12-3-2012

Block-scoped access restriction technique for HTML content in web browsers

Timothy Watt

Block-scoped Access Restriction Technique for HTML Content in Web Browsers

By

Timothy Watt

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Security and Information Assurance

Rochester Institute of Technology

B. Thomas Golisano College of Computing and Information Sciences

December 3, 2012

Rochester Institute of Technology

B. Thomas Golisano College of Computing and Information Sciences

Master of Science in

Computing Security and Information Assurance

Thesis Approval Form

Student Name: __ Timothy Watt _____

Thesis Title: _ Block-scoped Access Restriction Technique for HTML Content in Web Browsers _

Thesis Committee

Name	Signature	Date
Dr. Bo Yuan Chair		12/3/2012
Dr. Sumita Mishra Committee Member		12/3/2012
Dr. Yin Pan Committee Member		12/3/2012

Abstract

Web sites, web browsers, web site authors, web component authors, and end users interact in a complicated environment with many recognized and unrecognized trust relationships. The web browser is the arena in which many important trust relationships interact, thus it bears a considerable burden in protecting the interests and security of web end users as well as web site authors. Existing proposals, draft standards, implemented features, and web application techniques go a long way towards allowing rich and compelling content interactions, but they do not provide for rich, mutually-distrusting content to be safely embedded in a single page. This proposal suggests a declarative policy mechanism that permits untrusted content to be safely embedded in a web site while still retaining some richness. It also suggests a policy integration approach to allow multiple cooperative (but not necessarily trusting) parties to provide components of a policy that combine together in a safe manner. It incorporates techniques including fine-grained and coarse-grained permission dropping and white-listing protections for retained capabilities. Finally, the proposed concepts are applied to a number of real-world CVE vulnerabilities, and it is explained how the proposal does or does not prevent or mitigate the attack. The solution is shown to be effective against cross-style-scripting style attacks, and to not be effective at preventing incoming cross-site request forgery attacks.

1. Table of Contents

1.Table of Contents....1
2.Introduction....2
3.Trust....6
4.Weaknesses....9
4.1.Web Application....9
4.2.Web Browser and Infrastructure....12
4.3.End-User....14
5.Security Policies....16
6.Cross-Site Scripting....20
7.Related Work....22
7.1.CSRF Mitigation....22
7.2.Distrusting Embedded / Enclosed Content....24
7.3.Declarative Policy Using Custom HTTP Headers....25
7.4.The Gap....26
8.Solution: HTML Privilege Dropping....26
9.Attack Resilience Analysis....32
9.1.Samy Worm (XSS)....33
9.2.WMF Image Flaw....34
9.3.General Non-Persistent XSS....35
9.4.CVE-2008-5249 - Cross-site Scripting (XSS) Vulnerability in MediaWiki....36
9.5.CVE-2008-0971 - Barracuda Networks Products Multiple Cross-Site Scripting Vulnerabilities 36
9.6.CVE-2006-4308 - Multiple Cross-site Scripting (XSS) Vulnerabilities in Blackboard Learning System....37
10.Review of Complementary Solutions....37
11.Conclusions....38
12.Unanswered Questions and Future Work....39
13.Bibliography....41

2. Introduction

The concept of “web security” cannot be explored without an understanding of the complex environment of numerous entities and supporting technologies, each with different responsibilities, trust relationships, assumptions, and weaknesses. At the simplest level, the environment includes the end-user (a person), web browsing software (“the web browser”), a computing device (a laptop, computer, or mobile device), a network linking the end-user's computing device to the web server, a web server (software running on a computer), web content being hosted by the web server, and a content producer (“web site owner” or “web site author”; sometimes, also a “web application author”). The communication between the web browser will involve DNS (domain name service) queries, one or more TCP/IP connections, HTTP (hypertext transport protocol), and sometimes SSL (secure sockets layer) or TLS (transport layer security). This highly simplified environment represents the end-user's perspective of most web activities. In many web sites, however, the “content producer” scenario suggested above is overly simplified: web site authors commonly integrate content produced and hosted outside of their control on their web sites (as simple examples, advertisements and end-user comments / discussions). For the purposes of this work, the primary focus will be on content with multiple authorship across multiple organizations (sometimes called 'mash-ups' or 'web 2.0', but also common in traditional web sites).

It is useful to elaborate on the meaning of “content” when comes to modern web sites. For this work, “content” extends to include HTML (hypertext markup language), images, CSS (cascading style sheets), JavaScript, Flash Applets, PDF (portable document format) files, and Java Applets. There are other kinds of data that can be created and published by web site authors, but most other types are not directly consumed by end-user web browsers. Even among this list of content types, only some are directly processed by the web browser: HTML, images, CSS, and JavaScript are the web browser's

purview, while Flash and Java Applets as well as PDF files are handled by plug-ins (software that is dedicated to assisting the web browser to process non-core content types). It is important to note that many of these content types are themselves inter-dependent and complex: HTML can contain JavaScript in several different forms (both direct and by way of linking to a separate URL), HTML can contain CSS (both direct and by way of linking to a separate URL), JavaScript can produce or modify HTML, CSS, and other JavaScript, and CSS can contain JavaScript (through advanced expression formats) and images (indirectly, by linking to a separate URL). The non-core formats (PDF files and Flash and Java Applets) can also contain one or more of these technologies, but that aspect is not explored further within this work. Of the core formats, only images are incapable of containing other core formats [22, 23, 24].

Content authorship is also more complex than the simple scenario presented earlier. A modern web page will include content produced by a number of authors. Web advertising is a multi-author affair all by itself (usually consisting of at least an advertising network and an advertisement author). Further, many web sites accept (or are designed around) content authored by its users or other parties. Google News, for example, self-hosts articles produced by the Associated Press and also serves headlines and article summaries that are automatically derived from external news sites. Social networking sites such as Facebook and Google Plus serve user-generated content, their own content, and advertising content; user-generated content includes profile pages, videos, images, and comments, while self-authored content consists of the core content management functionality (comment posting forms, authentication forms, and the like), and advertisers and advertising networks produce advertisements. Of all the authorships listed, it is usually user-generated content that is given the least amount of trust by the site owner (due, e.g., to lack of a business relationship), though advertising content is often given less trust than self-authored content. This trust relationship between the different

entities will be explored further, as it plays a significant role in shaping web security policies and capabilities.

