contents and map them into the safe DOM.

```
// create JSA document to check on
JSA.doc = JSA.create();
//Copy document and assign random ID values
JSA.doc.documentElement.innerHTML ='<html><head>' +
document.getElementById(JSA._random).textContent;
```

Listing 1.3. Creating a virtual DOM and assigning random IDs to each element

(3) Enforcing the given Policies. This is done by iterating over the virtual DOM tree we have created. The enforcer first requests all elements matching the JSAgents policy selectors and, upon receiving one or more elements, passes them to the protected *_enforce()* method. This method is being provided by the JSAgents enforcer module and compares the policy defined capabilities with the actual object's capabilities, eventually taking action in case of any mismatches

[4] DOM Clobbering describes malicious declarative DOM manipulation: http://www.thespanner.co.uk/2013/05/16/dom-clobbering/.

[5] For example, the HTML `<img src=x name=getElementById>` would overwrite the method `document.getElementById()` with an image-object.

being identified. The final goal is the removal of either the attributes or the specific attribute values, the prefixing of resource URIs or even the removal of entire elements and there-attached child nodes. However, to make sure the policy directives and their selectors cascade properly (selector precision over selector order), the enforcer initially only flags elements for deletion or manipulation. Only after the final selector's rules have been enforced, the elements are actually removed or modified (see below). By design, the enforcer is being defined and kept as "a module" because adding it to the core library would cause unnecessary overhead – we acknowledge that possible forks and adaptations of JSAgents might prefer building their own enforcers and keep the core library untouched. Using it as a module allows an easy extension and customization; one imaginable scenario would be that a more exotic or legacy browser needs to be supported.

(4) Remove Elements Flagged as `del`, Domain White-Listing. After all necessary enforcement iterations are completed, the final state of element and attribute flagging is being used as an indicator of whether the element or attribute should be removed or kept in place. Elements are removed if they are flagged as `del`, or if their source domain does not match any of the whitelisted domains.

```
this.filter = function () {
// remove elements with kill-switch
var elements = JSA.doc.querySelectorAll('*');
for(var index in elements){
if(elements[index].tagName) {
    if(elements[index].getAttribute(JSA._random) ==='del') {
        elements[index].parentNode.removeChild(elements[index
            ])
}}}}
```

Listing 1.4. Deleting marked elements

Note that during the flagging and enforcement no other script in the protected website can be executed[6]. Furthermore, beware that the deletion flags are applied with a token value to make sure that an attacker cannot inject those attributes and force legitimate elements to be deleted. The token changes every time JSAgents runs.

(5) Rendering the Document. Read- or write-access restrictions must be enforced in the "real" DOM. This signifies that the flagged virtual DOM with forbidden elements already removed, is now copied back – as shown in the simplified example: `document.body.innerHTML = JSA.doc.body.innerHTML` (script content is, if permitted, reactivated separately). During subsequent parsing, restrictions on the parsed elements imposed by the attached flags will be enforced.

[6] Only if an external window opened from the same origin injects code, a malicious script may run concurrently to JSAgents. This may happen if only parts of the website are protected with the JSAgents library. Partial library usage is considered a dangerous implementation misbehavior and leads to race conditions that allow policy bypasses.

The core library uses DOM Mutation Observers to get on-time notification on changes happening to the write-protected DOM elements. Only by monitoring access and changes to the existing elements with working observers, a continuous level of write-protection can be guaranteed. All DOM objects flagged with access restrictions are protected – all their property getters are set to return `null`.

```
var access = document.querySelectorAll('*['+_random+'access
    ]')
for(var elm in access) {
    if(access[elm].tagName) {
        for(var i in access[elm]){
            // null all properties of the protected element
            Object.defineProperty(access[elm], i, {value:
                null});
}}}
```

