# A Measurement Study of the Content Security Policy on Real-World Applications

Kailas Patil[1] and Braun Frederik[2]

*(Corresponding author: Kailas Patil)*

Department of Computer Engineering, Vishwakarma Institute of Information Technology[1]

Survey No: 2/3/4, Kondhwa, Pune-48, Maharashtra, India

(Email: kailas.patil@viit.ac.in)

Mozilla Corporation[2]

331 E. Evelyn Avenue, Mountain View, CA, 94041, USA

*(Received Aug. 03, 2014; revised and accepted Jan. 16 & May 05, 2015)*

## Abstract

Content Security Policy (CSP) is a browser security mechanism that aims to protect websites from content injection attacks. To adopt CSP, website developers need to manually compile a list of allowed content sources. Nearly all websites require modifications to comply with CSP's default behavior, which blocks inline scripts and the use of the *eval()* function. Alternatively, websites could adopt a policy that allows the use of this unsafe functionality, but this opens up potential attack vectors. In this paper, our measurements on a large corpus of web applications provide a key insight on the amount of efforts web developers required to adapt to CSP. Our results also identified errors in CSP policies that are set by website developers on their websites. To address these issues and make adoption of CSP easier and error free, we implemented UserCSP a tool as a Firefox extension. The UserCSP uses dynamic analysis to automatically infer CSP policies, facilitates testing, and gives savvy users the authority to enforce client-side policies on websites.

*Keywords: Content restrictions, content security policy, security policy, web security*

## 1 Introduction

The web browser security model is rooted in the same-origin policy (SOP) [22], which isolates one origin's resources from other origins. However, attackers can subvert the SOP by injecting malicious content into a vulnerable website through attacks such as Cross-Site Scripting (XSS) [3]. According to the OWASP vulnerability assessment in 2013, XSS attacks are among top five vulnerabilities [23]. The root cause of code injection problem on websites is that browsers are unable to distinguish between legitimate and maliciously injected content in a web application. To mitigate threats of XSS attacks, Mozilla proposed Content Security Policy (CSP) [30] a defense-in-depth. CSP has become a part of W3C specification and CSP 1.0 is in the state of Candidate Recommendation [36]. It aims to solve this problem by providing a declarative content restriction policy in an HTTP header that the browser can enforce. CSP defines directives associated with various types of content that allow developers to create whitelists of content sources and instruct client browsers to only load, execute, or render content from those trusted sources. However, writing an effective and comprehensive CSP policy for websites is laborious. A policy can break website functionality if legitimate content is overlooked during policy generation. Web developers at large technology companies may not have direct access to change the CSP header on web servers, making it difficult to iterate over policies. This is hindering the adoption of CSP by real-world web applications as shown by our results in Section 2.

The goal of this paper is to systematically understand the difficulties in adopting a CSP policy by developers, discuss probable ways to bypass CSP protection, and develop a basic understanding of CSP usage in large, real-world web applications today.

**Our Study.** In this work, we study the usage of CSP policy on real-world desktop and mobile websites and identify errors and inconsistency in CSP enforcement. To do this, we used Scrapy framework to crawl real-world websites using various user agent strings to record CSP policies used by websites.

Based on empirical data collected in October 2013, we draw several inferences about the hurdles in CSP adoption. Our results show that there are three major reasons that are hindering CSP adaption. First, developers are unwilling to sacrifice functionality for security because they are worried about losing customers. Second, the limited knowledge of developers about the correct usage of CSP, shows that they have made mistakes while setting CSP policies for their websites. Third, the amount of

efforts required by developers to make their website compatible to CSP is hindering adoption of CSP in the wild.

Moreover, web browsers do not currently expose a policy enforcement mechanism directly to users, and users lack control over their own security when websites do not implement CSP. Security savvy users may prefer security over rich functionality. We argue that, if developers and users do not experiment with CSP, it is difficult for the community to iterate on the CSP specification [37] to come to a more usable solution.

To assist website administrators in constructing Content Security Policies, CSP AiDer [9] uses a crawler to crawl all the pages associated with a website and recommends a CSP policy based on the types of content found and the sources of that content. However, CSP AiDer is unable to recognize dynamically added scripts. It examines static HTML code to infer CSP policy rather than running the website in a web browser to infer policy based on content loaded by the browser.

In this paper, we propose UserCSP, a Firefox extension to address above mentioned problems and ease in CSP adoption. It helps developers and users to derive a CSP policy for a website. UserCSP automatically infers Content Security Policies, providing the strictest possible policies without breaking websites. To infer a CSP policy, UserCSP analyzes the content on a particular page and recommends a policy based on the types and sources of content used. UserCSP provides the inferred policy in the correct syntax for the CSP header, so a developer can immediately start using the policy for their website. Furthermore, UserCSP allows savvy users to voluntarily specify their own CSP policies on websites that may not have implemented CSP. UserCSP is an open-source project available for download on the Mozilla Add-on gallery [25] as well as on GitHub [26].

**Contributions.** The goal in this paper is to study usage of CSP on real-world web applications, aware developers to avoid mistakes observed on real-world web applications. We propose a solution, UserCSP, to ease adoption of CSP policy. The goals of UserCSP are two-fold: i) to allow security savvy users to specify their own CSP policies, and ii) to allow developers to experiment with CSP policies on their production pages. Moreover, UserCSP assists users and developers in constructing comprehensive CSP policies by providing them automatically inferred Content Security Policies that they can use as a starting point for experimenting with CSP on a website.