As identified earlier, there are multiple technologies that operate in concert to deliver content from a web host to an end-user's web browser. The DNS system (including DNS servers, which operate independently from web servers, and DNS resolvers, which live in web browsers and operating system libraries) is responsible for mapping host names (text names assigned to web and non-web servers) to IP addresses, which are needed to carry out TCP/IP connections. The HTTP protocol defines a standardized way for web browsers to communicate with web servers to request web content and to express user-supplied operation instructions (such as logging in, updating a Facebook profile, or posting a comment. The SSL and TLS protocols define a standardized way for web browsers to authenticate web servers and provide confidentiality and integrity for communications between the web browser and web server. Beyond the communications, the HTML, CSS, and JavaScript languages provide web authors with the tools to compose their content for end-users to view (with various image formats providing support for defining images to embed in this content). Applet formats such as Flash and Java provide web authors with advanced and rich opportunities for end-user interaction, such as playing videos with sound. Web browsers and web servers themselves also play a heavy role in the technology stack, as the end-user and web authors use them directly to accomplish their respective goals.

Web browsers serve as the primary face of all web sites as experienced by end-users. They are responsible for requesting, fetching, assembling, interpreting, and showing web authors' contents in an integrated form (the web page). They present web content to end-users on display devices such as computer monitors or mobile phone screens. They incorporate a myriad of operating system services, third-party functionality, and internally-implemented functionality. The list of responsibilities include

evaluating JavaScript, evaluating and applying CSS styling, using fonts to draw text, using OS services and application libraries to render image content, performing DNS querying, performing TCP/IP communications, performing HTTP communications, performing SSL/TLS certificate validation, and providing the end-user with information about the current state of the web browser (what web site is being displayed as well as what security and integrity guarantees exist about the currently-displayed web site) [25].

Together with web servers, web applications provide the web author with functionality comparable to that provided to end-users by web browsers. A web server provides core functionality such as HTTP communications, TCP/IP communications, and SSL communications. A web application is then responsible for processing the HTTP requests and providing the content to send to the requesting end-user's web browser. In the simplest case, a web application is not necessary: the web server can act like a simple file server and send unchanging web sites to visiting end-users. Modern web sites have more dynamic behavior, where the web site is expected (by end-users and web authors) to remember personal information and preferences for individual end-users and to provide different content (“dynamic content”) based on many factors (such as what comments other end-users have posted or what geographical region the end-user is located in). Web applications are typically written in programming and template languages such as PHP, Java (and GWT, JSP, JSF, or Struts), C#, Ruby (and Ruby on Rails), Groovy (and Grails), and they often involve structured storage managers such as relational databases as well as domain-specific languages such as SQL. Web applications may further rely on earlier-mentioned technologies such as HTTP, JavaScript, TCP/IP, and SSL/TLS in the process of communicating with other services (possibly published by other organizations) to produce or react to the end-users' requests and commands. Many complex choices inherent to modern web sites fall under the control of the web authors and the web applications they write and/or use.

3. Trust

Trust is an important concept inherent in the web technology stack. Trust exists between end-users and web authors, between web authors and web application providers and technologies, between web authors and web service/component providers (a form of web author who publishes re-usable fragments or services instead of end-user-facing web sites), between web authors and web browser providers, between end-users and web browser providers, between web browser providers and DNS and related provider entities (such as web registrars, Internet Service Providers, and SSL certificate authorities), and between web authors and the web application and server technologies (including the DNS system, SSL/TLS, and so on).

Some of the listed trust relationships are not apparent to the parties engaging in them. A great example of this is DNS: web authors and end-users will trust that the DNS infrastructure will resolve 'www.google.com' to an IP address owned by Google[1]. At a web application level, web site / application authors can inadvertently trust the integrity and predictability of input values from HTTP requests (e.g., by expecting that only values permitted by a particular HTML form could occur in that input). At the other end of the relationship, the end-user may trust (really, assume) that information they provide within a web form will be submitted to the web site they are currently visiting (or some site legitimately operating on its behalf).

In addition to non-apparent trust relationships, web authors and end-users also participate in overt trust relationships. An end-user will express trust in a web author providing a service by entering their credit card information into a form on that web author's site. They may trust that the appearance of a “pad lock” icon in the web browser guarantees that information they submit in a form will be protected when it is sent to the web site and that the site they are viewing has not been tampered with

1 The DNSSEC protocol adds integrity assurances to the DNS protocol, but DNSSEC merely shifts the trust relationship from the DNS infrastructure to the DNSSEC signing authorities; SSL is a more reliable way to verify this trust, though it mainly shifts trust from the network infrastructure to the SSL certificate authority infrastructure.

[26]. Web authors may trust their web hosting provider (such as GoDaddy, RackSpace, or similar services) to prevent unauthorized changes from being made to their content.

At the web browser side, some important trust decisions and assumptions are built into the web browsers (often because they are built into the standards the web browser implements). One important trust expectation is that all content served within a single origin all deserves to be treated as equal [15, 27]. In this context, 'origin' is a technical term that refers to key portions of the URL that content originated from: the contents of 'http://www.google.com/ig/' has the origin '(http, www.google.com, 80)', where 80 comes from the implied default port for the HTTP protocol. The 'treated as equal' aspect means that JavaScript running within a given origin will have access to any cookies, HTML DOM (document object model) objects, and any JavaScript variables and functions sharing the same origin that it has access to (including other frames, and also including permission to use advanced web browser capabilities to interact with that origin without the end-user's explicit consent, such as with AJAX[2]). Based on the HTML standards, web browsers will also extend an 'origin' to include JavaScript loaded from web servers with a different origin: 'SCRIPT' tags with a 'SRC' attribute pointing to a remote resource will extend the current page's origin to that script, regardless of where the URL points (for example, if 'http://www.google.com/ig' includes "<SCRIPT SRC='http://evil.example.com/evil.js'>", that JavaScript will run with the origin '(http, www.google.com, 80)', NOT '(http, evil.example.com, 80)'). This trust relationship is often referred to as the same-origin policy (SOP). Less surprising but no less important, web browsers also impart an HTML page's origin to any JavaScript directly contained in it (in the form of in-line scripts and event handlers). Within a single page, the standardized way to change the origin is by the web author including content with a 'FRAME' or 'IFRAME' tag whose URL is sufficiently different from the

2 AJAX is a term that refers to background communications carried out by the web browser on behalf of JavaScript code; it means Asynchronous JavaScript and XML, because it is often used to send and receive XML documents or other background information to the web server it was retrieved from.

current page (i.e., where the scheme, host name, and/or port number are different from the original page).