Listing 1.5. Read access protection for DOM elements

It is possible to allow access for certain trusted DOM methods if required. By default, however, all property access is prohibited. Once write-access to a DOM element with `write-access:false` is communicated to the JSAgents core function, a wide range of actions becomes available. Depending on the JSAgents policy, the library can block write-access (through restoring the element to its original state), return empty strings upon read-access, and even report attempts of read or write to protected elements.

```
// freeze flagged elements
var freeze = document.querySelectorAll('*['+_random+'freeze
    ]')
for(var elm in freeze) {
    if(freeze[elm].tagName) {
    var observer = new MutationObserver(function(mutations) {
        mutations.forEach(function(mutation) {
        alert('form tamper detected');
        });
    });
    var config = {
        attributes: true,
        childList: true,
        characterData: true
    };
    observer.observe(freeze[elm], config);
    Object.defineProperty(freeze[elm],'value', {set: function
        (){
        return alert('form value tamper detected');
    }});}}
```

Listing 1.6. Code to handle write-access control to elements

From this point forward none of the elements and attributes that are violating the JSAgents policies are present. Please note that changing element properties

through the keyboard is not registered as a mutation event by the DOM Mutation Observers, as opposed to the write-access by a script. Thus, for example, for write-protected form elements, user input from keyboard is not considered to be write-access, thus laying the foundation for basic access-control functionality.

3.3 JSAgents Modules

The JSAgents library further ships a set of modules that provide functionality not yet available in modern browsers.

(1) A JavaScript implementation of the MD5 hashing algorithm `md5.js` is being loaded via module; MD5 is being used despite security concerns for performance reasons. The library allows to upgrade to SHA1 and later releases are planned to be shipped with a JavaScript implementation of SHA256. It is important to note that the use cases for hashing algorithms in the JSAgents library do not depend on collision resistance.
(2) An `enforcer.js` script is used to enforce various JSAgents policies and iterate over the target elements. It also imposes the restrictions or permissions the developer wishes to enforce and grant (as discussed in Sect. 3.2). Furthermore, an extended enforcer allows creation of additional rules – complementary to the already available rules and policies. The enforcer is again not considered a part of the core library because it might be subject to customizations, for instance for a website that uses a specific JavaScript framework.

3.4 JSAgents Policy Files

JSAgents policy files are composed in a JSON format and make use of a very simple and intuitive dictionary of instructions. This allows even novice developers to understand the concept and impact of the policy files rather quickly. Note that the dictionary of available configuration directives might be subject to change as it is now in its prototypic state. The code shown in Listing 1.7 and Listing 1.8 demonstrates the flexibility of JSAgents policy composition. The syntax is designed to closely resemble CSS selectors.

Listing 1.7 makes use of the asterisk-selector which causes the JSAgents engine to indeed choose all available DOM elements on the loaded document and impose the following (very restrictive) directives. As dictated by this policy, no Script, Iframe, Object, Embed, Applet, or SVG elements will be present in the modified DOM. JavaScript and data URIs will be removed from the DOM and so will be the event handlers. All remaining elements will be frozen, for example they cannot be modified by the DOM meta-programming from now on. Read-access to all remaining elements is blocked.

The code shown in Listing 1.8 is a bit more permissive, and demonstrates the "cascading" features of the policy language. Here we can observe an overall of four selectors: the asterisk selector, a selector for head-elements, a selector for form elements and their expected descendants, and, finally, a selector for the

```
{ "*" : {
    "iframe-elements": false,
    "object-elements": false,
    "embed-elements" : false,
    "applet-elements": false,
    "svg-elements":    false,
    "script-elements": false,
    "javascript-uris": false,
    "data-uris":       false,
    "event-handlers":  false,
    "write-access":    false,
    "read-access":     false
}
```

Listing 1.7. A high-security policy: All forms of scripting and read/write-access to DOM elements prohibited

```
{ "*" : {
    "javascript-uris": false,
    "data-uris":       false,
    "event-handlers":  false,
    "script-elements": false,
    "style-elements":  false
  },
  "head" : {"script-elements":
    "same-domain"
  }
  "form, input, textarea" : {
    "read-access":  false,
    "write-access": false
  },
  "#widget" : {
    "script-elements": true
} }
```