In summary, this paper makes the following contributions:

- We performed a large-scale study on Alexa Top 100,000 websites and 289 mobile websites to find usage of CSP in the wild.
- We draw inferences on the likely reasons that are hindering CSP adoption in real-world websites.
- We design and prototype UserCSP to automatically generate Content Security Policies and then we evaluate the compatibility of the inferred security policies on websites.
- We propose an approach for applying security policies on the client-side. Our approach allows savvy users to specify their own custom Content Security Policies.

Our experiments show a lack of Content Security Policy implementations in real-world websites and the necessity for tools like UserCSP to help promote adoption. UserCSP provides developers with an easy mechanism to create an effective, comprehensive, and strict Content Security Policy that secures their users and does not break website functionality.

The rest of this paper is organized as follows: Section 2 presents our experimental evaluation and analysis. Section 3 describes the design of UserCSP. Section 4 describes evaluation of our approach, and we conclude the paper in Section 6.

## 2 Experimental Evaluation and Analysis

We conducted empirical measurements to obtain the data for evaluating Content Security Policy (CSP) usage in wild. Our measurements are mainly conducted on a Dell server running Ubuntu 12.04 64bit, with Xeon 4-core 2.67GHz CPUs and 32GB RAM.

### 2.1 Measurement Goals

Our measurements aim to measure the following:

**Goal_1:** Measure inconsistency in real-world websites in enforcing CSP.

**Goal_2:** Identify errors in existing CSP policies applied by developers on their websites that nullify the defense provided by CSP.

**Goal_3:** Estimate the amount of efforts required by web developers to adapt to CSP for their websites.

### 2.2 Measurement over Alexa Top 100,000 Desktop Websites and 289 Mobile Websites

In our experiments, we used Scrapy framework to crawl desktop and mobile websites. Our results show that out of 100,000 Alexa top websites [29] only 27 unique websites are using CSP policies. In particular, only 20 desktop websites actually enforced CSP policies and remaining 07 websites using CSP policy in report-only mode. Similarly, we analyzed 289 mobile websites [1]. In our experiments we noticed only one mobile website `http://mobile.twitter.com/` uses a CSP policy.

We also observed that 24 unique websites are using *unsafe-inline*, *unsafe-eval* or *eval-script* options. According to W3C standard and effective CSP protection on website, it is crucial to move all inline scripts and style sheets to the trusted external sources to allow web browsers to identify injected scripts by an attacker.

Next, we explain how we measure these metrics and present their results.

**Goal_1: Inconsistency in CSP Enforcement.** To measure the inconsistency in enforcing CSP policies in real-world websites, we measured different headers used by developers to send CSP policy to clients. There is an inconsistency in CSP supporting browsers in CSP enforcement headers they obey. Firefox version 4.0 onwards supports *X-Content-Security-Policy* header and Google Chrome and Safari support *X-WebKit-CSP* header for CSP enforcement. Whereas, Firefox doesn't respect *X-WebKit-CSP* header and Google Chrome neglects *X-Content-Security-Policy* header used by websites for CSP enforcement.

Due to this inconsistency across web browsers, Candidate Recommendation of CSP specification of the W3C Working group of Web Application Security proposed a standard header name for CSP enforcement: `Content-Security-Policy`. At the time of writing of this paper, Firefox version 23 and Google Chrome version 25 support *Content-Security-Policy* header.

We scanned in total 100,000 Alexa top desktop websites and 289 mobile websites and checked response for various possible CSP headers such as *X-Content-Security-Policy*, *X-WebKit-CSP*, and *Content-Security-Policy*. Web applications can detect user agents and send appropriate CSP header in the response; therefore, we scanned all websites multiple times by using separate user agent strings [35]. The user agent strings we used are listed in the Table 1.

Figure 1 shows our finding of CSP headers usage on real-world websites. We observed that out of 28 unique websites that are using CSP policies some websites are using multiple headers to set CSP policies. In our experiment we observed,

- One website `http://start.funmoods.com/` used all three headers *X-WebKit-CSP*, *X-Content-Security-Policy*, and *Content-Security-Policy*.
- Three websites namely `http://mega.co.nz/`, `https://github.com/EllisLab/CodeIgniter/wiki`, and `http://lastpass.com/` used both *X-Content-Security-Policy* and *Content-Security-Policy* headers.
- Four websites namely `http://blog.twitter.com/`, `business.twitter.com`, `demo.phpmyadmin.net`, `http://papa.me/` were serving both *X-Content-Security-Policy* and *X-WebKit-CSP*.
- Two websites `http://files.acrobat.com/`, `http://web.tweetdeck.com/` was serving both *X-WebKit-CSP-Report-Only* and *X-Content-Security-Policy-Report-Only* headers.
- One website `http://support.twitter.com/` used both *X-Content-Security-Policy-Report-Only* and *Content-Security-Policy-Report-Only* headers.
- One website `http://hootsuite.com/` used both *X-WebKit-CSP-Report-Only* and *Content-Security-Policy-Report-Only* headers.
- One website `http://mobile.twitter.com/` used both *X-WebKit-CSP* and *X-Content-Security-Policy-Report-Only* headers.

