DOMino : Towards an Automated DOM-Based Cross-Site Scripting Defense

Anonymous Author(s)

ABSTRACT

DOM (Document Object Model) - Based Cross-Site Scripting (DOM XSS) has remained a persistent threat to the security of web application since its initial classification [29]. The search for a readily-compatible, lightweight, and developer-friendly defense against DOM XSS is motivated by the attack's prevalence and the potential for DOM XSS to become more common as websites become heavier on the client-side.

Based on our understanding of current website implementations, we design and implement DOMino, a DOM XSS defense which leverages Content Security Policy (CSP) for script blocking and solves CSP's main adoption roadblocks for top sites, mainly incompatibility with inline and dynamically loaded scripts. We also contribute DOMino++, a more robust version of the defense that takes advantage of an additional CSP source directive (disable-dynamic) from a modified Chromium browser to prevent DOM XSS previous to the DOMContentLoaded event. With large-scale evaluation on top Alexa sites, we demonstrate that DOMino has negligible performance overhead (near 1% on average) and is already compatible with current practices on 80% of the Alexa Top 200 websites with little to no code changes, including those using popular "single page" JavaScript applications. We integrate DOMino within a deployment of DOMinoEdit, an open sourced latex editor developed by the team to test compatibility of DOMino within a deployed website undergoing active development, demonstrating how DOMino can be used within existing website infrastructure to mitigate DOM XSS against insecure usage of DOM APIs (Application Programming Interface). Finally, we build a formal web security model in Alloy to confirm the effectiveness of DOMino's security properties, used to identify and repair a loophole in our implementation. We believe DOMino is a compatible, light-weight, and secure defense against DOM XSS.

CCS CONCEPTS

• **Security and privacy → Web application security**; **Browser security**;

KEYWORDS

DOM XSS, Content Security Policy, Chromium

1 INTRODUCTION

Cross-Site Scripting (XSS) has arguably been *the* most prominent threat to Web application security, as exemplified by its continued presence on vulnerability watch lists [35] for a decade or more. Recent Web development trends toward applications heavy in client-side JavaScript code; this steadily increased the attack surface for Document Object Model (DOM)-based XSS [36], also known as type-0 XSS, in which malicious scripts are executed on a user browser via DOM functionality without involving a victim server. We believe that DOM XSS is a unique and serious Web security threat that requires specifically designed defenses, for several reasons:

- DOM XSS vulnerabilities are widespread, this problem is aggravated by the majority of DOM XSS vulnerabilities which are a result of common domains used for advertisements and analytics that become integrated within a majority of the world's most visited websites [31].
- XSS injection via the DOM bypasses all server-based XSS defenses, even state-of-the-art techniques such as contextual auto-escaping [18, 43].
- Vulnerabilities in the JavaScript DOM are nearly impossible for common commercial security tools such as black-box vulnerability scanners to detect [14].
- The JavaScript content of modern web pages can vary greatly across time [31] and potentially from user to user. Thus any browser-based defense must handle dynamically changing script content without much performance overhead while maintaining compatibility, since many of these scripts are advertisements that drive the Web economy.
- DOM XSS defenses should account for the possibility of Persistent Client-Side XSS vectors, some of which may leverage the browser's local storage to pass tainted information to sensitive sinks for execution of DOM XSS attacks while evading Server-Side detection. [45]

In this paper, we contribute DOMino, a browser-based defense against DOM XSS requiring no server changes. DOMino leverages the script blocking features of Content Security Policy (CSP) to provide automatic, compatible, and low-overhead protection against XSS, especially DOM based XSS. We design this system such that the defense uses script nonces, introduced as early as CSP 1.1. It is meant to be compatible with current web development practices as surveyed in recent works despite inline scripts and dynamically created scripts [49] . We show that our implementation of DOMino causes negligible page-load overhead on the order of milliseconds(less than 1% overhead on average) and is compatible with legitimate inline and dynamically-loaded scripts, a great enhancement over out-of-the-box CSP-based defenses which would reject these scripts. Our survey over the Top 200 and Top 1000 Alexa Global sites finds that DOMino is fully compatible with 88% and 62% of those sites respectively. We include DOMino within a deployment of DOMinoEdit, an open sourced, real-time collaborative latex editor developed to show DOMino's capacity to rectify insecure usage of DOM-manipulating API's and prevent DOM XSS, another contribution of this work. Finally, we verify the efficacy of the DOMino script injection blocking by extending an existing web security model built in Alloy [11, 16] to model DOM script loading and execution (a contribution in its own right), in the process uncovering and repairing a novel script injection vector which evaded our initial implementation. Furthermore, our design of DOMino is

intended to be as applicable as possible to browsers that implement script nonces.

Because of its compatibility, low-overhead, and security, DOMino is a timely Web security contribution and eliminates some common roadblocks towards broad CSP adoption.

The alignment of the most popular browsers on the browser implementation of DOM APIs suggests the new, Chromium specific feature addition (the 'disable-dynamic' CSP source directive) leveraged by DOMino++ is compatible with WebKit based browser engines (Safari), other Chromium based browser engines, and FireFox after an investigation of Gecko's source code and open source developer communications for a potential browser coverage as high as 85%. [3].

This paper is organized as follows. In Section 2, we survey current DOM XSS defenses and explain why state-of-the-art XSS defenses are unsuited to prevent DOM XSS for most Web users. Section 3 proposes the requirements of an effective DOM XSS defense and evaluates unmodified Content Security Policy (CSP) as a straw man DOM XSS defense. Section 4 discusses the compatibility flaws of the CSP defense, motivating the design of DOMino. Section 5 describes in detail the implementation of DOMino in JavaScript and Chromium, and Section 6 evaluates the security, performance, compatibility, and ease of deployment of DOMino with formal security model-checking and real-world experiments. Section 7 concludes.

Ultimately, we will open source DOMino, DOMino++, and DOMinoEdit such that we can facilitate a systematic means of developing a DOM XSS defense that is applicable as possible to modern websites and the their respective development practices.

2 RELATED WORK

We next survey techniques for XSS mitigation in light of their suitability for stopping DOM-Based XSS. The first class of these techniques primarily seeks to *detect* XSS in a security review, so that the existing vulnerability can be fixed in the Web application code. DOMino is fundamentally different from these techniques in that it make no attempt to flag vulnerable code, but rather protects vulnerable code from being exploited on the browser at runtime. These **XSS Detection Techniques** include

(1) General-purpose black box web application vulnerability scanners [1, 10, 40]. Most of these tools are structured around the network-layer, and previous studies have found these tools lacking in JavaScript interpretation capabilities [14], making them unlikely to detect DOM XSS.
(2) Black-box tools specifically tailored towards domxss, such as DOMinator [33] While these tools are effective in finding vulnerabilities for the developer, they do not protect the runtime user from domxss, which is the goal of DOMino.
(3) JavaScript flow-tracking and symbolic execution frameworks [27, 30, 34, 38, 44, 46, 50] that can be used or adapted to analyze code to find flows between untrusted (attacker-provided) sources and vulnerable sinks (such as `innerHTML` or `document.write`). However, the analysis usually involved creates performance penalties (for example, at least 73% slower page loads for [27], an order of magnitude for [44]) that currently prevent these techniques from being used for runtime user protection.
(4) Automated web crawling combined with dynamic taint tracing using an instrumented Chromium browser [31] has shown success in detecting domxss. This kind of detection helps to provide a better understanding of the DOM XSS attack surface but would be unwieldy for the average web developer to use in attempting to secure a specific website. Even more recent research has expanded upon data collected from an instrumented Chromium browser to facilitate information flow-tracking, these JavaScript functions have been used to develop and train DNN's to determine if a given javascript function is vulnerable to DOM XSS [32] while also mitigating the performance hit shouldered from JavaScript flow-tracking. Similarly, crawling and instrumenting a website to expose vulnerable information flows to craft payloads and evaluate DOM XSS vectors has shown promise in exposing vulnerabilities but does not provide a defense for DOM XSS at runtime [39].

We note that, since their main audience is the developer, none of these detection techniques are in conflict with DOMino, which can be used in tandem with any of them to provide greater end-to-end security for the site-user.