Listing 1.8. A low-security policy: Scripting is permitted for scripts living in the page header and a widget container - read-/write-access to form content is prohibited

element(s) applied with the "widget" ID. Depending on the selector, different policies are assigned and will thus be enforced by the JSAgents prototype. None of the elements are permitted to contain script-elements – aside from the head-element which can comprise of script-elements as long as their source points to a same domain resource, and also the element with the "widget" ID. The selected form elements are being protected from arbitrary access. No script can have access to their value properties, all attempts to set their values via JavaScript will be blocked, a read-access will return an empty value. This is interesting for websites which wish to impose better protection for user-generated content in the form elements (account data, passwords, credit card numbers).

The grammar used for the selectors is identical to the CSS grammar and will be parsed by the DOM `document. querySelectorAll()` method for element selection. No deviations from the standard are implemented. Developers can freely use any selector string that is available and supported by the browser. Please note that although the selectors in Listing 1.8 are ordered according to generality, the ordering of selectors is not relevant for the correct functionality as JSAgents will always give preference to stronger selectors. Later versions of the library will also support detailed style directives to avoid HTTP leakage via backgrounds, list bullets, fonts, cursors, and alike. The set of the currently available directives and policies for the JSAgents prototype is described as follows:

(1) iframe-elements, **object-elements**, **embed-elements.** These policy directives can be set to `true`, `false` or a domain reg-ex. The elements can be permitted, prohibited or only be permitted if the `src` attribute

matches the given domain string. If a directive is set to `false`, all such elements will be removed from the DOM by the JSAgents core library.[7]

(2) applet-elements. This policy directive can be set to `true`, `false` or a domain reg-ex. Java applets can be permitted, prohibited, or only be permitted if the code-base or archive attribute matches the given domain string.

(3) svg-elements. This policy directive can be set to `true` or `false`. If set to false, no SVG elements can be used inside the selected nodes. This does not exclude the option of using SVG embedded via image elements or CSS. Recent browser versions have proven to be able to safely deal with SVG content – once the SVG data is being loaded as an image rather than a document.

(4) script-elements. This policy directive can be set to `true`, `false` or a domain reg-ex. Script elements can be permitted, prohibited, or only be permitted if the src attribute matches the given domain string. Note that the "same-domain" setting does not utilize a regular expression but rather an exact string matching between origin and domain part of the URL that the script element is supposed to load from.

(5) style-elements. This policy directive can be set to `true`, `false` or a domain reg-ex. Style (and link) elements can be permitted, prohibited, or only be permitted if the href attribute/ import URIs match the given domain string.

(6) img-elements. This policy directive can be used to permit or prohibit images loaded from external URLs. Especially for web-mail software, embedded images and comparable resources allow for advertisers and other parties to track and monitor reception of and reaction to a HTML mail. With prohibition of external sources, an additional layer of privacy will be added. To cope with the needs of modern web-mailers, an additional function `ask()` was added. By using this function, JSAgents leaves the decision of loading or blocking external images to the user, instrumenting a permission-dialog.

(7) javascript-uris. This policy directive can be set to `true` or `false`. Once set to `false`, none of the elements hosted by the selected element can be applied with JavaScript URIs. This holds for all attributes supporting URL strings. Note that an element using JavaScript URIs will be completely removed in case that the policy setting prohibits its existence. Several existing tools attempt to rewrite the URL to become a harmless placeholder value, JSAgents nevertheless removes the entire element for security reasons.

(8) data-uris. This policy directive can be set to `true` or `false`. Once set to `true`, none of the elements hosted by the selected element can be applied with data URIs. This holds for all attributes supporting URL strings.

[7] It should be noted that Java applets can be loaded via object element as well. Future versions of the JSAgents prototype will warn the developer in case a policy prohibits the usage of applets yet allows the arbitrary object usage.

(9) event-handlers. This policy directive can be set to `true` or `false`. If set to `false`, all event handlers will be removed from the selected elements.

