Enhancing Web Page Security with Security Style Sheets

SCS Technical Report TR-11-04

Version: February 10, 2011

Terri Oda
Carleton University
terri@ccsl.carleton.ca

Anil Somayaji
Carleton University
soma@ccsl.carleton.ca

Abstract

Although the web security community now has a variety of techniques that could help web developers to defend against common attacks such as cross-site scripting and cross-site request forgery, this work is not in a form suitable for general use. What is needed is a web standard that unites these techniques using syntax and semantics that are easy for web developers to learn and straightforward for browser makers to implement. Here we propose such a standard, *Security Style Sheets*, a browser-enforced policy language modelled on Cascading Style Sheets. Security Style Sheets provides an extensible policy framework that allows for policy to be separated from content and to be specified at both coarse and fine levels of granularity. In this paper we present the syntax and semantics of Security Style Sheets, explain its relationship with past web security proposals and CSS, and give examples of how it could be used to protect mainstream websites such as Facebook. Also in the model of CSS and the Acid3 tests, we present a conformance suite for Security Style Sheets.

1 Introduction

While the web was originally conceived as a research tool, it has quickly become an essential tool for business, entertainment, news, and a variety of other facets of the modern world. But with increased utility and use has come increased incentives for attackers to exploit flaws in web interfaces. Unfortunately, despite the increased risks, there is ample evidence that web sites have not been hardened against attack. For example, estimates suggest that 70% of sites may be vulnerable to cross site scripting, the most popular form of web attack [5].

Historically, the bulk of web security work has focused upon the server side. This is fairly logical in many ways: input validation can stop many web attacks, and server-side validation, while imperfect, is straightforward to implement. Those most interested in security are often the web site developers and operators who work heavily on the server side. And more cynically, it is much easier to sell a customer on a pre-packaged server-side solution such as a web application firewall than it is to sell the customer on the idea that *their* customers should all be using specific browsers or running additional software on the client side.

However, the client side is a compelling place for web security solutions because many attacks occur in the browser: cross site scripting and cross-site request forgery can take place with minimal to no server-side involvement. There is a great potential for good if we can limit the scope of attacks on the client side. But while the potential is great, the difficulty of implementing solutions for any given site is high: any developer interested in client-side protections must learn a great variety of techniques and syntax to implement the current state of the art protections. Further, the situation is getting worse as security researchers discover new attack techniques and corresponding defenses.

If site developers and operators are going to implement security policy on the client side, it should be easy for them to understand and implement, and it needs to be worth their time. Similarly, if browser developers are going to support a policy language, it should be easy for them to understand, adopt, and test. This policy language should be extensible without making common use cases more complex: we do not want to develop tools that are so complex that ignoring them becomes a rational trade-off [11].

To achieve these goals, we propose that several of the most promising client-side security techniques be made available through a standardized web security policy. Previous work on standards within the web security space has concentrated on issues other than client side security. For example: Secure Sockets Layer (SSL) [10] provides communications security between the client browser and the host server, the OASIS standard for Web

Services Security (WS-Security) [2] concentrates on securing SOAP messages and Security Assertion Markup Language (SAML) [1] is used for handling authentication and authorization between organizations. However, no standard has yet been defined to deal with issues of within-page security.

To ease adoption, we wish to model our policy language on what browser developers and web site operators already know, and to combine and modify past proposals to fit within this framework. Many policy languages have used XML-based syntaxes because of the availability of parsers and the basic familiarity had by many people working within the web space. Here, however, we propose to follow the pattern of another ubiquitous web technology: Cascading Style Sheets (CSS).

This paper proposes the first web policy standard framework designed to stop or mitigate modern web security threats such as XSS and CSRF. Our proposal, *Security Style Sheets*, gathers several known client-side security techniques together through a consistent CSS-inspired syntax. Additionally, we have created a conformance suite that tests whether browsers implement the correct semantics for Security Style sheets, again modelled upon conformance tests for CSS and other web standards.

The rest of this paper proceeds as follows. We first discuss the client side web security issues that are targeted by Security Style Sheets. We next describe CSS and explain why it is a suitable template for a web security policy language. In Section 4 we describe the syntax and semantics of Security Style Sheets; in Section 5, we discuss what types of web attacks the proposed standard will be able to address relative to past proposals. Section 6 gives a detailed example of how Security Style Sheets could be used to secure comments in common types of web applications. The conformance tests for the proposal are then described in Section 7. Section 8 discusses how Security Style Sheets could be extended, and Section 9 concludes.

2 Client-side Web Security Issues

When looking at vulnerabilities and exploits within web sites, it is sometimes difficult to sort them into distinct categories because terminology and definitions can differ depending upon the classification in question. As a basis for our discussion, let us consider the Open Web Application Security Project's Top 10 list for 2010, shown in Figure 1.

Of these most common vulnerabilities and issues, some simply cannot be safely handled in the client browser. For example, *A7 - Insecure Cryptographic Storage* cannot be remedied on the client-side. Others are more of a grey area such as *A9 - Insufficient Transport Layer Protection*, which cannot entirely be handled on the client side, but solutions such as ForceHTTPS [13] can force the client to use secure protocols when available (i.e. the original website must be available over HTTPS, which is a server-side issue). Of these top 10, *A2 - Cross-Site Scripting* and *A5 - Cross-Site Request Forgery* are good targets for client-side solutions and mitigations.

A1:	Injection
A2:	Cross-Site Scripting (XSS)
A3:	Broken Authentication and Session Management
A4:	Insecure Direct Object References
A5:	Cross-Site Request Forgery (CSRF)
A6:	Security Misconfiguration
A7:	Insecure Cryptographic Storage
A8:	Failure to Restrict URL Access
A9:	Insufficient Transport Layer Protection
A10:	Unvalidated Redirects and Forwards

Figure 1: Open Web Application Security Consortium Top 10 2010

Terms such as cross-site scripting are very broad. OWASP states that, "XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation and escaping. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites." [26] While the OWASP classification is a convenient one for our purposes because it lists the threats in order of popularity, other classifications such as the one given by the Web Application Security Consortium [4] may give slightly different definitions. Despite the name "cross-site scripting" a successful XSS attack may be neither cross-site nor involve scripting. A carefully placed closing tag or carefully crafted CSS could equally be used to deface a site. As such, defenses may stop some types of XSS but not others. And it can be even more confusing to categorize specific attacks: an attack might use cross site scripting in order to do clickjacking that then causes cross-site request forgery.