DOMino fits in the next class of XSS mitigation techniques, focusing on **Systematic XSS Escaping or Blocking** to protect end-users of Web applications. These techniques include:

(1) Uniform and Contextual auto-escaping techniques. Examples of Uniform auto-escaping techniques are those in the Django template system [6]. Contextual auto-escaping techniques are those of XHP for Facebook [18] and CSAS [43] for Google. These analyze user or database inputs on the server and attempt to convert them to escaped forms suitable for their usage context in the browser. However, because they rely on server-side functionality, they cannot mitigate domxss, where the attack vector never reaches a server.
(2) Caja [21], FBJS [17], AdSafe [4], TreeHouse [25], Akhawe et. al. [12] and other language or browser-based code sandboxes. These work well for sandboxing untrusted scripts from trusted JavaScript content, often at some compatibility cost of source code rewriting, wrapping, and filtering or performance overhead due to ferrying data to sandboxed environments. They also do not protect improperly written trusted JavaScript code from attack through DOM inputs such as document.location.hash.
(3) XSS filters built into browsers, such Chrome's XSS Auditor [13] These prevent scripts from executing if they are detected as part of an attack attempt – e.g. being present in the request. However, defenses like XSS Auditor do not stop all types of DOM XSS – for example if the XSS vector is carried in multiple different query parameters, XSS Auditor cannot detect and prevent the attack attempt. Other recent Client-Side filters proposed for Firefox handle potentially malicious inputs from input data in GET- and POST-formats but struggles with attack vectors such as 'svg' and 'object' tags [48]. In contrast to these examples, DOMino,

which operates as fully-formed scripts are being installed into the DOM, prevents the example attacks. We again note, however, that XSS Auditor is not in conflict with DOMino. Both can be used in tandem as defense-in-depth.

(4) Browser-enforced security policies, such as Alhambra [47] and Content Security Policy [2]. These allows Web application authors to declare a policy of allowed behaviors in order to block unwanted script execution:
   (a) Alhambra prevents browser Javascript engine from executing tainted input from untrusted inputs such as document.location and also enforces document structure integrity (DSI): preventing the DOM from deviating significantly from previous visits.
   (b) CSP blocks scripts whose provenance does not match a manifest declared by the application author.

   These policies present an effective defense and can have negligible performance impact (although enforcing DSI in Alhambra may cause 100% overhead and requires specially modified browsers). Indeed, DOMino leverages CSP, which is on the W3C standardization path, to enforce its script blocking. However, adopting CSP is still a non-trivial task for a site (as acknowledged in the spec itself), and using CSP naïvely can lead to compatibility problems with inline and dynamically loaded scripts, as we examine in Sec 4.1. As we will then describe in Sec. 4.2, this motivates the design of DOMino to achieve compatibility with inline and dynamically loaded scripts by automatically adapting a site's CSP. Recent research surveys also examine the compatibility problems of deploying CSP with nonce and cryptographic driven defense to existing websites [49], their efforts yield a CSP source-directive called 'strict-dynamic' which attempts to propagate trust to dynamically added scripts created via document.createElement(). This defense is tightly coupled to each browser as it relies on each underlying browser to implement the source-directive, while DOMino's JS-based approach to propagating a nonce to <script> elements created via the document.createElement() function should be browser independent which exemplifies the challenge of developing a comprehensive defense (Safari still has yet to implement the "strict-dynamic" source directive) [37]. Other research examines the extent to which popular browser extensions manipulate web pages in-the-wild and how website CSP's can be altered by these extensions [23]. Their contribution of an endorsement mechanism for allowing browser extensions and webservers to alter CSP on the fly further exemplifies the need for more readily compatible standards for CSP usage across the web.

(5) Proxy-based protection. Golubovicet al. propose a reverse HTTP proxy to generate CSP policies automatically and externalize inline script [20]. The good thing about the proxy-based method is that it requires little or no developer efforts. However, the proxy-based approaches have a fundamental security shortcoming that they cannot provide any protection when they are in the "learning mode" to generate CSP policies. Unfortunately, the proxy-based approaches have to switch to "learning mode" frequently to be compatible with the websites since the websites change frequently.

(6) XSS filters via HTML sanitizers such as DOMPurify that can operate on the client-side[24]. DOMPurify uses element and attribute allow-lists to determine if sample markup text should be modified to remove potential DOM XSS by DOMPurify's sanitize function. This approach appears to provide a capable means of defending against DOM XSS will require modifications to a web page's source code or configurations that don't come-out-of-the-box. Ultimately, it is the developer's onus to locate sensitive sink functions and understand what inputs must be sanitized such that DOMPurify is able to work at its full capacity. Our team aims to create a defense that is, for the most part, automated so as to eliminate improper developer usage of an XSS defense as a potential XSS vector. Moreover, a goal of our defense is to maintain compatibility with DOMPurify and the aforementioned XSS defenses so as to maximize the flexibility of a developer to use our defense in tandem with current state-of-the-art defenses.

3 DESIGN FOR EFFECTIVE DOMXSS DEFENSE

3.1 Developer and Attacker Model

We will first clarify the developer model in which our system operates, which is also summarized in Table 1. The web application codebase which we defend is developed by "security aware" programmers who follow general best-practices but may not be security experts. As such, they freely use any Javascript and DOM methods except those which convert strings to code, i.e. `eval` and its kindred. Also, they may statically and dynamically create any non-script DOM content without limitation. Exploitable input escaping vulnerabilities may exist in any method used to create non-script content (though this paper focuses on vulnerabilities in dynamic methods), allowing the attacker to change the content to scripts or to add scripts. The developers may also load scripts statically using external or inline `<script>` tags and inline handlers and dynamically by creating and attaching DOM nodes using `document.createElement`. The uses of `document.write` and `javascript:` URIs to create legitimate dynamic scripts are ruled out, however, which will be addressed in our compatibility discussion. Creating script nodes using `document.createElement` *is* allowed. Furthermore, as developers have clearly signaled the intention to introduce code by using `document.createElement` to create script nodes, we leave them to their devices and do not further audit this mechanism.

Our adversary is a Web attacker who tries to exploit Web application developer mistakes and user naïveté to execute malicious JavaScript on a victim page, to the detriment of the page user. The attacker may cause the user browser to make HTTP requests to an URL of the attacker's choice (such as through a disguised shortened URL), and to load the fetched page as a `document` in a browser window or iframe with the attacker's chosen URL. We thus assume that all user-controllable inputs, including browser-layer inputs

such as `document.location` and `referrer`, are controllable by the attacker.

API	Content Injection	
	non-script	script
`Static HTML source`	✓v	✓
`eval`, etc.	N/A	✘
`document.createElement`	✓v	✓
`innerHTML/outerHTML`	✓v	✘
`document.write`	✓v	✘
`javascript: URI`	N/A	✘

Table 1: Content Creation API Model. A ✓ indicates that the developer can directly use the API, and a ✘ indicates they are assumed not to. ✓v indicates that defensible vulnerabilities are assumed to exist in the developers' usage of the API.

3.2 Design Criteria

In the manner of existing commercial XSS defenses like Chrome's XSSAuditor [13], we feel that a suitably deployable DOM XSS defense will protect against vulnerabilities caused by common developer errors, as we proposed in the previous section. Compatibility and performance should be balanced with the security provided by a DOM XSS defense. Therefore, our runtime defense against DOM XSSS has the following properties:

- **Compatible**. We do not want to burden Web application developers and operators with major changes to their existing code and operational infrastructure. The defense must thus not create compatibility issues with existing static or dynamic scripts in the page, nor disrupt existing site operation and monetization.
- **Low-overhead**. Because this is a runtime protection in the browser, the defense cannot cause unacceptable slowdown in JavaScript execution or page loading.
- **Good coverage**. The defense should work for all attack vectors of DOM XSS that can be plausibly controlled by a remote attacker.

3.3 Content Security Policy

We now briefly describe the history of Content Security Policy (CSP) functionality [2] in order to point out both its suitability as a basis for DOM XSS defense (mainly its low-overhead and security due to native browser support) and its adoptability and compatibility shortcomings (the admitted *raison d'être* for DOMino). We will also use this opportunity to highlight the CSP features which the DOMino implementation will leverage.