Our results indicate that website developers use custom browser headers as well as CSP header specified by the W3C CSP 1.0 specification. Inconsistency in supporting CSP header across web browsers creates confusion in web developers. As at a time of writing this paper, Chrome and Firefox web browsers supports *Content-Security-Policy* header as per W3C CSP 1.0 specification. We recommend web developers to transition their websites to using the *Content-Security-Policy* header.

Furthermore, inconsistency in CSP directive support at browser level could create confusion among developers while deriving CSP policy for their website. For example, Firefox web browser supports `frame-ancestors` directive in CSP policy whereas it is not in CSP specification and it is not supported by other web browsers. The *frame-ancestors* directive is not a part of the CSP specification because web browsers support *X-Frame-Options* header. But *X-Frame-Options* only checked the parent, and not the grandparent, or great grandparent. However, *frame-ancestors* would check all ancestors. Recently, IE changed their *X-Frame-Options* support so that it checks all ancestors. The other problem with *X-Frame-Options* is that you can only list one URI in *allow-from* whereas, *frame-ancestors* lets you have a list of them. Therefore, the webappsec working group is going to put a *frame-ancestors* like directive (probable name is *frame-options*) into CSP spec [38].

We observed total four (4) websites set *frame-ancestors* directive out of 28 websites that use CSP policies including report-only mode.

**Goal_2: Identify Errors in CSP Policies Enforced by Websites.** We performed an analysis of CSP policies used by developers to protect the users of their websites from content injection attacks. The aim of this study to answer questions such as, Do developers understood how to use CSP policy? And, how many websites enforce CSP policy incorrectly and nullify the defense of CSP mechanism? A summery of our analysis is given below:

- *Incomplete Mediation:* Our empirical study results show that 21 websites are using CSP policy to protect their home pages rather than enforcing it on all internal web pages.

Table 1: A list of user agent strings

| Browser | Version | User Agent String |
|---|---|---|
| Firefox | 23 | Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:23.0) Gecko/20130602 Firefox/23.0 |
| Google Chrome | 29.0.1547.2 | Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/29.0.1547.2 Safari/537.36 |
| Internet Explorer (IE) | 10.6 | Mozilla/5.0 (compatible; MSIE 10.6; Windows NT 6.1; Trident/5.0; InfoPath.2; SLCC1; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; .NET CLR 2.0.50727) 3gpp-gba UNTRUSTED/1.0 |

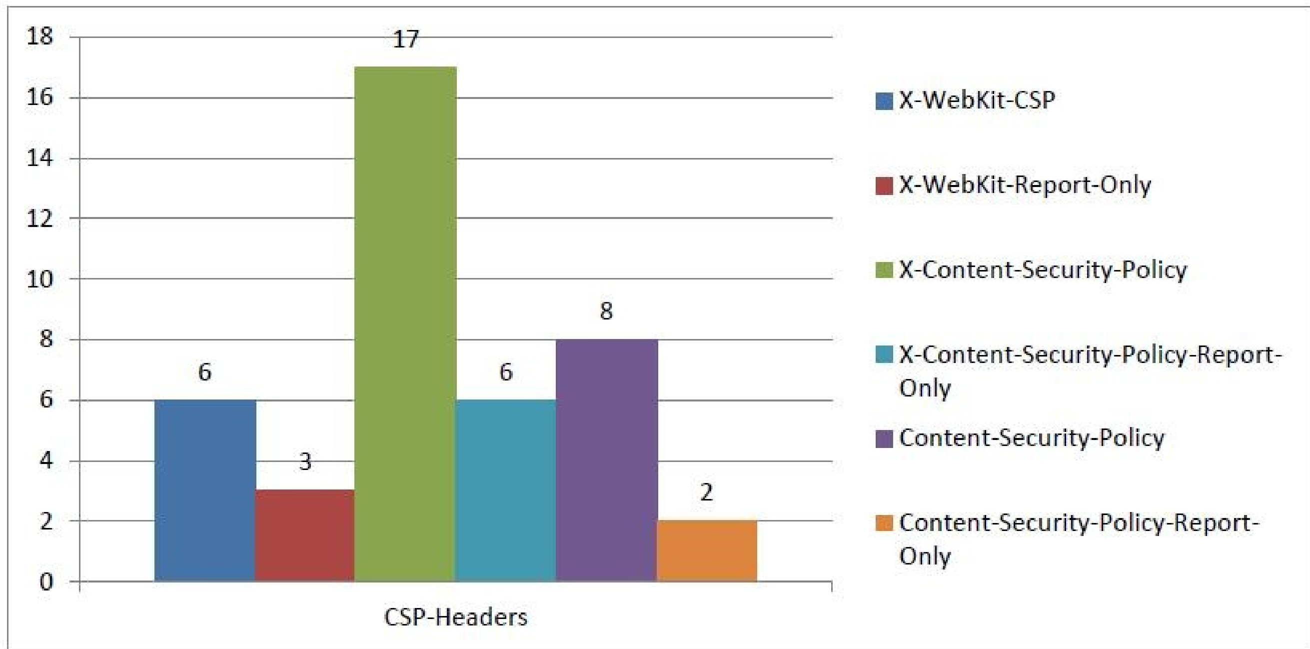


Figure 1: CSP headers usage on real-world websites

- *Non-standard CSP directive usage:* We observed that three (3) websites are using *xhr-src* directive that is supported by only Firefox web browser. In addition, four (4) websites are using *frame-ancestors* directive which is also Firefox specific.