To facilitate talking about specific protections, we are going to consider five types of web security issue, and these five types do not line up exactly with the existing definitions.

Script Injection The situation wherein an attacker is able to insert executable code (typically JavaScript) into a web page. This is typically a subset of Cross Site Scripting (A2) although it can sometimes be grouped with the other injection attacks. Once script is inserted, it can be used to read or modify any other part of the page. Often, script injection is combined with information leakage in order to get information obtained in the browser sent to the external attacker.

Content Injection The situation wherein an attacker is able to load static content (e.g. Images, HTML, fonts) into a web page. This is generally considered to be a subset of Cross Site Scripting (A2) or sometimes part of a Cross-Site Request Forgery Attack (A5). This can be used for defacement or placing false information in the page.

Information Leakage This refers to the case where an attacker is able to get information from a user's browser that s/he would not have had access to otherwise. This is often considered to be part of Cross Site Scripting (A2) although in other classifications such as the one used by WhiteHat security [5] it merits a separate category. In many cases, information is leaked using requests to load content by encoding that information in the URI to be loaded, so information leakage and content injection are very closely linked categories.

Cross Site Request Forgery An attack where a user visiting one site is forced to send unauthorized requests to another site, generally performing some sort of action on that second site. This correlates directly with Cross Site Request Forgery (A5) in the OWASP classification.

Clickjacking An attack where a user's clicks may be hijacked, so that the browser registers the click as going somewhere other than where the user thought s/he was clicking. This is not part of the OWASP top 10 as it is not sufficiently common, but it can be found elsewhere in the OWASP attack classifications.

This is not an exhaustive list of issues that can be mitigated within the browser, but it highlights several high-profile issues that have some interesting solutions not yet fully deployed in the browser.

3 Cascading Style Sheets

Over a decade ago, cascading style sheets (CSS) were introduced to allow web page developers to separate style and layout information from the content of a web page. This made it easier for styles to be updated, maintained, or completely reworked without requiring changes to the HTML document itself.

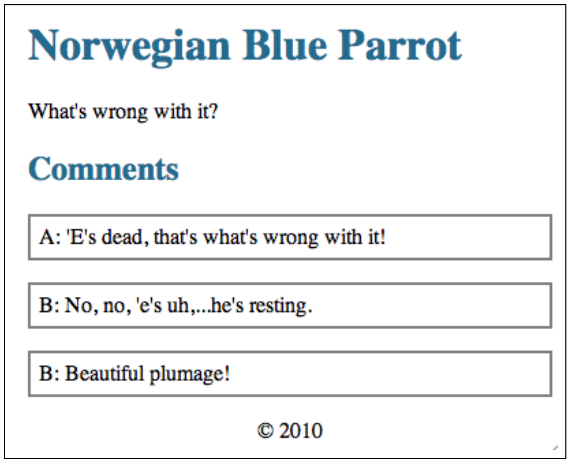
CSS is a way for developers to specify style and layout for a web page. Rather than using (now deprecated) tags like `<center>` within the HTML document, each HTML element can be "styled" either inline or in a separate document. For example, to center a paragraph one might use `<p style="text-align: center">`. But the real power comes when the CSS is separated out and can be applied to entire classes of items.

```
 1  <html>
 2  <head>
 3     <title>Sample Document</title>
 4     <style type="text/css">
 5         .comment {
 6            padding: .4em;
 7            border: 2px solid gray;
 8         }
 9         h1, h2 {
10            color: #236B8E;
11         }
12         #footer {
13            text-align: center;
14         }
15     </style>
16  </head>
17  <body>
18  <h1>Norwegian Blue Parrot</h1>
19  <p>What's wrong with it?<p>
20
21  <h2>Comments</h2>
22  <p class="comment">A: 'E's dead! </p>
23  <p class="comment">B: No... he's resting. </p>
24  <p class="comment">B: Beautiful plumage! </p>
25
26  <div id="footer">
27     &copy; 1969
28  </div>
29
30  </body>
31  </html>
```

Listing 1: Sample HTML document with CSS

In Listing 1 and the corresponding Figure 2, we can see how a style might be applied to a small document. The style is given in lines 4-16, while the rest of the listing gives the rest of the HTML. In the CSS from lines 5-8, we set the properties of any element with class comment so that it will have a padded box with a solid gray line around it. The padding is specified as ".4em" which is relative to the existing font size so that the padding is 40% the size of one text unit. In lines 9-11 we set the color of both header one (h2) and header two (h2) elements to be a shade of blue. In lines 12-14, we ensure that the element with the id of "footer" has centered text. Figure 2 shows the page as it would be rendered in a modern browser.

Right now, security technologies are often interwoven with the HTML, and security improvements often require extensive rewrites to pages in order to make them most effective. But what if we could separate the security from the document and apply security policy as a style



Norwegian Blue Parrot

What's wrong with it?

Comments

A: 'E's dead, that's what's wrong with it!

B: No, no, 'e's uh,...he's resting.

B: Beautiful plumage!

© 2010

Figure 2: Web page corresponding to the code in Listing 1.

sheet? This could make the management of web security policy much easier as the policy would be gathered in one place, allowing for easier auditing, replacement, and modification.

