Towards Client-side HTML Security Policies

Joel Weinberger
University of California, Berkeley

Adam Barth
Google

Dawn Song
University of California, Berkeley

Abstract

With the proliferation of content rich web applications, content injection has become an increasing problem. Cross site scripting is the most prominent examples of this. Many systems have been designed to mitigate content injection and cross site scripting. Notable examples are BEEP, BLUEPRINT, and Content Security Policy, which can be grouped as HTML security policies. We evaluate these systems, including the first empirical evaluation of Content Security Policy on real applications. We propose that HTML security policies should be the defense of choice in web applications going forward. We argue, however, that current systems are insufficient for the needs of web applications, and research needs to be done to determine the set of properties an HTML security policy system should have. We propose several ideas for research going forward in this area.

1 Introduction

Content injection attacks, and in particular cross-site scripting (XSS) attacks, are a significant threat to web applications [26, 3]. These attacks violate the integrity of web applications, steal from users and companies, and erode privacy on the web. Researchers have focused a great deal of effort at preventing these attacks, ranging from static detection to dynamic checking of content.

The classic approach to stopping content cross-site scripting attacks is by the careful placement of *sanitizers*. Sanitizing, or filtering, is the removal of potentially harmful content or structure from untrusted data. Sanitization places a heavy burden on developers; they must identify where untrusted data appears in their application, what that data is allowed to do, and the context the data appears in on the final page. Unfortunately, these are not straightforward processes. Sanitization is a very brittle process, prone to error. Besides often missing untrusted data in their application, it is not always obvious what sanitization method to apply in what context [25, 16].

In response, several alternatives have been proposed in the literature, most notably, BEEP [8], BLUEPRINT [21], and Content Security Policy (CSP) [18]. These proposals suggest very different mechanisms for preventing XSS and, in some of the cases, more general content injection. Previously, these have been viewed separate proposals with different approaches to the same problem. However, we identify these proposals as instances of a more general notion of *client-side HTML security policies*.

We argue that HTML security policies are superior to sanitization and should be the core defensive mechanism against content injection in future web applications. However, we also argue and show that the current proposals for HTML security policies fall short of their ultimate design goals. We argue that HTML security policies should be at the core of web application security, but much research still needs to be done in building successful HTML security policy systems. We put forth several suggestions for future research, but generally leave next steps as an open problem.

HTML Security Policy Systems We focus on BEEP, BLUEPRINT, and Content Security Policy. We examine these systems to understand their respective power and limitations. While all three provide working systems for expressing and implementing HTML security policies, CSP is of particular importance because it is deployed in Firefox 4 [17]. We evaluate the efficacy of all three, including the first empirical evaluation of CSP on real applications by retrofitting Bugzilla [1] and HotCRP [2] to use CSP. We conclude that none of these systems solves the content injection or XSS problems sufficiently:

- BEEP has serious limitations in dealing with dynamic script generation. This is especially problematic in today's web frameworks that support and

encourage templating across all types of code. Additionally, it does not provide any support for non-XSS content injection.

- BLUEPRINT has significant performance problems. These are not superficial; there are inherent to its approach in avoiding use of the browser's parser.

- CSP can cause significant performance problems in large applications because of the restrictions on code it enforces. Additionally, it does not fit well into web framework the programming models.

Proposals We argue that going forward, web frameworks should use HTML security policies instead of sanitization as the basis of their protection against content injection. However, the security community needs to decide on a set of requirements that HTML security policies should hold going forward. In the end, new HTML security policy systems, and perhaps new browser primitives, need to be developed by the research community to solve the content injection problem.

2 HTML Security Policies

Content injection occurs when untrusted user input changes the intended parse tree of a document [5, 13]. In the case of a web page, this happens when a user input is placed on a page, and the user input contains control structures (such as HTML tags or JavaScript code) that the developer did not intend to be present. By definition, this allows an attacker to modify the behavior of the page in unexpected ways, changing the developer's intended semantics. XSS is a specific type of content injection where the attacker modifies the document's structure to place a script on the page.

Web developers generally use sanitization to defend from content injection, but sanitization can be difficult to get right. Thus, researchers have proposed several other mechanisms for protecting web applications from content injection. Three well known proposals, BEEP, BLUEPRINT, and Content Security Policy (CSP), are instances of HTML security policy systems. HTML security policy systems provide mechanisms for an application to specify how a document is parsed and interpreted by the browser on the client, in contrast to the syntactic approach of sanitization on the server, where unwanted syntax is removed before it reaches the client.