3.3.1 CSP version 1.0. As of CSP 1.0, Web application authors declare, statically in an HTTP header, a list of acceptable sources (`script-src`) for scripts allowed to execute on the application page. These sources can be specified as a web origin [9] tuple of (scheme, host, port), or with "origin-like" syntax denoting wildcard matches or specifying sub-origin paths. Furthermore, two special keywords, **`unsafe-inline`** and **`unsafe-eval`**, can be specified as sources:
`unsafe-inline`: When `script-src` or `default-src` is specified, CSP disables all inline scripts from executing. (That is, scripts must all have an external `src` attribute in order to execute.) Including `unsafe-inline` in the list of script sources allows inline scripts without an external `src` attribute to execute.
`unsafe-eval`: When `script-src` or `default-src` is specified, CSP disables the conversion of a data string into executable JavaScript code via `eval`, `Function`, and their kindred. Including `unsafe-eval` in the list of script sources allows this type of conversion.

3.3.2 CSP version 1.1. CSP version 1.1 introduced an experimental directive, **`script-nonce`**, in addition to `script-src`. This directive specified a predetermined random nonce in the policy, and `<script>` tags on the page must contain an attribute `nonce` with value matching the policy-specified nonce in order to execute. This directive also has the specified side-effect of disabling all `javascript:` URIs and inline event handlers specified as HTML attributes. The **`script-nonce`** is now declared as a **`nonce-[nonce]`** within script-src which has seen continued support in the most recent version of CSP, CSP Level 3.

3.3.3 Strawman DOM XSS Defense using a CSP configuration without CSP 1.1's introduced features. CSP already meets a number of our design criteria, it is supported by popular browsers (Safari, Chrome, Safari mobile, and Firefox and on). Its browser-native implementation for blocking undesirable scripts based on their source performs with acceptable overhead. Also, its integration with the browser's JavaScript loading process delegates site security to the browser, making any CSP bypass a high-profile browser vulnerability which vendors are very motivated to fix. Web applications, especially new ones designed from scratch, can implement a secure and low-overhead defense against DOM XSS (and all XSS) using a header-specified version of Content Security Policy.

Recent surveys of top commercial Web applications which arguably have the most resources and incentive to adopt CSP, have found that CSP defenses pose significant compatibility and adoption hurdles due mainly to the prevalence of dynamic and inline scripts [49]. This research into roadblocks to adoption of CSP brings to light the challenge of integrating CSP within a website due to incompatibility with web developer practices. The new features that were introduced as of CSP 1.1 and received continuous support to the current CSP Level 3 have the opportunity to alleviate some compatibility headaches introduced by dynamic and inline scripts; We will further examine the related work as motivation for a readily-compatible CSP-based defense and subsequently present our solution to aforementioned issues of proper CSP adoption–DOMino.

4 DOMINO DESIGN

4.1 Examining Related Works about Issues with CSP DOM XSS Defenses

Related works have found that implementation of a CSP-based XSS defense on leading contemporary commercial Web applications faces major roadblocks [49], the two of greatest interest to the team are the following:

Inline `<script>`s: To ensure security in a CSP XSS defense, the site owner must *not* set the `unsafe-inline` directive, or else escaping or filtering mistakes could lead to the injection of HTML

<script> tags or inline event handlers (say through document.write). However, inline <script>s are valuable to application developers for site performance (as pointed out by developers [19, 22]), since they can reduce the number of connections and network round-trips.

Dynamic <script>s: Script-source declarations under CSP must be made in the HTTP header prior to any page loading. While some support for wildcard matching exists, by and large the policy author must know a-priori the acceptable JavaScript content of the site. While this requirement makes sense for JavaScript libraries, it presents a compatibility challenge for another class of JavaScript content – advertising. Because of dynamic targeting, even the most reputable advertising exchanges (e.g. Google ads network) can dynamically load JavaScript content from an origin unanticipated by the site owner (publisher).

These two roadblocks indicate a-priori declaration of script-source origins can lead to incompatibility with modern websites that intend to load scripts dynamically, often from more than a single origin.

4.2 DOMino Design: Improving Compatibility

The prior subsection details how inline and dynamically loaded scripts are the primary stumbling blocks to the adoption of CSP as an XSS defense. We now describe two DOM XSS defenses that leverages some features introduced in CSP 1.1: one which maximizes compatibility (**DOMino**) and one that maximizes security (**DOMino++**). As with CSP, both of our defense are opt-in and pose no compatibility issues if not used. While CSP is designed to defend against all types of XSS, our solutions focus on stopping DOM XSS, since most of the existing XSS defenses (as per our own surveys, the surveys of recent research into CSP-based XSS defenses [49], and other related works in section 2) are effective against reflected and stored XSS, thus allowing our designs to trade some security features of CSP for enhancements in compatibility with existing Web sites.

DOMino is our DOM XSS defense that maximizes compatibility. Page loading under DOMino works as follows:

(1) No CSP policy is initially specified, thus *all* scripts are allowed to load and execute until the DOMContentLoaded event, which signals that the browser parser has finished processing the entire static page source, including static scripts.
(2) Upon the DOMContentLoaded event, create a page CSP using a <meta> tag including a script nonce directive with a random nonce to begin blocking javascript: URIs and inline event handlers. Use wildcards in script-src to allow dynamic loading of all external <script>s, so long as the <script>s have the proper nonce.
(3) Convert static inline event handlers to external scripts, see Figure 2.
(4) Modify the dynamic script creation function document. createElement to automatically add the chosen nonce to <script> tags it creates, allowing them to execute. In effect, this allows trusted scripts to propagate their trust to JavaScript libraries or code that they wish to load.

This defense admittedly does not stop scripts injected via DOM functions such as document.write before the DOMContentLoaded event. However, many popular and emerging web applications are "single page", relying solely on client-side JavaScript and AJAX to implement all functionality. For these web sites, the vast majority of dynamic content insertion using DOM methods occur after the DOMContentLoaded event. A recent survey of websites indicates support for dynamic content insertion after the DOMContentLoaded event is key to improved compatibility and adoption of CSP-based defenses [49]. Thus, the class of vulnerabilities that DOMino can prevent represents a significant share of DOM XSS vulnerabilities. We will address this limitation more in Section 4.3.

Further, for sites that *do* want to protect themselves from possible DOM-based script injection before the DOMContentLoaded event, we propose a possible extension to DOMino, called **DOMino++**. DOMino++ uses an addition CSP directive called **disable-dynamic**. If this directive is enabled, the browser will disable script execution for content dynamically added to the DOM. Page loading under DOMino++ will now work as follows:

(1) Specify an initial CSP with unsafe-inline to allow all static inline <script>s to be loaded and wildcards to allow all static external <script>s to be loaded, but with disable-dynamic to disallow dynamic <script> loading.
(2) After the DOMContentloaded event fires, Add a second meta tag that includes a script nonce, in effect enabling the loading of dynamic <script>s that match the required nonce and blocking javascript: URIs and inline event handlers.
(3) (Same as before) Convert inline event handlers to external scripts, adding the chosen nonce to allow these to execute, see Figure 2.
(4) (Same as before) Modify the dynamic script creation function document.createElement to automatically add the chosen nonce to <script> tags it creates, allowing them to execute. In effect, this allows trusted scripts to propagate their trust to JavaScript libraries or code that they wish to load.

4.3 DOMino Limitations

As we presented in the previous section, both DOMino and DOMino++ rely on CSP to provide DOM XSS protection. Because CSP is enabled only after the page is parsed (i.e., after the DOMContentLoaded event), DOMino cannot block DOM-based XSS attacks that happen before this stage without the proposed disable-dynamic directive. Table 2 depicts the types of scripts blocked by our defense. DOMino by itself indeed does not fully mitigate all DOM-based XSS attacks. However, we believe this to be an acceptable limitation for two reasons. First, we believe that the usability of DOMino clearly merit its use as defense-in-depth, outweighing its drawbacks. DOMino is lightweight by design, consisting of a single JavaScript library that can be incorporated easily into existing defenses (such as those mentioned in Section 2) to provide more thorough vulnerability coverage. Second, we believe a light-weight and easy-to-adopt defense with limited vulnerability protection can be of greater benefit than a defense that offers greater vulnerability protection but also

Relative to DOMContentLoaded	Injection method	DOMino	DOMino++
Before	HTML source	allowed	allowed
	document.createElement	allowed	disallowed
	other JavaScript methods	allowed	disallowed
After	document.createElement	allowed	allowed
	other JavaScript methods	disallowed	disallowed

Table 2: Types of scripts allowed by DOMino and DOMino++.

