Reining in the Web with Content Security Policy

Sid Stamm
Mozilla
sid@mozilla.com

Brandon Sterne
Mozilla
bsterne@mozilla.com

Gervase Markham
Mozilla
gerv@mozilla.org

ABSTRACT

The last three years have seen a dramatic increase in both awareness and exploitation of Web Application Vulnerabilities. 2008 and 2009 saw dozens of high-profile attacks against websites using Cross Site Scripting (XSS) and Cross Site Request Forgery (CSRF) for the purposes of information stealing, website defacement, malware planting, clickjacking, etc. While an ideal solution may be to develop web applications free from any exploitable vulnerabilities, real world security is usually provided in layers.

We present content restrictions, and a content restrictions enforcement scheme called Content Security Policy (CSP), which intends to be one such layer. Content restrictions allow site designers or server administrators to specify how content interacts on their web sites—a security mechanism desperately needed by the untamed Web. These content restrictions rules are activated and enforced by supporting web browsers when a policy is provided for a site via HTTP, and we show how a system such as CSP can be effective to lock down sites and provide an early alert system for vulnerabilities on a web site. Our scheme is also easily deployed, which is made evident by our prototype implementation in Firefox and on the Mozilla Add-Ons web site.

Categories and Subject Descriptors

H.4.3 [**Communications Applications**]: Information Browsers; H.3.5 [**Online Information Services**]: Web-based Services; D.4.6 [**Security and Protection**]: Information Flow Controls

General Terms

Design, Security

Keywords

content restrictions, web security, security policy, http

1. INTRODUCTION

In the shadow of all the problems stemming from web mash-ups and content injection attacks on the web, it seems attractive to tighten control on the domains. Web "hackers" are often able to use Cross–Site Request Forgeries [14] or Cross–Site Scripting [2] to move data between domains, exploiting browser or site-specific vulnerabilities to steal or inject information.

Additionally, browser and web application providers are having a hard time deciding what exactly should be a "domain" or "origin" when referring to web traffic. With the advent of DNS rebinding [8] and with the gray area regarding ownership of sibling sub-domains (like `user1.webhost.com` versus `user2.webhost.com`), it may be ideal to allow the service providers who write web applications the opportunity to specify, or fence-in, what they consider to be their domain.

1.1 Uncontrolled Web Platform

Web sites currently execute in a mostly uncontrolled web browser environment. The sole protection currently afforded to websites with regards to policies restricting content is the same–origin policy (SOP) [20]. Although this policy is deployed in browsers, attackers are still able to subvert the policy by directly attacking the site and injecting their own script into the content. For example, an attacker may post a message to `messageboard.com` that is rendered for all future visitors to the site. In his message, he includes some HTML that loads a script from `evil.com`, his website. Suddenly all visitors to the message board site are running arbitrary `evil.com` code *within* the `messageboard.com` domain.

More generally, the attacker could also insert references to arbitrary images or style sheets to alter the appearance of the web site; though this is often considered as a less significant attack, it is yet more evidence that a vulnerability in a web site can lead to significant changes to its appearance or operation.

This lack of control is exemplified by iframe injections used to poison search engine rankings of some popular sites [5]. In this attack, some popular sites' validation input is circumvented to inject an `iframe` onto a site's search results page. After the injection, all browsers that render the page inadvertently load an `iframe` that points to data served by the attacker. That page then attacks the visitor's browser through a browser vulnerability like codec installation, ActiveX objects or other drive-by downloading techniques, and search engines reduce the ranking of these victim sites since they may be classified as malware. This problem exploited by the attackers has two parts: (1) the web application does not properly validate input and (2) after the data is injected, a visit to `victimsite.com` causes a browser to load the attack page on `evilsite.com`. We argue that although the input validation is important, it is never perfect; the victim site should be able to specify which sites are trusted and then rely on the visitors' browsers to forbid loading re-

WWW 2010, April 26–30, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

sources from untrusted sites like `evilsite.com`, reducing the abilities an attacker gains through a successful XSS hack.

A way for a web site designer to dictate the behavior of the site is needed, and with assistance from web browsers, such a feature should limit the site's behavior to what is expected. Any other content loads, requests, or abnormal behavior should be blocked as deviation from what is expected. Such a *content restrictions* feature would give web authors more control over data on their site—even as third party content is used. We argue that content restrictions should be used to control sites and propose an implementation of content restrictions called Content Security Policy (CSP), that does exactly this.

Cross-Site Scripting Attacks. In 2008 and 2009, cross-site scripting (XSS) attacks have remained the most wide-spread and frequently occuring vulnerability on web sites [6]. While it may not be clear why XSS is so common, this evidence is an incentive to find a new way to block XSS. While many vendors and framework providers have helped with XSS filters, attackers regularly find new ways to inject and run script on a victim page. Perhaps it is time to approach the problem from a different angle: instead of only filtering scripts from being inserted into a page, we can also disable invading scripts as they attempt to run.