In HTML security policy systems, a policy is provided to the browser when a web page is loaded, and as the browser renders the page, it enforces that the page matches the HTML security policy. This is particularly useful in the context of a web server providing dynamically generated content where a policy is hard to enforce statically (such as a page containing user comments, a news aggregator, etc.).

The advantages of HTML security policy systems over sanitization are several fold. The key improvement is that developers do not need to search for the precise place in code that untrusted data may appear. In sanitization, these places are hard to find, and even when found, it is not necessarily clear what sanitizer to apply (i.e. does the untrusted data appear between HTML tags, or is it part of an attribute, or another context entirely?) [25]. In comparison, one of the goals of HTML security policy systems is to specify allowed behavior, not to limit the syntax at these precise points.

Additionally, in HTML security policy systems, there is an explicit policy to enforce, in contrast to the ad-hoc application of sanitizers. This suggests that HTML security policies are easier to audit than sanitization. A developer can check a policy against their security model, rather than searching an application for sanitizers and building a policy from that.

Because of these properties, and the unreliability of and difficulty in using sanitizers, we argue that HTML security policy systems should be used going forward in web applications to help solve the content injection problem instead of sanitization. However, the current set of available HTML security policy systems have several major problems in their design.

2.1 Existing Policy Systems

While there are many different HTML security policy systems, we focus on three of the most cited, BEEP [8], BLUEPRINT [21] and CSP [18]. These three systems take two very different approaches to the same problem: how to stop untrusted content from injecting additional structure into a web page. They are all particularly concerned with cross-site scripting attacks.

BEEP BEEP focuses on XSS instead of the more general content injection problem. BEEP implements a whitelist of trusted scripts that may execute and rejects the execution of any script not on the whitelist. Thus, only trusted, developer-built scripts may execute on the client, and any injected scripts will fail to do so.

Unfortunately, BEEP's handling of dynamically generated scripts does not match the web framework model. BEEP requires that the hash of a script be statically determined or that the script is added by a trusted script. By definition, if the script is generated dynamically, its hash cannot be determined statically. One can imagine a system with scripts that add dynamically generated scripts, but this is very different from how web applications and frameworks currently handle code generation.

Additionally, BEEP does not handle content injection other than XSS, which have recently been seen in real sites [22]. Also, attacks on BEEP have been developed similar to return-to-libc attacks [4]. While more complex than traditional XSS, the existence of such attacks is cause for concern.

BLUEPRINT BLUEPRINT presents a system for parsing document content using a trusted, cross-platform, JavaScript parser, rather than browsers' built in parsers. The authors view HTML parsers in different browsers as untrustworthy because of browser quirks and view the cross-site scripting problem as fundamentally arising from this. Their approach provides the browser with a "blueprint" of the structure of the page, and a JavaScript library builds the page from the blueprint rather than trusting the browser's HTML parser.

This "blueprint" is an HTML security policy. The server parses the document itself and generates the structural blueprint of the document. This is communicated to the browser where it is used by the BLUEPRINT JavaScript library to build the document. The blueprint is a step-by-step instruction set for the structure of the page, and if any of the content violates this structure, it violates the policy and is removed.

One of the key assumptions of the authors is that server applications "know how to deal with untrusted content." Unfortunately, the authors make this assumption without defending it. There certainly are numerous cases of server applications that do not understand how to properly deal with untrusted content; this is the basis of SQL injection attacks [23]. A tool that could help well-intentioned developers stop *potentially* untrusted content would help to alleviate this.

Additionally, BLUEPRINT unfortunately suffers from several performance problems. In the original paper, the authors report 55% performance overhead in applying BLUEPRINT to Wordpress and 35% performance overhead in applying it to MediaWiki. Because of its very nature, BLUEPRINT cannot use the efficient parsing primitives of the browser; it relies entirely on building the document from the blueprint with the JavaScript parser.

Content Security Policy (CSP) To our knowledge, CSP is the first HTML security policy system to be implemented and released by one of the major browser vendors (in Firefox 4). CSP takes a different view of the browser than BLUEPRINT. Instead of "not trusting" the browser's parsing decision, CSP implements a declarative policy that the browser then enforces on the application. CSP trusts the browser for enforcement, conceding that an application may be flawed.

CSP does this by developing a large set of properties that may be set on a *per page basis*. All trust is based on the page level by CSP properties that state trusted servers for scripts, images, and a variety of other content. This provides a strong fail-safe property. Because the entire page is covered by the policy, the policy will apply to all untrusted content wherever it appears on the page.

