EasyCSPeasy: A Server-side and Language-agnostic XSS Mitigation by Devising and Ensuring Compliance with CSP

Beliz Kaleli[1], Manuel Egele[1], and Gianluca Stringhini[1]

Boston University, Boston, USA
{bkaleli,megele,gian}@bu.edu

Abstract. Scripts enable much of the functionality of the modern Web. At the same time, attackers may utilize them in cross-site scripting (XSS), leading to malicious code execution. Content Security Policy (CSP) is a mechanism to prevent XSS attacks by restricting the scripts that can be loaded on a website. Devising an effective CSP policy by hand is a daunting task due to the complexity of modern Web applications. Previous attempts to automate this process are either specific to certain server-side programming languages, require modifications to the Web application's source code, fall short in mitigating XSS, or require third-party cooperation. To assist Web developers and facilitate the adoption of CSP, we propose a server-side system that crafts a *safe CSP* configuration and modifies the script content in the server response to comply with the set configuration. EasyCSPeasy overcomes various limitations of previous systems as it is language-agnostic, standalone, and does not require source code modification. We evaluate our system on six open source Web applications (five PHP-based, one Perl-based) and show that all continue to provide their *commonly interacted* functionalities when integrated with EasyCSPeasy. We quantify the minimal overhead introduced by our system. We deploy known vulnerable versions of three Web applications and demonstrate that the CSP policies automatically generated by EasyCSPeasy block known attacks against these applications.

Keywords: Web Security · CSP · XSS

1 Introduction

Cross-site Scripting (XSS) vulnerabilities are a persistent and ongoing security issue on the Web [7]. XSS attacks occur when attackers inject malicious JavaScript (JS) that executes on otherwise benign and trusted websites. These malicious scripts may access session cookies (unless the HttpOnly flag is set, which prevents JS from accessing cookies) and sensitive information used within the website context. Researchers and Web organizations (e.g., the W3C [24]) have been battling against XSS by constructing both server-side [27,28,38,44,45] and client-side [39,49] defenses. Mitigation techniques include filtering user input,

encoding output, and Content Security Policy (CSP) [6]. CSP allows developers to declare approved origins of content (images, scripts, etc.) that browsers are permitted to load on their websites, providing protection against XSS by only allowing developer-approved scripts. Consequently, adding new scripts to a website requires the developer to update the CSP. Otherwise, the browser may block the execution of the new scripts, leading to functionality and rendering issues. Moreover, the included scripts may also create other scripts during runtime. Hence, it is challenging for developers to keep track of these script inclusions. This makes creating and maintaining a CSP a burdensome and error-prone process when done manually. As a testament to this, Roth et al. [42] found that several website administrators tried to maintain their CSP for months, but eventually gave up. Moreover, they found that 56% of the websites that deploy CSP policies do so in an alert-only fashion, without enforcing any content limitation, likely due to the difficulty of getting the policies right.

To overcome the issues with manually devising policies and improve CSP adoption, researchers proposed several automated approaches that generate a CSP and modify the server-side code or server response [30,33,34,41]. Upon our analysis of previous work, we conclude that the studies that focus on mitigating XSS by leveraging CSP have one or more of the following shortcomings: deriving a single allowlist-based CSP [34,30,41,33,49], third-party dependency [30,49], being specific to certain server-side languages [34,33] or not supporting commonly practiced [37] dynamic websites (generating JS on the fly via server-side scripting) [34,33,49]. In an allowlist-based CSP approach, only the sources listed by the site-operator in the policy are allowed to execute, while all other sources are blocked by default. However, Roth et al. [42] showed that 90% of the CSPs in the wild are bypassable due to insecure allowlists. Moreover, Lekies et al. introduced Script Gadgets [36] (developer-intended JavaScript code fragments that can be reused to execute arbitrary JavaScript) and showed that some gadgets can bypass all CSP configurations where a single CSP header is set, including allowlist-based ones. Weichselbaum et al. [47] suggested that a safe solution must have two CSPs: one that inspects the source URL of a script and one that inspects an assigned token (a nonce value [6]) of a script. Hence, we see a need for a solution that can help Web developers automatically devise a *safe CSP* and rewrite the pages of their website to comply with the devised CSP (i.e., ensuring that the approved JS sources will be allowed to execute in the browser). A *safe CSP* can fully utilize the capabilities of CSP to block XSS attacks propagated by injecting JavaScript into a website.

In this paper, we propose and develop EasyCSPeasy, a server-side solution to assist Web developers to automatically devise a safe CSP for their target Web application and rewrite the pages to ensure that only the script content intentionally included by the site-operator (i.e., developer-intended) will execute in the Web browser (i.e., ensuring compliance with the set CSP). EasyCSPeasy is language-agnostic, standalone (works independently of third parties), and supports dynamic websites. Our system devises a safe CSP by visiting the pages of a target website in a clean (i.e., uncompromised) setup which does

not contain any attacker-injected content, and collecting the developer-intended script sources. Then, EasyCSPeasy's filter rewrites the server HTML response on the server side. We built EasyCSPeasy to work with Apache Web servers. However, our approach can be adapted to work with other Web servers with minor engineering effort. We evaluate EasyCSPeasy on six open source Web applications (PHP-based: WordPress [23], Litecart [21], Squirrelmail [18], PhpMyAdmin [15], PhpBB3 [14], and Perl-based: TWiki [19]), showing that our approach does not break their *commonly interacted* functionalities (i.e., functionalities that are commonly exercised by the users [26]). We also evaluate the security aspect of our system by deploying three known-vulnerable Web applications with known XSS vulnerabilities, and showing that the safe CSP generated by EasyCSPeasy is effective in preventing them from being exploited. To encourage further research in this space, we will make the source code of EasyCSPeasy open source upon acceptance of this paper.

2 Related Work

CSP is designed to control content inclusion from a single point to make enforcement and security checks easier. However, adopting CSP itself is shown to be challenging [42,29,48]. One of the first examples to facilitate the CSP adoption is deDacota [33]. Dedacota works by securing Web applications programmed in ASP.NET by rewriting an application so that the code and data are clearly separated in its Web pages and then leveraging CSP. AutoCSP [34] leverages dynamic taint analysis to identify allowed content and modifies the server-side code to generate pages. CCSP [30] proposes a modified CSP where Web developers have to provide upper bounds and content providers have to provide a dependency list for each script in the website. JSCSP [49] is a client-side solution that offers a self-defined security policy that enforces essential confinements to JavaScript functions and DOM elements etc. CSPAutoGen [41] trains templates, generates a single allowlist-based CSP, and rewrites server responses.

Each of these previous works has limitations that EasyCSPeasy remedies. Both JSCSP [49] and CCSP [30] require client-side modification and also depend on third parties. EasyCSPeasy is a standalone server-side solution. AutoCSP [34] and deDacota [33] are specific to websites programmed in PHP and ASP.NET server-side languages respectively, whereas EasyCSPeasy is language agnostic. Adopting a dynamic approach is a prevalent practice among site-operators [37]. However, unlike EasyCSPeasy, AutoCSP [34], JSCSP [49] and deDacota [33] are not compatible with dynamic websites. CSPAutoGen [41] enables the execution of any script observed during the training phase within any page of the target Web application. This could lead to unintended or malicious behavior, akin to exploiting script gadgets, if an attacker injects a developer-intended script from one page of a domain into another page of the same domain where the developer did not intend it to be included. EasyCSPeasy permits script execution on a target page only if the script was observed on that page during training (with potentially varying query parameters as detailed in Section 5.3). All five

systems mentioned above [34,30,49,33,41] follow a single allowlist-based CSP approach which makes them inherit the security issues we discuss in Section 3.2. Most importantly, unlike all these systems, EasyCSPeasy generates a safe CSP configuration.

3 Background and Motivation

The main building block of Web security is the Same Origin Policy (SOP). Under the SOP, a Web page is only able to access resources or data from the same origin as the page itself, unless the resource or data is explicitly shared through mechanisms such as Cross-Origin Resource Sharing (CORS). XSS attacks bypass the SOP [43] and compromise the confidentiality of cross-site data that should be kept secret. Previous research identified three main types of XSS vulnerabilities as Reflected, Stored, and DOM-based XSS [31]. CSP has been developed to mitigate the risk of content injection vulnerabilities such as XSS.

3.1 Content Security Policy

Modern webpages are an amalgamation of code and resources from various origins. JavaScript code can be included directly (i.e., inline) in the HTML code of a page, either within script tags or as part of inline event handlers. Alternatively, an external script can be included by using the `src` argument in a script tag (i.e., non-inline). Inline event handlers are HTML attributes (e.g., `onclick`) where the attribute value contains the JS code (e.g., `alert()`) that will run when the event occurs.