There are a variety of benefits specifically to the format of style sheets:

- This format is already familiar to the web designers and programmers who are frequently in charge of making and maintaining changes.
- We can take advantage of semantic knowledge already contained within the layout: CSS already contains labels for many different types of content, and information about which types are similar or are grouped.
- Using classes, one can provide security for similar elements of the page using a single policy. For example, a web developer might choose to have the same security settings for all comments, whether they appear under a news article or in a list of recent comments in a sidebar.
- Web applications often can be customized using themes or "skins" that are largely done by switching style sheets (such as demonstrated on the popular CSS Zen Garden site [20]). By providing security as a style sheet, we can hook in to such existing mechanisms to apply security policy to existing code.

4 Security Style Sheets

There are a small variety of client-side security solutions for the web, but three types stand out as particularly good candidates for integration into a security stylesheet. First there are systems which limit the sources for any externally loaded content, second are systems which constrain within-page communication, and finally there is the idea that it should be possible to have parts of the page where no scripts can be run.

Security Style Sheets works with three properties:

1. `approved-domains` contains a whitelist of domains which have been approved as suppliers of third party content, be that JavaScript, images, video or any other form of data.
2. `approved-elements` contains a whitelist of other web page elements which have been granted access to this particular part of the page.
3. `execution` is a binary property which indicates whether JavaScript or other code may be executed in a given web page element.

Each of these can be applied to any web page element, be it a specific `div`, all paragraphs with a given class, or even all links. The syntax used is fundamentally the same as the syntax used for cascading style sheets. Because this is the case, the definition of each property is given in a similar syntax to that used to describe CSS properties.

These three properties have been inspired by existing work in web security. They are not a comprehensive list of what could be part of a web security stylesheet specification, but it is our hope that they can form the initial basis for a larger stylesheet-based security policy language.

4.1 approved-domains

'approved-domains'	
Value:	all | none | self | [<uri>,] *
Initial:	all
Applies to:	all
Inherited:	yes
Percentages:	N/A
Media:	security
Computed value :	N/A

The `approved-domains` property gives a list of domains from which content (such as images and JavaScript) can be loaded. The "domains" are actually listed in the form of URIs, so the web developer can specify not only the domain, but also the protocol and port to be used. We say domains rather than URIs because the current assumption is that paths would not be specified in order to facilitate maintenance over time and allow one script to load helper scripts from the same domain. However, it is possible that actual implementations would allow more specific inclusions.

The idea behind `approved-domains` is that many current web exploits rely upon the ability to load additional malicious code from external sites [19], and still other exploits rely upon the ability to load external content as a backwards way of getting information out to an attacker in the URL.

By requiring approval of domains in advance before content can be loaded, we have limited the range of places from which an attack may come. An attacker thus would have to compromise an approved domain to insert attack code there, or would have to insert the entire attack code there, and would have to leak data in some other manner than through HTTP requests.

There are two systems which have heavily influenced the `approved-domains` property in security style sheets. On the assumption that one often knows what code and content that should be included on a given web page, these systems have looked at ways to whitelist the domains from which content can be included. The simpler Same Origin Mutual Approval (SOMA) policy [18] provides a per-domain content whitelist for all external requests, while the more customizable but also more complex Content Security Policy (CSP) [21] allows site designers the ability to specify allowed domains on a per-page basis and specify which domains can be used for which type of content. For example, including a domain in order to allow photos does not automatically mean that JavaScript may also be loaded from that domain.

Security style sheets borrows the "approved" terminology from SOMA, but rather than trying to provide both sides of the approval, it takes a cue from CSP and relies instead upon the Origin header [7] to help with stopping cross site request forgery attacks. Although it would be possible to provide CSP-style whitelists for specific types of content, the usability and security trade off is as yet unclear, so we have opted for the larger-grained but simpler approval per-domain for all types of content. However, not everything about `approved-domains` is larger-grained than its predecessors: while SOMA did approvals for all pages in a given domain and CSP handles whitelisting for a single page, `approved-domains` is a property which can be applied to any web page element within a page. This means that one might allow an advertisement to be included within a banner-sized space at the top of the page, but not give that same advertising server access to other page elements such as a login box.

It is not always true that one can predict all sources of content in advance. One common counterexample is actually advertisements, where one might know the advertising service contracted but not necessarily know every ad that might be displayed in advance. However, despite the fact that whitelisting may not be easy to use in all cases, CSP and SOMA assert that it could be a useful tool for many sites, such as banks, which maintain stricter controls over included content. By allowing whitelisting for single or groups of HTML elements rather than per-page, security stylesheets sidesteps the problem of some sites needing open areas by allowing for both protected and open areas within a given page, and (as described in the next section) controls about how information is disseminated between these areas of differing security values. This allows a site the ability to protect some areas even if not all of the page can be restricted in this way.

The goal of `approved-domains` is to allow sites the ability to constrain the potential attack vectors for script inclusion and content inclusion (e.g. several forms of XSS) and constrain the potential vectors for loss of sensitive information through third party communications. In addition, it limits the ability for the protected site to be used in a CSRF attack against arbitrary third parties.

```
<style type="text/sss">
.picture {
    approved-domains: all;
}
.comment {
    approved-domains: none;
}
#searchresults {
    approved-domains: self;
}
#advertisement {
    approved-domains: ads.example.com, stats.↩
         example.com;
}
</style>
```

Listing 2: Examples for approved-domains

Listing 2 shows some examples of how to use `approved-domains`. The first example, with approved-domains set to *all* is the current and default behavior of web pages: any code can be inserted in this location. While more secure default behavior could be provided, this would break existing pages, so this choice was made in the interest of providing backwards compatibility. The second example, *none*, prevents any additional code from being loaded in this location. This might be particularly useful for places where user input may be displayed but that code is not desired, such as comments on an article or the display of search results. The third example gives a simple way of approving inclusions from the same origin as the web page itself. And finally, the fourth example shows how external inclusion locations can be approved.

4.2 approved-elements

'approved-elements'	
Value:	all | none | [<id>,] *
Initial:	all
Applies to:	all
Inherited:	yes
Percentages:	N/A
Media:	security
Computed value :	N/A

The `approved-elements` property functions similarly to the `approved-domains` property, only it constraints within-page communication rather than external communication to third party sites.