The types of properties that CSP provides include trusted servers for images, scripts, and other content, but it also includes one particularly important property. This is the `inline-scripts` property, which, by default, is disabled. When disabled, this means that the browser will not allow any scripts to execute within the page; the only scripts that may be run are ones included by the `src` attribute of a `<script>` tag. This is fundamentally how CSP prevents XSS attacks. Because no script content is allowed to run within the page, and the developer may set a small number of trusted servers for scripts to come from, an injection attack can add a script but it either will not run because it is inline, or it will not run because it will have a `src` pointing to an attacker's untrusted server.

CSP rules are a declarative way of specifying the dynamic semantics of a web page. CSP specifies a set of semantic rules on a per page basis. However, content injection is a syntactic problem where the abstract syntax tree of a document is modified by an untrusted source [19, 13]. It would be possible to keep adding semantic rules to CSP, but a new rule would be needed for each semantic consequence of all possible syntactic changes. Because of this, CSP only provides rules for stopping a small set of injection attacks, namely XSS and specific types of content (such as `<iframe>` tags whose `src` attribute points to an untrusted server). CSP does not stop general content injection, and for it do so would require an ever growing set of rules.

CSPs declarative, page level, fail safe architecture is enticing. However, it places severe restrictions on how web application pages can be structured. We evaluate how these restrictions affect real web applications.

3 Evaluating the Application of CSP

To evaluate the efficacy of CSP as an HTML security policy, we apply it to two popular, real world applications. We determine the HTML security policies necessary to stop injection attacks and apply the policies to the applications. We modify the applications to work with these policies. We evaluate the performance of these applications using CSP, and measure the effort of modifying these applications.

3.1 Methodology

The applications we experiment on are Bugzilla [1] and HotCRP [2]. Bugzilla is a web application for orga-

nizing software projects and filing and tracking bugs, used by many large companies and open source projects, including RedHat, Mozilla, Facebook, and Yahoo! [6]. HotCRP is a conference manager used for paper submission and review by several major conferences.

We retrofit the applications to execute properly with a known CSP policy that blocks XSS. As a manual process, we run the program and explore its execution space by clicking through the application. We modify the applications to correct any violations of the CSP policy by the applications. This does not provide completeness but we feel this most accurately represents what a developer would need to do to apply CSP to her application. While static and dynamic analysis tools have the potential to help, we are unaware of such tools for the Template Toolkit [20] language that Bugzilla is written in.

3.2 Application Modifications

The major part of transforming Bugzilla and HotCRP is converting inline JavaScript to external JavaScript files that are sourced by the HTML page. Because CSP policies are of page level granularity, it cannot reason about individual scripts on a page. Thus, in order to prevent XSS with CSP, it must reject inline scripts and only source scripts from trusted servers. The consequence of this is that completely trusted scripts must be moved to separate files.

Data Access In the implementations of Bugzilla and HotCRP, there are a variety of inline scripts that reference data and variables generated by the templating languages. such as configuration information or the number of search results. This data is not untrusted input. Unfortunately, when the scripts are segregated into separate files from the templated HTML, the variables and data can no longer be referenced by the templating language in the script source file. This is because the templating languages for both Bugzilla and HotCRP treat the individual page as a scoping closure; variables are not shared between separately sourced pages. We address this by creating additional hidden HTML structure and storing the necessary data in an attribute. Later, we extract this via additional JavaScript on the client.

DOM Manipulation DOM manipulation becomes necessary in a number of other contexts as well. Take as an example dynamically generated JavaScript. In HotCRP there are several scripts that interact with lists of papers. For each of the papers in the lists, there are inline JavaScript handlers for each entry, such as `onclick` handlers. Because CSP does not allow inline scripts, including handlers, these handlers must be inserted dynamically by JavaScript. Thus, for each paper entry in the list, we use PHP to generate a `span` with a predictable name, such as `name="paper-name-span"`, and also contains a `data-paper-name` attribute. When the JavaScript executes, it searches for all elements with the name `paper-name-span`, and extracts the name of the element to add a handler to.

Page	No Inline JS	Async JS
index.php	$14.78\% \pm 4.5$	$-3.0\% \pm 4.25$
editsettings.php	$6.3\% \pm 4.7$	$5.1\% \pm 0.92$
enter_bug.cgi	$57.6\% \pm 2.5$	$44.2\% \pm 2.1$
show_bug.cgi	$51.5\% \pm 2.8$	$4.0\% \pm 3.0$

Table 1: Percent difference in performance between modified Bugzilla and original with 95% confidence intervals.