CSP works by limiting the resources that the user agent (i.e., the Web browser) is allowed to load for a target page. A CSP has the following form: `script-src <source-list>; img-src <source-list>`. A CSP can be delivered as an HTTP response header (most common) or as an HTML `<meta>` tag. When constructing a CSP, the developer needs to set a directive (e.g., `script-src`) to specify the type of content they want to control. Each directive in the CSP has its own `<source-list>` that defines the developer-intended sources. For example, to limit script inclusions, a site-operator could set the `script-src` directive. The browser will compare the script sources included in the webpage with the sources approved in the CSP (i.e., allowlisted) and only execute scripts that match the allowed sources, blocking all others. A site-operator can set multiple CSP compositions instead of a single one. In that case, the Web browser has to honor each CSP individually [5]. The `script-src` directive may contain the following keywords and values:

- `self`: Allows the execution of non-inline scripts with `src` attribute values that share the current URL's origin.
- `nonce-<value>`: Matches the specific script elements that contain the correct nonce value on the page. The nonce should be unique for each HTTP response and should be generated using a cryptographically secure random generator [20].

- `<allowed_hash>`: Allows the execution of an individual inline script by allowlisting its hash value.
- `unsafe-inline`: Allows the execution of all inline scripts and event handlers.
- `unsafe-eval`: Allows the usage of unsafe JavaScript function `eval` and other eval-like functions (e.g., `eval, new Function, setTimeout, setInterval`) that convert text to JavaScript.
- `strict-dynamic`: Specifies that the trust explicitly given to a script present in the page HTML, by accompanying it with a nonce or a hash, shall be propagated transitively to all the scripts loaded by that root script.
- `<allowed_url>`: Allows the execution of a non-inline script by allowlisting its source URL.

CSP Bypasses For CSP to mitigate XSS properly, it has to be configured securely, and Web browsers should support and enforce the CSP version used. Even so, policies that use a single CSP can still be bypassed. Open Redirects [42] and Script Gadgets [36] can defeat the single CSP configurations. If an allowlisted source redirects to a different target URL, browsers enforce CSP by ignoring the target URL's path. For instance, with a CSP policy like `script-src redir.com cdn.com/benign.js`, an attacker can exploit an open redirect vulnerability on `redir.com` to redirect to `cdn.com/evil.js`. Despite not being allowlisted, `cdn.com/evil.js` is executed due to CSP's partial path match. Script gadgets [36] are developer-intended script fragments within an application's code base. However, they can be reused by an injected HTML element, which may result in arbitrary JavaScript execution. In their study, Lekies et al. [36] introduce various types of script gadgets. Notably, two types can bypass single CSP configurations even if `unsafe-eval` and `unsafe-inline` are not enabled: gadget Type 1, bypassing nonce-based strict-dynamic enabled CSPs, and gadget Type 2, bypassing all single CSP configurations, including allowlist-based ones. Type 1 gadgets can be used to trigger new script element creation with potentially malicious code. We show an example of this type in Appendix Section A. Type 2 gadgets are out of scope since they do not create new script elements and CSP cannot detect and block them. In Section 3.2, we discuss the CSP configuration that can effectively defend against Type 1 gadgets.

3.2 Discussion on CSP Configurations

In Section 3.1, we presented the keywords offered by the CSP standard. These keywords can be combined to create different CSP configurations. However, not all CSP configurations effectively safeguard against XSS. Previous work [47] showed that a safe solution must have two CSPs: one that inspects the source URL of a script (an allowlist-based CSP, CSP C1) and one that inspects an assigned token of a script (a nonce-based CSP, CSP C2). The safe CSP configuration combines CSP C1 and CSP C2 to get the benefits of both worlds, specifically allowlisting individual URLs and enforcing a nonce value [47] and has the following form: `script-src <allowed_urls> <allowed_hashes>; script-src 'nonce-<value>' 'strict-dynamic'`. By combining both policies, safe CSP

avoids exhibiting their vulnerabilities. In Appendix Section B, we discuss the offered protection and the challenges in real-world deployment for C1 and C2 policies in detail. For instance, unlike the C1 policy, safe CSP cannot be bypassed by leveraging an Open Redirect vulnerability because an attacker-injected script would also need a valid nonce to execute in the presence of the C2 policy. Moreover, unlike the C2 policy, safe CSP can protect against Type 1 Script Gadgets. We conclude that safe CSP is the safest configuration among the discussed policies. Hence, throughout this paper, we refer to a CSP that is able to defend against the attacks discussed (excluding Type 2 Script Gadgets which cannot be prevented via CSP) as a safe CSP. However, this configuration inherits the challenges in real-world applicability of C1 and C2 policies, namely, building and maintaining a comprehensive allowlist and assigning a valid nonce to each script. Therefore, we also conclude that it is imperative, especially for complex websites, to have an automated system that can devise a safe CSP.

Since CSP can potentially block scripts on which website features may depend, the CSP standard offers `unsafe-inline` and `unsafe-eval` as compatibility modes that can ease deployment. These modes are not recommended for permanent use due to potential XSS vulnerabilities [8]. An attacker can exploit `unsafe-inline` by directly injecting scripts. Therefore, it is crucial to eliminate the need for `unsafe-inline` in a website's CSP. Since enforcing a CSP without `unsafe-inline` may lead to functionality issues, it is required to ensure compliance of the script content with the set CSP. EasyCSPeasy handles this by modifying the server response. Exploiting `unsafe-eval` requires passing a user-controlled string into an eval-expression and this text being converted to executable JavaScript, which can be prevented with sanitization practices [29]. Hence, the presence of `unsafe-eval` does not automatically result in website vulnerability. Our re-evaluation of Steffens et al.'s experiments [46] on the top 10K websites from the Tranco List [1] showed an 18.3% decrease in the use of `eval` (from 78.8% to 60.5%), 18 months after their experiments (results presented in Appendix Section C). Therefore, we have elected to leave the process of removing `eval` (from their code) to the discretion of site-operators.

4 System Overview

In this paper, we aim to build a system that i) generates a safe CSP for a target website and ii) ensures compliance with the set CSP to aid in the safe adoption of CSP. Addressing limitations in previous work (discussed in Section 2), we implemented EasyCSPeasy to be standalone (not dependent on third-party), language-agnostic and dynamic-website compatible. EasyCSPeasy automatically generates a safe CSP, allows disabling `unsafe-inline`, alters the server responses to ensure the execution of benign scripts, and prevents unwanted or malicious behavior by only allowing the execution of a script within the context of its developer-intended page (i.e., page-based script matching). In Appendix Section D, we compare existing systems to EasyCSPeasy on these design goals and show that EasyCSPeasy is the only system that accomplishes all of the goals.

We depict the architecture of EasyCSPeasy in Figure 1. EasyCSPeasy's operation follows two phases: Learning and Rewriting. During the Learning phase, in a clean setup of the Web application (i.e., does not contain any attacker-injected elements), EasyCSPeasy crawls the pages of the website to observe the script content and generate the CSP. During the Rewriting phase, EasyCSPeasy receives the server response and validates the script content received in this response by inspecting its prior occurrence (i.e., the validation process). If an element that contains JavaScript (i.e., JS-including element) was previously detected, EasyCSPeasy rewrites this element to allow its execution (i.e., the process of ensuring compliance). Conversely, if a JS-including element was not previously observed, it is removed from the server response or blocked by the CSP.

The Learning Phase consists of two modules: Crawler and CSP Generator. The Crawler creates a sitemap [22] (a list of URLs) of the website. Then, the CSP Generator visits the pages in the sitemap, records information about the JS-including elements in those pages (referred to as element knowledge in the rest of the paper), and generates a safe CSP. The Rewriting Phase has two stages: Filter and Mutation Observer. The Filter fetches the element knowledge recorded in the Learning phase, uses it to validate the JS-including elements received in the server response, and rewrites the validated elements to comply with the set CSP. The JS-including elements in a webpage can be pre-included [41] or runtime generated. Pre-included elements exist in the static HTML generated by the server, and runtime-included elements are dynamically generated in the Web browser. The JS-including elements that may be created at runtime will not be present in the server response received by the Filter. Hence, the Filter attaches the Mutation Observer script to the HTML server response which monitors runtime-generated elements.