At the web author side, there are a number of interesting trust relationships as well (some resulting primarily from technology or standards limitations). The ability to incorporate remote web components from other component providers comes with a trust-related trade-off: by using framing techniques (IFRAME or FRAME), the component can be fully isolated from the integrating web site, but this usage prevents the integrating web author from interacting with the loaded component (SOP applies in both directions: neither origin can interact with the other); alternatively, the integrating web author can extend full trust to the component and its provider by loading it in a SCRIPT, thus granting it the same access that the web author has over the integrating web page as well as background AJAX interactions with the integrating web server [15, 27]. A related decision occurs when loading first-party content (content procured by the web site author or by authorized contributors, such as journalists publishing their news articles or social network users publishing updates to their profile): the site author can choose to trust it fully by serving it from the same origin or embedding it in the same page (which allows the site author to interact with it from JavaScript and apply CSS styles to it), or the site author can serve it framed from a different origin (which may force the site author to choose less-preferred page layouts due to the visual restrictions imposed on frames). For site authors that want host such content from the same origin (e.g., to apply CSS styles to it or to interact with it using JavaScript) while also forbidding the use of JavaScript within that content, various types of web application filtering can be performed. As will be discussed later, successfully filtering content to impose policies on it (such as “styles are okay, but scripts and event handlers are not”) is difficult in the face of web browser implementation variations and ongoing HTML evolutions.

A recent addition to web application standards named 'Cross-domain messaging' [8] came into

existence before 2009. This addition offers a JavaScript mechanism for a web application author and a web component author to arrange for controlled communications between them across different origins within the web browser. This offers a new choice for web application authors without a need to support legacy web browsers: origin-based separations can be used without completely removing support for container-component interactions.

4. Weaknesses

There are weaknesses in many of the technologies that make up the web technology stack discussed earlier (including the end-user). In this section, weaknesses exhibited in many of the different steps will be discussed. Specific weaknesses will be explored in more depth.

4.1. Web Application

Within a web application, there are a number of weaknesses that can occur, depending on the nature of the web application. In typical implementations, web applications have access to resources intended to be available to any of its end-users. When resources take the form of files, the web application might reasonably use a path to access that resource. One weakness arises, called 'directory traversal', when the web application uses end-user provided input to determine what resource to provide, and the web application fails to sufficiently escape, filter, or check the end-user's input [28]. A naïve implementation, for example, might use a filename provided by an end-user to construct a path, starting from the end-user's assigned storage path. If the assigned storage path is '/storage/user1' and the requested resource is '../user2/passwords.txt', a naïve treatment of the requested resource might permit the end-user to obtain another end-user's data (in this case, 'passwords.txt' for the 'user2' user). More subtle variations of this weakness may depend on unexpected or unknown weaknesses in the underlying infrastructure, such as an operating system or application library that accepts illegal 'overlong' UTF-8 sequence 'C0 AE' to mean '.' (Unicode '002E') [29].

A common web application weakness is 'SQL injection', which refers to the ability for an attacker (or possibly an unaware end-user) to cause custom input to be sent to a relational database manager as part of a SQL (structured query language) statement [28]. SQL injection weaknesses represent improper web application programming techniques, and can stem from improper treatment of HTTP request parameter inputs. The ability to inject SQL often grants the attacker a number of capabilities, such as retrieving, tampering with, or destroying data accessible by the web application (possibly including sensitive data normally restricted to other end-users or web site authors). The ability to tamper with data gives the attacker another interesting level of access: other entities (including end-users) that access the tampered-with data may extend trust to it related to their expectation of the originator of the data (for example, if a student tampers with grades stored by a university web application, administrators who access the web application might trust the tampered-with grades reported by the web application; alternately, if arbitrary data is inserted into a database field that normally only contains unstructured text, another application which receives and interprets that field from that database might fail to operate correctly, possibly itself becoming a victim of the original attack).

Cross-site scripting is another weakness that is exhibited by web applications. It is structurally related to SQL injection in that it stems from improperly escaping, filtering, or validating data provided in the HTTP request parameters. The difference between SQL injection and cross-site scripting (XSS) is the component that suffers from the mis-handled input: for SQL injection, the database experiences the direct effects (by executing SQL statements that the web application author did not intend or anticipate), and for XSS, the victim is the end-user's web browser. In the case of XSS, a common goal is for the victim to run JavaScript code not intended by the web application author with the origin of the web site containing the weakness. As mentioned earlier, the ability to run JavaScript with a

particular web site's origin means that JavaScript can silently interact with the web site on behalf of the user. This JavaScript would have access to data stored by the web site as well as cookies set by that web site on the victim's web browser. When origin-agnostic JavaScript features are also brought to bear, the victim's web browser can be directed to send data the victim can access to a third party (commonly, the attacker). For example, is a bank has an XSS weakness, an attacker can coerce a victim end-user to click on an attacker-controlled link, visit an attacker-controlled web site, or visit a page on the bank's web site containing a previously-stored attacker-supplied fragment. In this example, the former variations (attacker-controlled link or web site), the weakness is called a 'reflected XSS' vulnerability, while the latter variation (page on the bank's web site) is a form of 'stored XSS' vulnerability. The mechanism for accomplishing these attacks is not pertinent for discussion here, since the impact is the same: the victim end-user's (and the end-user's web browser's) trust of the web site and its author is violated. Using this same XSS weakness, the attacker can also use the victim end-user's web browser to carry out unintended actions on behalf of that end-user, such as sending electronic payments to the attacker. This type of weakness, and ways to mitigate it, will be explored further.