Data Leak Attacks. Currently, web sites are at liberty to embed content from wherever else they wish. For example, `mysite.com/index.html` can embed images from `mysite.com`, or it may contain references to images anywhere else on the Internet like `webcounter.com/images/count.cgi`. The effects of this embedding policy are twofold: data is loaded from a third party site, and (less obviously) information is transmitted to that third party site in the form of the HTTP request. Not always is it the case that an adversary wants to embed malicious code on a site—it may be success enough for them to simply see HTTP requests.

Modern web browsers don't strongly enforce any rules on what can be embedded or referenced by a web site, opening up many vulnerabilities, especially considering the mash-up culture of Web 2.0. As a result, web applications may be leaking data back to one of the contributors, or an un-trusted third party. In the case of websites that have been compromised by content injection or those that contain cross-site-request forgeries generated by site contributors, the data leaked to a contributor could be as mild as IP addresses of all site visitors, or as severe as passwords or bank account numbers for all visitors. We call these types of attack *data leak attacks*.

Since there are many beneficial uses for the ability to embed off-site resources (web counters, traffic analyzers, advertisements), it is not in the best interest of anyone to outright reject this behavior. There are, however, some cases with scripts where there is enough potential for malicious code or sensitive data leak that web browsers should block certain requests, such as in the case of the iframe injection attack.

1.2 Contribution

We propose content restrictions be employed on websites—restrictions that give web application authors control over the content embedded on their site. We also propose an implementation of this called Content Security Policy (CSP); CSP will enable application developers to lay out content loading rules for their web site that will be enforced by the browser. When enforced, these rules limit what types of content may be requested for inclusion on the site, as well as where the content may be loaded from. We show how CSP can be employed as a mechanism to help discover and thwart cross-site scripting (XSS), data leak attacks, and other types of currently unknown attacks that lead to an unpredictable web site experience.

Even if the web developers lose control over the page's content, such as in the case of cross-site scripting and SQL-injection attacks, the attacker will have limited ability to affect the site's behavior since he will be unable to (1) import third party content onto the site and (2) make requests to third party URIs to extract content from the site. Our solution is implemented in the HTTP protocol—outside the scope of any code running in a browser document—so that the security controls can be erected *outside* of the sandbox where web content is rendered.

We also contribute a proof-of-concept implementation that provides a prototype implementation of our scheme on both a complex web site and in a popular web browser, showing that it is practical. We also compare our solution to others in the field and show how ours provides an early-warning system not present in others, does not create a large amount of network overhead, and cannot be used to reduce the security of a web site. Finally, we show our scheme is gradually deployable — it does not rely on complete adoption to work, and can be rolled out gradually onto web servers and browsers, increasing the general security of the web as it is deployed.

1.2.1 Goals of our Scheme

Our proposed scheme is intended to provide a site's administrators control over behavior and appearance of their site S with a rule set that dictates what may load onto the site and into what contexts (image, script, etc). Additionally, our scheme will help protect visitors of a web site S such that the information they provide the site will only be transmitted to S or other hosts authorized by S, preventing aforementioned data leak attacks. This policy will hold even if S is attacked by a web-based adversary who attacks the site through a web browser using some sort of data injection technique to perform cross-site scripting or inject additional code onto the site. Additionally, only scripts served from whitelisted origins will execute, minimizing the chance of XSS attacks on a site. Finally, our scheme will allow a site to specify in which frames it may be rendered; allowing a site to specify what other sites may enframe it minimizes potential for clickjacking.

Control Over Content Used on a Site: If our scheme is implemented correctly, an adversary who is able to augment the website with arbitrary JavaScript, HTML or CSS code will only be limited to the text he is able to inject: not only will he be unable to load third-party resources on the site, he also will not be able to transmit data to hosts not authorized by S.

Increased Security against XSS Attacks: A site employing our scheme and containing reflected or persistent XSS vulnerabilities will be protected when it is rendered in a browser that also supports our scheme. The script injected by an attacker will not execute, and the site owner will be warned of the vulnerability.

Clickjacking Avoidance: When our scheme is implemented correctly, an adversary who embeds a protected site S into his will not be able to invisibly overlay the protected content of S in order to "force clicks" that a user intends for the attacker's site to pass through invisibly to S. In essence, a site protected by CSP cannot be used as an unintended target for clicks stolen and redirected by an attacker.

Only Tightened Security: Additionally, our scheme will not introduce new ways for an adversary to glean information from visitors browsers, is robust (a site that mis-implements our scheme will have no less security than one that does not use our scheme), and cannot be used by an adversary to interrupt a site's operability.

Feasibility: Finally, our scheme will not incur major expense in construction, memory consumption, or processing time overhead, and will be easily adopted gradually; without change, current websites will operate properly in browsers supporting our new scheme, and conversely, web sites that do not support our scheme will still work properly in browsers that do.

1.2.2 Organization

The remainder of this paper is structured as follows: Section 2 describes how our scheme works and what it accomplishes, along with a detailed description of policy language and sample uses. In section 3 we compare our approach of content restrictions to other systems with similar goals, exhibiting the novelty and effectiveness of content restrictions and CSP. In section 4 we discuss how CSP is easily implemented, including a summary of how it can be adopted by browsers and sites. We show in section 5 how CSP can be effective even when gradually deployed, as well as how even a few supporting browsers can help protect a site's entire user base. Future extensions of our work are suggested in Section 6.