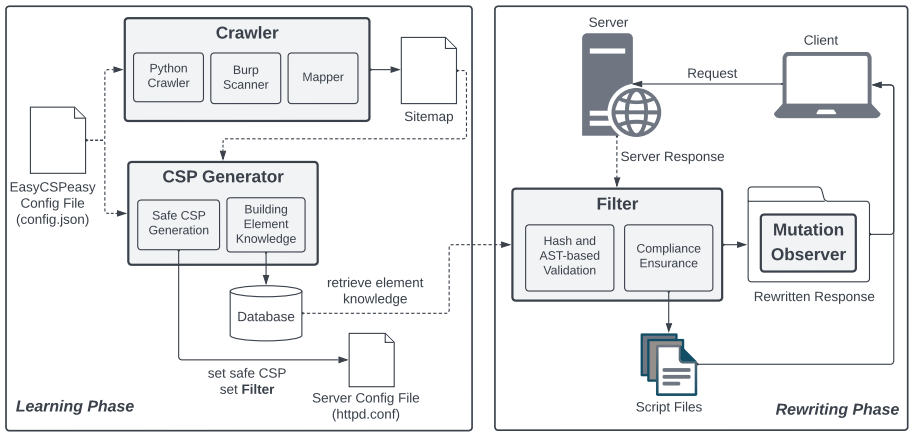

Fig. 1: *EasyCSPeasy System Architecture. (Dashed lines represent input to system modules.)*

4.1 Learning Phase

In the Learning phase, EasyCSPeasy creates the sitemap of a target website. Then, it visits the pages in the sitemap to collect the element knowledge and devise a safe CSP. Since the Learning Phase focuses solely on JS-including elements, it only needs to be executed once after changes in script content.

Crawling and CSP Generation The format of the safe CSP generated by EasyCSPeasy is shown in Listing 1.1. To create a comprehensive C1 policy for a target Web application, we gather source URLs of non-inline scripts and add them to the policy (`<allowed_urls>`). Additionally, we compute hashes of inline scripts and incorporate them into the C1 policy (`<allowed_hashes>`). To collect the inline and non-inline scripts included in a Web application, EasyCSPeasy requires a comprehensive sitemap. The sources not seen by the CSP Generator will not be allowlisted in the CSP. Hence, the CSP generated by EasyCSPeasy may block the execution of these elements. Generating a comprehensive sitemap is a difficult task and can be influenced by many factors such as website structure and authentication requirements [32,35]. Additionally, a website may introduce new JS code in a page due to various reasons such as GET and POST parameters, the state of the page, and script content generated after user interactions, making it difficult to accurately capture all possible JS code during the Learning phase. Web developers typically create a targeted testbed tailored to their website to ensure that their applications are thoroughly tested in an environment that closely mirrors production. The sitemap coverage shortfall may potentially be addressed by leveraging a website-specific testbed. Since we aim to evaluate our system on multiple Web applications, we build a generic crawler to generate sitemaps. To mitigate the sitemap coverage issues that arise from the reasons discussed above, our Crawler utilizes three independent techniques and combines the output of these methods to capture the script content of a website (discussed in Section 5). We note that Web crawling is an open research problem. EasyCSPeasy inherits the limitations of existing crawlers and may not generate a fully complete sitemap. Hence, EasyCSPeasy can automatically benefit from the advances in crawling research.

The Crawler, given a seed URL (typically the home page of a target website), generates a sitemap containing the pages of the website. Our Crawler uses authentication cookies to access behind-the-login pages and crawl a website. Then, the CSP Generator visits every page in this sitemap, records the element knowledge, and builds the C1 policy as described above. To build a C2 policy, as shown in Listing 1.1 line 2, the CSP Generator declares a nonce value and enables the `strict-dynamic` keyword in the C2 policy. We discuss the crawling and the CSP generation processes in detail in Section 5.

```
1 C1: script-src 'self' <allowed_urls> <allowed_hashes>
2 C2: script-src 'nonce-<nonce_value>' 'strict-dynamic'
```

Listing 1.1: *The CSP pair generated by EasyCSPeasy*

Building the Element Knowledge EasyCSPeasy must detect script content, validate that it is developer-intended, and make only the JS-including elements that pass the validation compliant with the set CSP. To accomplish that, EasyCSPeasy performs detection, validation, and compliance ensurance in various stages based on the script content's inclusion time (pre-included or runtime included) and inclusion mechanism (inline or non-inline). We summarize the responsible stages in Table 1. To help with validation and compliance, the CSP Generator detects each JS-including element and records the element knowledge which is later utilized by the Filter and the Mutation Observer. The CSP Generator visits the pages in the sitemap in a clean setup of the Web application to ensure only developer-intended elements are seen. Then, it computes the hashes of the `outerHTML` attributes for these elements (i.e., `tag_hash`), and records these tag hashes to a database. When an HTML server response is received in the Rewriting phase, the tag hashes of the received elements are compared with the hashes in the database for validation.

A dynamic website leverages scripts that contain dynamic values (e.g., current time, CSRF tokens). These values are set via server-side scripting then the dynamic script is shipped to the browser (pre-included in the response). An example of an inline dynamic script that contains such value is the following where the `time` value is not static: `<script>var userSettings="time":"1650395419" ;</script>`. Hence, if a dynamic script is inline, it is not possible to use the hash value of this script to validate it through hash comparison. To validate the inline dynamic scripts, EasyCSPeasy leverages the Abstract Syntax Trees (ASTs). The AST of a target inline dynamic script would contain the dynamic values in this script. When the dynamic values naturally change (e.g., during different sessions), the obtained AST for the target inline dynamic script will change as well. Hence, it is not possible to validate this script by comparing the ASTs directly. To overcome this issue, the CSP Generator first obtains the ASTs of inline scripts and modifies them to remove the dynamic values (the `Literal` nodes [2]). Then, it records the modified ASTs in the element knowledge (i.e., `script_ast`). The ASTs are later utilized by the Filter to validate the inline dynamic scripts (explained in detail in Section 5).

To ensure the compliance of the validated elements with the safe CSP, we need to ensure compliance with both C1 and C2 policies. To this end, the CSP Generator adds the source URLs of non-inline scripts and the hashes of inline scripts (except the dynamic ones) to the C1 policy. Inline events (due to no `unsafe-inline`) and inline dynamic scripts (due to everchanging hash value) need to be converted to non-inline for compliance with the safe CSP which is done in the Rewriting phase. To help with this, the CSP Generator records additional data in the element knowledge along with the tag hashes and script ASTs discussed above. The recorded element knowledge is shown in the following and further discussed in Section 4.2: `event`=[`tag_hash`, `event_id`, `event_listener_script`] and `script`=[`tag_hash`, `script_ast`, `script_id`]. To comply with the C2 policy, the Filter adds nonce values to the validated pre-included elements. Since the safe CSP enables `strict-dynamic`, transitive script

inclusions are allowed by default. Hence, runtime-included scripts (either by `self` or by third-party) inherit the nonce from their validated parent script.

		Validation		Compliance with the safe CSP			
		Stage	**Method**	**Stage**	**Method**	**Stage**	**Method**
Pre-included	**Inline Event Handlers**	Filter	tag hashes	CSP Generator	`self` in C1	Filter	convert to non-inline, add nonce
	Inline Scripts	Filter	tag hashes	CSP Generator	hash in C1	Filter	add nonce
	Inline Dynamic Scripts	Filter	modified ASTs	CSP Generator	`self` in C1	Filter	convert to non-inline, add nonce
	Non-inline Scripts	Filter	tag hashes	CSP Generator	source URL in C1	Filter	add nonce
Runtime Included	**Inline Event Handlers**	Mutation Observer	tag hashes	CSP Generator	`self` in C1	Mutation Observer	convert to non-inline, inherits nonce
	Inline Scripts	N/A	N/A	CSP Generator	hash in C1	N/A	inherits nonce
	Non-inline Scripts	N/A	N/A	CSP Generator	source URL in C1	N/A	inherits nonce

Table 1: *EasyCSPeasy phases that handle the validation and compliance ensurance of script content. Detections are done by the CSP Generator during the Learning phase.*

4.2 Rewriting Phase

In the Rewriting phase, EasyCSPeasy receives each server response generated by the Web application before it reaches the client to validate and rewrite script content for compliance. Hence, the Rewriting phase is run for each page request. The Filter executes the `validate()` function shown in Listing 1.2 for each JS-including element in the server response. This function first validates the elements and then removes any element that has failed the validation process from the HTML response. Then, we execute the `comply()` function shown in Listing 1.3 on all remaining elements to ensure compliance with the safe CSP.