The cross-site request forgery (CSRF) weakness is related to the XSS weakness, but it involves inappropriate trust by the web application author in the intent of a web browser submission of HTTP requests [19]. Unlike the XSS weakness, this weakness does not involve the inputs contained in the request; instead, the weakness is that the web application author trusts that the occurrence of a particular HTTP request occurred because the end-user intended it to occur. In this case, an example might make the subtle aspects more clear. If an end-user fills in a bank transfer web form and clicks “submit”, a reasonable observer would conclude that the resulting HTTP request (which includes the end-user's session cookie) expresses the end-user's intent for the bank to conduct transfer it described.

If, however, the end-user's web browser loaded evil.example.com while also carrying a live session cookie for bank.com, a reasonable observer would disagree that a hidden IMG in the 'evil.example.com' pointing to 'bank.com/transfer?from=victim&to=attacker&amount=10' represented the same end-user intent as the earlier example, even though it would include the same session cookie. However, the standards-compliant web browser would dutifully load the image in the second example, which could be interpreted by a naïve web application as the former situation. In this contrived example, the web application is exceptionally naïve at both ignoring HTTP standards recommendations that a 'GET' request (caused by the IMG) be used only to retrieve information and not to perform actions or have side-effects. A less contrived example would have 'evil.example.com' host a HTML FORM whose 'action' was 'bank.com/transfer' and which included hidden INPUT fields containing the earlier transfer parameters (JavaScript would then be used to submit the form without the user clicking anything). The challenge in dealing with this weakness is finding a way to restore the web application author's need to detect the end-user's intent (and differentiate it from automatic behavior by the web browser). Some of the existing mechanisms discussed later in this work can help defend against this weakness, but this work does not directly address it.

4.2. Web Browser and Infrastructure

The previous examples focused on weaknesses from the perspective of the web application and its author. There are also weaknesses that arise from engineering decisions in the web browser. One type of weakness that is common in web browsers is the trust extended by the web browser to plug-ins. A plug-in is a module provided separately from the web browser to handle data types not normally handled by the web browser (or to replace a data handler the web browser already provides). The weakness that arises from this trust relationship is that plug-ins can exhibit behavior beyond the confines of the web browser and the web site providing the content it is handling [20]. Examples of

plug-ins are Adobe's Flash plug-in, Adobe's PDF plug-in, and Oracle's Java plug-in. There are other popular plug-ins, but these are notable for an extensive series of security vulnerabilities that were found to affect them in the last several years. Variations of plug-in weaknesses can exist in core web browser functionality or in the operating system or application library support for core web browser functionality. An example of this related weaknesses is the WMF vulnerability identified in 2005, wherein a flaw in the Windows operating system was attacked by feeding a special image to the web browser, which would delegate to the operating system to display the data [13].

The web browser plays a non-trivial contributory role in the CSRF weakness presented earlier as a web application weakness. The presence of an HTTP request origin indicator (called after its HTTP header name, 'Referer') is optional, and so web application authors are often unable to rely on it for security enforcement (some web browsers and web infrastructure will remove this header, perhaps due to end-user privacy reduction its presence can introduce). A new 'Origin' header was proposed in 2008 [7] to provide a more reliable, less objectionable HTTP request header for web applications to use for mitigating the CSRF weakness. As of 2012, the header is defined as part of a larger W3C draft but is not formally standardized [18], though popular web browsers do include support for the header as part of certain activities [17].

The DNS infrastructure and libraries that web browsers rely on provide an important weakness that affects the trust placed in the same-origin policy by web browsers and web application authors. The weakness, commonly referred to as 'DNS rebinding', occurs when a web browser attempts to use DNS to resolve the same name multiple times and receives different answers for successive requests [30]. The impact of this scenario is that different web servers (controlled by different entities) would be lumped into the same origin by the web browser. The impact of a successful attack of this weakness is much the same as with XSS: the victim end-user's web browser will run attacker-controlled JavaScript

with the origin of a susceptible web application, which allows that JavaScript to freely interact with that web application. One particularly important distinction between this and a 'normal' XSS weakness is that the attacker does not need to be able to access the vulnerable web application directly; further, successful usage of this weakness requires the attacker to operate a DNS server that will answer requests as described.

4.3. End-User

Moving out of the technology arena, there are also weaknesses in how the end-user interacts with the web browser. Weaknesses in this category tend to be called 'phishing' as a broad term [26]. Modern web browsers provide a large amount of their display area to the web pages being displayed, which affords them a small number of opportunities to communicate the state of the web browser, the web site, and any important security information. Visual cues the web browser presents to the end-user include the address bar (URL bar), the security indicator (sometimes called the 'pad-lock' icon), and the status bar. The address bar is intended to inform the end-user what web site is being displayed at any given moment. Since modern web sites include content from many different addresses, the address bar is limited to only showing the outer-most web site address. Further, web browsers normally show the full URL in the address bar, since this information is normally important to help the end-user make well-informed choices about what to share with and expect from the web site. Unfortunately, the trustworthiness of different parts of the URL being shown is not apparent to many end-users: after the initial '/', the web site author has relatively unrestricted control over the text displayed, so 'evil.example.com/www.bank.com' points to a specific page on the web site 'evil.example.com', but an end-user could notice 'www.bank.com' and trust that to mean they are interacting with their bank. Due to the ease of copying HTML and replicating web sites' visual properties, this alternate web site could look and act like 'www.bank.com' enough to cause the end-user to try to log in with their bank

credentials. Recently as of 2012, web browsers have begun emphasizing the displayed web site's host name and de-emphasizing the path portion of the address.

Another web browser feature that can be used to confuse the end-user is the ability for JavaScript to open pop-up windows with no address bar. With this feature and basic planning, an attacker can create a web page to look like the web browser that is displaying it. By mimicking the web browser's interface, the attacker can present a fake address-bar displaying any security indicator and web site name they want the end-user to see.

Attempting to mimic the web browser's user interface within a web page is not necessary to confuse the end-user about the site being displayed. A simpler form of this weakness is often called 'typo-squatting', and relies on looking similar in the address bar to another web site. As an example, an attacker can prepare a copy of 'www.bank.com' and host it at 'www.bamk.com'. An end-user who does not pay close attention may fail to notice the 'm' where the 'n' should be. A more devious version of the same weakness takes advantage of advanced international display capabilities of modern web browsers: internationalized domain name (IDN) support allows non-Latin letters to be used in web site host names. Before this weakness was widely known and blocked by web domain name registrars, an attacker might register 'www.dell.com', where the 'e' is actually the Unicode character 'U+0435' (named 'CYRILLIC SMALL LETTER IE'). This weakness was identified as an IDN homograph attack when it was discovered [31], and current web browsers would show that URL as 'www.xn--dll-rdd.com'. In some cases, none of this care is needed: if an end-user chooses to not pay attention to the address bar and instead base trust on the contents of the web site, then the attacker would only need to copy the original web site and coerce the end-user into visiting the copy (for example, by sending an email with a disguised link).