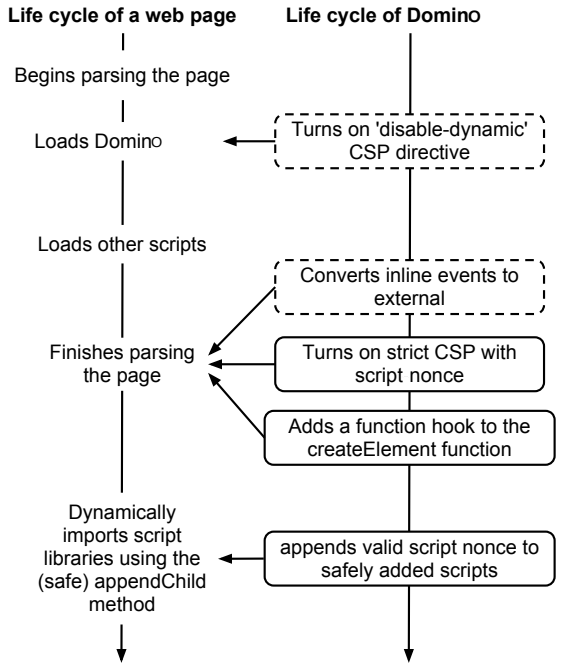


Figure 1: The life cycle of DOMino. The left column represents a canonical life cycle of web page rendered by the browser. The right column illustrates all the operations of DOMino and at what point of the web page's life cycle they are performed. Each solid box represents a mandatory operation of DOMino while a dashed box represents an optional operation.

greater difficulty in adoption, as many successfully deployed XSS defenses fall into the former category [13].

5 IMPLEMENTATION

We implemented DOMino as a JavaScript library that loads with the web page, and DOMino++ as an additional browser component that has the ability to disable dynamically injected JavaScript code if the site author chooses to. Unlike most existing XSS defenses (including vanilla CSP), deploying DOMino requires little to no change to existing code base. We provide a more detailed compatibility analysis in 6.3. The remainder of this section describes the two implementation components in detail.

5.1 DOMino JavaScript library

Our JavaScript library (DOMino + DOMino++) is implemented in 201 lines of code and illustrated in Figure 1. The library must execute before any potentially malicious script, therefore websites must include it at the top of the HTML document. If DOMino++ is chosen, we enable the `disable-dynamic` CSP directive for the host document; this is done by injecting an HTML `<meta>` tag into the page [5] (e.g., `<meta http-equiv= "Content-Security-Policy" content ="script-src 'self' 'unsafe-inline' 'disable-dynamic' * ">`). Since this is before `DOMContentLoaded`, `*` is set as a `script-src` to allow all external `<script>`s to load. If DOMino++ is *not* chosen, page loading continues without a CSP.

Once the browser stops parsing the web page, it will fire a `DOMContentLoaded` event. This triggers the core component of DOMino, which proceeds to perform the following three actions:

(1) DOMino first rewrites static inline event handlers (e.g., `<img onload="a()" />`) to allow them to execute. It searches the DOM for elements with inline events. Once an element is found, DOMino will remove the inline attribute, and rewrite its content to an external `<script>` establishing an event handler. A simplified version of the rewrite code is shown in Fig 2.

```
var tags =
   document.getElementsByTagName(tag);
for (var i=0; i<tags.length; i++) {
  //look over all the tags and
  //translate inline handlers
  var inlineString =
    tags[i].getAttribute(handler);
  if (inlineString &&
      typeof inlineString == 'string') {
    //create handler function
    var fn = new Function(inlineString)
    tags[i].addEventListener(
      this.handlerWhitelist[handler],
      fn, false);
    //remove the inline handler
    tags[i].removeAttribute(handler,0);
  }
}
```

Figure 2: Simplified rewrite code for converting inline event handlers to functions

Because our rewrite code uses the `new Function` method to convert inline handler strings to code, we must briefly enable the `unsafe-eval` CSP directive when this code runs and subsequently disable when the conversion is finished. Because JavaScript is single-threaded, this brief window poses no security threat.

(2) The second action DOMino performs is to generate a 32 bit random value using the `window.crypto` API to use as the nonce. After obtaining the nonce, DOMino tightens the page's CSP by adding the script nonce to script-src to remove the page's ability to execute inline or external JavaScript code unless the nonce is supplied. This step is done by adding a CSP meta tag, such as `<meta http-equiv="Content-Security-Policy" content="script-src 'self' 'nonce-[nonce]' *;">`. Notice that `*` is a `script-src`, allowing external `<script>`s to be loaded from any origin so long as they contain the proper nonce.

(3) Finally, DOMino installs support for Web sites to import dynamic JavaScript libraries using the `document.createElement` function, which allows sites to create `<script>` nodes and load them by attaching them to the page DOM. (We believe scripts created this way are safe because the installed code is clearly intended as such, with web developers fully aware of what type of content is being executed as JavaScript.) In order to make these dynamically imported scripts to work with CSP, DOMino adds a function hook to the `document.createElement` function that adds the secret nonce to every script node created using this method. This allows trust to be propagated without compromising security because only currently executing scripts have the permission to run additional scripts. Furthermore, since we only attach the secret nonce to scripts created securely, developers are protected from honest mistakes.

5.2 Chromium `disable-dynamic` implementation

In order to offer websites a way to maximize security by disallowing dynamically injected JavaScript from executing before the `DOMContentLoaded event`, we modified the Chromium browser (specifically, the Blink component) to include an additional CSP directive called `disable-dynamic`. We chose to modify the Chromium browser because it already supports script nonces in its implementation; because we mainly modified the Blink component, the same or a similar implementation should work with Safari's Webkit. In fact, Chromium still used the Webkit rendering engine during the advent of the original implementation of `disable-dynamic` in 2015. Furthermore, we believe our implementation to be a logical extension to the features originally introduced in the CSP 1.1 specification and can be implemented in all browsers that support CSP with relative ease.

Our Chromium implementation consists of 103 lines of C++ code. Most of our code is added to two places:

(1) The function that is responsible for creating document fragments

(2) The script runner object that is responsible for running all scripts element on the page.

Document fragments are objects used by functions such as `innerHTML` and `outerHTML` to insert additional nodes to the DOM tree. In Chromium, an HTML parser is created for each newly created document fragments. Scripting permissions can be assigned to the HTML parser to decide whether scripts in the fragment are allowed to execute. For our implementation, every time a new document fragment is created, we assign script permissions to the corresponding HTML parser depending on whether the `disable-dynamic` CSP directive is active. The existing enumerator for ParserContentPolicy, an object used for configuring permissions for parsing of document fragments, was set to kDisallowScriptingAndPluginContent to configure a document fragment's respective HTML parser to prevent script execution if `disable-dynamic` is detected

Changing document fragments alone is not enough because the `document.write` function does not operate on document fragments. Instead, `document.write` interferes with host page's HTML parser directly by injecting content into its input stream. This creates an implementation challenge because we cannot explicitly change the script execution permission of a substring in the input stream. Fortunately, the HTML5 specification (http://dev.w3.org/html5/spec-LC/tree-construction.html) allows us to determine the "nesting level" of a script; that is, when a script is loaded through `document.write`, its nesting level will be greater than 0. Our implementation modifies Chromium's html parser script runner object so when a script is trying to execute with a greater than 0 nesting level, we first check whether a `disable-dynamic` CSP directive is active on its host document. If the `disable-dynamic` CSP directive is active, we prevent the script from executing.

Code was also added to the document, content security policy, csp directive list, and source list directive objects to detect `disable-dynamic` and manage the state of `disable-dynamic` within the document object. These objects were changed to instrument Chromium to determine whether the changed behaviors for document fragment and script runner functions should be enabled.