Validation The `validate()` function calculates the tag hash of the received element (`receivedEl`) and attempts to find this hash value among the recorded element knowledge of the requested URL. If it finds a match, it labels the received element as benign. As discussed in the previous section, the inline dynamic scripts cannot be validated with this method. Any script that could not be matched by the hash comparison is treated as a potential inline dynamic script. For the potential inline dynamic scripts, the Filter calculates the modified ASTs (described in Section 4.1) and makes an AST comparison. If this script's modified AST is observed during the Learning Phase, the script is labeled as benign and dynamic (via `isDynamic` that will be utilized for compliance). Finally, the Filter removes the elements that are not labeled as benign from the response.

```
def validate(receivedEl, [recordedEls]):
    receivedEl.label = 'non-benign'
    if receivedEl.compare_hash([recordedEls]):
        receivedEl.label = 'benign'
    else if receivedEl.isInlineScript():
        if receivedEl.compare_AST([recordedEls]):
            receivedEl.label = 'benign'
            receivedEl.isDynamic = True
    if receivedEl.label = 'non-benign':
        receivedEl.remove()
```

Listing 1.2: *Simplified code of the `validate` function used by the Rewriting phase. (`receivedEl` refers to the received element in the server response.)*

Ensuring Compliance During the Rewriting phase, we make necessary modifications to the remaining elements to ensure compliance with the safe CSP by executing the `comply()` function (Listing 1.3) for each element. The Filter receives the pre-included elements in the server response. Hence, it can only modify pre-included elements and not runtime-included elements. We leverage the Mutation Observer to perform required modifications during runtime. As discussed in Section 4.1, inline event handlers and inline dynamic scripts have to be converted to non-inline to comply with the C1 policy so that they are not blocked. To this end, the Filter runs the `comply()` function on these pre-included elements to perform the conversion. The `comply()` function also adds a nonce attribute to the validated pre-included elements for compliance with the C2 policy. To handle the runtime-generated elements with inline events, the Filter attaches the Mutation Observer to the server response as a non-inline script. The Mutation Observer validates and converts runtime-generated elements with inline events to non-inline elements. The script content included during runtime by parent third-party scripts is handled in the same way by the stages shown in Table 1.

```
def comply(receivedEl):
    if receivedEl.isEvent() or receivedEl.isDynamic:
        receivedEl.convert_to_noninline()
    receivedEl.nonce = csp_nonce
```

Listing 1.3: *Simplified code of the* `comply` *function used by the Rewriting phase.*

5 System Implementation

5.1 Learning Phase: Crawler

To populate a sitemap for a Web application, we utilize the following three methods: i) Python Crawler based on an existing library for sitemap generation (`python-sitemap` [4]), ii) Burp Scanner [3] and iii) Mapper that maps server-side files to URLs. The Python Crawler and the Mapper run automatically when the Learning Phase is executed. We run the Burp Scanner manually since Burp does not offer an API. Then, we combine the output sitemaps of these methods and remove duplicate URLs (i.e., exact matches of full URLs). The combination of these three methods gives us better coverage of the target website compared to using only one of them. In Appendix Section E, we support this by presenting the contribution (in number of URLs) of the three methods to the sitemaps of the 6 Web applications that we experiment on in Section 6.1.

The Python Crawler works by starting at a designated seed URL, and then following all links on that page to other pages. It only follows the links that have the same domain as the seed URL (including the ones with relative paths) since we focus on the script content of only the target website.

During crawling, Burp Scanner navigates around the application. It follows links, submits forms, and logs in where necessary, to catalog the application's navigational paths. Burp Scanner goes beyond what the Python Crawler covers

by modifying URL parameters and making POST requests. We utilize the Scanner feature of Burp Suite Professional v2023.3.3. Burp provides various modes, each of which presents different trade-offs between coverage and speed. We employed the mode with the greatest coverage (i.e., *mostcomplete*) and configured the time limit to 60 minutes and the crawl depth to 10. Azad et al. [26] evaluated the coverage offered by BurpSuite Spider v2.0.14beta on four open source Web applications. They used additional three methods (Tutorials, Monkey Testing and Vulnerability Scanner) and compared the results. The Spider outperformed the other methods by discovering 94%, 90%, 83% and 68% of all the discovered pages for the four applications.

Mapper runs a command on the top directory on the server-side, which searches for all files with a certain extension (e.g., `.php` for PHP-based and `.pl` for Perl-based) in the current directory and its subdirectories. For example, for PHP-based applications, the mapper executes the following command: `find . -type f -name "*.php"`. It uses the output of this command to generate links for a target Web application. These links are generated by concatenating the seed URL and the relative location of the files.

5.2 Learning Phase: CSP Generator

Crafting the safe CSP The CSP Generator leverages `selenium` [17] to visit each URL in the sitemap and poll the HTML of the page every 0.2 seconds. We determined this (configurable) number specific to our setup with an average page load time of 0.2-0.4 seconds. It declares the page loaded upon detecting that the HTML content has remained unchanged for the past three consecutive checks. This way, it allows runtime DOM modifications such as runtime script generation to complete. As discussed in Section 3.2, the CSP Generator is designed to generate a safe CSP, as shown in Listing 1.1. In our CSP policy configuration, two CSPs, a CSP C1 and a CSP C2, are set and enforced at the same time. The CSP Generator creates a single CSP C1 for the entire website by progressively adding the source URLs and hashes of the scripts encountered in each page visited, to the C1 policy (Listing 1.1 line 1). We cryptographically generate the nonce value in the C2 policy (Listing 1.1 line 2) on the server side by leveraging the `mod_cspnonce` Apache module. Our implementation enables the nonce value to be changed with each request of a page, thereby protecting the website against nonce reuse attacks. Note that, although `unsafe-eval` is not present by default in our generated safe CSP, it can be enabled if necessary.

Building the Element Knowledge The CSP Generator records the element knowledge in the database separately for each URL present in the sitemap. When the Filter receives a server response after a client request, it fetches the element knowledge only for that specific page. This way, EasyCSPeasy limits the potential for an attack in which an adversary injects a developer-intended script from one page of a website to another page, aiming to induce malicious behavior. Additionally, this method decreases the performance overhead caused

by the hash comparison (Listing 1.2 line 3) and AST comparison (Listing 1.2 line 6) functions during the validation process. Several HTML attributes might change during runtime (such as `width` and `height`) which will affect the value of `tag_hash`. To avoid this, before calculating the hash, EasyCSPeasy removes the element attributes except for the inline event attributes (e.g., `onclick`) and the `src` attribute if present. The values of the removed attributes cannot be interpreted as JavaScript code and cannot be executed. Hence, removing those attributes does not have any security implications that concern CSP generation.

Dynamic Scripts To validate the inline dynamic scripts in the Rewriting phase, the CSP Generator first obtains the ASTs of these scripts by using the Esprima [12] parser. It then modifies the ASTs so that they do not contain any dynamic values and adds the modified ASTs (`script_ast`) to the element knowledge. According to the documentation [13], the `Literal` data type is assigned to the lexical elements that have one of the following value types: `boolean`, `number`, `string`, `RegExp` or `null`. We modify the generated AST by removing the values of any `Literal` types seen in the tree. Among the `Literal` value types, `string` and `RegExp` may be potentially dangerous if they are generated by processing user input without any sanitization. However, these string literals must be interpreted as code, to be dangerous. That may only be accomplished through an eval-like function. Since EasyCSPeasy sets a safe CSP where `unsafe-eval` is disabled, this potential vulnerability is eliminated.

5.3 Rewriting Phase

Validating the Elements in the Server Response To validate the received JS-including elements, the Filter retrieves the element knowledge for the requested URL and executes the `validate()` function. This function ensures that only developer-intended elements are present in the rewritten response. As discussed in Section 4.1, a server may generate a new page or new content based on the GET and POST parameters in a URL. For example, PhpBB generates new separate pages for each new topic. The URLs of these pages (topic URL) contain the topic's ID number. The topic URLs have the following form: `phpbb.com/viewtopic.php?t=<topic_id>`. If a new topic URL is created after running the Learning Phase, this URL will not be present in our sitemap and its element knowledge will not be recorded in the database.