2. CONTENT SECURITY POLICY

CSP is activated by a client's browser when the `X-Content-Security-Policy` HTTP header is provided in a HTTP response. The contents of the header either state the policy that will be enforced by the browser, or point to a file that contains the text of the policy; this file must be served from the same origin (scheme, host and port) as the protected document.

The purpose of the policy is to specify which types of resources may be loaded and from where they may be requested; additionally, options can be specified that modify the strictness of enforcement.

2.1 Base Restrictions

When CSP is activated on a page, a few features are automatically disabled to support XSS protection. These features can be re-enabled through the `options` directive, but when re-enabled may open a site to XSS through various techniques. These restrictions are fundamental to any content restrictions implementation, not just CSP, since a script injected by an attacker can easily manipulate content on the site.

Base Restriction 1: No Inline Scripts Will Execute. XSS attacks are possible because the browser has no way to differentiate between content the server intended to send and content injected by an attacker. Content Security Policy forces the separation of code from content and requires authors to be explicit about the code they intend to execute by placing such code in externally referenced files. The result of disabling inline scripts renders any script injected into a document inoperable. Specifically, features disabled by this restriction are:

- Text content of `<script>` tags.
- `javascript:` URIs (those that cause script to execute in the context of the protected document).
- event-handling attributes of HTML tags

Behavior previously obtained through use of these now-forbidden features can be maintained using other still-permitted features:

- Text content of `<script>` tags can be moved to externally referenced files.
- `javascript:` URIs are generally used as a substitute for `onclick` event handlers, and can be converted to JS functions and initiated as an event handler.
- event-handling on HTML tags can be accomplished through JavaScript by obtaining a reference to the element and then either (1) setting the `on*` properties of an element, e.g.:
  `element.onclick = myFunction;`
  or (2) using:
  `element.addEventListener("event", myFunction);`

Base Restriction 2: Strings May Not Become Code. The `eval()` function and related functions make trivial the task of generating code from strings, which commonly come from untrusted sources. These strings are often loaded via insecure protocols, and can become tainted with attacker controlled data. Once tainted data has been introduced to a JavaScript program, it is extremely difficult to control its propagation and calls to eval and similar are likely to incorporate tainted strings containing malicious code. As a result, calls to the JavaScript function `eval()` are blocked by CSP, as are any equivalent functions `setTimeout`, `setInterval` and the `Function` constructor that all take a string representing code as an argument.

We argue that most code using `eval()` can be rewritten without the function call itself, and that calls to `setTimeout` and `setInterval` can be rewritten to use their non-string-argument variants.

2.2 Policy Language

Aside from the base restrictions, a Content Security Policy is composed of *directives*; each directive states how the behavior of the browser should be modified on the protected document. Most of the directives control from where a type of resource may be loaded. A few are different, and will be described separately. This section only serves to summarize the policy language; a detailed syntax can be found in [4].

URI Directives. The content restrictions provided by CSP are available through *URI directives*. These directives specify classes of network requests that may be issued by the browser, and the directives' values specify to where the requests may be made. The directives supported by CSP, and what they regulate are:

- `font-src`: requests generated by `@font-src` CSS code.
- `frame-ancestors`: regulating what sites may embed the protected resource as an iframe or frame element.
- `frame-src`: requests that will be rendered as subordinate frames of the protected page.
- `img-src`: requests that will be loaded as images.
- `media-src`: requests targeted by a `<video>` or `<audio>` element.
- `object-src`: requests targeted by an `<object>`, `<embed>` or `<applet>` element.
- `script-src`: requests that will be interpreted and executed as scripts.
- `style-src`: requests that will be interpreted and executed as style sheets.
- `xhr-src`: requests generated by `XMLHttpRequest`s.

Aside from these directives, a catch-all directive called `allow` must be defined in every policy. This directive regulates requests in two cases: if the request cannot be classified into any other URI directives, or if the directive corresponding to the request type is not defined in the policy. The `allow` directive is *required* by CSP in order to stipulate the "default behavior" for missing directives or unclassified requests.

The `policy-uri` *Directive.* If present in the HTTP header, the value of this directive points to a file that contains the policy to enforce. This directive must appear alone in the header or it is ignored. The contents of the policy file are identical to the contents of an equivalent CSP HTTP header, only served via an external file so the policy (which may be large) can be cached. The URI specified as the `policy-uri` must be served by the same origin (scheme, host and port) as the protected document.

The `options` *Directive.* This directive can be used to modify the underlying behavior of CSP, including to remove one or both of its base restrictions. The directive's value is a space-separated list of tokens. Inserting `inline-script` in the options directive removes Base Restriction 1. `eval-script` in the directive value removes Base Restriction 2.

The `report-uri` *Directive.* If present in a policy, its value is a URI where notifications of policy violations will be sent. The URI must possess the same scheme, and public suffix plus base domain name, or it will be ignored.[1] This prevents sensitive data (cookies or session information) from being transmitted off-site to an attacker.