The idea with `approved-elements` is to limit the scope of vulnerability should an error be found within the page. WhiteHat security estimates that 71% of sites are vulnerable to cross site scripting and 70% of websites are vulnerable to information leakage [5]. Using `approved-elements`, sites can mitigate the effects of these attacks by ensuring that code injected into one part of a page cannot always access sensitive information entered within another part of the page.

There are several systems which have heavily influenced the design of the `approved-elements` property. One is Visual Security Policy [17] (ViSP), which describes a way of applying security policy to visual elements of the page. ViSP concentrated on providing within-page restrictions in a manner tied to visual elements as they are laid out on the page, while security stylesheets reverts to attaching policy to HTML Document Object Model (DOM) elements. This adherence to the DOM allows security stylesheets to behave more like cascading stylesheets, and should make things more consistent for developers who are familiar with the CSS model. (It may also make implementation easier because the code will be more similar to that for CSS.)

Another inspiration is AdJail [16], an advertising-specific web security policy. This framework is designed specifically to help secure third-party advertisements, protecting the web page from attack while allowing advertisers enough access to continue their existing business models. One particularly interesting part of the AdJail setup is that the size of the advertisement jail is constrained. This allows for protection against click-jacking, an attack where a malicious person might alter the page such that a user thinks they are clicking in one location but instead the page has forced their click to be used on another part of the page. This could be used for maliciously submitting forms, adding to the click-count of an advertisement, or any number of other actions.

To limit the damage caused by click-jacking, security style sheets could also limit the size of elements such that only things within the `approved-elements` list can write to that space. This is in some ways closer to the intent of ViSP's visual model of the page.

And security stylesheets also owes much to the previous work in web mashup security (e.g. [25, 14, 9, 15, 8]), which described in greater detail the risks inherent in within page communications.

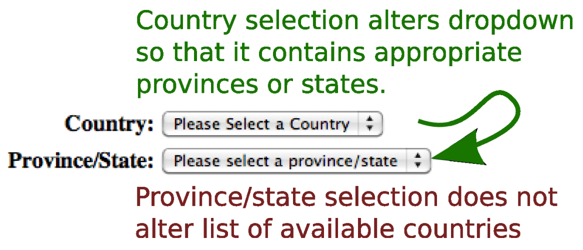

Country selection alters dropdown so that it contains appropriate provinces or states.
Country:
Please Select a Country
Province/State:
Please select a province/state
Province/state selection does not alter list of available countries

Figure 3: An partial address form demonstrating a non-symmetric use of approved-elements.

Note that `approved-elements` is not a symmetric property: for example, one might want the "country" drop-down in an order form to change the "province" drop-down, but there would be no reason for the province drop-down to change the country one, as shown in Figure 3.

```
<style type="text/sss">
#drawingcanvas {
   approved-elements: all;
}
.comment {
   approved-elements: none;
}
#currentstatus {
   approved-elements: status-box, menu;
}
</style>
```

Listing 3: Examples for approved-elements

Listing 3 shows some examples of the use of the `approved-elements` property. The first example of *all* is the current and default behavior, where all elements of the page can access all other elements of the page. The second of *none* is the "sandboxed" behavior, where an element is completely isolated from the rest of the page. This could be combined with a `approved-domains: none` property to provide a sandbox at the domain include level as well. However, it may be more common that it would be combined with a list of specific domains so that one could have a partial sandbox for an advertisement or other external widget that you wish to include without incurring risks for the integrity of the rest of the page. Finally, the last example allows the specification of one or more HTML elements that will be granted approval. These are specified by their unique id values, although it is possible that this could be extended to approve elements by class as well. Implementations will have to take special care that ids cannot be overwritten or changed.

4.3 execution

'execution'	
Value:	yes | no
Initial:	yes
Applies to:	all
Inherited:	yes
Percentages:	N/A
Media:	security
Computed value :	N/A

The idea of having data be non-executable is hardly a new idea, and is commonly discussed with respect to buffer overflows. However, it can also be applied in the browser as a method for defeating cross-site scripting by making it impossible for the maliciously injected script to execute. Browser-Enforced Embedded Policies (BEEP) [15], for example, includes a "noexecute" sandbox upon which security stylesheets' `execution` property is based.

In many ways, the `execution` property is the simplest of the properties, with the fewest edge cases. Anything within an element where `execution: no` is set cannot execute any `<script>` tags that are contained within. It may be logical to extend this to include not allowing certain types of object as well. In this way, we have created a safer sandbox within which code cannot be injected. This could be used for the purpose of displaying user-submitted content such as a comment on a news story: something that should not contain any JavaScript and thus can be constrained so that none will work in that area.

4.4 Backwards compatibility

All three of the security stylesheet properties have very permissive defaults: `approved-domains` defaults to allowing inclusions from all domains, `approved-elements` defaults to allowing access to all elements, and `execution` allows execution of any included code. This is contrary to typical security design, where defaults should be minimally secure, but these defaults were chosen so that they reflect the current behavior of the web. In this way, security stylesheets can be backwards compatible and enabling it will not cause breakages in pages that have not yet written policies.

4.5 Inheritance

One of the big features of cascading style sheets is its cascading inheritance model, but using this model for a security policy can be very problematic. Of large concern is the problem of privilege escalation. In particular, we want to ensure that included code (be it intentionally included or injected) cannot override permissions set by the site operator. However, we potentially do want included code to be able to choose permissions, since third party content providers such as advertising networks may want to provide code that can be cut and pasted, but may want to protect this code from attacks such as click fraud.

To avoid privilege escalation, security style sheets needs to have a model where subsequent policies can only be more restrictive than the pre-existing policy. To ensure consistency, we need a standard ordering for the application of policy.