To mitigate this issue, the Filter follows the algorithm shown in Listing 1.4. This function proceeds sequentially through the following steps, advancing to the next step only if the preceding one fails: i) attempt to match the full URL (i.e., the URL with its query parameters if present) with the recorded URLs (`db_urls`), ii) find the recorded URL that has the same path and query parameters, and the most matching parameter values, as the received URL (`max_params()`), iii) match the URL to the special regex pattern (may be provided by the Web developer) and iv) match the URL without the query string (via `remove_qs()`). With this algorithm, if the full URL has not been observed during the Learning Phase, the Filter can find a best effort match for a received

URL. For the above PhpBB example, the Filter can fetch a previously created topic page's element knowledge for a newly created topic page. We note that this method is reliable when the pages of the two best effort match URLs (such as different PhpBB topics) contain script content that is interpreted as the same (i.e., may contain inline dynamic scripts with matching ASTs) by EasyCSPeasy. We evaluate our algorithm in Section 6.1.

```
1  def best_effort_match(received_url, db_urls):
2      best_match = None
3      if received_url in db_urls:
4          return received_url
5      best_match = max_params(parse_qs(received_url),parse_qs(db_urls))
6      if best_match:
7          return best_match
8      best_match = re.match(special_pattern, db_urls)
9      if best_match:
10         return best_match
11     if remove_qs(received_url) in remove_qs(db_urls):
12         return remove_qs(received_url)
```

Listing 1.4: *The pseudo-code algorithm to find the closest URL in the database to the received URL. (parse_qs() parses the query string.)*

Ensuring Compliance with the safe CSP The Filter converts the inline event handlers to non-inline elements using the `comply()` function. For instance, it alters the inline element depicted in Listing 1.5, transforming it into the non-inline format shown in Listing 1.6. It fetches the non-inline `event_listener_script` from the event element knowledge and creates the `inline_event_secureN.js` file (`secureN`=`event_id`). Then, it appends a non-inline script element pointing to the location of `inline_event_secureN.js` (Listing 1.6 line 2) to the HTML response. This script can locate the HTML element to register the event listener by leveraging the `event_id` attribute (Listing 1.7).

The `comply()` function also converts inline dynamic scripts to non-inline scripts as shown in Listing 1.8. The `script_id` is leveraged to assign a location to this script by naming the script file as `inline_js_secureN.js` (`secureN`=`script_id`). Both scripts located at `inline_event_secureN.js` and `inline_js_secureN.js` would be allowed to execute due to the presence of the `self` keyword in the C1 policy, as the scripts are located in the same origin as the seed URL. An adversary may attempt to retrieve the script files to gain access to any sensitive information they might contain. To mitigate this, we generate a cryptographically secure random number (`secureN`) and use it to assign the file locations. This prevents an attacker from guessing the file names to accomplish unauthorized access.

The elements that are generated during runtime can manifest in two ways, either through automatic generation within the browser upon a user's visit to the page or through generation following user interaction with the page. For EasyCSPeasy to capture the elements generated only after user interaction, the necessary user interactions should be simulated during the Learning phase. This limitation could be addressed by the Web developer utilizing a targeted testbed that exercises user interactions. The Filter also generates the script files

(`inline_event_secureN.js`) shown in Listing 1.6 to allow the Web browser to load these scripts during runtime. The final rewritten response which is compliant with the safe CSP is then served to the client.

```
1 <button onclick="alert('clicked')">X</button>
```

Listing 1.5: *An example HTML element with an inline event handler.*

```
1 <button event_id="secureN">X</button>
2 <script src='./inline_event_secureN.js'></script>
```

Listing 1.6: *An example conversion of an HTML element with an inline event handler.*

```
1 document.querySelector("[event_id='secureN']").addEventListener("onclick",
      function(){alert('clicked')});
```

Listing 1.7: *JavaScript code of an inline_event_secureN.js.*

```
1 <script src='./inline_js_secureN.js'></script>
```

Listing 1.8: *An example conversion of a previously inline script.*

Mutation Observer The Mutation Observer script is appended to the page in the Rewriting Phase by the Filter to validate the developer-intended runtime-included inline events and convert them to non-inline elements to comply with the C1 policy. This script observes changes in the DOM during runtime. It detects any runtime-generated inline event handlers. For each of these elements, the Mutation Observer computes the tag hash to validate the element (via `validate()`) and obtain the corresponding `event_id` using the element knowledge. Then, it makes the conversion shown in Listing 1.6. The generated script (Listing 1.6 line 2) inherits the Mutation Observer's nonce, achieving compliance with the C2 policy. The browser then loads the *inline_event_secureN.js* files that were already generated by the Filter. Note that, runtime-included inline and non-inline scripts do not need to be validated since the safe CSP should block any script that is not developer-intended.

5.4 Implementation Details

The CSP Generator appends the CSP header pair to the primary Apache configuration file (*httpd.conf*) as shown in Listing 1.9 lines 4 to 7. The appended CSP is effective for all pages of the specified domain. We implement the Filter as an external output filter (programmed in PHP and Python3). We set this filter by leveraging the Apache module, `mod_ext_filter` and adding the lines 1 to 3 shown in Listing 1.9 to the Apache configuration file. We programmed both the Crawler and the CSP Generator in Python3. We built EasyCSPeasy to work with Apache Web servers. However, our approach can be implemented to work with other Web servers by applying minor changes to our code. Specifically, a developer can modify their Web server's configuration file by using our approach and get the same behavior. For example, in an NGINX Web server, we can simply implement the Filter by setting a `sub_filter` through the `ngx_http_sub_module` module.

```
1 LoadModule cspnonce_module mod_cspnonce.so
2 ExtFilterDefine theFilter intype=text/html mode=output cmd="php filter.php
      <website_dir> <website_url>"
3 SetOutputFilter theFilter
4 <Directory />
5 Header set Content-Security-Policy "<C1>"
6 Header append Content-Security-Policy "<C2>"
7 </Directory>
```

Listing 1.9: *EasyCSPeasy's modification on Apache configuration file*

6 Evaluation

EasyCSPeasy needs to mitigate the aforementioned attacks in Section 3.1, preserve commonly interacted functionalities within any website, and have a low performance overhead. Hence, we evaluate EasyCSPeasy for functionality, security, and performance aspects. Our approach to evaluating EasyCSPeasy is threefold: i) Evaluating functionality on open source Web applications by setting up an EasyCSPeasy working environment, ii) Demonstrating protection on the vulnerable versions of open source Web applications and iii) Measuring the performance impact on open source Web applications.

6.1 Functionality Experiments on Open Source Web Applications

The modifications made by EasyCSPeasy must not disrupt the functionality of a target website. We evaluate our system on open source Web applications that are designed to offer different features. Specifically, we evaluate the following five PHP-based applications: Litecart [21] (eCommerce), PhpBB3 [14] (forum), PhpMyAdmin [15] (database administration), Squirrelmail [18] (email), WordPress [23] (CMS), and one Perl-based application, TWiki [19] (wiki). We aim to evaluate the above Web applications with a comprehensive set of test cases.

Azad et al. [26] prepared Selenium tests to systematically exercise commonly interacted functionalities in older versions of WordPress and PhpMyAdmin. The goal was to ensure that their testing scenarios align with how real users typically engage with these web applications, providing a more realistic and meaningful assessment of their behavior. Hence, we manually replicate their tests to evaluate the latest versions of WordPress and PhpMyAdmin. Since there is no available set of test cases for the other Web applications, we develop our own test cases following the methodology presented in [26] to simulate common Web user interactions. We present the test cases and our results in Table 2. To deploy the Web applications, we leverage an XAMPP [25] setup with PHP version 8, Perl version 5.32, and Apache version 2.4. The evaluated versions of LiteCart, TWiki, and WordPress leverage eval-like functions. To evaluate EasyCSPeasy on those applications, we enable `unsafe-eval` in the CSP.

Test Num.	LiteCart v.2.3.3	PhpBB3 v.3.3.5	PhpMyAdmin v.5.1.3	Squirrelmail v.1.5.2	TWiki v.6.0.0	Wordpress v.4.7.1
1	Login, logout	Login, logout	Login, logout	Login, logout	Login, logout	Login, logout
2	Add to cart	View forum	Create database	View email	Search	New post (c)
3	View cart	View existing topic	Create, browse, drop, optimize table	Send email	Create topic (c)	Theme
4	Increase, decrease quantity in cart	Create topic	Input data to table	Delete email	Edit topic	Setting
5	Remove product from cart	Preview topic and reply	Run SQL query	Search functionality	Attach file	Category tag
6	Change regional settings	Post reply	Create index	Flag, unflag email	Rename topic	Comment
7	Edit account	Send, receive private messages	Export backup		Set new topic parent	Export
8	Search product	Search functionality	Add user		Delete topic	User
9	View order history		Check variables, charsets, engines			Media
10	Send message to customer service		Import SQL			

Table 2: *Functionality Experiment Results on Open Source Web Applications. ((c): Conditional pass)*

Functionality Experiment Results In Table 2, we show that out of 51 test cases executed on the six Web applications, 49 passed, 2 conditionally passed (i.e., after providing a special pattern to the `best_effort_match` algorithm shown in Listing 1.4) and none of the test cases failed. We observe similar occurrences as the PhpBB example discussed in Section 5.3 during our evaluation of PhpMyAdmin and PhpBB. By using the algorithm presented in Listing 1.4, EasyCSPeasy successfully executed 4 tests for PhpBB (case 4 - 7) and 6 tests for PhpMyAdmin (case 2 - 7) that otherwise would have failed.