(10) write-access. This policy allows setting an element to an immutable state by freezing it and prohibiting access to any of its child properties. This is particularly interesting for form elements as means of keeping external scripts and other active content from varying values, actions and other potentially sensitive data stores. This policy directive can be set to `true` or `false`. Upcoming versions will also allow to define an array of allowed setters, making sure that trusted JavaScript methods are permitted whilst untrusted methods are blocked from modifying form values.

(11) read-access. This policy allows to manage read-access to an element. If set to false, all read-access to its sensitive DOM values will be blocked. Similar to the *freeze*-policy, this directive is particularly interesting for the protection of sensitive data in form elements. The policy directive can be set to `true` or `false`. Upcoming versions will also allow to define an array of allowed setters, ensuring that trusted JavaScript methods are permitted while untrusted methods are blocked from modifying form values.

4 Security Evaluation

Since no formal methods are available to test security against MI attacks (XSS filter bypasses are nearly always found through manual inspection), several semi-formal empirical evaluations have been performed.

Generic Security Features. Unlike other filter tools, JSAgents cannot be bypassed by an attacker utilizing obfuscation, unusual character sets, compressed markup (WBXML) or even mXSS attacks. JSAgents takes the information about the markup that is to be analyzed directly from the browser's DOM. That means that even if a certain version of a given user agent has exploitable flaws that lead to broken markup being parsed into something active and executable, JSAgents can still maintain its protective functionality through analyzing the markup after the browser has processed it.

State-of-the-art Test Vectors. Two major sources of state-of-the-art XSS attack vectors, "RSnake XSS Cheat Sheet" (now maintained by OWASP, 107 unique vectors[8]) and "HTML5 Security Cheatsheet" (139 unique vectors[9]) were used for an initial hardening of JSAgents.

Public Challenge. To test the security features of the JSAgents core, a demo was made available online and announced publicly ("DOMPurify" project hosted on GitHub). Composing an arbitrary HTML string and sanitizing it from XSS and DOM clobbering attacks using our library is made possible through this demo. We received feedback from 31 researchers, with an approximate total of 13,000 attempts to break JSAgents. Only 15 bypasses based on unexpected

[8] https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet.
[9] http://html5sec.org.

browser and DOM behaviors as well as DOM clobbering were identified after that test phase and could be mitigated successfully (e.g., DOM-clobbering attacks using a form-node, two input elements applied with the name "attributes" and similar).

Empirical Security Model Based on Browser Capability Tests. The currently employed version of the JSAgents core has been hardened against XSS, DOM Clobbering, Tag Splitting, XML injections, and mXSS attacks by one the authors who is considered as an expert in this field. Further, capability tests for HTML, MathML and SVG elements to harden JSAgents have identified several formerly unknown methods of script execution. This research has resullted in several new attack vectors like using SVG and the `<animate>` element to execute JavaScript from seemingly harmless attributes such as *from*, *to* and *values*, for example leading to XSS Auditor bypasses in WebKit and Blink. Consequently, the JSAgents core was updated and now successfully mitigates these attack vectors.

Preventing Information Leakage. A test-suite was created to enumerate all currently documented ways for browsers to leak information via HTTP requests to third-party servers (images, CSS, videos, HTML manifests, proprietary MSIE features, CSS *image()*). JSAgents was then optimized to spot and later remove those data leaks. This was motivated by a need to allow a web-mailer to present HTML mails without risking data leakage and unwanted tracking. At the same time, it made it possible for the web proxies to provide better anonymity (which they lack based on the fact that proxied HTML is filtered on the server and thereby prone to attacks using obfuscation and exotic HTML features).