Our Chromium implementation thus provides a proof-of-concept for the `disable-dynamic` CSP directive. DOMino++, integrated within a deployed branch of DOMinoEdit, uses `disable-dynamic` to stop execution of common DOM XSS payloads injected previous to the DOMContentLoaded event. As befits a software security feature, we also formally evaluated the security of our implementation using model-checking, which we will discuss in detail in Sec. 6.1. In the process, we found and fixed some vectors (previously unknown to us) for dynamic content injection that evaded the defenses in our first implementation. For example, we discovered with the help of our model that by injecting an HTML `<meta>` '`refresh`', the attacker can cause the host document to be navigated to a JavaScript URI of the attacker's choice (e.g., `<meta http-equiv='refresh' content='0;url=javascript:evil()'>`), bypassing our defense. We fixed this bug by removing the ability of a `<meta>` tag to redirect the host page to a JavaScript URI if the `disable-dynamic` CSP source directive is set. For the defense against this vector, Blink's document object was updated to prevent a refresh in the event that `disable-dynamic` is detected

6 EVALUATION

6.1 Formal Security Analysis

Our security modeling approach [15] consists of defining the security goals of DOMino, then modeling our implementation and attacker powers by extending the web security framework described in [11]. We then check whether our model satisfies its security goals, which are stated as invariants, logical expressions over the states in the models. Our model is constructed using Alloy [26], a declarative modeling language based on first-order relational logic.

6.1.1 Security Assumptions and Goals. The high level security goal of DOMino is to stop attacker controlled JavaScript code from executing in the security context of a benign web page. Since DOMino is only designed to stop DOM-based XSS, we consider all statically included JavaScript content to be trusted. We formalize this assumption in our model using an Alloy *fact* equivalent to the following statement:

• The attacker cannot tamper with the initial state of the document, or its protections.

Furthermore, our model only allows dynamic JavaScript content to be loaded and execute using one of the safe methods described in Section 5.1 (`document.createElement`). Our security goals are modeled using Alloy *assertions* equivalent to the following statement:

• If a JavaScript object is scheduled for execution (by being attached to the DOM), it must meet one of the following conditions:

(1) The JavaScript object is associated with a node that exists inside the initial state of the document.
(2) The JavaScript object must not be associated with a node where either it or one of its parents was inserted dynamically into the DOM using an unsafe function described in Section 5.1, namely `document.write` and `innerHTML`. (Nodes inserted via `document.createElement` are not considered a successful attack, as we argued in Sec. 5.1 because developers are explicitly inviting code-injection by using this code idiom.)

6.1.2 Model Construction. In order to create a model that accurately portrays the parsing of the document and the construction of the DOM, we added several additional components to the web security framework described in [11]. An architectural summary of our model is shown in Figure 3 and described below.

Parser – The parser in our model is a simplified abstraction of the HTML parser from the HTML 5 specification [8]. It is simplified because we are not interested in the decoding of network byte stream, the preprocessing of input to the HTML tokenizer, or the tokenization process. The input to the parser is a sequence of nodes sorted in the same textual order as they appear in the HTML source. One can also imagine this input sequence as the output of the HTML tokenizer (which we did not include as a part of our model). The output of the parser is the document's DOM tree.

DOM – The DOM is represented in our model using a *Document* object and a set of *Element*s, as shown in Alloy *signature*s in Fig 5.

Inside each document, the set *elements* represents all elements reachable from *root*. For each element, the set *children* represents all its direct children from the DOM tree. Additional constraints are included in our model to uphold logical relations between signature instances. For example, one Alloy *fact* states that there can be no loops inside the DOM tree.

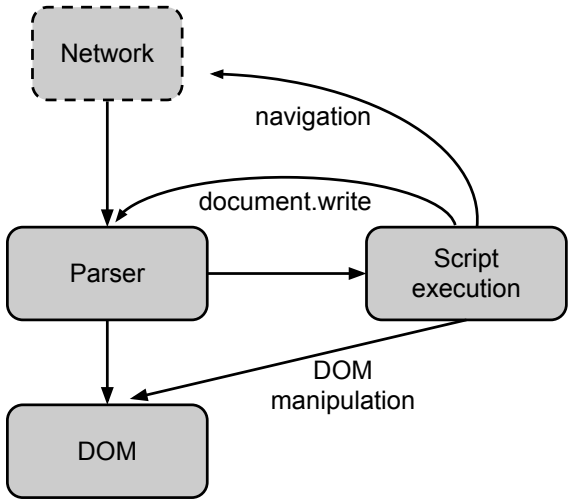


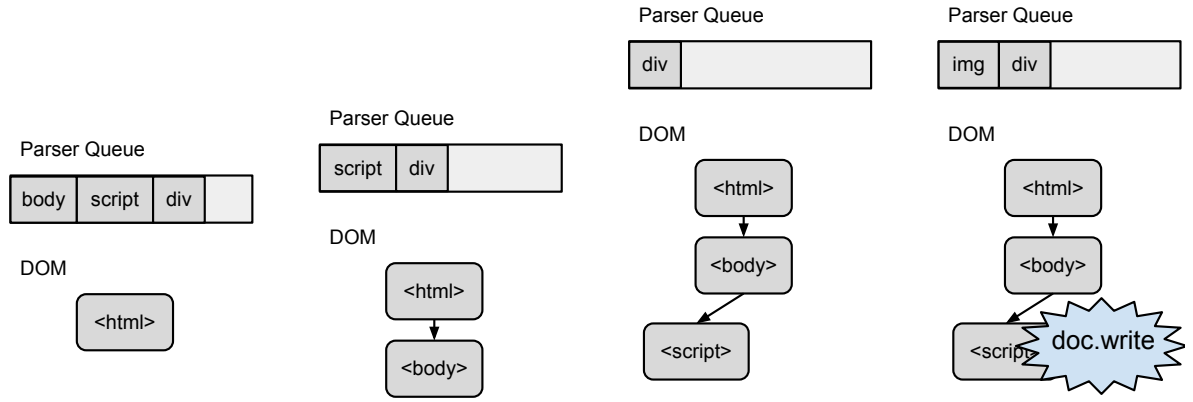
Figure 3: The main components of our model. The dashed box contains the component of the web security framework described in [11] and the solid boxes represent the components added by us.

Script execution – As the parser encounters script nodes during its tree construction process, the script will be scheduled for execution. All scripts attached to the DOM are executable. An executing script can perform several actions in our model. First, it can issue a `document.write` call, which appends a new node to the beginning of the parser's input stream. Second, it can manipulate the DOM directly using functions such as `appendChild` or `innerHTML` to attach nodes. These functions are subject to DOMino++'s `disable-dynamic` policies, i.e. `innerHTML` and `document.write` are forbidden from attaching *script* nodes.

Lastly, an executing script can trigger a navigation event that forces the document to be navigated to a specific origin. It is important to note that the target origin of a navigation event can contain a *javascript* scheme. When a document is navigated to an origin with a *javascript* scheme, a new JavaScript node is added to the parser's input stream and scheduled for execution.

Parsing Process – The original web security framework that we built our model upon contains no notion of time in constructing the DOM tree. However, the parsing of an HTML document and the construction of its DOM tree is an iterative and stateful process. To simulate the notion of states, we utilized the polymorphic linear ordering module that comes with Alloy. Each state of our parser (illustrated in the Alloy *signature* in Fig 6) consists of a *tokenQueue* containing the sequence of elements waiting to be inserted into the DOM and its associated *document* object.

At each iteration of the parsing process, two things can happen: 1) the parser consumes the next element in the token queue and inserts it into the DOM, or 2) the next element in the token queue is a script and the script executes. Figure 4 depicts a sample execution



(a) The initial state of a page load. Static elements are inside the parser queue, waiting to be inserted into the DOM.

(b) The first element is removed from queue and inserted into the DOM.

(c) The second element is removed from queue and inserted into the DOM. Since this element is a script, it is then executed.

(d) The script in 4(c) causes a new element to be added to the queue via `document.write`.

Figure 4: A sample execution path of our Alloy model.

```
sig Document {
        ... //code removed for clarity
        root: HTMLElement,
        elements: set Element, }

sig Element {
        ... //code removed for clarity
        host: one Document,
        tag: HTMLtag ,
        children: set Element }
```

Figure 5: Alloy signatures for Document and Element in our model.

```
sig ParserState {
        tokenQueue:  seq Element,
        document:  Document }
```

Figure 6: Alloy signature for ParserState

path of the HTML parser in our model. In this particular execution path, the parser starts off with three elements in its token queue. At each iteration it pops the first element of the queue and inserts it into the DOM tree, until a script element is reached (Figure 4(c)). This script element then invokes the `document.write` function call which appends a new image element to the beginning of the queue.