Another weakness occurs when an end-user is fooled into clicking on a button that carries out

an unexpected action. Commonly called 'click-jacking', the weakness arises from the ability for web sites to frame other web sites while also controlling how the enclosing frame is or will be displayed [14]. Click-jacking behaves similarly to cross-site request forgery, but it targets the end-user instead of the web browser. Like CSRF, it requires the end-user to be logged in to the web application being abused. Abusing this weakness also requires a related weakness within the web application: it must be possible for an untrusted (different-origin) web site to control the contents of an end-user-submitted form (using cross-origin customizable aspects such as URL parameters). An example of click-jacking would be demonstrated if a non-Google web site could pre-filll (but not submit) an email in Google Mail on behalf of an end-user; the attacking web page could open and hide a frame containing the pre-filled email, then position the frame under the end-user's mouse immediately before a click occurs. The end result is that the attacker submits a form on behalf of the end-user with information controlled by the attacker.

5. Security Policies

Web browsers are faced with the combined tasks of enabling web site authors to present elaborate web sites with interactive content and reusable components and services while also providing a framework for web site authors and end-users to establish and confirm trustworthy relationships and engage in communications and commerce. Web browsers apply different security policies for executing interactive content than they do to including or displaying content [15]. For loading and displaying resources such as images, JavaScript, framed content, and CSS, web browsers implement more lenient security policies. For the execution phase of resources such as in-line and separate JavaScript, web browsers exercise careful and context-sensitive control over what the executable content is permitted to do and interact with. Legacy and modern web browsers alike implement a baseline security policy called the same-origin policy (SOP) [27, 30]. Based on the same-origin policy, web browsers have

structured later security contexts around the 'origin'.

When discussing origins, it is important to realize what an origin is. A web browser always deals with a resource by its URL (universal resource locater). For reference, an example of a fully-qualified URL might be 'http://www.google.com:80/ig/'. When informally referring to a web site or a web page, it is often convenient to omit the scheme ('http') and the port ('80'), but they are still part of the full URL. The 'origin' in 'same-origin policy' comes from several specific parts of this URL: the scheme ('http'), the full host name ('www.google.com'), and the port number ('80') combine to form the origin. In the context of a web page, the origin comes from the address of the HTML page loaded in the web browser (shown in the address bar). In the context of framed content (a FRAME or IFRAME pointing to any kind of resource, HTML or otherwise), the origin is based on the URL assigned to that frame[3]. Origin-based policy applies in the context of JavaScript, and it controls which resources and sub-frames that JavaScript can interact with (and in what ways).

Understanding why origin-based policies are a useful mechanism is easier when one understands what capabilities exist for JavaScript to use. JavaScript itself is a relatively simple language, providing no direct support for TCP/IP communications, HTML, or web content interaction. Support for most of JavaScript's value comes from the 'DOM', or document object model, provided by the web browser running that JavaScript. The DOM includes a number of standardized global variables and types that are exposed to any JavaScript that runs in a web browser, such as 'window' and 'document'. For example, JavaScript could read a form field using 'var data = document.getElementById("field1").value' and then send it using the 'XMLHttpRequest' type or by using 'document.createElement("IMG")' and assigning a URL including 'data' into it. When two windows or frames are open to two different origins, the web browser will prevent JavaScript running

3 JavaScript running in an origin is permitted to explicitly relax its own origin by setting the global variable 'document.domain' to a more general origin (e.g., from 'users.example.com' to 'example.com').

inside one of the web pages from accessing the DOM for the other web page. This, for example, prevents 'evil.example.com' from including a frame of 'gmail.google.com' and reading the end-user's email if they were logged in. In addition to preventing cross-origin frame access, web browsers also prevent cross-origin uses of XMLHttpRequest to dynamically fetch another web site's content. Unlike XMLHttpRequest, JavaScript is permitted to create new elements (such as IMG or SCRIPT tags) that point to remote origins; this capability, however, does not give the JavaScript the ability to receive the data returned by the remote origin.

The same-origin policy forms the basis for the web browser's behavior with cross-domain JavaScript, but it coexists with an important and opposite security policy: remote script inclusion. When JavaScript code occurs within an HTML document (within a SCRIPT tag), it is called an in-line script. Another supported way to include JavaScript in an HTML file is by reference to another URL. The URL of the referenced script does not play a part in determining the origin for it: because the original HTML document referenced it, the remote JavaScript is treated as though it were entered in-line in the document.

Over time, web applications and web components became more complicated, and interactions between web applications and externally-hosted web components became more dynamic. A common name for these dynamic interactions is 'mash-ups'. In a mash-up, an integrating web site depends on one or more local and remote application components [5]. An example of this interactivity involves Google Maps, where Google provides the mapping components (capable of drawing a map of a certain address with markers positioned over places of interest) and imagery, and the integrating web site plots interactive data (such as store location, crime hot-spots, or the locations of an end-user's friends and contacts) onto the map. In this situation, the integrating web site author is forced to choose between interactivity and isolation: to have the integrating web page or the component' react to changes in the

other, the web page must fully trust the remote component and include it via SCRIPT tag (permitting the remote component full access to the integrating page's origin); to isolate the remote component, the author must use an origin-splitting FRAME or IFRAME and forgo interactivity, though initial data could be shared to the component by passing in parameters through the URL.

In response to web site authors' evolving needs for cross-origin interaction without loss of isolation, web standards bodies have incorporated a cross-origin communication mechanism, 'postMessage' [8]. With the postMessage addition, integrating web sites authors and component authors must mutually opt in to cross-origin message passing. Part of the opt-in procedure requires a message sender to identify the targeted receiver's origin (to prevent mis-delivery of a possibly sensitive message to an unintended origin). Message receivers must opt in to receive messages, and all recipients are given the unforgeable origin of the sender. This standard provides integrating web site authors and web component authors with full flexibility in defining what policies they wish to adhere to; however, this flexibility comes at the cost of forcing both sides to implement all policy aspects they wish to enforce.