Our proposed ordering uses the tree structure of HTML. Policy is prioritized based on where it is added within the tree, so policy in the HTML header (where stylesheets are usually included) would take precedence over policy placed in the branches of the tree. Within a given element the policies will be applied in order, so if a page includes `security1.sss` and `security2.sss` in that order, `security2.sss` would only be allowed to be more restrictive than `security1.sss`. Any directive that granted more permissions in `security2.sss` would be silently ignored, as would any additional permissions granted within the body of the document. The permissions can be applied based on a breadth-first ordering of the HTML DOM tree. In addition, we should standardize ordering based on specificity of selector as described in the CSS standard's rules for inheritance [23].

This ordering is arbitrary, and may not be the ideal one for the purpose of implementation and usability, so before a final decision is made on appropriate ordering one would need to consult browser manufacturers and potential policy writers to ensure an ordering that will minimize difficulty and confusion.

5 Attack Coverage

Table 1 gives a brief comparison of some of the existing solutions and our proposed security style sheets. We have grouped existing solutions into several broad categories. Not all of the solutions discussed fit clearly into these categories, but this is intended to give a rough picture of the current solution space. Notably, AdJail [16] is not represented in the table.

Whitelisting Whitelist-based solutions such as CSP [21] and SOMA [18] work by providing a list of ap-

Issue → Solution ↓	Script Injection	Content Injection	Information Leakage	CSRF	Clickjacking
Whitelisting (e.g. CSP, SOMA)	whitelist	whitelist	whitelist	whitelist	no
Sandboxing (e.g. MashupOS, HTML5)	limit damage	no	limit damage	no	limit damage
Script re-writing (e.g. Caja, JSReg)	limit damage	no	limit damage	no	no
Security Style Sheets	whitelist, limit damage	whitelist, limit damage	whitelist, limit damage	whitelist	limit damage

Table 1: A comparison of some browser-based web security solutions.

proved sites with which the target site can communicate. This limits the range of sites from which malicious scripts/content can be loaded, the sites with which the target site can communicate, and even the sites for which the target site can be used in a CSRF attack.

Sandboxing Sandboxing within the page is something explored by a large variety of mashup work including MashupOS [25], Subspace [14], SMash [9], OMash [8], ViSP [17] and many others. More recently, sandboxing has become part of the proposed HTML5 standard as a sandbox property for iframes [24]. Sandboxing is a mitigation strategy that limits the parts of the page that a malicious script might access, thus limiting an attack's ability to gain information or overwrite areas for the purpose of clickjacking.

Script re-writing Script rewriting techniques such as Caja [3] or JSReg [12] concentrate on a different method for limiting the scope of attacks by rewriting the JavaScript to a more constrained subset.

Security style sheets integrates ideas from all of these pre-existing systems to unify them under a single syntax, thus making it easier to consolidate security policy for a given web site. While similar results could be achieved by using several of these systems, the policy would be fragmented and users wishing to create policy would need to learn several policy languages and techniques in order to ensure full coverage.

Security style sheets handles these issues in very similar ways to its predecessors:

Script Injection Like CSP and SOMA, security stylesheets allows for maintaining a whitelist of "approved" content, thus limiting the attack vectors to a (potentially very small) list of previously known sites. Similar to ViSP and the HTML5 sandbox, it can constrain the actions of a script to a much smaller area of the page, so that even if a malicious script is inserted it may not be able to access sensitive data. Finally, security style sheets allow for areas where scripts will not be executed, thus nullifying any script-based attack in those regions.

Content Injection Much like with script injection, security stylesheets uses a white list to limit the potential attack vectors, and then places further limits on the page by limiting within-page communication and constraining the area within which the content may be displayed.

Information Leakage Again, security style sheets restricts the flow of information out of the page by limiting the requests that can be made. All requests out must be part of a known whitelist. In addition, security stylesheets limits the potential damage caused by an information leak by limiting the within-page communications so that it may be impossible for sensitive information to leak from one part of the page to another which may have a more permissive whitelist.

Cross-Site Request Forgery SOMA has a "double-sided whitelist" which is a way of indicating that it is in fact several whitelists: one of the site of the site being visited, and another for each third party content provider for that site. Both the original site and the content provider's whitelists must agree that a site is approved before anything is requested and loaded. CSP however has a single sided whitelist, where only the site being visited provides an approval list and it is presumed that the content-providing site will handle their own requests via use of the Origin header. Security style sheets takes the same approach as CSP, assuming that content providers will handle their own content requests using Origin: or other technologies such as Adobe Flash's `crossdomain.xml` file [6]. This client-side whitelist stops requests from going out, thus preventing the website from attacking other sites (but not protecting it from CRSF from other sites).

Clickjacking Inspired by the work of AdJail, security style sheets places limits on the visual display of content in special regions. This makes clickjacking attacks more difficult, since many rely upon large page overlays or specifically targeted areas where the user is likely to click. If these regions have been protected, no content will be overlaid on them and clickjacking attacks should be curtailed.

6 A real world example: User-generated comments

To get a better sense of how security style sheets might work in practice, here we present a policy example.

Cross site scripting is often injected into a page via forms used to allow users to submit information. For example, rather than submitting a comment which simply says `All your base are belong to us` a malicious user can submit `All your base are belong to us. <script src=‘‘http://example.com/evil.js" />` and then `evil.js` will be loaded into the page to wreak whatever damage it was designed to do.