- *Non-effective CSP policies:* We observed websites are setting CSP policy incorrectly and thus keeping open doors for content injections. For example, `http://www.metro-partner.ru/` website sets following CSP policy:

  *X-Content-Security-Policy: allow 'self'; img-src *; script-src *; options eval-script inline-script;*

  The `http://www.metro-partner.ru/` website has defined CSP policy in incorrect way and thus made it non-effective to protect users from content injection attacks. It allows scripts to be executed from any domain and images to be loaded from any arbitrary domains. Furthermore, it also allows execution of inline scripts and eval() usage.

We observed CSP policy errors on eight (8) websites. Table 2 shows a few examples of incorrect policy enforcement on real-world websites. It shows incorrect CSP enforcement nullifies the defense-in-depth CSP protection mechanism.

Our results show that developers have limited knowledge about CSP usage and as a result of that errors made by them in setting CSP policies nullifies the protection provided by CSP and provides attackers opportunity to exploit content injection vulnerabilities.

**Goal_3: Estimate the Amount of Efforts.** To measure the amount of developer efforts required to change their website to adapt to CSP, we measured the number of inline scripts, and inline event handlers used in real-

Table 2: A few examples of CSP policy errors on real-world websites

| CSP Policy | Description |
|---|---|
| X-WebKit-CSP: default-src * 'unsafe-inline' 'unsafe-eval'; script-src 'self' 'unsafe-inline' 'unsafe-eval' s.ppsrc.com 'unsafe-inline' 'unsafe-eval' www.google-analytics.com 'unsafe-inline' 'unsafe-eval' ssl.google-analytics.com 'unsafe-inline' 'unsafe-eval' zhushou.360.cn 'unsafe-inline' 'unsafe-eval' zs.91.com 'unsafe-inline' 'unsafe-eval' zy.91.com 'unsafe-inline' 'unsafe-eval' www.wandoujia.com 'unsafe-inline' 'unsafe-eval' wandoujia.com 'unsafe-inline' 'unsafe-eval' js.tongji.linezing.com 'unsafe-inline' 'unsafe-eval';report-uri http://papa.me/csp/report; | 1. Developers misunderstood the usage of unsafe-eval and unsafe-inline, and used them incorrect way. 2. Injected content will not be prevented by CSP because inline scripts are allowed. |
| X-Content-Security-Policy: allow *; options inline-script eval-script; frame-ancestor', "allow *; options inline-script eval-script; frame-ancestor 'self'; | Arbitrary domains are allowed and inline scripts are also allowed. Thus nullifies CSP protection. |
| X-Content-Security-Policy: allow 'self'; img-src *; script-src *; options eval-script inline-script; | It allows scripts and images from arbitrary domains as well as allows inline scripts. |
| Content-Security-Policy: default-src https: 'unsafe-eval' 'unsafe-inline' | It allows arbitrary https: sources and inline scripts. |

world websites. Inline scripts mean JavaScript code that is embedded in < *script* > tag, JavaScript URIs, and inline event handlers such as onclick, onmouseover, etc. In our test bed we examined home page as well as three internal pages of desktop and mobile websites using the Scrapy framework. We noticed on an average seven inline scripts and eleven inline event handlers are used on Alexa top 100,289 desktop and mobile websites.

Moreover, we measured the amount of changes require to remove inline scripts using phpBB a real-world web forum application [27]. Our modifications of phpBB were 18 files modified that includes in total 174 line deleted and 218 lines added.

To regulate inline scripts and allow developers to specify which *script* elements on a webpage are intentionally included, an experimental directive *script-nonce* is added to the CSP 1.1 draft specification [37]. The *script-nonce* directive allows developers to use inline scripts and inline event handlers by whitelisting them, and hence reduces the number of changes required for developers to implement CSP. There are also recent discussions about an additional experimental *script-hash* directive that computes the hash of JavaScript and only allows the script to execute when its hashed value matches the value in the directive. Both directives have great potential to reduce CSP violations and increase CSP adoption.

**Evaluation Summary.** Our evaluation results show that only 28 including both desktop and mobile websites (out of 100,289) are using CSP policies to protect their users from content injection vulnerabilities. This infers that web developers are unwilling to sacrifice functionality over security and limited knowledge of CSP among developers resulted in incorrect CSP policy enforcement. Moreover, the list of resource origins changes on websites that use rotating advertisements, DNS load-balancing, etc. So, there is a need of tool that automatically infers CSP policy and avoids mistakes that developers can make such as, only enforcing CSP on the home page, incorrect CSP policy enforcement, etc. To address above mentioned challenges we proposed the UserCSP approach.

## 3 UserCSP Design

The goal of UserCSP is to allow users to specify and apply security policies on web content. UserCSP helps developers and users write comprehensive policies for websites by providing them with a GUI to add and modify CSP policies.