Related to the above exception to the same-origin policy is the cross-origin resource sharing protocol and feature ('CORS') [17]. Unlike 'postMessage', which operates across already-loaded web pages or frames, CORS permits a web application to use XMLHttpRequest to communicate with a different origin. Much like the postMessage scenario, the CORS feature requires both the initiator and the recipient (the remote server being contacted with XMLHttpRequest) to opt in to the communication. Unlike postMessage, CORS does not directly address the 'mash-up problem', because one communication end-point is still a remote server.

The need for cross-origin resource sharing predates the feature. Before the CORS protocol was introduced, the JSONP (JavaScript Object Notation with Padding) mechanism was described to provide a less-safe way to accomplish the same goal: retrieve JSON, XML, or other text from a server in a

different origin, and pass it to the running JavaScript. JSONP achieves what non-CORS XMLHttpRequest is forbidden from achieving; it does so by dynamically adding a SCRIPT tag to the current web page. With XMLHttpReqquest, the web site author's JavaScript would retrieve text from a remote server, carefully process it using JSON rules, and feed it to a JSON handler. With JSONP, the web site author caused the web browser to load and blindly execute a remote script within the current origin (with any parameters passed through the URL to the script). Part of JSONP's interaction involves the web site author informing the JSONP resource what function to call, and the blindly-executed script is trusted call that function with the proper data. Because it involves including a remote JavaScript resource with a SCRIPT tag, all the risks of loading a remote component are introduced: it has the same permissions (page origin access) as any script the web site author wrote.

Origin-based security policies represent a powerful and capable mechanism in modern web browsers, but one limitation bears repeating. Origin-based security cannot separate a single origin from itself. That is, executable JavaScript content embedded within or included within a single HTML page (or indeed, all HTML pages served from the same origin) all receives equal access. In other computing environments, self-isolation is a useful methodology to constrain the effects of software bugs or weaknesses [16]. For example, Linux and UNIX services can drop privileges by changing their effective user identifier after they are done using elevated privileges. This concept is often referred to as the principle of least privilege. This work will explore applying the principle of least privilege to HTML in order to augment origin-based security policies and mitigate some of the identified weaknesses.

6. Cross-Site Scripting

The work presented herein is primarily interested in mitigating the effects of cross-site scripting weaknesses. As discussed earlier, cross-site scripting occurs when the web browser executes scripts

which the web site author did not intend in the origin of that web site [1, 2, 3, 10]. In modern web sites with user participation, end-users may publish content which will be presented to other end-users in a controlled manner.

When a web application permits end-users to supply content to the web site to display to other end-users, the web application author is faced with providing the 'controlled manner' aspect of the user-to-user interaction. As mentioned previously, HTML pages can contain embedded scripts that execute within the same origin as the web page. Unless the web site author and all end-users trust each other a great deal, it would normally be undesirable for the web application author to permit end-users to provide JavaScript that will execute within the web site for other end-users. Web application authors are able to use web application content-escaping features to trivially prevent this risk: before displaying end-user supplied content, all symbols normally treated as special by web browsers (such as '<', '>', and '&') can be escaped or filtered out. The web application author can also serve user-supplied content as a framed web page served with a restricted display type (e.g., 'text/plain').

The process of filtering or transforming end-user supplied content becomes more challenging when the web application author wishes to permit richer HTML-like or HTML-subset capabilities to the end-users. One way to permit full HTML capabilities while preventing end-user content from operating within the web site's origin is to serve the end-user content from a separate origin (i.e., a separate web site) within a frame. One down-side to this approach is that it fails to prevent end-users from attacking other end-users' web browsers, such as by using vulnerable plug-ins.

Permitting end-users to supply rich content while preventing abuse of other end-users is currently a job that falls entirely on the web application author: end-user web browsers will expect rich content to be expressed as HTML, so the web application must take care to limit that HTML to only the subset desired. As the Samy MySpace worm demonstrated in 2005 [2, 10], filtering out some rich

content while permitting other rich content can lead to surprises. In the case of the Samy worm, some web browsers interpreted HTML that seemed to have no embedded JavaScript as though it had embedded JavaScript. More simply put, the web browser supported a different variation of HTML than the web application's filter was modeling. Even though the web application author did not intend for the end-user's content to support executing JavaScript, the web application author had no way to express this intent to the web browser. This work intends to provide a mechanism for the web application author to communicate that intent to the web browser (in this case, to not execute JavaScript within end-user-provided content).

7. Related Work

There are numerous projects and products pursuing the goal of protecting end-users from falling victim to the various weaknesses outline above. Disabling JavaScript is often approached as a far-reaching protection strategy, as JavaScript exposes many capabilities that improve the likelihood of a successful attack. Disabling JavaScript is a difficult approach to pursue, because many web applications are written with the assumption that JavaScript is available. Disabling JavaScript would result in an inability for the end-user to conduct on-line banking and many forms of on-line commerce. This approach also does not help end-users who do not disable JavaScript, thus the approach does not constitute a technique that helps a web application author protect end-users. Also, disabling JavaScript fails to protect against certain forms of CSRF, since simple images can trigger particularly egregious CSRF weaknesses (i.e., web applications that accept GET requests that carry out important activities).

7.1. CSRF Mitigation

There are a number of mitigation and preventative techniques for the cross-site request forgery weakness. One straight-forward approach that is supported in many web application frameworks is to generate a random number or token to include with each form being filled out by the end-user [7].

Upon form submission, the random number is compared with the one sent with the original form. Since the token only exists within the form's web page (and not in an automatically-submitted web browser cookie), JavaScript executing within a different origin in the end-user's web browser cannot obtain the token (i.e., it cannot be stolen). The Samy worm (described in section 9.1) was only able to obtain the CSRF token in its attack because the JavaScript was running within the proper origin.

Related to CSRF tokens, custom HTTP request headers are also proposed as a technique for preventing CSRF attacks. Similarly to the CSRF token approach, a random token is generated and included with each blank form. Unlike the CSRF token approach, the token would be submitted in the HTTP request with a custom request header and not as a standard form input field [7]. This approach relies on two useful constraints imposed by web browsers: first, custom headers may only be set on XMLHttpRequest queries, which are forced to adhere to same-origin restrictions; and second, origin-agnostic elements such as images, scripts, and styles have no means for supplying custom headers.