5 Performance Evaluation

To measure the execution time of JSAgents we created random HTML files with valid elements and attributes, and arbitrary values. Table 1 shows the performance for files with 10, 100, 608, 1,000, and 10,000 elements. For each test case, Table 1 contains the average time in milliseconds after 25 tests with the web API interface `console.time` (reconstructed for IE10). Next to `console.time`, we worked with the JavaScript profiler of Firebug for Firefox and the native profiler of Chrome to analyze the execution time of our JSAgents functions. Therefore, there are five additional measurements for each test case in FF and GC. We used the policies of Listings 1.7 and 1.8 with the following modifications: (1) For policy 1.7, we used `script-elements: "same-domain"`. (2) For policy 1.8, we omitted the `#widget` definition, since this value was not present in our sample files. Each tested browser was installed on the same virtual machine with an Intel Xeon E5-2470 processor (2,3 GHz), four assigned cores and 4 GB RAM. By reading out the measurements via the profiler, we noticed that the attribute enforcer, source extraction, and innerHTML modification need more execution time than any other parts of JSAgents. By comparing our results, the attribute enforcer is the slowest component if there are at least 37 HTML elements on the website.

Table 1. Performance evaluation in milliseconds.

Elements	IE10	IE11	FF16	FF29	GC36
Policy of Listing 1.8 with					
`script-elements: "same-domain"`					
100	21	23	45	53	25
608	83	114	182	129	77
1,000	132	181	297	208	121
10,000	1,131	1,437	3,064	1,643	1,073
Policy of Listing 1.7 without the `#widget` definition					
100	21	23	49	52	25
608	81	107	179	126	74
1,000	124	173	279	202	117
10,000	1,043	1,403	2,923	1,524	1,031

To make our measurements applicable to real life applications, we computed the average number of HTML elements of the following main pages: Google (145), YouTube (1,302), Facebook (383), Twitter (402), and Yahoo (810). This average number is 608, and for this number (cf. Table 1) JSAgents needs 117 ms to be fully executed for the modified policy of Listing 1.8 with 11 directives inside of one selector, and 114 ms for the modified policy of Listing 1.7 with eight directives inside of three selectors. The modified policy of Listing 1.7 is a little faster than the policy of Listing 1.8; the enforcer is responsible for this behavior because its execution time increases with a higher number of directives.

6 Future Work

JSAgents is a library and framework that can reliably enforce fine-grained policies on the DOM of a website or any other browser-based document. Deploying a security tool on this specific layer has many benefits and enables several novel use cases and docking points for future work and extensions.

Extensibility Through Modularity. Given that the developer can deploy a module right in the time window between the document content being fully loaded and the document being rendered, a large set of additional security and usability enhancements can be implemented. For example accessibility factors of the document can be enriched by JSAgents, since the subtitles can be automatically displayed for videos, markup can be annotated from linked content sources, visibility aspects can be adjusted by applying additional contrast or manipulating font sizes.

Enhancements of the Policy Language. Future revisions will cover policy directives capable of managing permissions to use arbitrary non-HTTP protocol handlers, a flag to enforce “SSL only” resources, and a possibility to pipeline

any existing binary resource through a configurable proxy-URL. An implementation of fine-grained DOM property access management will be offered. This is advantageous for developers who wish to use JSAgents with applications that already make use of a plenitude of JavaScript code and DOM interaction.

A Comparable Approaches

A.1 From XSS Filters to CSP 1.0

Client-side XSS Filters. JSAgents is *not* a classical XSS filter. This is due to the fact that each and every XSS filter must be able to make distinctions between user-supplied and application-supplied markup. Conversely, JSAgents only sees the combination of both (aside from common DOMXSS sources and sinks like `location.href`). However, by employing an approach inherently different from that of any common XSS filters, JSAgents can reliably mitigate several kinds of XSS and markup injection – including DOMXSS and, in part, Scriptless Attacks. If a web application uses third-party input to build some parts of the DOM tree, regardless of whether it is user-supplied, stored, or DOM-based (e.g., `document.URL`, `document.href`, `document.referrer`), it may specify a whitelist of the allowed HTML elements for that very part of the DOM tree. Any other type of element will be deleted by JSAgents. Thus, if JavaScript execution and other potentially malicious HTML5 elements are not allowed in certain parts of the website, these attacks will be blocked. An example of this approach is given in Sect. 5, where a common webmail application assumes that the Iframe containing the body of the rendered email should not contain any active markup.