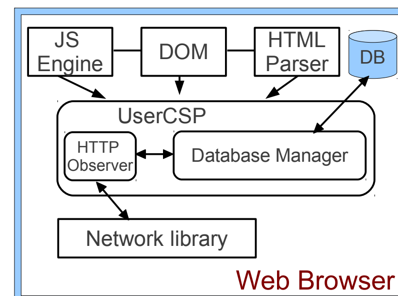


Figure 2: UserCSP architecture

Figure 2 illustrates the architecture of UserCSP. UserCSP monitors the browser's internal events (including HTML parsing, HTTP requests, and XHR requests

Table 3: UserCSP behavior

| Website CSP | User Defined CSP | Global CSP | Action |
|---|---|---|---|
| Yes | No | Yes/No | No change to browser behavior. Website CSP is enforced. |
| No | Yes | Yes/No | Browser enforces User Defined CSP. |
| Yes | Yes | Yes/No | User selects between Website CSP, User Defined CSP, or combination of both. |
| No | No | Yes | Browser enforces Global CSP. |
| No | No | No | No change to browser behavior. No CSP policy is enforced. |

triggered by scripts running in the JS engine). It then dynamically analyzes the content type loaded by a webpage and the source of that content. The HTML parser component in the browser parses the webpage and initiates HTTP requests to load resources such as images, scripts, and stylesheets included in the page. The Database manager component is responsible for storing the webpage's user specified policy in a local database and later retrieving the policy when the user visits the webpage in the future.

As shown in Table 3, when users visit a website, UserCSP performs one of the following actions:

- If the website has defined a CSP policy, but the user hasn't, then UserCSP does not interfere with the website defined policy. However, it does allow the user the option to amend the website's policy.

- If a user has specified a CSP policy for a website, but the website administrator hasn't, then the user's policy is enforced.

- If both a user specified CSP policy and a website defined policy exist, then the user has a choice to either apply their own policy or adopt the website defined policy. Moreover, users can choose to combine their custom policy with an existing website policy by selecting a strict (intersection) or loose (union) combination policy.

- If neither the user nor the website specify a CSP policy, but the user has specified a global policy that can be used for websites that do not have site-specific policies defined, then UserCSP will apply the global policy.

- If neither the user nor the website specify a CSP policy, and there is no global policy, then UserCSP does not affect the content loading on the website.

### 3.1 Automatic Policy Enforcement

There are several challenges in automatic CSP policy enforcement as listed below.

- Dynamic content on a website that can introduce new code into the website at run-time after web page load.

- The list of resources origin changes on websites that use rotating advertisements, DNS load-balancing, etc.

- Heavy usage of inline-scripts on websites make it difficult to derive strict CSP policy that blocks inline-scripts, eval, etc.

- Run-time content injection into websites by browser extensions.

To allow automatic policy inference for websites, UserCSP uses an algorithm that performs dynamic analysis to monitor content loaded by a webpage and recommends a CSP policy based on the content types and content sources included in the webpage. It also monitors the resources dynamically added to the webpage by JavaScript. To record new content introduced by websites at run-time, UserCSP during learning phase continuously monitor websites even after website is completely loaded. It records inferred policy into local database. Next time, when the user visits the same site UserCSP takes previously inferred policy and combine it with the currently inferred policy. Due to rotating advertisements that change periodically may lead to a load request from different origins, therefore, the continuous inferring process of UserCSP helps users to detect changes in the resource origin and apply changed domain to reload resources. Our inferred policy derives strict CSP policy, which blocks inline scripts, styles, eval, and event handlers. However, UserCSP also provides features to users to allow inline scripts and eval on their favorite websites manually. In modern browsers, extensions are high privilege than websites and run with the privileges of the browser. Browser extensions are used to enhance user experience and provide new functionality. Therefore, UserCSP honors content included by extensions into web pages and includes them into inferred policy.

## 4 Evaluation of UserCSP

We tested UserCSP's user defined CSP feature and automatically infer CSP feature with the Alexa Top 100

websites[2] [29]. Manually defined CSP policies are harder to evaluate since they require several rounds of refinement and HTML source code inspection to record content sources. We initially seeded the policies with same-origin restrictions and then expanded them since many websites require content from CDN's and sub-domains.

To test compatibility of the automatically infer CSP feature of UserCSP, the extension inferred policies for each of the Alexa Top 100 websites and then applied the policies onto their respective website home pages (appendix Section 4 includes some examples). Reports were created for each website and examined for CSP violations[3].

The number of whitelisted origins per-policy ranged from 1 to 33, with a mean of just over 7 origins per-policy and a standard deviation of 6.52. Over 25% of websites required more than 10 origins, indicating that creating a comprehensive and effective CSP policy is a challenging task. When there are more than a handful of resources to whitelist, developers are likely to whitelist everything by including "*" in a directive instead of searching for all the necessary origins; this makes the policy less restrictive than it could be. By providing a mechanism to infer the policy, UserCSP provides a quick, effective, and comprehensive policy for developers to set on their websites.

The tests performed for the automatically infer CSP feature have some limitations. The tests infer a CSP policy on page load, but do not interact with the page to determine if further resources are loaded after initial load time. Certain events like clicks on a page may cause additional resources to be loaded. UserCSP can account for these additional loads, but requires the developer to interact with the page during policy inference. This limitation is of little impact to webmasters who are familiar with their website and can make sure that all relevant documents are visited during UserCSP policy inference. Since our tests do not interact with the page content, the average number of origins per-policy is an underestimate of the number that is actually needed for a comprehensive policy. This indicates that creating a CSP policy is even more difficult than previously stated and shows the importance of UserCSP.

After applying UserCSP's inferred policies, all the Alexa Top 100 websites generated CSP violation reports that showed violations for the inline script default restriction. In addition, total 11 websites generated CSP violation reports for using *eval()*[4]. This experimental survey implies that websites commonly use inline scripts.

[2] Three websites containing adult content were excluded from our testing.