Let us suppose that we have a section of the page which includes a comment that was submitted by a user, who may or may not be trustworthy. What sort of security stylesheet policies would protect the rest of the page from attack? Listing 4 gives a possible highly restrictive policy which allows no execution, no loading of external content, and gives no other parts of the page access to this section just in case.

```
<div class="comment"> Comment goes here </div>

<style type="text/sss">
.comment {
   execute: no;
   approved-domains: none;
   approved-elements: none;
}
</style>
```

Listing 4: Restrictive comment policy

However, this may be too restrictive. What about on a real site? Facebook allows users to post status messages that have a similar use case: we don't want code executed, but we might want to attach a photo or video. The message itself is in a span with the class `messageBody`. As such, we could use a policy like the one shown in Listing 5, which allows inclusions from other sources, but does not allow execution. In addition, in order to allow users to "like" the update, we need to allow JavaScript which has been included in the headers the ability to modify the page to give feedback when the user has pressed the "like" button.

```
<style type="text/sss">
.comment {
   execute: no;
   approved-domains: all;
   approved-elements: document.head;
}
</style>
```

Listing 5: Potential Facebook status update policy

7 Conformance testing

In order to facilitate and evaluate any given implementation of security style sheets, we have developed a set of conformance tests to ensure that the implementation successfully creates the necessary encapsulation and communication.

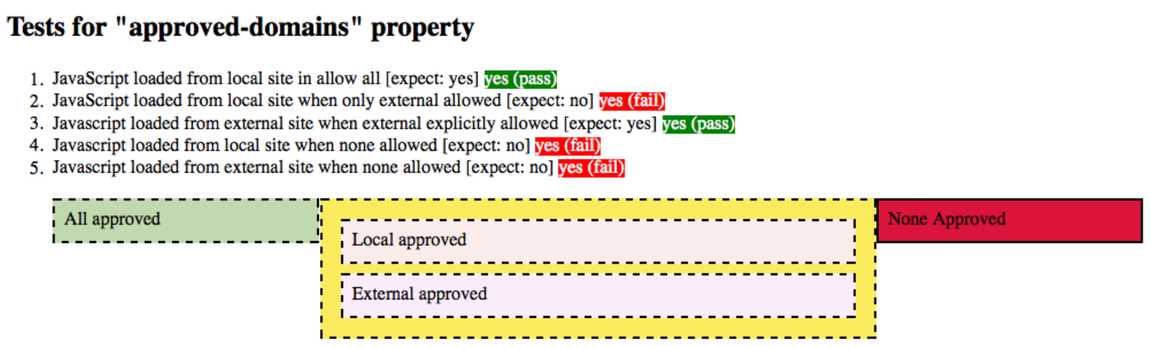
Because security style sheets have been built upon the idea of the web layout, this conformance suite has been based upon another type of conformance suite used for web standards. These are the acid tests created by the Web Standards Project [22]. The acid3 test is a web page with a JavaScript test harness that actually runs one hundred subtests related to web specifications. This particular test focuses mainly upon web 2.0 technologies including HTML4, CSS3 and ECMAScript.

One benefit to this test is that as well as testing the expected security properties of the page, it can also be used to test the final layout by comparing the in-browser rendering to that of a reference image.

7.1 approved-domains

To test the `approved-domains` property, we need to load additional materials in boxes with various properties. We test loading within a fully open box, loading within two types of semi-permeable boxes (one which allows local scripts, and one which allows external ones from a specific domain), and attempting to load within a box which has no allowed domains. The expected values for these tests are shown in Figure 4, alongside the results which actually occur in an unmodified browser. (If this test were run within a fully compliant browser, the results would all show green with "(pass)" marked beside each one.)

The content loaded from both local and external servers is simply a short snippet of JavaScript which attempts to modify the result (given in a `span` tag). For


Tests for "approved-domains" property

1. JavaScript loaded from local site in allow all [expect: yes] yes (pass)
2. JavaScript loaded from local site when only external allowed [expect: no] yes (fail)
3. Javascript loaded from external site when external explicitly allowed [expect: yes] yes (pass)
4. Javascript loaded from local site when none allowed [expect: no] yes (fail)
5. Javascript loaded from external site when none allowed [expect: no] yes (fail)

Figure 4: `approved-domain` conformance test: results for an unmodified browser

example, one such piece of JavaScript code is shown in Listing 6. If we were to constrain different types of content in different manners as with CSP, we would have to alter this test to include more content-types and to attempt to load things which are not JavaScript. However, for the simplified approved-domains property, we can load just a single line of JavaScript for testing.

```
document.getElementById("result-domain5").innerHTML↩
      = "yes";
```

Listing 6: JavaScript code used to test approved-domains

7.2 approved-elements

There are five tests for the `approved-elements` property's basic communication abilities. The tests here use the metaphor of "finding a needle in a haystack" to identify the parts of the page being used. All of the code in this case is contained within the element designated as the "searcher" which attempts to find the "needle" elements both inside and outside of a "haystack" element.

The first test ensures that one can find a needle when safely placed outside of the haystack, and when full access has been given to the needle element. This test should pass in even unmodified browsers. The next test tries to find a needle which has been hidden within the haystack. While the haystack allows all elements access, the needle allows no access to other elements. As such, this should not be found in a security-stylesheet compliant browser. This tests both the inheritance from the haystack and the behavior of the `approved-elements: none` setting of the property.

The next test is one to ensure that needles are not being found in areas where there are no needles. This is more of a test of the JavaScript in the browser than a specific test of security stylesheet behavior. Finally, the last two needles in the haystack have permissions between the extreme of all access and no access: one gives access to the searcher element (and thus should be found), and the other gives access to an element that is not the searcher element (and thus should not be found).

Figure 5 shows the tests run in a web browser that does not support security stylesheets. Listing 7 shows the `findNeedle` function and the tests run within the XHTML shown in Figure 5.