While executing two of the test cases (Twiki case 3 and WordPress case 2), we observe that new URLs that are unseen during the Learning Phase are introduced to the Web application instances. When a user creates a post, WordPress generates a page for the post with the URL of the following form: `wordpress.com/<year>/<month>/<day>/<post_name>`. Similarly, for new topics, Twiki generates a URL of the following form: `twiki.com/bin/edit/Main/<topic_name>`. Hence, we craft one special pattern for each WordPress and Twiki to match a new post or topic URL to any post or topic URL seen during the Learning Phase. The special pattern for WordPress is the following: `wordpress.com/(\d{4})/(\d{1,2})/(\d{1,2})/(.+)`. After we provide the special patterns, both test cases pass. We label those cases as conditionally passed. We create multiple topics on each Web application and observe that for both WordPress and Twiki, any topic page within each application contains the same script content (i.e., interpreted as the same by EasyCSPeasy). We note that any special pattern may be configured in `config.json` by the Web developer. Moreover, Web developers may optionally disable the strict "page-based script matching" described in Section 4 instead of adding a special pattern (forgoing the added protection). When this option is disabled, our system will not attempt to find a URL match. It will fetch all recorded element knowledge in the database instead of the specific page's element knowledge relieving the need for special pattern construction. We conclude that by default EasyCSPeasy mostly retains the commonly interacted functionalities of the Web applications. When special patterns are provided, EasyCSPeasy retains all functionalities.

To demonstrate EasyCSPeasy's successful script content rewriting, we need to show that a target Web application actually executes scripts. We utilize CSP's report-only mode (CSP-RO) which allows the execution of script content vio-

lating the set policy while reporting such violations. The reports contain the inline script content (i.e., inline scripts and event handlers) and non-inline script content (i.e., scripts loaded via `src` attribute) that would be blocked by the set policy. To quantify the script content, first, for each Web application, we set the CSP-RO as `script-src 'none'` so that all script content will be reported as a violation allowing us to log them. Then, we run all test cases in the experiment setup described above without integrating EasyCSPeasy. This way we log the script content executed when the test cases are exercised. We show that the number of executed scripts, detailed in Appendix Section F, peaks at 91 on a single page.

6.2 Security Experiments on Open Source Web Applications

To demonstrate that EasyCSPeasy can help block all 3 types of known XSS attacks, we set up vulnerable versions of 3 open source Web applications (LiteCart, WordPress, and MyBB). These applications contain disclosed vulnerabilities that have been assigned CVEs [16]. We leveraged XAMPP with Apache version 2.4. For LiteCart and MyBB, we used PHP version 5.6 and for WordPress, we used PHP version 8. After setting up, we successfully reproduce the disclosed attacks on these Web applications. Then, we integrate EasyCSPeasy to the applications and attempt to reproduce the same attacks. Upon the integration of our system, all attack attempts fail. Hence, we conclude that EasyCSPeasy is able to protect against the 3 types of XSS as promised.

Reflected XSS Prevention (Case Study: LiteCart) LiteCart versions before 1.3.3 are found to have multiple reflected XSS vulnerabilities in `search.php` (CVE-2014-7183 [9]). This vulnerability is caused by the Web application taking the user input inside the search box and reflecting it directly into the title HTML tag without any sanitization. We deploy LiteCart version 1.3.2 in our experiment environment and craft two separate URLs to reproduce two reflected XSS attacks. The first attack URL, shown in Listing 1.10 line 1, contains an inline script as the malicious payload. The second URL, shown in Listing 1.10 line 2, contains a link tag with an `onerror` inline event handler. We execute both attacks successfully on a default LiteCart setup. After that, we deploy EasyCSPeasy on our LiteCart setup. Our system generates a safe CSP specifically crafted for LiteCart. We observe that the first attack is unsuccessful, as the generated C1 policy does not contain the hash of the injected inline script in Listing 1.10 line 1. Similarly, the second attack in Listing 1.10 line 2 fails, since EasyCSPeasy disables the dangerous `unsafe-inline` keyword.

```
1 http://litecart.net/en/search?query=";></title><script>alert(1)</script>
2 http://litecart.net/en/search?query=";></title><link rel='stylesheet' type
      ='text/css' href='blah' onerror=alert(1)></link>
```

Listing 1.10: *Reflected XSS payloads*

Stored XSS Prevention (Case Study: WordPress) WordPress features a plugin architecture where one can integrate a plugin into their WordPress setup. A stored XSS in the Absolutely Glamorous Custom Admin plugin (versions <= 6.8) has been disclosed in CVE-2021-36823 [11]. The stored XSS attack can be executed via unsanitized input fields of the plugin settings. A user can input any one of the three payloads shown in Listing 1.11 line 1 to 3 to execute arbitrary JavaScript. This user input is also stored in the database, so the potentially malicious payload will be served to every user that visits the vulnerable page. After reproducing the stored XSS attack with three different payloads shown in Listing 1.11, we deploy EasyCSPeasy on the WordPress setup. We observe the payloads in Listing 1.11 lines 1 and 2 fail to execute since the CSP C1 generated by EasyCSPeasy does not contain `unsafe-inline`. Similarly, the payload with a non-inline script in Listing 1.11 line 3 is blocked because its source URL is not allowlisted in the CSP C1.

```
1 "><img src=x onerror=alert(1) />
2 "><script>alert(1)</script>
3 "><script src=evil.com/evil.js></script>
```

Listing 1.11: *Stored XSS payloads*

DOM-based XSS Prevention (Case Study: MyBB) MyBB is a free and open-source forum software. In MyBB (version <= 1.8.24), the custom code (BBCode) for the visual editor does not escape input properly when rendering HTML, resulting in a DOM-based XSS vulnerability (CVE-2020-15139 [10]). The weakness can be exploited by pointing a victim to a page where the visual editor is active (e.g., such as for a post or Private Message) and operates on a maliciously crafted BBCode message. We use the payload in Listing 1.12 to exploit this vulnerability in our MyBB setup. After running the attack successfully on a default MyBB setup, we integrate EasyCSPeasy to MyBB and attempt to reproduce the same attack. EasyCSPeasy blocks the DOM-based XSS attack attempt as expected since it disables the `unsafe-inline` compatibility mode.

```
1 [size=10;"><img/onerror='alert(1)'/src=1/>]PoC[/size]
```

Listing 1.12: *DOM-based XSS payload*

6.3 Performance Evaluation

The Learning Phase of EasyCSPeasy has to be executed only once by the site-operator, whereas the Rewriting Phase runs on every page request and may affect the user experience. Therefore, we evaluate EasyCSPeasy's performance impact by measuring the performance of the Rewriting Phase. We conducted experiments on each of the six Web applications shown in Table 2. We integrate EasyCSPeasy to the Web application environments we created for functionality experiments. To profile our code we leverage the `cProfile` python library which shows the execution time for each individual function. We visit ten pages (of the

most commonly interacted pages) per application and profile EasyCSPeasy. We show the key results in Table 3. EasyCSPeasy introduces a total median overhead of 53 ms which is under the 100 ms limit for having the user feel that the system is reacting instantaneously [40]. The median overhead corresponds to 9.6% (53ms) of the total page load time (550ms), where 15 ms is attributed to the function that fetches the element knowledge from the database and 35 ms is attributed to the HTML parsing functions. We observe the maximum overhead case on a WordPress page where the total overhead introduced is 429ms which corresponds to 11.9% of the total page load time (3.6s). The main difference between the median and the maximum cases, which is causing the additional overhead, is the increased execution time of the AST operations. The WordPress page where we observe the maximum overhead contains 8 long scripts with dynamic variables. Collectively these scripts contain 319,708 characters. While generating the ASTs, the Esprima library works on each character one by one. The high number of characters in the maximum overhead case compared to the other cases is the main reason for the increased overhead.