[1]A public suffix is a domain under which users may register their own domain name. For example, .co.uk, .com, or .pvt.k12.wy.us. See `http://publicsuffix.org`.

2.2.1 Violation Reports

To provide an early-warning system, so that a site's administrators can be notified when an injection attack may be occurring, CSP implements violation reporting. When a `report-uri` is provided and a policy is violated, information about the protected resource and the violating content is transmitted to the report-uri via HTTP POST if available in the employed scheme, otherwise an appropriate "submit" method is used. This report is an XML document containing the following fields:

request HTTP request line of the resource whose policy is violated (including method, resource, path, HTTP version)

request-headers HTTP request headers sent with the request (above) for the CSP-Protected content

blocked-uri URI of the resource that was blocked from loading due to a violation in policy

violated-directive The policy section that was violated (e.g., "script-src *.mozilla.org").

original-policy The original policy as served in the X-Content-Security-Policy HTTP header (or if there were multiple headers, a comma separated list of the policies)

An XML Schema for the violation reports can be found in [4]. In the case where a protected resource is not rendered because the frame-ancestors directive was violated, blocked-uri is not sent and is assumed to be the same as the request URI. The reason for this is because this situation is different from other policy violations: no third-party content was blocked, rather the protected content elected not to load since it does not trust the sites that have enframed it.

2.2.2 Example Policies

Often, the easiest way to explain semantics of a policy language is through examples. Here we provide samples that are illustrative, though not exhaustive.

Example 1. Site wants all content to come from its own domain:

```
X-Content-Security-Policy: allow 'self'
```

In this example, requests for all types of resources (images, scripts, etc) must come from the same origin as the protected document.

Example 2. Auction site wants to allow images from anywhere, plug–in content from a list of trusted media providers (including a content distribution network), and scripts only from its server hosting sanitized JavaScript:

```
X-Content-Security-Policy: allow 'self'; img-src *;
        object-src media1.com media2.com *.cdn.com;
        script-src trustedscripts.example.com
```

Example 3. Online payments site wants to ensure that all of the content in its pages is loaded over SSL to prevent attackers from eavesdropping on requests for insecure content:

```
X-Content-Security-Policy: allow https://*:443
```

2.3 Protections via CSP

Control over Content Used on a Site: Because of the whitelisting nature of CSP, any site not explicitly allowed by a policy cannot be the target of any web requests that are automatically generated during the page's parsing and runtime. Simply put, if something on the page attempts to initiate a connection to any host on the Internet, it first must be authorized by CSP. Web links are the exception to this rule: if a user clicks a link, the page will redirect to that site without asking if the target is allowed by CSP. This is different than the automatically-triggered types of requests blocked by CSP, and deception (convincing a user to click a link) is beyond the scope of CSP.

Increased Security against XSS Attacks: *Type I-II XSS:* In order for an attacker to run script on a page protected with CSP he must:

1. Inject a `<script src="...">` tag into the document
2. Point that tag to a script served from a whitelisted origin (See Section 2.2).
3. Control the contents of the referenced script.

This protection is mainly accomplished via the CSP Base Restrictions. Since arbitrary inline script in the protected page cannot execute, injected scripts will simply not be evaluated. Additionally, if an attacker were able to taint a string that is passed to `eval()` (or any related functions that evaluate arbitrary strings), the evaluation of the tainted string would not proceed since these functions are disabled by CSP. Finally, if an attacker is able to inject arbitrary HTML into the DOM of a page, resulting in the addition of a `<script>` tag that points to a script he controls, it is subject to the content restrictions of the CSP (as enumerated in the `script-src` directive). If his site is not explicitly trusted by the creators of the protected page, the request will not be issued to fetch his script, and the attack will fail.

Clickjacking Avoidance: With the `frame-ancestors` directive, a protected page can decide what other pages it trusts to embed it. When loaded by an attacker's site, a protected page will not be rendered since the attacker's site is not explicitly allowed by the `frame-ancestors` directive — or if it is not present in the policy, the `allow` directive.

Only Tightened Security: When CSP is used by both client and server (or by only one of the pair), it *will not* reduce the security of a web site. This is because CSP does not provide *new* functionality or abilities to a site, but rather locks down what the site can do. An attacker who is able to compromise a CSP header (or policy file) cannot write a policy that makes a site less secure than it would have been without CSP. When implemented properly as a *layer* of security (and not the only protection mechanism), CSP can effectively tighten a site's security by restricting what it can do under control of an attacker.

Feasibility: As we will show in Section 4, CSP can be easily deployed, even gradually, in a popular browser and a complex and popular web site. A CSP–supporting browser will simply not enforce any policy on a site that does not provide a CSP. Additionally, CSP–supporting web sites will still function without any obvious flaws in a non-supporting browser.

3. RELATED WORK

Reis et al write about a fundamental need for a way to draw boundaries around programs, unwanted code, programs in the browser, and other pieces of web sites [19]. They explain how uniform security policies can't be applied since there are many different types of code that execute in a browser. We take the problem of boundaries identified by Reis et al [19], and define a way to specify a boundary for a given web application, and then enforce it.