[3] In order to adhere to the same-origin-only report-uri restriction in Firefox without alerting websites with our custom CSP testing, we used *http-on-modify-request* to capture and then cancel HTTP requests that contained violation reports.

[4] Websites that generated CSP violation reports for the use of *eval()*: http://www.youtube.com/, http://www.qq.com/, http://bbc.co.uk/, http://adobe.com/, http://sohu.com/, http://aol.com/, http://youku.com/, http://cnn.com/, http://dailymotion.com/, http://imgur.com/, http://neobux.com/

```
default-src      'self';
script-src       http://ads1.msads.net
                 http://kaw.stj.s-msn.com;
img-src          http://udc.msn.com
                 http://kaw.stb.s-msn.com
                 http://b.scorecardresearch.com
                 http://c.in.msn.com
                 http://www.bing.com
                 http://kaw.stb01.s-msn.com
                 http://kaw.stc.s-msn.com
                 http://kaw.stb00.s-msn.com;
style-src        http://kaw.stc.s-msn.com;
frame-ancestors  *;
```

Figure 3: Inferred CSP for msn.com

We used a hack to capture CSP violation reports in Mozilla Firefox during our evaluation, because Firefox allows sending violation reports only if the web page domain and the "report-uri" directive domain are the same. Hence to capture violation reports for the test bed websites, "report-uri" was set to the actual domain but with a fake-path (e.g. http://example.com/fake-report-path). To prevent alerting websites of this custom CSP testing, Firefox's *http-on-modify-request* event was used to capture HTTP requests and cancel the HTTP requests with *fake-report-path* that contained the violation reports after collecting the violation data.

```
default-src      'self';
script-src       http://ads.yimg.com
                 http://l.yimg.com
                 http://mi.adinterax.com;
object-src       http://ads.yimg.com;
img-src          http://l.yimg.com
                 http://l1.yimg.com
                 http://ads.yimg.com
                 http://tr.adinterax.com
                 http://mi.adinterax.com
                 http://b.scorecardresearch.com;
style-src        http://l.yimg.com;
frame-src        http://ad.yieldmanager.com;
frame-ancestors *;
```

Figure 4: Inferred CSP for in.yahoo.com

**Examples of Inferred CSP Policy by UserCSP.** CSP policies that were automatically inferred by UserCSP for http://msn.com/ and http://in.yahoo.com/ are shown in Figure 3 and Figure 4 respectively.

# 5 Related Work

In addition to Content Security Policy, several other solutions exist to mitigate Cross-Site Scripting attacks.

The majority of these solutions use server-side sanitization. Content sanitizers attempt to remove potentially harmful characters from untrusted data. To be effective, sanitization must be performed at each and every entry

point where untrusted data is present in a web application; leaving even one untrusted data source unsanitized makes the web application vulnerable to XSS attacks. Correct placement of context aware sanitizer routines is a challenging task for web application developers [39].

ScriptGard [28] uses a mechanism that helps detect mismatches between sanitization routines and the context in which the routines are invoked. The XSSAuditor filter [2], implemented in the Google Chrome browser, observes HTTP requests and corresponding responses to detect reflected XSS attacks. However, client-side XSS filters are limited to detecting reflected XSS attacks only. Furthermore, recently discovered flaws in the XSSAuditor show that attackers can find complicated bypasses for these blacklist-based filters [7]. Filtering alone cannot be relied upon to prevent XSS. Whitelisting trusted resources via an applied security policy like CSP is a safer choice.

Blueprint [21] uses an alternative approach to protect against XSS. Blueprint treats the HTML parsing component of a browser as untrustworthy and instead uses web servers to parse the document and create output representing the structure of the webpage (the blueprint). This is sent to the browser which uses the blueprint to build the document. A significant amount of overhead is created in order to avoid using the browser's parser. Content Security Policy does not have this issue, since it relies on websites to declare a policy, uses the browser's parser, and trusts the browser's enforcement mechanism to apply the provided policy.

NoScript [20] is a Firefox extension that allows users to disable JavaScript on a per-domain basis and aims to mitigate XSS attacks by detecting reflected XSS. NoScript only blocks scripts, whereas CSP enforcement is applied to various content types on a per-page basis. With UserCSP, users can define policies with a finer granularity and achieve better website usability than with NoScript.

Browser-Enforced Embedded Policies (BEEP) [10] allow web applications to specify the scripts that can run on a website. Similar to the limitations in NoScript, BEEP can only restrict JavaScript on a website; other content such as images, frames, and style sheets are not restricted.

XSS-Guard [3] uses a mechanism to determine which scripts are intended to be on website and which scripts are not. To learn which scripts should be allowed, XSS-Guard first identifies the set of scripts present in the actual HTTP response from a website. XSS-Guard then replicates output statements uninfluenced by user input to get a shadow response. The actual and shadow responses are then compared to identify scripts that were injected into the actual response. XSS-Guard is useful when dynamic and rich HTML content make it challenging to create a comprehensive set of server-side sanitizers. However, XSS-Guard is limited because it can only detect reflective XSS attacks and doesn't protect against persistent XSS attacks. Content Security Policy, on the other hand, can prevent both.

Extensive research efforts focus to improved multi-stage secrete sharing techniques using cryptography [4, 6, 12, 15]. Researchers proposed multi-stage secret sharing techniques based on one-way functions or factorization problem. These techniques could be used by web servers to send content security policies to the web browsers securely.