HTTPonly Cookies. By setting the JSAgents directive `read-access: false` for properties such as `document.cookie`, we effectively turn any cookie into an HTTPonly cookie, so that `document.cookie` can no longer be accessed by scripts. Similar access control can be imposed on form elements to prevent malicious script from stealing its contents or sniffing keystrokes.

Sandboxed Iframes. In their default configuration, sandboxed Iframes have a virtual origin that is different from any other kind of origin. By default, they neither allow script execution nor form submission and they are not permitted to navigate the top level frame (although those restrictions can be lifted gradually). Two of these properties can easily be modeled with JSAgents: We can remove all script and form elements from a selected Iframe. Sandboxed Iframes, however, feature additional properties to gradually release the default security constraints. In its current state JSAgents is not yet able to emulate this functionality.

HTTP Leak Detection/Proxy Injection. Since JSAgents is targeting extractability, it is capable of detecting HTML elements that attempt to load external resources. Depending on the use case, leaking information via direct HTTP requests might compromise privacy promises of a web application. This especially holds for web-mailers and web proxies, where a HTTP request sent to an arbitrary URL or IP would leak user data and timing information, essentially

enabling localization and tracking. JSAgents can be instructed to change any of the existing URLs that point to external resources to be prefixed with a proxy URL. This would mean that leakage of sensitive user data is avoided.

A.2 Content Security Policy

Content Security Policy 1.0. Large parts of CSP 1.0 can be implemented and extended with the use of JSAgents. This is achieved by creating a policy that prohibits any form of inline scripting, objects, embeds and applets, while implementing a prefix for external resources (or simply blocking the use of external resources that are coming from a non-whitelisted domain). Therefore, one possible application scenario is to (partly) implement CSP in legacy browsers. However, since JSAgents is not part of the browser core, it is less resistant to higher-privilege attacks (e.g., web injections by local malware). JSAgents can be used as a CSP replacement in browsers that do not support it and as a CSP supplement in browsers with partial or full support.

```
header('X-Content-Security-Policy: script-src'self'; style-
    src 'self'; img-src'self' images.mysite.com);
```

Listing 1.9. Example CSP policy.

The code instances in Listings 1.9 and 1.10 show how an example CSP policy can be emulated using JSAgents, even if the browser itself does not support CSP.

```
{"*" : {
  script-elements: "same-domain",
  style-elements: "same-domain",
  img-elements: "same-domain", "images.mysite.com"
}}
```

Listing 1.10. CSP-emulating JSAgents policy.

CSP 1.1 Script Nonces. Introduced with CSP 1.1, script nonces are a way to permit execution of only those (inline) script elements that have a nonce

Table 2. Comparison between CSP 1.0, 1.1 and JSAgents (Y: available, m: available via module, n: not available).

Feature	CSP 1.0	CSP 1.1	JSAgents
connect-src, font-src, frame-src, img-src, media-src, object-src, script-src, style-src	Y	Y	Y
base-uri, frame-ancestors	n	Y	n
default-src, form-action, plugin-types, referrer, sandbox	n	Y	Y
reflexted-xss, report-uri	n	Y	m
Cascading Properties	n	n	Y
DOM node read-access	n	n	Y
DOM write read-access	n	n	Y
ask() function	n	n	m

attribute with a value identical to a nonce value transmitted in the HTTP header. This makes inline script injection nearly impossible. In cooperation with the web application, JSAgents, inspired by Noncespaces [18], can achieve the same goal: Inline scripts are marked with a fresh nonce value by the web application and the same nonce value is written into a copy of the configuration file. As a result of the `jsa.js` execution with this unique policy, all inline scripts unmarked with the nonce value from the configuration file will be deleted (Table 2).