Jackson et al have previously presented a more restrictive same-origin policy (SOP) [11, 10, 9] that is designed to protect victims of attacks like invasive browser history sniffing [12]. Their idea is that each domain's state should be completely isolated with respect to history and cache. This creates a sandbox for each domain, but does not address the threat of scripts loaded unintentionally or inadvertently from another domain. While this approach provides additional privacy in the form of restricting a site A loaded in a visitor's browser from inferring information about other websites, it does not provide the inverse: prevention of a site X leaking data about itself to A, an external site *not* loaded by the visitor's browser.

In order to stop data leak attacks through a SOP, the policy must be strict about where resources are located; for example, a website from `x.com` would only be able to load dynamic content such as scripts and plug-in data (SWF files, Java Applets, ActiveX controls) from the domain `x.com`. This is not a desirable approach (blocking all content from outside a domain), since many sites depend on loading resources from other domains. In fact, this external resource loading is a pivotal feature of what people are calling Web 2.0: sharing and disseminating information freely. Instead of a strictly DNS-based origin that is enforced by the browser, it would be more flexible to allow the creators of the site identify which domains or hosts can be part of its origin. In this fashion, a web site A is not always separated from *all other hosts*, but instead A should be able to identify from which other hosts it should be isolated. A content restrictions implementation like CSP gives web authors this type of control over their web pages.

Relaxing Same-Origin. James Burke proposed a policy [1] that allows a web page to specify from where scripts are allowed to be loaded and where they are not allowed to come from. This policy refines the way that XMLHttpRequests (AJAX) can behave much in a similar way to how we address all resources, not just scripts. While it is useful to control where scripts come from since they are much more powerful than images, there is potential that a script from a trusted origin may be corrupted through a vulnerability in the web application. As a result, it is also important to catch requests that might be "phoning home" to an attacker; these can be generated not only through XMLHttpRequest objects, but also through writing tags to the DOM.

W3C is developing a mechanism that allows cross-site requests to be performed using XMLHttpRequest objects [21]. The authors realize that the "all or nothing" scheme (which is nothing when it comes to cross-domain AJAX) is too limiting for today's Web. Their proposed scheme uses a HTTP header to specify that the request is cross-origin; the web server is then responsible for deciding whether or not to serve the requested data and also specify how the data can be used. Our proposed CSP uses a similar technique with HTTP headers, but extends the policy to that of *all requests* on a web page that are not already subject to the very restrictive but currently implemented same-origin policy [20]. Additionally, we rely on the web browser to enforce the policies instead of the web server — this is because our goal is to protect the visitors of a site, and not only the site itself.

Browser-Enforced Authenticity Protection. As noted by researchers at Purdue University, the web browser can be a powerful aid to secure the Web. They propose a system called BEAP [15] in which the browser limits the types of sensitive data (cookies, authentication tokens, etc) that are transmitted with requests based on a policy inferred by the operation of the web site—it is not stated by the site explicitly, but rather interpreted based on the details surrounding the action that caused the HTTP request. The effect of this system is that authentication and session tokens aren't sent when they aren't deemed "needed" by BEAP, but the requests are still transmitted, albeit in a safer way.

CSP is similar in that it bases policy decisions based on the context in which a request is generated (what tag, whether it was a script, etc.). The main difference is that CSP leverages the browser to block all requests except for those that the web site has explicitly granted, only allowing cookies and other session data to be transmitted to sites trusted by the protected page, not necessarily just in "safe" contexts.

Browser-Enforced Embedded Policies. Browsers are effective at acting as an enforcer, since they are the platform on which web sites "run," and given a rule set, program execution can be mechanically regulated. One model that allows sites to specify exactly what types of data are allowed is a content-enforcement model called Browser-Enforced Embedded Policies [13] (BEEP) aims to allow web applications to specify precisely which scripts can run on its site. The browser is then tasked with enforcing which scripts can run and which cannot. The result is that—while an XSS attack may insert a script into a victim web site—the script will not run.

BEEP uses a parse-hook technique in the browser to decide if a script should or should not run immediately before it is executed. This parse-hook is a JavaScript function that runs just before a script in question and is provided parsed code for the script as well as DOM node of the element that caused the script to be invoked. The implementation of the parse-hook function is left up to the owner of the victim web site. In short, the function calculates a cryptographic digest string of the script's code value and decides whether or not the script is expected to appear on the page. Any script whose digest value is not accepted, is canceled by the parse-hook.

BEEP is limited to restricting specific scripts that may run on a web site. Other types of content, such as images and style sheets are not restricted. In addition, plug-in content (such as Adobe Flash content or Java Applets) are not restricted by BEEP, while they are still able to run scripts in the context of the page. A more global approach to content restrictions, like CSP, is able to better lock down scripting on a site due to a more holistic view of what content might be dangerous.