A group of researchers studied and proposed user authentication technqiues [5, 14, 16, 18, 31, 40]. User authentication is the most important protocol for verifying users to get the system's resources. Password based authentication is the most convenient mechanism. Such techniques could be combined in the web server security mechanisms to extend support for personalized policies by web servers.

Other research efforts [11, 17, 24, 41, 43] proposed techniques of encryption to delegate authority of signing to the proxy. They also allow a multi-proxy signature scheme in certificate-less settings. A rich set of proxy signature schemes [8, 13, 19, 32, 33, 34, 42] have been widely researched. The proposed mechanisms non only succeeded in proxy delegations, but also achieved non-repudiation, revocation, verification properties. The extension of such techniques could allow to transfer the burden of writing CSP policies from web server to web proxy.

# 6 Conclusion

Content Security Policy is an effective mechanism to prevent against content injection attacks. In this paper, we did a large-scale study of CSP usage and infer difficulties in the CSP adoption. CSP has not been widely adopted because of the challenges involved in creating a comprehensive and functional policy, and limited knowledge of CSP among developers. Since adoption is controlled by developers, users lack control over their own security. Users do not have a mechanism to apply Content Security Policies on the websites that they visit and cannot protect themselves from Cross-Site Scripting and Clickjacking attacks.

UserCSP helped to break down the challenges involved in adopting Content Security Policy with UserCSP feature to automatically infer policies and puts control into the users hands by providing them a mechanism to protect themselves with custom policies that they can create and modify.

Our analysis and results show that another barrier to Content Security Policy adoption is the use of inline JavaScript. To overcome this, we would like to experiment further with the proposed *script-nonce* and *script-hash* directives that are under discussion for inclusion in the CSP 1.1 specification.

# References


[1] Alexa Internet, Inc., *Top Sites*, 2013. (`http://www.alexa.com/topsites`)

[2] D. Bates, A. Barth, and C. Jackson, "Regular expressions considered harmful in client-side xss filters," in *Proceedings of the 19th ACM International Conference on World Wide Web, (WWW'10)*, pp. 91–100, New York, NY, USA, 2010.

[3] P. Bisht and V. N. Venkatakrishnan, "XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks," in *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment, (DIMVA'08)*, pp. 23–43, Berlin, Heidelberg, 2008.

[4] T. Yi Chang, M. S. Hwang, and W. P. Yang, "A new multi-stage secret sharing scheme using one-way function," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 1, pp. 48–55, 2005.

[5] T. Yi Chang, M. S. Hwang, and W. P. Yang, "A communication-efficient three-party password authenticated key exchange protocol," *Information Sciences*, vol. 181, pp. 217–226, 2011.

[6] T. Yi Chang, M. S. Hwang, and W. P. Yang, "An improved multi-stage secret sharing scheme based on the factorization problem," *Information Technology and Control*, vol. 40, no. 3, pp. 246–251, 2011.

[7] G. Heyes, *Bypassing XSS Auditor*, 2013. (`http://www.thespanner.co.uk/2013/02/19/bypassing-xss-auditor/`)

[8] M. S. Hwang, S. F. Tzeng, and S. F. Chiou, "A non-repudiable multi-proxy multi-signature scheme," *Innovative Computing, Information and Control Express Letters*, vol. 3, no. 3, pp. 259–264, 2009.

[9] A. Javed, "CSP aider: An automated recommendation of content security policy for web applications," in *IEEE Oakland Web 2.0 Security and Privacy (W2SP'12)*, 2012.

[10] T. Jim, "Defeating script injection attacks with browser-enforced embedded policies," in *Proceedings of the ACM International Conference on the World Wide Web (WWW'07)*, pp. 601–610, 2007.

[11] Z. Jin and Q. Wen, "Certificateless multi-proxy signature," *Computer Communications*, vol. 34, no. 3, pp. 344–352, 2011.

[12] C. C. Lee, M. S. Hwang, and I-En Liao, "On the security of self-certified public keys," *International Journal of Information Security and Privacy*, vol. 5, no. 2, pp. 55–62, 2011.

[13] C. C. Lee, T. C. Lin, S. F. Tzeng, and M. S. Hwang, "Generalization of proxy signature based on factorization," *International Journal of Innovative Computing, Information and Control*, vol. 7, no. 3, pp. 1039–1054, 2011.

[14] C. C. Lee, C. H. Liu, and M. S. Hwang, "Guessing attacks on strong-password authentication protocol," *International Journal of Network Security*, vol. 15, no. 1, pp. 64–67, 2013.

[15] C. Ta Li and M. S. Hwang, "An online biometrics-based secret sharing scheme for multiparty cryptosystem using smart cards," *International Journal of Innovative Computing, Information and Control*, vol. 6, no. 5, pp. 2181–2188, 2010.

[16] I-En Liao, C. C. Lee, and M. S. Hwang, "A password authentication scheme over insecure networks," *Journal of Computer and System Sciences*, vol. 72, no. 4, pp. 727–740, 2006.

[17] C. Lin, K. Lv Y. Li, and C. C. Chang, "Ciphertext-auditable identity-based encryption," *International Journal of Network Security*, vol. 17, no. 1, pp. 23–28, 2015.

[18] C. W. Lin, C. S. Tsai, and M. S. Hwang, "A new strong-password authentication scheme using one-way hash functions," *International Journal of Computer and Systems Sciences*, vol. 45, no. 4, pp. 623–626, 2006.