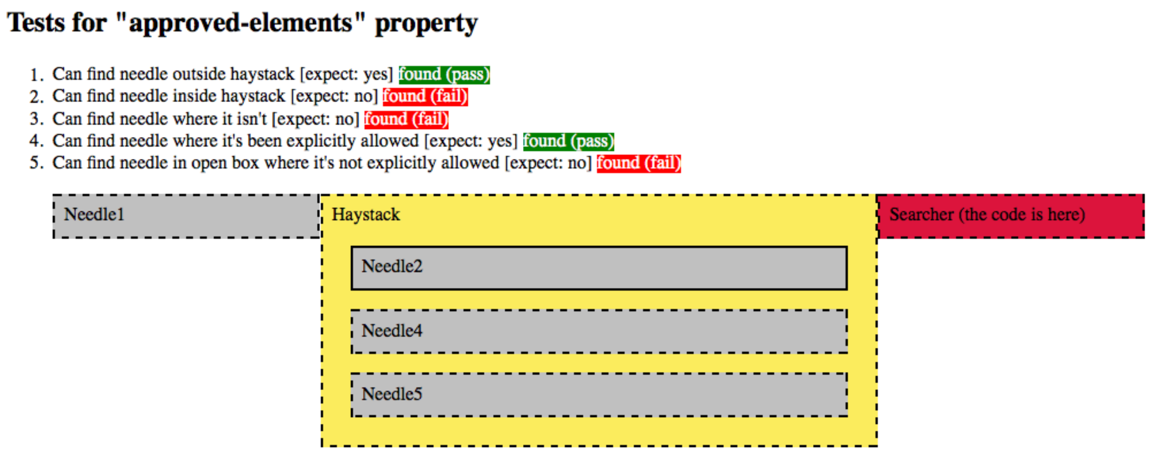
```
function findNeedle(testname, needle, start) {
    var elements = start.getElementsByTagName("*");
    var i = elements.length - 1;
    var result = document.getElementById(testname);
    result.innerHTML = "not found";
    while (i >= 0) {
        if (elements[i] && elements[i].innerHTML &&
                elements[i].tagName != "SCRIPT" &&
                elements[i].innerHTML.match(needle)↩
                    ) {
                result.innerHTML = "found";
                return;
        }
        --i;
    }
}

// tests for allow-elements
findNeedle("result-needle1", "Needle1", document);
findNeedle("result-needle2", "Needle2",document.↩
     getElementById("haystack"));
findNeedle("result-needle3", "Needle3", document);
findNeedle("result-needle4", "Needle4", document);
findNeedle("result-needle5", "Needle5", document);
```

Listing 7: The findNeedle function used to test approved-elements

7.3 execution

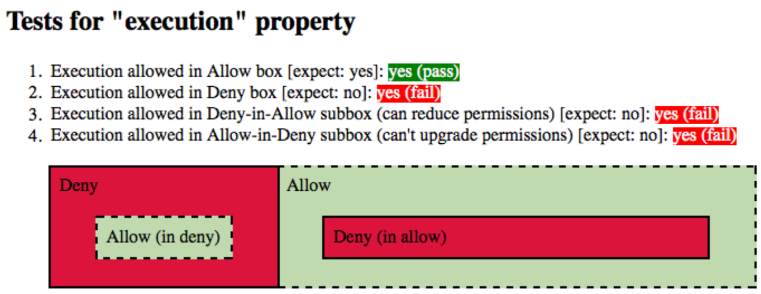
The `execution` conformance tests are shown in Figure 6, as they appear in a browser that does not support



Tests for "approved-elements" property

1. Can find needle outside haystack [expect: yes] found (pass)
2. Can find needle inside haystack [expect: no] found (fail)
3. Can find needle where it isn't [expect: no] found (fail)
4. Can find needle where it's been explicitly allowed [expect: yes] found (pass)
5. Can find needle in open box where it's not explicitly allowed [expect: no] found (fail)

Figure 5: `approved-elements` conformance test: results for an unmodified browser



Tests for "execution" property

1. Execution allowed in Allow box [expect: yes]: yes (pass)
2. Execution allowed in Deny box [expect: no]: yes (fail)
3. Execution allowed in Deny-in-Allow subbox (can reduce permissions) [expect: no]: yes (fail)
4. Execution allowed in Allow-in-Deny subbox (can't upgrade permissions) [expect: no]: yes (fail)

Figure 6: `execution` conformance test: results for an unmodified browser

security stylesheets.

There are four tests for the `execution` property. The first two tests verify behavior for the two possible settings: `yes` and `no`. The test for `execution: yes` will pass even in a normal browser, while `execution: no` will fail in a normal browser because all code is automatically executed.

The next two tests are for the behavior of sub-trees. Here, we must be careful of privilege escalation: it should not be possible to make and use a `execution: yes` area within an existing `execution: no` one, otherwise a clever attacker would simply escalate privileges to ensure that their malicious code will run. We do, however, support the restriction of privilege, so it should be possible to make a `execution:no` area within a `execution: yes` area and have both behave as described.

The code used for these tests is very simple: it attempts to change the value of the test result section (the red or green box at the end of each list item) to be `yes` instead of `no`. The actual colors and the pass/fail note are provided by another function. A sample of this simple code, including the surrounding XHTML, can be seen in Listing 8.

```
<div id="deny-in-allow" class="deny subbox">
    Deny (in allow)
   <script type="text/javascript">
    document.getElementById("result-deny-in-allow")↩
         .innerHTML="yes";
   </script>
</div>
```

Listing 8: Sample execution testing code (the deny subbox within the allow box)

8 Discussion

It is our hope that Security Stylesheets can be used as the basis for a more extensive security policy language for the web. Using the oft-quoted aphorism "complexity is the enemy of security" we have chosen to minimize the number of options we specify here. However, we recognize that developers may in practice require more fine-grained control. In order to accommodate additional expressiveness without compromising simplicity for users who do not need it, we suggest that shorthand properties be used much like they are elsewhere in CSS. For example, if one wants to set the margin of an element, a simple `margin: 10px` could suffice, but if one needs more fine control over the margin, the `margin-left, margin-right, margin-top, margin-bottom` properties are all available.

Consider the idea of detailed object-type based whitelisting, as proposed in CSP. If we wanted to add these to our `approved-domain` whitelist, we could have `approved-domain-image, approved-domain-script, approved-domain-object` etc. If the web developers want to allow all content from `https://example.com` they could still use `approved-domain: https://example.com` but if only images were desired then the policy could read `approved-domain-images: https://example.com` to be more specific.