MashupOS. Wang et al. from Microsoft Research have taken an operating systems approach, called MashupOS [22], to provide granularity not present in the same-origin policy. They proposed a trio of new HTML tags that help a site express its relationship to other sites it may want to use as content libraries. These tags allow a site to specify one-way trust for content it embeds or other content it is embedded into. This is an improvement over the same-origin policy, since it is no longer all-or-nothing, but there is more granularity involved in the access control granted. Our solution allows a site to specify a policy for an entire page that is then worked into the page regardless of the content injected. CSP is a separate ruleset enforced by the browser that is disjoint from the page so any modifications to the DOM or the raw source of the web page minimize risk of breaching the security policy.

SOMA. Terri Oda et al. from the Carleton Computer Security Lab's SOMA policy [17] restricts resource inclusions on web pages by requiring approval from both the target site and resource provider. This approach restricts content based on cooperation from both client and server sides: web sites serve "manifest" files that tell a browser which domains will be contributing content to the site, and the contributing domains (who are providing resources) provide a service that replies with "yes" or "no" when provided a domain name. The browser then decides whether or not to allow requests to be dispatched from a web page by checking its manifest and the results of the yes/no service queries.

> "SOMA constrains JavaScript's ability to communicate by limiting it to mutually approved domains. Since many attacks rely upon JavaScript's ability to communicate with arbitrary domains, this curtails many types of exploitive activity in web browsers. Whereas currently any web server can be used to host malicious JavaScript or to receive stolen information, the list of potential attackers is narrowed significantly, either to insiders at the web site in question, or to one of its approved partners. As we explain below, this change would provide substantial additional protection in practice." [17]

The SOMA system involves a large amount of communication between not only client and server for the document in question, but also between client and all other third-party content providers. While SOMA may provide a solid negotiation scheme for embedding content, in practice it may be too costly to wait for the number of round-trip queries required to render a page that contains content from many different origins. Additionally, SOMA provides no hardening of JavaScript like CSP's base restrictions, so a site victim of content injection will still execute inline scripts even though they're provided by the attacker and not the web site designers.

Noncespaces. Van Gundy and Chen proposed Noncespaces as a mechanism that helps browsers separate trusted and untrusted content by use of XML namespaces [7]. When a web application implements nonspaces, its server is set up to randomize the XML namespace in a document each time it is served. With the random nonce inserted, items lacking the nonce as the namespace can be deemed "untrustworthy" since they were not expected, and thus are probably injected content.

While effective against some types of XSS attacks, Noncespaces does not protect from persistent data injection; this is because the server that inserts the randomized namespace may also insert such namespace into server-persisting injected content. Additionally, Noncespaces is limited to XHTML documents, so other document types not based on XML may not benefit from this technique.

Policies on JavaScript. Aside from the content protection systems mentioned above, there are various policies that have been proposed to lock down JavaScript execution to only code that is trusted by the authors of a web application. Although these mechanisms are not intended to restrict types of content that are not JavaScript, CSP can still be used to lock down JavaScript.

BrowserShield [18] aims to provide some generic safeguards to help minimize the chance that a user may inadvertently run code through a browser that will exploit his computer. (BrowserShield is similar, but not quite the same as SpyProxy, developed by Moshchuk et al. from the University of Washington [16].) BrowserShield intercepts all JavaScript code on a page as it executes, and rewrites it so it is subject to an execution policy provided by the web site designers. If the script is determined to violate the policy, its execution is aborted. The idea of subjecting content to a policy generated by the site designers is common to content restrictions, but BrowserShield does not discern content generated by the content providers from that provided by some unknown third party who can inject code into the page. While for purposes of one type of XSS, reflected XSS, BrowserShield is still effective, this difference illustrates how CSP stops XSS through a different approach than JavaScript rewriters like BrowserShield.

4. IMPLEMENTING CSP

In order to test the feasibility of CSP, we constructed a prototype implementation on both the client and the server side. On the client side, we modified the Firefox web browser to support CSP. On the server side, we added a CSP header to pages served from a copy of Mozilla's Add-Ons web site, and updated the site to support the CSP base restrictions.

4.1 CSP in Firefox

We implemented Content Security Policy in Firefox by patching various parts of the code. Details and proof-of concept can be found on Mozilla's Bugzilla [3]. In summary, modifications to the Firefox code required the creation of a CSP parser, a service to watch for requests (and reject ones that are forbidden by policies) and then connection to all the bits of Firefox that control the base restrictions. The implementation adds approximately 4000 lines of code to Firefox, and is written in a combination of C++ and JavaScript (XPCOM components, mainly). A preview build with a prototype implementation of CSP can be obtained from `http://people.mozilla.org/~bsterne/content-security-policy/download.html`. Details about its implementation can be found in Bugzilla [3].

4.2 CSP on `addons.mozilla.org`

To ensure that it is feasible to deploy CSP on a web site, we decided to apply a policy to the website hosted at `https://addons.mozilla.org` (AMO). As to not disturb users with interruptions, we erected a mirror of the site on a private network for development, and modified the site to serve an `X-Content-Security-Policy` header with all web pages on the site. Before actually testing CSP as we prototyped it, we did a quick scan of the AMO web site to anticipate how much of it would have to change. We noted the following potential modifications:

- 125 instances of script tags with inline content
- 18 `javascript:` URIs
- 182 HTML-attribute event handlers
- 58 occurrences of `eval()` (mostly for the unpacking routine provided by packed versions of shared libraries such as JQuery)
- 3 occurrences total of `new Function()`, `setInterval()` and `setTimeout()` that accepted a string argument. Most of these were used as a substitute for `eval()` in Safari, and were part of shared libraries used by the site.