Additional Application Logic Another pattern we observe is the required movement of templating logic into JavaScript. Because the JavaScript is no longer inlined, the conditional branching in the templates no longer can affect it. Thus, it must replicate the same conditionals and checks dynamically. Using the same techniques as we discussed earlier, we replicate where necessary templating conditionals in JavaScript based on data passed in DOM element attributes. This adds additional performance costs to the execution, but also provides additional points of failure for the transformation.

Total Modifications Overall, the types of modifications to Bugzilla and HotCRP closely mirrored one another. This was particularly interesting given that they used two unrelated templating frameworks, Template Toolkit and PHP, respectively. In both cases, it was necessary to transfer data and variables to JavaScript through HTML structure, create significant additional DOM manipulations to build the application, and to move and duplicate application logic from the server to the client.

Our modifications were substantial, adding 1745 lines and deleting 1120 lines of code in Bugzilla and adding 1440 lines and deleting 210 lines of code in HotCRP. We also observed an increase in the number of `GET` requests and data transferred for Bugzilla and HotCRP.

3.3 Performance

The performance of the applications are affected in several different ways. During the application modification, we observe several particular modifications that relate to performance:

- **Additional script source files** In order to remove inline scripts from Bugzilla and HotCRP, we add a number of new script source files. When possible,

Page	No Inline JS	Async JS	JS Template
index.php	$45.3\% \pm 6.3$	$37.2\% \pm 5.0$	$27.9\% \pm 3.7$
search.php	$52.9\% \pm 5.4$	$50.4\% \pm 3.7$	$20.2\% \pm 3.9$
settings.php	$23.3\% \pm 2.7$	$16.1\% \pm 8.2$	—
paper.php	$61.1\% \pm 9.5$	$58.5\% \pm 8.7$	$19.1\% \pm 2.5$
contacts.php	$67.8\% \pm 4.8$	$35.5\% \pm 4.9$	—

Table 2: Percent difference in performance between modified HotCRP and original with 95% confidence intervals and JQuery Templating performance.

we consolidate what were separate inline scripts into one source file, but this is not always possible. For example, if a script uses `document.write`, the script must be placed in a specific location. Additionally, many scripts are conditional on templating branching, and should only occur if specific elements exist or particular properties hold. These extra files have the potential to add latency to page load and execution.

- **New DOM manipulations** As discussed above, to transfer data and variables from the template to the script, we attach the data to HTML elements and use JavaScript to extract the data through the DOM. These extra DOM accesses can be costly operations in what would have otherwise been static data.

Results Our performance evaluation results can be seen in Tables 1 and 2 for a random set of pages for each application, measured in the Chrome web browser network performance tool. After our experiments finished, we observed that one of the major performance slow downs appeared to be the synchronous loading of script sources. Since our modifications required an increase in the number of external scripts, we modified the pages to use the `async` attribute in the `<script>` tags where possible. This allows those scripts to load asynchronously. This change substantially improved the performance of the applications, but in most cases, not enough to approach the original performance.

Our results show that using CSP for Bugzilla and HotCRP is both a complex task and may harm performance. We show that CSP requires changes to how both applications are structured. While CSP has several desirable properties, such as page level granularity and a fail safe architecture, this shows that, like BEEP and BLUEPRINT, it would be difficult to deploy with a secure setting on complex applications.

4 Related Work

There is extensive work how to discover and eliminate XSS vulnerabilities in web applications [7, 27, 9, 10, 28]. There has been work on both eliminating these vulnerabilities on the server and in the client. This work has focused on treating XSS as a bug to be eliminated from an application, keeping XSS vulnerabilities from even reaching production systems. This means that much of this work is static analysis, but some work has focused on dynamic techniques on the server [24]. Other work, specifically KUDZU [14] and FLAX [15], have focused on symbolic execution of client-side JavaScript.

Sanitization, or content filtering, is the elimination of unwanted content from untrusted inputs. Sanitization is applied explicitly by an application or a framework to the untrusted content. Sanitization is almost always done on the server; generally, the goal is to remove the unwanted content before it reaches the client. Sanitization is usually done as a filter over character elements of string looking for "control" characters, such as brackets in HTML. XSS-GUARD [5] and ScriptGard [16] argue that it is necessary to look at sanitization as a context sensitive problem.

There have also been a number of other HTML security policy systems proposed [11, 12]. We focused on three of the most discussed in the literature, but future evaluations would, of course, need to take these into account as well.

5 Towards HTML Security Policies

While current HTML security policy systems are not sufficient for today's web applications because of their performance problems and requirements on how applications are built, they provide very enticing properties. Research should evaluate HTML security policies and how to build better HTML security policy systems. Towards this goal, we start the conversation with several points and questions about HTML security policy systems.