In addition to using custom tokens to verify the legitimacy of an HTTP request, another technique is to check certain aspects of the request more closely. One header that is a good candidates for scrutiny is 'Origin' [7, 12]. While the 'Referer' header is an older standard, it is also optional and is stripped out in some cases (by the browser and sometimes by proxy servers) [7].

A more thorough—though less convenient—approach to blocking CSRF attacks is to force the end-user to re-authenticate before certain actions [19]. This attacks the foundation of the CSRF weakness: mis-use of web browser state (such as the end-user's current active sessions). This approach does have the negative side-effect of annoying legitimate end-users by forcing them to log in again while using the web site.

7.2. Distrusting Embedded / Enclosed Content

Several proposals proscribe mechanisms for demarcating untrusted areas of a web page have been published. One proposal similar to restricted-block introduces an 'untrusted' marker on the DIV tag. Within such a marked tag, JavaScript would be dynamically rewritten to prevent access to the enclosing web page, XMLHttpRequest and frame/document access would be prevented to the enclosing origin, and CSS names would be modified to prevent name collisions with the enclosing document [9]. Another proposal suggested a new frame-like '<jail>' tag, which would include a remote document within it (like other framing tags) while prohibiting JavaScript within the framed content [1, 4]. Similar to the '<jail>' tag, Microsoft implemented a 'security=restricted' flag to its '<iframe>' tag [8]. A compromise between 'dumb frame' and 'detailed policy' was proposed which allowed un-framed content to be prevented from running JavaScript by specifying a 'noexecute' flag in a ' ' tag wrapping the content; this proposal also suggested a script-permissive approach using cryptographic hashes to white-list permitted JavaScript [3]. One issue with applying policy to untrusted nested content is the ability for such content to embed closing HTML tags to prematurely end a sand-boxed block. One novel approach was suggested to include random tokens in opening tags that had to be repeated in closing tags [1]. A more standards-compliant approach is to escape the content in JavaScript strings and embed them using 'innerHTML', which prevents the content from breaking out of the existing element [3].

A proposal from several Microsoft engineers suggested a set of mash-up-focused tags and associated security policy definition capabilities [5]. This proposal stuck to the framed-content model common to other proposals. It introduced tags such as '<Sandbox>', '<OpenSandbox>', '<ServiceInstance>', and '<Friv>'. One unusual aspect of this proposal—compared to others—was the allowance for cross-origin DOM access (from the parent to the child) in the '<OpenSandbox>' case (while still imposing separate-origin status, like with '<Sandbox>' and other proposals). Also novel in

this proposal are the background-execution '<ServiceInstance>' concept and DIV-like '<Friv>' viewport for service instances.

7.3. Declarative Policy Using Custom HTTP Headers

Several engineers from the Mozilla web browser maker put forward a proposal and design for Content Security Policy, a technique to mitigate XSS weaknesses using declarative security policies [6]. This proposal takes the form of a new HTTP response header that dictates document-wide policy for the web browser to apply to the returned document. This proposal is similar in scope and approach to this restricted-block mechanism, but it focuses on document-wide policy, and it builds on the remote framed content techniques. Similarly to restricted-block, content security policy allows fine-grained control over various HTML features, such white-listing permitted targets for request types and disabling or constraining the execution of JavaScript. Unlike the restricted-block approach, content security policy does not define a policy stacking mechanism, including integrating overlapping policies from different contributors and applying parent restrictions to descendant restricted blocks. One feature CSP proposes that RB does not is a mechanism for reporting policy failures to the policy-imposing web site, permitting web site authors to notice when their web sites are being targeted by attackers (or when policy is unexpectedly acting to restrict intended functionality). An important gap in CSP coverage is that when content is saved or served through a web proxy server, the policy headers may be lost; in this scenario, the fall-back behavior is that the served content may have complete permissions.

Some web-browser-based techniques have been implemented to address specific weaknesses by accepting and imposing simple document-wide policies that affect how the web browser treats served content in various scenarios. These techniques use HTTP response headers to communicate the policies and policy enforcement intent. The 'X-XSS-Protection' header provides a simple toggle to opt-in or opt-out of web browser pattern-based detection and prevention of reflected XSS attacks within the

served content (i.e., links that contain HTML-like constructs within the URL) [11]. Similar pattern-based reflected XSS prevention can be found in the NoScript Firefox add-on [noscript.net]. The 'X-Content-Type-Options' header was implemented to prevent web browser 'content sniffing' from executing malicious content when it was included in benign contexts (for example, serving a JavaScript file while declaring it to be an image type, relying on the web browser to recognize and treat it as HTML instead of an image) [11]. Since click-jacking relies on the ability for an attacker to embed the targeted web site in a frame, the 'X-Frame-Options' header was introduced to declaratively forbid embedding the response in a frame [11].

7.4. The Gap

This work seeks to permit several key concepts side-stepped or partially addressed by the related work. The first goal is to permit the restricted content to occur in the trusted web page, preventing a linear increase in the number of HTTP requests needed to serve multiple sand-boxed content pieces in a single page. The second goal is to permit integrating policies defined by multiple interested parties at the web browser; this is intended to reduce the maintenance burden on web site authors when out-sourced content (such as advertising network resources) wish to change policies without forcing the web site author to blindly accept such third-party policies. The third goal is to permit fine-grained as well as coarse-grained policies to being defined and imposed; the intent behind allowing coarse-grained privileges is to allow some amount of intent-based future-proofing, reducing the maintenance burden on web site authors as web standards evolve to introduce new capabilities. Of the work reviewed above, the more flexible ones were limited to framed remote contents, while the ones that supported direct embedding tended to be overly coarse (e.g., deny-all).

8. Solution: HTML Privilege Dropping

By pro-actively declaring any content interpretation intents and non-intents, the web

application author can ensure that a compliant web browser will provide only the sub-set of HTML and JavaScript features intended for a particular piece of content. Essentially, the web application author can “drop privileges” for end-user-generated content (for example), taking advantage of the principle of least privilege. Before getting to the details, it is important to remember that this solution does not prevent the web site end-user from conducting the restricted actions (or from modifying a compatible web browser to disable the restriction enforcement); instead, it prevents malicious end-users from coercing an honest end-user's web browser from automatic, undesirable behavior.