[19] E. J. L. Lu, M. S. Hwang, and C. J. Huang, "A new proxy signature scheme with revocation," *Applied Mathematics and Computation*, vol. 161, no. 3, pp. 799–806, 2005.

[20] G. Maone, *Noscript*, 2009. (`http://noscript.net`)

[21] M. T. Louw and V. N. Venkatakrishnan, "Blueprint: Robust prevention of cross-site scripting attacks for existing browsers," in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pp. 331–346, Washington, DC, USA, 2009.

[22] Mozilla, *Same Origin Policy for javascript*, 2012. (`https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript`)

[23] OWASP, *The Ten Most Critical Web Application Security Risks*, 2013. (`https://www.owasp.org/index.php/Top_10_2013-Top_10`)

[24] C. Pan, S. Li, Q. Zhu, C. Wang, and M. Zhang, "Notes on proxy signcryption and multi-proxy signature schemes," *International Journal of Network Security*, vol. 17, no. 1, pp. 29–33, 2015.

[25] K. Patil, T. Vyas, F. Braun, and M. Goodwin, *Usercsp: Add-ons for Firefox*, 2012. (`https://addons.mozilla.org/en-US/firefox/addon/newusercspdesign/`)

[26] K. Patil, T. Vyas, F. Braun, and M. Goodwin, *Usercsp. Github*, 2012. (`https://github.com/patilkr/userCSP`)

[27] phpBB, *Free and Open Forum Software*, July 29, 2015. (`https://www.phpbb.com/`)

[28] P. Saxena, D. Molnar, and B. Livshits, "Scriptgard: Automatic context-sensitive sanitization for large-scale legacy web applications," in *Proceedings of the 18th ACM conference on Computer and Communications Security (CCS'11)*, pp. 601–614, New York, NY, USA, 2011.

[29] scottdb56, *mobi*list - A List of Mobile Device-friendly Websites*, 2013. (`http://mobi.sdboyd56.com/`)

[30] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proceedings of the 19th International Conference on World Wide Web*, pp. 921–930, 2010.

[31] C. S. Tsai, C. C. Lee, and M. S. Hwang, "Password authentication schemes: Current status and key issues," *International Journal of Network Security*, vol. 3, no. 2, pp. 101–115, 2006.

[32] S. F. Tzeng, M. S. Hwang, and C. Y. Yang, "An improvement of nonrepudiable threshold proxy signature scheme with known signers," *Computers and Security*, vol. 23, no. 2, pp. 174–178, 2004.

[33] S. F. Tzeng, C. C. Lee, and M. S. Hwang, "A batch verification for multiple proxy signature," *Parallel Processing Letters*, vol. 21, no. 1, pp. 77–84, 2011.

[34] S. F. Tzeng, C. Y. Yang, and M. S. Hwang, "A nonrepudiable threshold multi-proxy multi-signature scheme with shared verification," *Future Generation Computer Systems*, vol. 20, no. 5, pp. 887–893, 2004.

[35] User Agent String.com, *User Agent String Explained*, 2013. (http://www.useragentstring.com/)

[36] W3C Candidate Recommendation, *Content Security Policy 1.0*, 2012. (http://www.w3.org/TR/CSP/)

[37] W3C Editor's Draft, *Content Security Policy 1.1*, 2013. (https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html)

[38] W3C WebAppSec Working Group, *User Interface Safety*, 2013. (https://dvcs.w3.org/hg/user-interface-safety/raw-file/tip/user-interface-safety.html)

[39] J. Weinberger, A. Barth, and D. Song, "Towards client-side html security policies," in *Proceedings of 6th USENIX Workshop on Hot Topics in Security (HotSec'11)*, pp. 8, 2011.

[40] H. C. Wu, M. S. Hwang, and C. H. Liu, "A secure strong-password authentication protocol," *Fundamenta Informaticae*, vol. 68, pp. 399–406, 2005.

[41] H. Xiong, Z. Chen J. Hu, and F. Li, "On the security of an identity based multi-proxy signature schemee," *Computers and Electrical Engineering*, vol. 37, no. 2, pp. 129–135, 2011.

[42] C. Y. Yang, S. F. Tzeng, and M. S. Hwang, "On the efficiency of nonrepudiable threshold proxy signature scheme with known signers," *The Journal of Systems and Software*, vol. 73, no. 3, pp. 507–514, 2004.

[43] M. Zhang and T. Takagi, "Efficient constructions of anonymous multireceiver encryption protocol and their deployment in group e-mail systems with privacy preservation," *Systems Journal*, vol. 7, no. 3, pp. 410–419, 2013.

**KAILAS PATIL** received the PhD in Computer Science, National University of Singapore (NUS), Singapore, in 2014. He is currently an Associate Professor with the Department of Computer Engineering at VIIT, University of Pune, India. He is a Mozilla Rep in India. His research interests include information security, cloud security, and web security. He also served as a reviewer in many SCI-index journals, other journals, other conferences.

**BRAUN FREDERIK** is a Security Engineer at Mozilla, which means testing (and breaking) upcoming features before release. Frederik also develops security tools like ScanJS and helps with improving security features in Firefox OS. Frederik prefers distributed over centralized, free over proprietary, and Mate over Cola. He also takes part in CTF hacking competitions with the team Fluxfingers.