Execution Time (ms)			**Script Content**		
Type	**Median**	**Max**	**Type**	**Median**	**Max**
Database Fetch	15 (2.7%)	15 (0.4%)	**Inline Event**	2	1
HTML Parse	35 (6.4%)	12 (0.3%)	**Inline Script**	3	75
AST Operations	3 (0.5%)	402 (11.2%)	**Dynamic Script**	3	8
Total Overhead	53 (9.6%)	429 (11.9%)	**Non-inline Script**	7	91
Total Page Load	550 (100%)	3600 (100%)			

Table 3: *Performance Overhead Experiment Results.*

6.4 Limitations

EasyCSPeasy has two limitations: i) the generic crawler of our system may not generate complete sitemaps and ii) runtime-generated script content that is only generated after user interaction has to be explicitly triggered during the Learning phase. However, both of these limitations may be overcome by utilizing application-specific testbeds. Testbeds are commonly used for testing an application after code changes. A targeted testbed is likely to contain the sitemap to visit the pages of the application and to simulate user interactions.

7 Conclusion

In conclusion, we propose EasyCSPeasy, a server-side, standalone XSS mitigation system that helps Web developers adopt CSP by generating a safe CSP for any target website. EasyCSPeasy also ensures compliance with the set CSP by rewriting the server response. We evaluate and show our system can successfully protect against all three types of XSS. In addition to offered mitigation, EasyCSPeasy does not interfere with the most commonly interacted website functionalities and introduces a minimal performance overhead.

Acknowledgement

This paper was supported by a seed grant from the Center for Information & Systems Engineering (CISE) at Boston University and by the NSF under grants CNS-2127232 and CNS-2211576.

References

1. A Research-Oriented Top Sites Ranking Hardened Against Manipulation. https://tranco-list.eu/
2. Abstract Syntax Trees. https://docs.python.org/3/library/ast.html
3. Burp Scanner. https://portswigger.net/burp/vulnerability-scanner
4. C4software/python-sitemap: Mini website crawler to make sitemap from a website. https://github.com/c4software/python-sitemap
5. Content-Security-Policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy
6. Content security policy level 3. https://www.w3.org/TR/CSP3/
7. Cross site scripting (XSS). https://owasp.org/www-community/attacks/xss/
8. CSP:script-src - HTTP—MDN. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src
9. CVE-2014-7183. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-7183
10. CVE-2020-15139. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-15139
11. CVE-2021-36823. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-36823
12. Esprima. https://esprima.org/
13. Esprima: Parser. https://docs.esprima.org/en/latest/syntax-tree-format.html
14. PhpBB - Free and Open Source Forum Software. https://www.phpbb.com/
15. PhpMyAdmin. https://www.phpmyadmin.net/
16. Search CVE list. https://cve.mitre.org/cve/search_cve_list.html
17. Selenium. https://selenium.dev
18. Squirrelmail - Webmail for nuts! https://squirrelmail.org/
19. The Open Source Enterprise Wiki and Web Application Platform. https://twiki.org/
20. Using a nonce with CSP. https://content-security-policy.com/nonce/
21. Websites using LiteCart. https://trends.builtwith.com/websitelist/LiteCart
22. What is a Sitemap? https://developers.google.com/search/docs/crawling-indexing/sitemaps/overview
23. Wordpress.com: Built a Site, Sell Your Stuff, Start a Blog & More. https://squirrelmail.org/
24. World Wide Web Consortium (W3C). https://www.w3.org/
25. XAMPP Installers and Downloaders for Apache Friends. https://www.apachefriends.org/index.html
26. Azad, B.A., Laperdrix, P., Nikiforakis, N.: Less is more: Quantifying the security benefits of debloating web applications. In: Proceedings of USENIX Security Symposium. Santa Clara, CA, USA (Aug 2019)

27. Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Saner: Composing static and dynamic analysis to validate sanitization in web applications. In: Proceedings of IEEE Symposium on Security and Privacy (S&P). Oakland, CA, USA (May 2008)
28. Bisht, P., Venkatakrishnan, V.N.: Xss-guard: Precise dynamic prevention of cross-site scripting attacks. In: Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA). Paris, France (Jul 2008)
29. Calzavara, S., Rabitti, A., Bugliesi, M.: Content security problems? In: Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS). Vienna, Austria (Oct 2016)
30. Calzavara, S., Rabitti, A., Bugliesi, M.: Ccsp: Controlled relaxation of content security policies by runtime policy composition. In: Proceedings of USENIX Security Symposium. Vancouver, BC, Canada (Aug 2017)
31. Cui, Y., Cui, J., Hu, J.: A survey on xss attack detection and prevention in web applications. In: Proceedings of the International Conference on Machine Learning and Computing (ICMLC). Shenzhen, China (Feb 2020)
32. Doupé, A., Cavedon, L., Kruegel, C., Vigna, G.: Enemy of the state: A State-Aware Black-Box web vulnerability scanner. In: Proceedings of USENIX Security Symposium. Bellevue, WA, USA (Aug 2012)
33. Doupé, A., Cui, W., Jakubowski, M.H., Peinado, M., Kruegel, C., Vigna, G.: Dedacota: Toward preventing server-side xss via automatic code and data separation. In: Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS). Berlin, Germany (Nov 2013)
34. Fazzini, M., Saxena, P., Orso, A.: Autocsp: Automatically retrofitting csp to web applications. In: Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE). Florence, Italy (May 2015)
35. Huang, Y.W., Huang, S.K., Lin, T.P., Tsai, C.H.: Web application security assessment by fault injection and behavior monitoring. In: Proceedings of the International Conference on World Wide Web (WWW). Budapest, Hungary (May 2003)
36. Lekies, S., Kotowicz, K., Groß, S., Vela Nava, E.A., Johns, M.: Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets. In: Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS). Dallas, TX, USA (Oct 2017)
37. Lekies, S., Stock, B., Wentzel, M., Johns, M.: The unexpected dangers of dynamic JavaScript. In: Proceedings of USENIX Security Symposium. Washington, D.C., USA (Aug 2015)
38. Louw, M.T., Venkatakrishnan, V.: Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In: Proceedings of IEEE Symposium on Security and Privacy (S&P). Oakland, CA, USA (May 2009)
39. Meyerovich, L.A., Livshits, B.: Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In: Proceedings of IEEE Symposium on Security and Privacy (S&P). Oakland, CA, USA (May 2010)
40. Nielsen, J.: Usability engineering. Kaufmann (2009)
41. Pan, X., Cao, Y., Liu, S., Zhou, Y., Chen, Y., Zhou, T.: Cspautogen: Black-box enforcement of content security policy upon real-world websites. In: Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS). Vienna, Austria (Oct 2016)

42. Roth, S., Barron, T., Calzavara, S., Nikiforakis, N., Stock, B.: Complex security policy? a longitudinal analysis of deployed content security policies. In: Proceedings of Network and Distributed System Security Symposium (NDSS). San Diego, CA, USA (Feb 2020)
43. Saiedian, H., Broyle, D.: Security vulnerabilities in the same-origin policy: Implications and alternatives. IEEE Computer (2011)
44. Samuel, M., Saxena, P., Song, D.: Context-sensitive auto-sanitization in web templating languages using type qualifiers. In: Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS). Chicago, IL, USA (Oct 2011)
45. Saxena, P., Molnar, D., Livshits, B.: Scriptgard. In: Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS). Chicago, IL, USA (Oct 2011)
46. Steffens, M., Musch, M., Johns, M., Stock, B.: Whos hosting the block party? studying third-party blockage of csp and sri. In: Proceedings of Network and Distributed System Security Symposium (NDSS). San Diego, CA, USA (Feb 2021)
47. Weichselbaum, L., Spagnuolo, M., Lekies, S., Janc, A.: Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In: Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS). Vienna, Austria (Oct 2016)
48. Weissbacher, M., Lauinger, T., Robertson, W.: Why is csp failing? trends and challenges in csp adoption. In: Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID). Gothenburg, Sweden (Sep 2014)
49. Xu, G., Xie, X., Huang, S., Zhang, J., Pan, L., Lou, W., Liang, K.: Jscsp: A novel policy-based xss defense mechanism for browsers. IEEE Transactions on Dependable and Secure Computing (2020)

A Type 1 Script Gadgets

This type of gadget can be used to trigger new script element creation with potentially malicious code. We show an example script gadget present in a website's HTML in Listing 1.13 lines 2 to 5. This gadget finds all `button` elements in a page and assigns their `data-text` attribute to their `innerHTML` (the text shown on a button). An attacker can inject a `button` element to this page and assign their malicious script as the value of the button's `data-text` attribute. We show an example of injected HTML to trigger malicious script inclusion in Listing 1.14. When the gadget is triggered, it creates a new script element with the attacker's malicious code (Listing 1.15).

```
1 <button data-text='mybutton'></button>
2 <script nonce='r4nd0m'>
3 bts=document.getElementsByTagName('button');
4 bts=Array.from(bts);
5 bts.forEach(element => element.innerHTML=element.getAttribute('data-text')
      ); </script>
```