Starting with a wide-open policy (one that disables the CSP base restrictions and allows all hosts), we observed site behavior and logged violations. Next, one by one, we re-enabled the two base restrictions and observed what was blocked. We then made updates to the site so that it supported the last policy (with content restrictions and both base restrictions enabled). Finally, we enabled content restrictions, only allowing requests back to the host that served the main page, and observed if anything was blocked.

`allow *; options inline-script eval-script.` To begin with, we installed the most liberal policy possible, disabling the base restrictions and allowing all hosts. In this control phase of our experiment, CSP should be entirely disabled even though a policy is specified. We browsed the site[2] with the CSP-supporting Firefox (see Section 4.1) and observed errors that were posted to the JS console as well as what on the site failed to function. We observed no CSP-related errors posted to the JS console, and didn't observe any obviously broken parts of the site.

`allow *; options eval-script.` Next, we re-enabled Base Restriction 1 (inline scripts) and, while browsing the site, noticed 121 inline script violations (and no other violations) posted to the JS console. We also noticed a few parts of the site did not function as desired — mainly some drop-down menus were not populated, and the overlaid image preview mechanism was replaced by one that loaded images in a new page when clicked.

[2]while we did not walk the complete site (as is difficult to do with thousands of listings on AMO), we made a diligent effort to view each unique part of the site once, in the same order for each part of these tests.

`allow 'self'; options inline-script`. Next, we once again disabled Base Restriction 1 and re-enabled Base Restriction 2 (eval scripts) and, while browsing the site, noticed one CSP eval violation posted to the JS console (and no other violations). This was only evident in the error console, and did not seem to affect the behavior of the site.

`allow *`. Next, we updated the site to support the base restrictions (by modifying the majority of the code segments identified above). This took approximately four hours for the authors (who were not on the AMO development team and did not have any prior knowledge of how the site operated) to perform on the moderately sized web site. We re-enabled both base restrictions and browsed the site, noting no violations were posted to the JS console. We found many of the potential violations previously identified were in unused parts of third-party libraries, and may not have caused any violations if those parts of the libraries were not used by AMO.

`allow 'self'`. Next, we added content restrictions to the policy, and once again methodically browsed the AMO site that was updated to consider the base restrictions. We noticed no violations were posted to the JS console, indicating that no out-of-site requests were taking place.

Simulating vulnerabilities. Simulating an XSS vulnerability, we inserted an inline script, and watched it blocked by CSP as we loaded it in Firefox. We also inserted an `<img>` tag that pointed to an image hosted on another domain, and watched as CSP blocked the load, posting a message to the JavaScript error console.

5. EVALUATION

CSP can be implemented quickly by web sites, and without requiring complete adoption by all browser vendors to be effective. Take, for instance, all pairs of clients and servers that may or may not implement our scheme:

Server and Browser Both Implement: In this ideal case, a CSP is provided by the service provider and enforced by the browser on the client's computer.

Only Server: In this case, any CSPs are ignored by the client's browser, and the current policy (data can be loaded from everywhere) is effective. This does not break any web application, but may cause more data leaking than is ideal. This client is not protected, nor is the service provider.

Only Browser: In this case, the absence of CSP from the service provider suggests to the browser to apply the most relaxed policy so that data is not accidentally blocked from loading. User's experience is the same as without CSP support in the browser.

No Implementation: This is the case as it is today. The server does not specify which sources can be trusted, and the browser trusts all.

Benefit from limited support. CSP does not have to be widely adopted to be beneficial; a few visitors to a web site with CSP-supporting user agents can help provide security for the rest of a site's visitors. This is accomplished through the CSP violation reporting mechanism: it can serve as an early warning system to alert the owners of a site that CSP has discovered potential attacks or flaws in their pages. We believe that receiving reports from even a small minority of CSP-supporting visitors will make it worthwhile for a site to implement CSP if they are worried about XSS attacks, clickjacking or data leaks.

Policy Evolution. As CSP can be slowly adopted by user agents and the whole time prove beneficial to adoptive sites, these sites can gradually deploy more strict policies as they are able. While the base restrictions are necessary to provide adequate XSS protection, they can be turned off to provide a more lenient version of CSP; turning off the base restrictions is not advised, however.

Though even without the base restrictions, CSP can be useful in a limited capacity: restricting what types of requests may be issued automatically on the site. A site may be more worried about blocking image loads (triggered from inside user-provided content) from most third-party sites. Additionally, a site's developers may want to continue using inline scripts until they can figure out how to efficiently move the scripts to external files; by adding the `inline-script` option, this base restriction can be disabled until the site's designers have updated the site.

We also propose a report-only variant of CSP: when deployed, this separate HTTP header triggers CSP to only send reports and not enforce a policy. This variant would simulate CSP without actually blocking any scripts that may break a site. While there are no protections afforded by this variant, deploying a report-only policy can help site designers gauge how much work will need to be done in order to support CSP.