6.1.3 DOMino Modeling Results. To check for the security of DOMino, we model the DOM XSS attack by giving our attacker the ability to execute arbitrary JavaScript functions (e.g., `innerHTML`, `document.write`) to insert **nodes** into the document. This is a stronger form of our attacker model stated in Sec. 3.1, because it presumes that a flow already exists by which a remote attacker can inject DOM nodes.

We then ran our model, checking if the parser reaches at a state where a **script** that 1) did not exist in the initial token queue and 2) was not injected via `document.createElement` becomes scheduled for execution. If this state is reached, our security assertions are violated, as attaching an unintended script via an unsafe method simulates an attacker successfully causing execution of a dynamically-injected script. If no such state is found, we assume the model to be secure up to the number of states we explored in our model (10 of each signature).

After running our model using the Alloy Analyzer, we immediately discovered a violation to our security assertions (depicted in Figure 7). This violation occurred because a navigation event to an origin with a *javascript* scheme caused a JavaScript node to be executed by our parser. At the first glance, this violation should have never occurred because our attacker only had the ability to inject nodes into the document, without the explicit ability to navigate the document. However, upon further inspection, we realized that this navigation event was caused by an injected `<meta>` element. When a `<meta>` nodes with its `http-equiv` attribute set to `refresh` was added to the DOM, a navigation event is fired using the value of the element's 'content' attribute as its target URL. If the scheme of this url was *javascript*, then the navigation event will create a dynamic script and schedule it for execution. We confirmed that this violation existed in our implementation at that time, and updated our implementation as described in Section 5.2. When the ability for the attacker to introduce `<meta> refresh` nodes is removed from the model, our model reported no further security violations up to size 10 for each signature.

We must note that our model does not yet exhaustively incorporate all of the method that could lead to DOM script execution. Due to the growing complexity of the HTML 5 specification, the

diversity in browser implementation, and the number of undocumented browser APIs, we believe solving this problem requires a significant work, which we plan to undertake in the near future.

6.2 Performance

The biggest non-constant-time computation performed by DOMino (and DOMino++) after the `DOMContentLoaded` event is the conversion of inline event handlers to functions. We measured this conversion time using Chromium's built-in logging to the terminal and the JavaScript Performance API [7]. This affords us both fine-grained context as to when key functions fire in C++ while the JavaScript Performance API yields timestamps before and after the inline event handler conversion completes. On the average, our conversion function adds 1.22% of overhead, the 2015 implementation on the version of Chromium using Webkit yielded a performance overhead of 0.74%. The change in performance overhead is negligible and could be explained by the increasing popularity in single page client-side heavy web applications resulting in more inline event handler conversions. DOMino performs no computations more complex than creating a single CSP `<meta>` node aside from the event handler conversion. Thus, DOMino appears to cause negligible performance overhead on a page's *perceived* load time, which is usually measured as the time to `DOMContentLoaded`. (An oft-used rule of thumb is that users can perceive delays of 200ms.).

All selenium web crawls used a Chrome driver compatible with Chrome 87.0.4277.0. The desktop computer crawling tested websites has an i7 processor, 32 GB of RAM, and an NVIDIA GeForce RTX 2070 Super running on the Windows 10 operating system, the same machine used to edit and build our own branch of Chromium from source to support DOMino++. All logging preferences were enabled for both the browser and the chromium driver such that the team could best understand the behavior of the browser for each visited website, status of the crawl, and issues with the chromium driver.

Using our selenium web crawler, we visited sites from Alexa top 1000 and logged times to the DOMContentLoaded event as well as the time taken for inline event handler conversions on page loads, both of which are used to determine added performance overhead from DOMino. Fiddler Classic, a popular web debugging tool, was used to create a proxy such that the researchers could add the DOMino.js file or the DOMino++.js file to tested web pages before being rendered on the browser. For each domain, a new process is created in which selenium runs for a maximum of 500 seconds before terminating to circumvent situations where the selenium crawl hangs or a site is unreachable. Each domain is visited 5 times to determine the average performance overhead of DOMino for each domain. We ensure each website has finished loading and wait a minimum of 10 seconds before visiting the same website again or progressing to the next domain, doing so ensures we can capture and log instances of scripts and event handlers loaded after the DOMContentLoaded event which are made incompatible due to DOMino's CSP configuration.

Fiddler provides FiddlerScript, a language based on JScript.Net, that was used to enable modification of a webpage response to the browser on the fly via its OnBeforeResponse() function. [5]. The OnBeforeResponse() function was modified to automatically perform the following actions:

(1) Determine if a response is returning html content
(2) Remove compression and chunking on response
(3) Use regular expressions to find the start of the <head>tag within HTML document
(4) Read DOMino or DOMino++ local file to DOMino-script-content string
(5) Replace the <head>with <head><script>DOMino-script-content </script>
(6) Return modified response body to browser

The OnBeforeResponse() function is called before the Fiddler proxy returns a response to the browser. FiddlerScript can also ensure only requests to websites the researchers intend to test are manipulated and automates the human-error prone manual process of changing head tags in the response bodies to contain JavaScript files of interest, DOMino.js and DOMino++.js, for each website.

6.3 Compatibility

DOMino is an opt-in mechanism, making it backward compatible by design. However, one important question remains unanswered: Can DOMino be deployed on real world sites with little to no change to their existing codebase? To answer this question, we used the same selenium webcrawler referenced in performance testing, in visiting each domain 5 times we are able to average the number of scripts and inline handlers that are incompatible with DOMino on each domain. This survey aims to determine the percentage of Alexa top 1000 global sites that can deploy DOMino with no change to their existing codebase.

To determine the ratio of real world websites that can adopt DOMino with little to no changes to their existing codebase, we crawled the Alexa Top 1000 Global sites with our Chromium implementations of both defenses (DOMino and DOMino + `disable-dynamic`). On each site, we measured the percentage of scripts (tags and inline handlers) which were incompatible with our defense and thus were not loaded into the DOM. Figure 8 presents this compatibility data. As we can see, DOMino is compatible with the majority of the sites, with only 38% of sites experiencing any broken scripts. On the other hand, because DOMino + `disable-dynamic` disables dynamic script injection until `DOMContentLoaded`, it breaks scripts on 44% of the sites. We emphasize that these results do not imply that our defense breaks 38% and 44% of websites respectively, they only suggest that 62% and 56% of Alexa top 1000 global sites can deploy DOMino and DOMino + `disable-dynamic` with relative ease. We also tracked the number of incompatible scripts on each visited domain without DOMino deployed in the web crawl finding 3% of websites in the Alexa top 1000 experience broken scripts due to their own configurations of CSP.

We also examine the compatibility of DOMino and DOMino++ on the top 200 domains in figure 9, finding both to be more compatible on average with this portion of the world's most visited domains. Of this most popular subset of domains, DOMino is compatible with 88% of websites while DOMino++ is compatible with 81% of websites.

Overall, we are encouraged by these compatibility numbers, which exceed that of out-of-the-box CSP. (Without `unsafe-inline`, CSP would break scripts on nearly every site.) Furthermore, the 3% of the Alexa top 1000 websites with an improperly configured CSP