Similarly, it seems possible that the developers might want finer control over reading and writing, rather than the blanket access provided through `approved-elements`. So to provide more specific access, one might have `approved-elements-write` and `approve-elements-read` depending on what actions can be performed on a given piece of content.

As long as shorthand properties are carefully thought out, it should be possible to increase the complexity of security style sheets without irreparably compromising the simplicity of the basic design.

9 Conclusions

Security style sheets is a way to harmonize currently separate client-side web security techniques. By combining the ideas of whitelisting, sandboxing and others into a single syntax, we aim to limit the amount of time required for interested web site owners to provide greater security for their sites. Because this syntax is inspired by cascading style sheets, policy writers will be able to leverage information already provided as part of a web site's design, thus easing the creation of new policy.

Security style sheets can provide a platform for integration of other client-side web security techniques within a standardized framework. We hope that this paper and the initial conformance test suite can inspire discussion within standards bodies who are interested in providing greater security for the modern web.

10 Availability

At time of publication, the security style sheets conformance suite will be made available online for use in testing.

References

[1] Authentication context for the oasis security assertion markup language (saml) v2.0. Tech. Rep. saml-authn-context-2.0-os, OASIS, Mar 2005.

[2] Web services security: SOAP message security 1.1 (ws-security 2004). Tech. Rep. wss-v1.1-spec-os-SOAPMessageSecurity, OASIS, Feb 2006.

[3] google-caja: A source-to-source translator for securing javascript-based web content. *Google* (2010). `http://code.google.com/p/google-caja/`.

[4] Web application security consortium: Threat classification v2.0. Tech. rep., Web Application Security Consortium, 2010. `http://projects.webappsec.org/Threat-Classification`.

[5] WhiteHat website security statistic report: Fall 2010, 10th edition – industry benchmark. Tech. rep., WhiteHat Security, Fall 2010.

[6] ADOBE SYSTEMS INCORPORATED. External data not accessible outside a Macromedia Flash movie's domain. Tech. Rep. tn_14213, Adobe Systems Incorporated, Feb 2006.

[7] BARTH, A., JACKSON, C., AND MITCHELL, J. C. Robust defenses for cross-site request forgery. In *Proc. of the 15th ACM Conference on Computer and Communications Security (CCS '08)* (Oct 27-31 2008), ACM, pp. 75–87.

[8] CRITES, S., HSU, F., AND CHEN, H. Omash: Enabling secure web mashups via object abstractions. In *Proc. of the 15th ACM Conference on Computer and Communications Security (CCS '08)* (Oct 27-31 2008), ACM, pp. 99–107.

[9] DE KEUKELAERE, F., BHOLA, S., STEINER, M., CHARI, S., AND YOSHIHAMA, S. Smash: Secure cross-domain mashups on unmodified browsers. Tech. rep., IBM Research, Tokyo Research Laboratory, Jun 11 2007.

[10] FREIER, A. O., KARLTON, P., AND KOCHER, P. C. The ssl protocol version 3.0. Tech. Rep. draft-freier-ssl-version3-02.txt, Internet Engineering Task Force (IETF), 1996.

[11] HERLEY, C. So long, and no thanks for the externalities: The rational rejection of security advice by users. *Proc. of The 2009 New Security Paradigms Workshop (NSPW'09)* (Sep 8-11 2009), 133—144.

[12] HEYES, G. JSReg update, Jul 2009. `http://www.thespanner.co.uk/2009/07/20/jsreg-update/`.

[13] JACKSON, C., AND BARTH, A. Forcehttps: Protecting high-security web sites from network attacks. In *Proceedings of the 17th International World Wide Web Conference (WWW2008)* (Beijing, China, Apr 21–25 2008).

[14] JACKSON, C., AND WANG, H. J. Subspace: Secure cross-domain communication for web mashups. In *Proc. of the 16th International World Wide Web Conference (WWW2007)* (Banff, Alberta, May 8-12 2007).

[15] JIM, T., SWAMY, N., AND HICKS, M. Defeating script injection attacks with browser-enforced embedded policies. In *Proc. of the World Wide Web Conference (WWW2007)* (Banff, Alberta, May 8-12 2007).

[16] LOUW, M. T., GANESH, K. T., AND VENKATAKRISHNAN., V. Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *19th USENIX Security Symposium* (Washington, DC, USA, Aug 2010).

[17] ODA, T., AND SOMAYAJI, A. Visual security policy for the web. In *USENIX Workshop on Hot Topics in Security (HotSec '10)* (Aug 10 2010).

[18] ODA, T., WURSTER, G., VAN OORSCHOT, P., AND SOMAYAJI, A. SOMA: Mutual approval for included content in web pages. In *ACM Computer and Communications Security (CCS'08)* (Oct 2008).

[19] PROVOS, N., MCNAMEE, D., MAVROMMATIS, P., WANG, K., AND MODADUGU, N. The ghost in the browser: Analysis of web-based malware. In *Workshop on Hot Topics in Understanding Botnets (HotBots)* (Apr 2007), vol. 10.

[20] SHEA, D. CSS zen garden: The beauty in CSS design, Checked Jan 2011. `http://csszengarden.com`.

[21] STERNE, B. Security/csp/spec. Tech. rep., Mozilla Corporation, 2009.

[22] THE WEB STANDARDS PROJECT. Acid3 browser test. `http://www.webstandards.org/action/acid3/`.

[23] W3C. 6.4 the cascade. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification: W3C Working Draft 07* (Dec 2010).

[24] W3C. 4.8.2 the iframe element. *HTML5: A vocabulary and associated APIs for HTML and XHTML. Editor's Draft 9* (Feb 2011).

[25] WANG, H. J., FAN, X., HOWELL, J., AND JACKSON, C. Protection and communication abstractions for web browsers in MashupOS. In *21st ACM Symposium on Operating Systems Principles (SOSP)* (2007).

[26] WICHERS, D. OWASP top 10 2010. *The Open Web Application Security Project* (2010).