References

1. Klein, A.: DOM based cross site scripting or XSS of the third kind (2005). http://www.webappsec.org/projects/articles/071105.shtml
2. Heiderich, M., Schwenk, J., Frosch, T., Magazinius, J., Yang, E.Z.: mxss attacks: attacking well-secured web-applications by using innerhtml mutations. In: CCS (2013)
3. 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)
4. Heiderich, M., Niemietz, M., Schuster, F., Holz, T., Schwenk, J.: Scriptless attacks-stealing the pie without touching the sill. In: ACM Conference on Computer and Communications Security (CCS) (2012)
5. Stone, P.: Pixel perfect timing attacks with html5. http://contextis.co.uk/files/Browser_Timing_Attacks.pdf
6. Sterne, B., Barth, A.: Content security policy 1.0," W3C, Candidate Recommendation, November 2012. http://www.w3.org/TR/2012/CR-CSP-20121115/
7. Barth, A., Veditz, D., West, M.: Content security policy 1.1, w3c editor's draft 12 November 2013. https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html
8. Barth, A.: HTTP State Management Mechanism, RFC 6265 (Proposed Standard), Internet Engineering Task Force, April 2011. http://www.ietf.org/rfc/rfc6265.txt
9. Hickson, I.: Html living standard - last updated 21 february 2014. http://www.whatwg.org/specs/web-apps/current-work/multipage/the-iframe-element.html
10. Ross, D.: IE8 XSS Filter design philosophy in-depth, April 2008. http://blogs.msdn.com/b/dross/archive/2008/07/03/ie8-xss-filter-design-philosophy-in-depth.aspx
11. 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, ser. WWW 2010, pp. 91–100. ACM, New York (2010). http://doi.acm.org/10.1145/1772690.1772701
12. Zuchlinski, G.: The anatomy of cross site scripting. Hitchhiker's World **8**, November 2003
13. Bisht, P., Venkatakrishnan, V.N.: XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In: Zamboni, D. (ed.) DIMVA 2008. LNCS, vol. 5137, pp. 23–43. Springer, Heidelberg (2008)
14. Johns, M.: Code injection vulnerabilities in web applications - exemplified at cross-site scripting. Ph.D. dissertation, University of Passau, Passau, July 2009
15. 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)

16. 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)
17. Gourdin, B., Soman, C., Bojinov, H., Bursztein, E.: Toward secure embedded web interfaces. In: Proceedings of the Usenix Security Symposium (2011)
18. Gundy, M.V., Chen, H.: Noncespaces: using randomization to defeat cross-site scripting attacks. Comput. Secur. **31**(4), 612–628 (2012)
19. Nadji, Y., Saxena, P., Song, D.: Document structure integrity: a robust basis for cross-site scripting defense. In: NDSS. The Internet Society (2009)
20. 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, ser. SP 2009, pp. 331–34. IEEE Computer Society, Washington, DC (2009). http://dx.doi.org/10.1109/SP.2009.33
21. 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)
22. Nava, E.V., Lindsay, D.: Abusing Internet Explorer 8's XSS Filters. http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf
23. Zalewski, M.: Browser Security Handbook, July 2010. http://code.google.com/p/browsersec/wiki/Main
24. Zalewski, M.: The Tangled Web: A Guide to Securing Modern Web Applications. No Starch Press, San Francisco (2011)
25. Bug 29278: XSSAuditor bypasses from sla.ckers.org. https://bugs.webkit.org/show_bug.cgi?id=29278
26. Heiderich, M.: Towards Elimination of XSS Attacks with a Trusted and Capability Controlled DOM (2012). http://www-brs.ub.ruhr-uni-bochum.de/netahtml/HSS/Diss/HeiderichMario/diss.pdf
27. 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, ser. SEC 2011, p. 1. USENIX Association, Berkeley (2011). http://dl.acm.org/citation.cfm?id=2028067.2028068
28. Kolbitsch, C., Livshits, B., Zorn, B., Seifert, C.: Rozzle: De-Cloaking internet malware. In: Proceedings IEEE Symposium on Security & Privacy (2012)
29. Nava, E.V.: ACS - active content signatures. PST_WEBZINE_0X04, no. 4, December 2006
30. Di Paola, S.: Preventing xss with data binding. http://www.wisec.it/sectou.php?id=46c5843ea4900
31. Heiderich, M., Frosch, T., Holz, T.: IceShield: detection and mitigation of malicious websites with a frozen DOM. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 281–300. Springer, Heidelberg (2011)