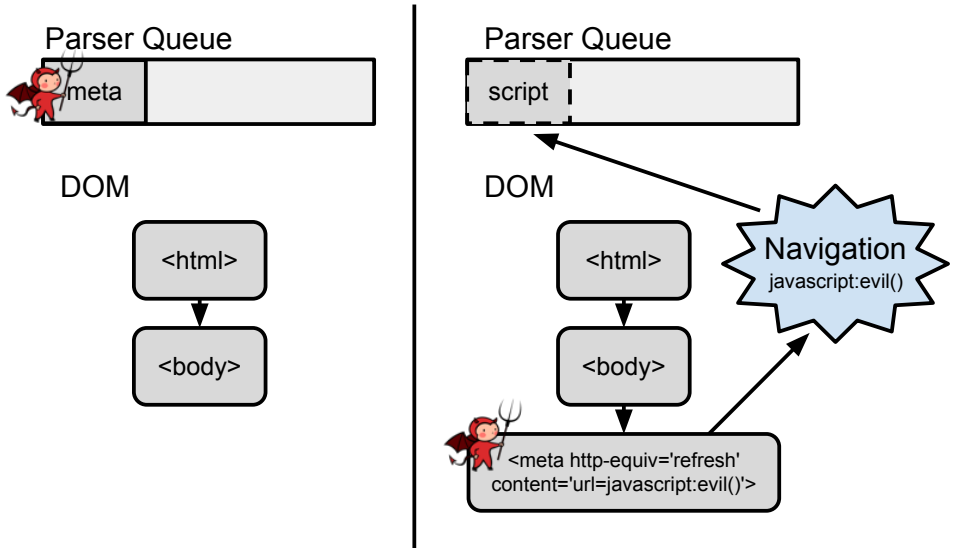


Figure 7: `<meta>` Refresh Bug Found by Alloy Model

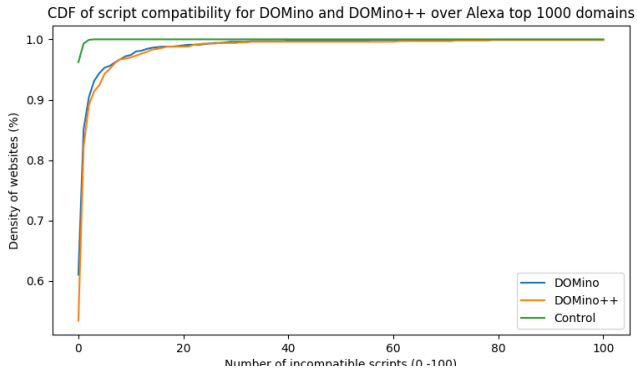


Figure 8: Compatibility study Top 1000 Alexa Global Sites

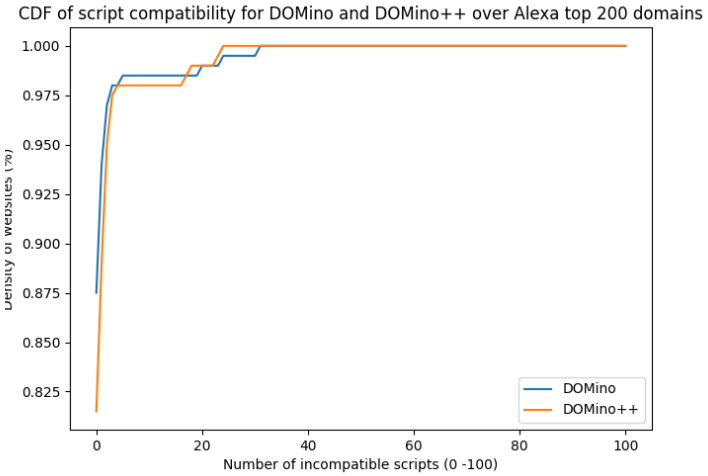


Figure 9: Compatibility study Top 200 Alexa Global Sites

exemplify the need for a solution to make CSP easily compatible with current websites. We thus believe that DOMino provides a valuable, much-needed approach for simple CSP adoption.

6.4 Deployment

A web-based, open-sourced latex editor was developed to test the efficacy of DOMino when deployed to an existing web application, especially one that experiences continuous changes to front-end infrastructure from any number of developers. The team intended to build website with a heavy web client that accomplishes a reasonably involved task using the browser while remaining compatible with DOMino's defense. DOMino.js is integrated within a deployment of this open-sourced latex editor, meant to be a free, open-sourced platform that enables concurrent collaborative editing and compilation of latex files.

This editor, named DOMinoEdit by the team (will be later renamed and referenced as contribution to this latex editor continues beyond the scope of this research), is developed using Ace editor, a popular open-sourced project that uses the DOM for rendering and manipulating text on the browser. Ace is used in production by organizations like AWS, Khan Academy, Wikipedia, and Codeacademy. Collaborative editing is facilitated by ShareDB and websockets, allowing the web application to reflect changes in a latex project in real time across all of a project's viewers and editors. ShareDB is the real-time database the team chose for managing concurrent, multi-user editing of JSON documents (used to represent latex files within a latex project) via operational transformations. The front-end features for user, project management, and login are developed using jquery, server-side data is templated using EJS (Embedded JavaScript templating). The web application server runs Node, using Express to build DOMinoEdit's API layer. DOMinoEdit's latex build service is bundled within a docker container to easily manage

```
... // websocket code calling setChat(message)
function setChat(passed) {
  let chatroom = document.getElementById("chatroom")
  if (chatroom) {
    const div = document.createElement('div');
    div.className = 'row';

    div.innerHTML = `
    <div>
      ${passed}
    </div>
    `;

    chatroom.appendChild(div);
  }
}
```

Figure 10: One example of DOMinoEdit's usage of an insecure DOM API

and allocate compute resources for compiling a project's latex files. These tools and libraries, specifically Ace and jQuery, were chosen as they are amongst the most popular front-end libraries, interface with the DOM, and could be a good measure of compatibility of DOMino's defense when integrated with these libraries.

DOMinoEdit implements a chat management component which ingests messages sent by users belonging to a given latex project via websocket communication, the chat management component was purposefully implemented such that it updates the front end (DOM) using insecure DOM API's (innerHTML, document.write()) in real time to display messages to users. An example implementation is shown in figure 10.

The deployment features two branches, the first of which uses DOMino.js and updates the chat component using the .innerHTML API such that without DOMino.js, payloads such as <img src="fake" onerror=alert("1")> and other tag-context-escaping inputs will execute. The second branch integrates DOMino++.js to prevent input payloads such as this quote-context-escaping example "><svg onload=alert(1)> from causing DOM XSS before the DOMContentLoaded event. Each deployment's DOM XSS vectors were also tested using popular payloads such as those seen on github XSS cheatsheets, OWASP, and PortSwigger [28, 41, 42]. Payloads for DOMino++ were tested against document.write(), document.inner HTML, and the http-refresh vectors, this is to test DOMino++'s efficacy against specific vectors 'disable-dynamic' aims to secure. The payloads for DOMino were tested against a a more general variety of vectors including the aforementioned vectors focused upon by DOMino++.

Within the DOMino branch deployment, DOMino stopped most all DOM XSS payloads from executing after the DOMContentLoaded event. For the one payload DOMino struggled to prevent from executing, DOMPurify was successfully used to sanitize attacker-controlled input to prevent script execution. The compatibility of DOMino in conjunction with other DOM XSS defenses on tested payloads further emphasizes DOMino's promise as a flexible DOM XSS defense.

The DOMino++ branch deployment is designed more specifically to ensure tested payloads are injected previous to the DOMContentLoaded event. To accomplish this, payloads were tested on uses of document.write(), document.innerHTML, and the http-refresh functions as included <scripts> within the HTML markup that attempts to write content directly to the webpage by pulling the payload from the URL, a commonly used attacker-controlled source of data.

DOMinoEdit shows how DOMino.js prevents DOM XSS after the DOMContentLoaded event and the capacity for DOMino++.js to prevent DOM XSS previous to the DOMContentLoaded event on the modified Chromium browser with the 'disable-dynamic' CSP source directive. DOMinoEdit integrates DOMino after development of most of the platform's web functionality, exemplifying DOMino's backwards and opt in compatibility.

Future work regarding effectiveness of DOMino on real-world websites could include modifying our crawler to also test payloads on the visited websites from the Alexa top 1000, this would require the permission of each tested website to test an incredible volume of payloads. Alternatively, our crawler could be modified to crawl our deployment of DOMinoEdit and automatically test payloads on the deployed branches upon changes to the source code of DOMinoEdit, DOMino, and DOMino++.

7 CONCLUSION

We have presented DOMino, our defense against DOM XSS that leverages Content Security Policy, and demonstrated its ease-of-adoption by showing its compatibility with current Web development practices in a real-world evaluation. We also showed that DOMino imposes negligible performance overhead. We additionally contribute infrastructure (DOMinoEdit) that will be open-sourced such that future changes to DOMino and DOMino++ can be tested for compatibilty and performance. Futhermore, we extended a formal Alloy web security model by adding a script execution model and confirmed the security properties of DOMino. Given the trend of Web development towards "single page" client-side JavaScript applications, DOM XSS ought only to become a more prominent threat, making DOMino a timely contribution. Because DOMino relies on features introduced as far back as CSP 1.1 (such as script nonces and <meta> policies), we encourage browser vendors to fully implement and continue to provide support for the CSP to facilitate broad adoption of defensive approaches such as DOMino.