6. FUTURE WORK

CSP has been shown to be a robust implementation of content restrictions, but extensions to CSP can be created to increase its usability, or add additional protections.

While the current model requires a policy to be delivered in the HTTP headers of the document being protected, the policy could instead be delivered once by the site and then enforced for all documents served from the same origin. This delivery mechanism has ramifications that are different from the "one document for one policy" scheme that warrant careful scrutiny before such a model is implemented.

It may also prove useful for a site to limit navigation away from the page; perhaps clicked links and submitted forms should also be part of the content restrictions language. While these rules would not restrict the types of content embedded on the page, they still dictate behavior of the site and may be desirable for many existing web applications.

To aid in constructing a safe policy for a site, a tool should be created that crawls the site and determines what resources are loaded and from where. This tool can create a policy based on the site's "expected behavior," and then implemented in CSP, any deviations from that policy can be reported. The current method of manually creating a policy may be difficult for more complex sites.

7. CONCLUSIONS

We propose the use of content restrictions to lock down web sites behavior, and have provided an implementation of content restrictions called Content Security Policy. CSP provides not only an ability for web sites to specify what types of content may be loaded (and from where), but also some protection from cross-site scripting and other common web attacks such as clickjacking.

While a site should not rely on something like CSP to provide a complete suite of security, CSP can be used as an early warning mechanism for attacks that appear in the wild, and even when not widely adopted by a majority of the web browser market, can prove a useful layer in protecting web applications and their users.

8. ACKNOWLEDGEMENTS

The authors would like to thank Adam Barth for all his rigorous scrutiny of CSP as it evolved. Robert "RSnake" Hansen also helped provide feedback in the early stages of the project, and helped publicize content restrictions as a new way of thinking about web security. The authors are grateful for recent feedback provided by Collin Jackson, Dan Boneh and the Stanford Security lab at large. Dan Veditz, Lucas Adamski, Jonas Sicking and other members of the Mozilla family have provided the authors indispensable support for architecting and implementing CSP in Firefox.

9. REFERENCES

[1] J. Burke. Jsonrequest, part 2 (cross domain policy for all). Blog, March 2006. URL: `http://tagneto.blogspot.com/2006/03/jsonrequest-part-2-cross-domain-policy.html`.

[2] S. Cook. A web developer's guide to cross-site scripting, January 2003. `http://www.giac.org/practical/GSEC/Steve_Cook_GSEC`.

[3] M. Corporation. Bug 493857: Implement content security policy. https://bugzilla.mozilla.org/show_bug.cgi?id=csp, May 2009.

[4] M. Corporation. Content security policy formal specification. https://wiki.mozilla.org/Security/CSP/Spec, May 2009.

[5] D. Danchev. Mass iframe injectable attacks, March 2008. `http://ddanchev.blogspot.com/2008/03/massive-iframe-seo-poisoning-attack.html`.

[6] J. Grossman. Whitehat website security statistics report. Whitepaper, WhiteHat, http://www.whitehatsec.com/home/assets/WPstats0808.pdf, August 2008.

[7] M. V. Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 8-11, 2009.

[8] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting browsers from dns rebinding attacks. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 421–431, New York, NY, USA, 2007. ACM.

[9] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Stanford safecache. `http://www.safecache.com`.

[10] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Stanford safehistory. `http://www.safehistory.com`.

[11] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 737–744, New York, NY, USA, 2006. ACM.

[12] M. Jakobsson and S. Stamm. Invasive browser sniffing and countermeasures. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 523–532, New York, NY, USA, 2006. ACM.

[13] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.

[14] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *the IEEE International Conference on Security and Privacy for Emerging Areas in Communication Networks (Securecomm)*, pages 1–10, September 2006.

[15] Z. Mao, N. Li, and I. Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In *Financial Cryptography and Data Security: 13th International Conference, FC 2009, Accra Beach, Barbados, February 23-26, 2009. Revised Selected Papers*, pages 238–255, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] A. Moshchuk, T. Bragin, D. Deville, S. D. Gribble, and H. M. Levy. Spyproxy: execution-based detection of malicious web content. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–16, Berkeley, CA, USA, 2007. USENIX Association.

[17] T. Oda, G. Wurster, P. V. Oorschot, and A. Somayaji. Soma: Mutual approval for included content in web pages. In *CCS'08: ACM Computer and Communications Security*, October 2008.

[18] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: vulnerability-driven filtering of dynamic html. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 61–74, Berkeley, CA, USA, 2006. USENIX Association.

[19] C. Reis, S. D. Gribble, and H. M. Levy. Architectural principles for safe web programs. In *Sixth Workshop on Hot Topics in Networks (HotNets) 2007*, Atlanta, Georgia, November 2007.

[20] J. Ruderman. In Mozilla Documentation, August 2001. URL: `http://www.mozilla.org/projects/security/components/same-origin.html`.

[21] W3C. Access control for cross-site requests. Technical report, February 2008. `http://www.w3.org/TR/access-control/`.

[22] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in mashupos. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 1–16, New York, NY, USA, 2007. ACM.