- Researchers should determine the set of properties that an HTML security policy system should have. For example, CSP's page-level granularity is simple as a policy, but puts an undue burden on developers in how they write and retrofit applications. Is this the right trade-off to make?

- From the problems of the systems we observe today, what can we learn? We identified a number of properties systems should not have, such as extensive restrictions on how code is written. Should we just accept these problems to reap the benefit of HTML security policies?

- Combining these systems may be fruitful. For example, a combination approach of BEEP and CSP that allows inlined scripts if they are on a BEEP-like whitelist but also allows external scripts may

be an improvement in usability over either system independently. What features can we extract and combine from current systems to build new ones?

- Should new HTML security policy systems work in legacy browsers or focus on state-of-the-art browsers? For example, it seems likely that BLUEPRINTs performance shortcomings could be addressed with better browser support. On the other hand, retrofitting applications for new browser primitives can be a struggle, as seen with CSP.

6 Conclusion

HTML security policies should be the central mechanism going forward for preventing content injection attacks. They have the potential to be much more effective than sanitization. However, the HTML security policy systems available today have too many problems to be used in real applications. We presented these issues, including the first empirical evaluation of CSP on real-world applications. New HTML security policy systems and techniques need to be developed for applications to use. As a first step, research needs to identify the properties needed in HTML security policy systems.

Acknowledgments

This material is based on work partially supported by the National Science Foundation under Grants No. 0311808, No. 0832943, No. 0448452, No. 0842694, and No. CCF-0424422, as well as by the Air Force Office of Scientific Research under Grant No. A9550-09-1-0539. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Bugzilla. `http://www.bugzilla,org/`.

[2] HotCRP. `http://www.cs.ucla.edu/~kohler/hotcrp/index.html/`.

[3] OWASP: Top 10 2007. `http://www.owasp.org/index.php/Top_10_2007`.

[4] E. Athanasopoulos, V. Pappas, and E. Markatos. Code injection attacks in browsers supporting policies. In *Proceedings of Web 2.0 Security and Privacy 2009*, 2009.

[5] 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, pages 23–43, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] F. Buclin. Bugzilla usage world wide. `http://lpsolit.wordpress.com/bugzilla-usage-worldwide/`.

[7] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 40–52, New York, NY, USA, 2004. ACM.

[8] T. Jim, N. Swamy, and M. Hicks. Beep: Browser-enforced embedded policies. *16th International World World Web Conference*, 2007.

[9] B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the Usenix Security Symposium*, 2005.

[10] B. Livshits, M. Martin, and M. S. Lam. SecuriFly: Runtime protection and recovery from Web application vulnerabilities. Technical report, Stanford University, Sept. 2006.

[11] B. Livshits and Úlfar Erlingsson. Using web application construction frameworks to protect against code injection attacks. In *Proceedings of the 2007 workshop on Programming languages and analysis for security*.

[12] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE Symposium on Security and Privacy*, May 2010.

[13] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. *Proceedings of the 16th Network and Distributed System Security Symposium*, 2009.

[14] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.

[15] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *Network & Distributed System Security Symposium, (NDSS)*, 2010.

[16] P. Saxena, D. Molnar, and B. Livshits. Scriptgard: Preventing script injection attacks in legacy web applications with automatic sanitization. Technical report, Microsoft Research, September 2010.

[17] S. Stamm. Content security policy, 2009.

[18] 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*, WWW '10, pages 921–930, New York, NY, USA, 2010. ACM.

[19] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. 2006.

[20] Template Toolkit. `http://template-toolkit.org`.

[21] Ter Louw, Mike and V.N. Venkatakrishnan. BluePrint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[22] TNW: The Next Web. YouTube hacked, Justin Bieber videos targeted. `http://thenextweb.com/socialmedia/2010/07/04/youtube-hacked-justin-bieber-videos-targeted/`.

[23] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 32–41, New York, NY, USA, 2007. ACM.

[24] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *Proceedings of the International symposium on Software testing and analysis*, 2008.

[25] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. A systematic analysis of xss sanitization in web application frameworks. In *Proceedings of 16th European Symposium on Research in Computer Security (ESORICS)*, 2011.

[26] WhiteHat Security. WhiteHat Webinar: Fall 2010 Website Statistics Report. `http://www.whitehatsec.com/home/resource/presentation.html`.

[27] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the Usenix Security Symposium*, 2006.

[28] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, pages 121–136, 2006.