Listing 1.13: *An example Script Gadget present in a website's HTML*

```
1 <button data-text="<script src='evil.com/evil.js'></script>"></button>
```

Listing 1.14: *An example attacker injected HTML*

```
1 <button data-text='...'>
2 <script src='evil.com/evil.js'>
3 </script></button>
```

Listing 1.15: *The final HTML*

B CSP Configurations: C1 and C2

C1 policy - Allowlist-based This configuration contains a list of allowed script source URLs and a list of hashes to allow individual inline script execution. The CSP C1 configuration has the form shown in Listing 1.16 line 1. Previous work found that allowlists are ineffective and bypassable by showing that they can potentially include unsafe endpoints (destination URLs) that contain vulnerabilities (such as JSONP and Open Redirects) which may be leveraged to perform XSS attacks [47]. To preserve all functionality in a website, the site-operator has to build a complete URL and hash list for the C1 policy. Additionally, configurations based on allowlists can be challenging to maintain, as websites may regularly add or remove scripts. As a result, the allowlist must be continuously updated to reflect these changes. Manually building and maintaining such comprehensive lists can be a lengthy and error-prone process (e.g., the site-operator might forget to allowlist some scripts or make a typo in the allowlist), especially for complex websites [42]. Hence, we argue that devising a C1 policy should be done automatically as has been done before [34].

```
1 script-src <allowed_urls> <allowed_hashes>
```

Listing 1.16: *CSP C1*

C2 policy - Nonce-based, `strict-dynamic` *enabled* This configuration contains a randomly generated nonce value which will only allow execution if a script has the same nonce value as the one listed in the CSP. The CSP C2 configuration has the form shown in Listing 1.17 line 2. Runtime script generation is a widespread practice on the Web. The presence of `strict-dynamic` makes this policy compatible with websites that perform runtime script generation since these scripts would not carry a valid nonce value and would otherwise be blocked by the CSP. However, because of this, a C2 policy will leave a page vulnerable to Type 1 Script Gadget attacks. When implementing a C2 policy, maintaining functionality requires site operators to assign a valid nonce value in each server-generated script. This process should be automatized since it may be time-consuming and error-prone.

```
1 script-src 'nonce-<value>' 'strict-dynamic'
```

Listing 1.17: *CSP C2*

C SMURF Experiments

In Table 4, we show the results of the experiments done by Steffens et al. [46] and our results that we obtain by rerunning their experiments. Specifically, we compare the two results to show that the `eval` usage went down by 18.3%. We aim to gauge the shift in `eval` usage among leading websites. To achieve this, we measure `eval` usage among today's most popular websites rather than those that held that status in January 2020 (Tranco list used in [46]). We consider this approach more representative of current `eval` usage trends. We used the March 28, 2022 version of the Tranco list.

		Inline Event Handler Writing Hosts	Inline Script Writing Hosts	Eval Using Hosts	Non-programmatic Script Including Hosts
SMURF experiments by Steffens et al. [46]	**Self OR third-party OR (self AND third-party)**	85.5%	95%	**78.8%**	N/A
	Only third-party	23.7%	0.4%	23.8%	N/A
	Only self	11.2%	50.6%	17.7%	N/A
	Only third-party OR (self AND third-party)	74.3%	44.8%	61%	17.6%
Our SMURF experiments	**Self OR third-party OR (self AND third-party)**	82%	97.7%	**60.5%**	N/A
	Only third-party	35.4%	9%	30%	N/A
	Only self	12.3%	40.6%	12.3%	N/A
	Only third-party OR (self AND third-party)	69.6%	57.1%	48.2%	12%

Table 4: *Original and Our SMURF experiments. Cells highlighted to show that* `eval` *usage is significantly decreased.*

D Comparison with Related Work

In Table 5, we compare existing systems to EasyCSPeasy on the design goals presented below and show that EasyCSPeasy is the only system that accomplishes all of these goals. We compose the design goals of our system by addressing the shortcomings of previous work [34,30,49,33,41], regarding their offered protection against XSS, genericness, applicability to real-world websites, and the potential introduction of additional security considerations. We present our design goals in the following:

Standalone: A standalone system performs its function without requiring any third party. Our system should be standalone so that a site-operator may easily integrate it with their Web application.

Generates a Safe CSP: Our system should devise a safe CSP as explained in Section 3.2.

Language Agnostic: To have higher real-world applicability, our system should allow integration with Web applications programmed in any back-end language.

Supports Dynamic Websites: Dynamic websites may generate JavaScript on the fly via server-side scripting. Adopting a dynamic approach is a prevalent practice among site-operators [37]. Hence, being incompatible with those sites would significantly decrease the number of potential websites that can benefit from the system. Our system should support dynamic websites.

No `unsafe-inline`: The dangerous `unsafe-inline` compatibility mode should be disabled in the CSP and our system should modify the webpages so that the script content is compliant with this CSP.

Page-based script matching: To prevent unwanted or malicious behavior, we should only allow execution of a script within the context of its developer-intended page.

	AutoCSP [34]	CCSP [30]	CSPAutoGen [41]	deDacota [33]	JSCSP [49]	EasyCSPeasy
Standalone	Yes	No	Yes	Yes	No	Yes
Generates a Safe CSP	No	No	No	No	No	Yes
Language Agnostic	No	Yes	Yes	No	Yes	Yes
Supports Dynamic Websites	No	Yes	Yes	No	No	Yes
No `unsafe-inline`	No	No	Yes	No	No	Yes
Page-based Script Matching	No	No	No	No	Yes	Yes

Table 5: *Comparison with Related Work on the Design Goals.*

E Obtained Sitemap Sizes

We show the number of URLs contributed to the sitemaps by each of the three methods for the six Web applications in Figure 2. Burp Scanner did not terminate on PhpBB3 because of an everchanging session ID URL parameter called `sid`. Hence, we show the contribution of Burp on Figure 2 (b) as zero. We successfully configured our Python crawler to ignore the `sid` parameter.

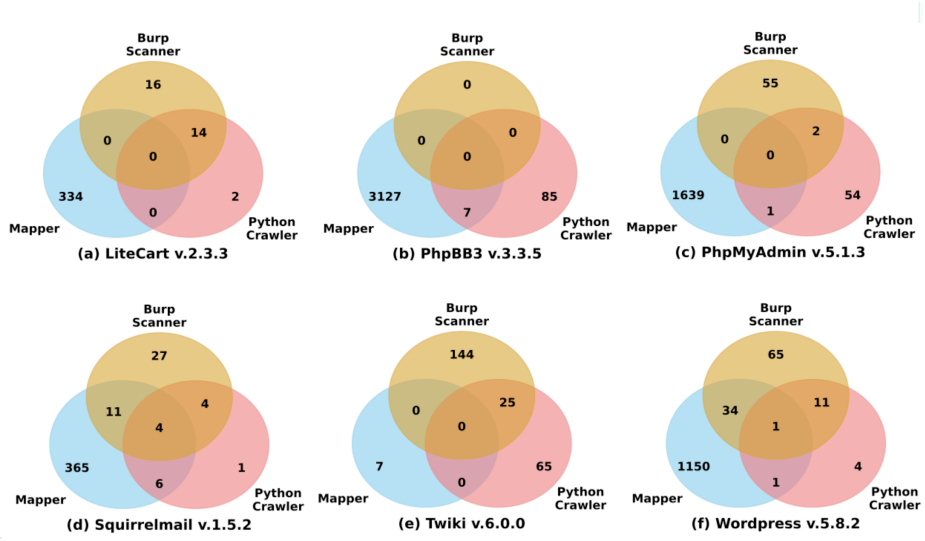

Fig. 2: *Venn diagrams showing the number of URLs contributed to the sitemaps by each of the three methods.*

F Script Content of Evaluated Web Applications

In Table 6, we show the number of scripts executed by each evaluated Web application.

	Type	LiteCart v.2.3.3	PhpBB3 v.3.3.5	PhpMyAdmin v.5.1.3	Squirrelmail v.1.5.2	TWiki v.6.0.0	Wordpress v.4.7.1
Median	**Inline**	2	2	1	7	6	12
	Non-inline	3	4	2	1	50	12
Max	**Inline**	5	5	5	11	9	84
	Non-inline	3	13	37	3	74	91

Table 6: *Test Case Script Content Execution. The numbers represent the number of scripts executed by the Web application.*