REFERENCES

[1] 2012. skipfish web application scanner. http://code.google.com/p/skipfish/. (2012).
[2] 2015. Content Security Policy 1.0. https://www.w3.org/TR/CSP1/. (2015).
[3] 2019. Gecko-Blink Script Nesting Level Definition Realignment. https://hg.mozilla.org/integration/autoland/rev/990c8a382cf3. (2019).
[4] 2021. ADsafe: Making JavaScript Safe for Advertising. http://www.adsafe.org/. (2021).
[5] 2021. Content Security Policy 1.1. https://www.w3.org/TR/CSP/. (2021).
[6] 2021. Django Templates. https://docs.djangoproject.com/en/dev/ref/templates/. (2021).
[7] 2021. JavaScript Performance API. https://developer.mozilla.org/en-US/docs/Web/API/Performance. (2021).
[8] https://html.spec.whatwg.org/multipage/parsing.html. Parsing HTML documents. http://www.whatwg.org/specs/web-apps/current-work/multipage/parsing.html. (https://html.spec.whatwg.org/multipage/parsing.html).

[9] RFC 6454. 2011. The Web Origin Concept. https://tools.ietf.org/html/rfc6454. (2011).

[10] Acunetix. 2021. Web Vulnerability Scanner. http://www.acunetix.com/vulnerability-scanner/. (2021).

[11] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell, and Dawn Song. 2010. Towards a Formal Foundation of Web Security. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium (CSF '10)*. IEEE Computer Society, Washington, DC, USA, 290–304. https://doi.org/10.1109/CSF.2010.27

[12] Devdatta Akhawe, Prateek Saxena, and Dawn Song. 2012. Privilege separation in HTML5 applications. In *Proceedings of the 21st USENIX conference on Security symposium (Security'12)*. USENIX Association, Berkeley, CA, USA, 23–23. http://dl.acm.org/citation.cfm?id=2362793.2362816

[13] Daniel Bates, Adam Barth, and Collin Jackson. 2010. Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the 19th international conference on World wide web (WWW '10)*. ACM, New York, NY, USA, 91–100. https://doi.org/10.1145/1772690.1772701

[14] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. 2010. State of the Art: Automated Black-Box Web Application Vulnerability Testing. *Security and Privacy, IEEE Symposium on* 0, 332–345. https://doi.org/10.1109/SP.2010.27

[15] J. Bau and J.C. Mitchell. 2011. Security Modeling and Analysis. *Security Privacy, IEEE* 9, 3 (may-june 2011), 18 –25. https://doi.org/10.1109/MSP.2011.2

[16] Eric Yawei Chen, Jason Bau, Charles Reis, Adam Barth, and Collin Jackson. 2011. App Isolation: Get The Security of Multiple Browsers with Just One. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS '11)*. ACM, New York, NY, USA, 227–238. https://doi.org/10.1145/2046707.2046734

[17] Facebook Inc. 2009. FBJS (Facebook JavaScript). https://developers.facebook.com/blog/post/189/. (2009).

[18] Facebook Inc. 2010. XHP: A New Way to Write PHP. http://www.facebook.com/notes/facebook-engineering/xhp-a-new-way-to-write-php/294003943919. (2010).

[19] Gmail Developer. 2015. Personal Correspondence. (2015).

[20] Nicolas Golubovic. *autoCSP-CSP-injecting reverse HTTP proxy.* Ph.D. Dissertation.

[21] Google Inc. 2010. google-caja: Compiler for making third-party HTML, CSS, and Javascript safe for embedding. http://code.google.com/p/google-caja/. (2010).

[22] Nicholas Green. 2013. Proposal for script-hash directive in CSP 1.1. http://lists.w3.org/Archives/Public/public-webappsec/2013Feb/0052.html. (2013).

[23] Daniel Hausknecht, Jonas Magazinius, and Andrei Sabelfeld. 2015. May I? - Content Security Policy Endorsement for Browser Extensions. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Magnus Almgren, Vincenzo Gulisano, and Federico Maggi (Eds.). Springer International Publishing, Cham, 261–281.

[24] Mario Heiderich, Christopher Späth, and Jörg Schwenk. 2017. DOMPurify: Client-side protection against XSS and markup injection. In *European Symposium on Research in Computer Security*. Springer, 116–134.

[25] Lon Ingram and Michael Walfish. 2012. TreeHouse: JavaScript sandboxes to helpWeb developers help themselves. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 13–13. http://dl.acm.org/citation.cfm?id=2342821.2342834

[26] Daniel Jackson. 2006. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press.

[27] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2010. An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications. In *Proceedings of the 17th ACM conference on Computer and communications security (CCS '10)*. ACM, New York, NY, USA, 270–283. https://doi.org/10.1145/1866307.1866339

[28] Robert RSnake Hansen Jim Manico. 2021. XSS Filter Evasion Cheat Sheet. https://owasp.org/www-community/xss-filter-evasion-cheatsheet. (2021).

[29] Amit Klein. 2005. DOM Based Cross Site Scripting or XSS of the Third Kind. http://www.webappsec.org/projects/articles/071105.shtml. (2005).

[30] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 1193–1204.

[31] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out domsday: Towards detecting and preventing dom cross-site scripting. In *2018 Network and Distributed System Security Symposium (NDSS)*.

[32] William Melicher, Clement Fung, Lujo Bauer, and Limin Jia. 2021. Towards a Lightweight, Hybrid Approach for Detecting DOM XSS Vulnerabilities with Machine Learning. In *Proceedings of The Web Conference. To appear.*

[33] Minded Security S.r.l. 2011. Dominator by Minded Security. https://blog.mindedsecurity.com/2011/05/dominator-project.html. (2011).

[34] Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2007. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *In Proceeding of the Network and Distributed System Security Symposium (NDSS'07)*.

[35] OWASP: The Open Web Application Security Project. 2017. OWASP Top 10, 2017. https://owasp.org/www-project-top-ten/2017/. (2017).

[36] OWASP: The Open Web Application Security Project. 2021. DOM Based XSS. https://owasp.org/www-community/attacks/DOM_Based_XSS. (2021).

[37] OWASP: The Open Web Application Security Project. 2021. Webkit Bug - Implementing 'strict-dynamic' source expression. https://bugs.webkit.org/show_bug.cgi?id=184031. (2021).

[38] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. 2015. Auto-Patching DOM-based XSS At Scale. *Foundations of Software Engineering (FSE)* (2015).

[39] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. 2015. DexterJS: robust testing platform for DOM-based XSS vulnerabilities. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 946–949.

[40] Qualys Inc. 2021. QualysGuard. https://qualysguard.qualys.com. (2021).

[41] PortSwigger Research. 2021. Cross-site scripting (XSS) cheat sheet. https://portswigger.net/web-security/cross-site-scripting/cheat-sheet. (2021).

[42] s0md3v. 2021. AwesomeXSS Vulnerability Payload List. https://github.com/s0md3v/AwesomeXSS. (2021).

[43] Mike Samuel, Prateek Saxena, and Dawn Song. 2011. Context-Sensitive Auto-Sanitization in Web Templating Languages Using Type Qualifiers. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS '11)*. ACM, New York, NY, USA, 587–600. https://doi.org/10.1145/2046707.2046775

[44] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A Symbolic Execution Framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10)*. IEEE Computer Society, Washington, DC, USA, 513–528. https://doi.org/10.1109/SP.2010.38

[45] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. 2019. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. (2019).

[46] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. 2014. Precise client-side protection against dom-based cross-site scripting. In *Proceedings of the 23rd USENIX security symposium*. 655–670.

[47] Shuo Tang, Chris Grier, Onur Aciicmez, and Samuel T. King. 2010. Alhambra: a system for creating, enforcing, and testing browser security policies. In *Proceedings of the 19th international conference on World wide web (WWW '10)*. ACM, New York, NY, USA, 941–950. https://doi.org/10.1145/1772690.1772786

[48] Andreas Vikne and Pål Ellingsen. In *Client-Side XSS Filtering in Firefox*. San Diego, CA, USA. https://doi.org/10.1145/2046707.2046775

[49] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. 2016. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1376–1387. https://doi.org/10.1145/2976749.2978363

[50] Michael Weissbacher, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. ZigZag: Automatically Hardening Web Applications Against Client-side Validation Vulnerabilities. In *24th USENIX Security Symposium (USENIX Security 15). USENIX Association.*