The earlier example that inspired this privilege-dropping mechanism was constraining rich content supplied by untrusted end-users. The problem extends beyond end-users, however: advertising networks enjoy a similar relationship with web site owners. Advertising networks accept rich content from advertisers and publish them to others' web sites. If a web site author directly embeds content from an advertising network, and if that advertising network makes the same filtering mistake MySpace did in 2005, the end-result would be permitting advertisements to carry out XSS-style attacks on otherwise-safe web sites [20. With web browser support for privilege dropping, a web site would not be at the mercy of the advertising network's filtering competence.

In order to embed policy directly in a web page, it needs to be expressed in a form that web browsers can recognize. For XML and HTML documents, which are the focus of this work, the declaration would take the form of a new tag, RESTRICTED. The purpose of choosing a tag is to provide a clear boundary for the privilege-dropping behavior, indicating both where it begins and where it ends. Further, because tags (unlike HTTP headers) can be nested, different privilege sand-boxes can be established within the same page where permissions need to be different (e.g., privileges given to an advertisement versus privileges given to end-user-generated content). Using tags instead of HTTP headers also removes the layout restrictions implied by frame-based policy

application: the web site author can inter-mingle restricted content with the web site's layout in whatever way is deemed appropriate (versus framed restricted content, which can only be integrated as rectangular regions within the parent web page).

In order to facilitate single definition of policy, a RESTRICTIONS tag will also be introduced for actual policy definition (either in-line or as a reference to a policy document). This follows the design pattern introduced by CSS, where STYLE blocks contribute reusable rules that can be referenced later in the document using the 'class' attribute on appropriate tags.

One challenge for combining the concept of privilege-dropping with a moving-target standard like HTML is how to handle standard evolution in a fail-safe manner. A related issue to the standard being a moving target is the varied implementations and interpretations of the standards exposed by web browsers. Further complicating the matter is the fact that web browsers frequently support non-standardized features or implement features from draft standards before they are finalized.

Two mitigation strategies are described to address the combined problems of standard evolution and web browser variety. First, the restriction mechanism will use a level of indirection when addressing specific web browser capabilities: instead of a restriction targeting a specific feature of a specific standard or web browser, it will target an intent or capability. The ramification of this is that a supporting web browser must group all controllable features under one or more capability categories. Second, because web browser capabilities are constantly evolving, a restricted block would impose a default policy of 'deny-soft' to all capabilities not mentioned within its policy list.

When a restriction is specified for a capability, it can take one of three declared states: 'allow', 'deny', and 'deny-soft'. The 'allow' declaration indicates the web browser should not further restrict access to the identified capability for the marked web content. In particular, the web browser is not required to grant access to the capability to the content: other security policies apply in tandem to this

restriction feature, and this mechanism shall not be used to override them. That is, this mechanism must not grant privileges that the content would not have already had. The 'deny' declaration indicates that the web browser must prevent the marked content from using the identified capability. The 'deny-soft' declaration is a special case which permits a later 'allow' rule to prevent restriction of the identified capability. The 'deny-soft' declaration is important to cover two different scenarios of applying restrictions. The first scenario is when multiple overlapping rules are applied as part of the same policy set. The second scenario involves marked content wrapped by more than one level of restriction block.

The restriction mechanism permits multiple overlapping or non-overlapping policies to be incorporated into the policy set for the same restricted block. This constitutes an important expectation of how policy definition is authored: a web author might both specify policy directly based on immediate knowledge, and a web author might delegate policy decisions to a second- or third-party entity. It is anticipated that delegation will play an important role in policy definition deployments: delegation permits a web author to incorporate a second party's policy into the web site. This creates an opportunity for the earlier-mentioned advertising network to specify the restrictions they wish to enforce on advertisements they accept in a web-browser-enforced manner. Delegating policy definition in this way permits the advertising network to change its policy over time, based on web browser or standard changes. The composition does not require the web site author to fully trust the advertising network: if any party (including the web author) incorporates a 'deny' policy, the denied capability cannot be reinstated by any other integrated policy. Similarly, nested restricted blocks are treated as an amalgamation of all ancestor policy sets (as well as the local policy set), which implies that the most restrictive policy is what applies in cases of conflicts. The interactions of white-lists in nested and coincident policies will be discussed later, when white-lists are presented.

In addition to providing policy answers, the restriction language must allow policy authors to

identify broad and specific features or capabilities to control. The broad categories to control include remote resource loading (including plug-ins, CSS style sheets, web-fonts, images, scripts, and XMLHttpRequest queries), remote resource execution and/or evaluation (including plug-ins, CSS style sheets, web-fonts, and scripts but not images, XMLHttpRequest queries, or CSS-identified images), plug-in execution, script execution (including remote, in-line, CSS-based script expressions, and event handler callbacks), and local window control (including reading and writing the current location, redirecting the page, mimicking HTTP headers with '<meta>', reading and writing cookies, accessing history entries, and triggering print functionality). Specific capabilities that would be controlled include script inclusion, script in-line specification (covering both event handlers and in-line scripts), images, stylesheet loading, XMLHttpRequest queries, cross-origin XMLHttpRequest queries, web-GL texture loading, and web-font loading.

Along with capability identification and policy decisions, there is value in supporting finer-grained focusing of policy with white-lists. Using white-lists, a policy author can further constrain capabilities, such as image loading or remote script loading, to specific destinations or MIME types. For CSS and for remote resource loading controls, further constraints can be imposed. If CSS style evaluation is permitted, a white-list can be supplied listing permitted properties as well as permitted value types (e.g., to permit value types other than 'url' and 'expression'); this would allow complex styling while forbidding position control, for example. If remote resource loading is permitted, multiple white-lists will control aspects of the request and response, including a white-list of URL patterns to permit, a white-list of acceptable response MIME type patterns, an indication of whether URL query parameters are permitted, and an indication of whether resource-related cookies and/or authentication credentials may be sent. The query-parameter, cookie, and authentication control mechanisms help prevent rich end-user content from being used to launch CSRF attacks.

One risk this policy mechanism would face is container break-out by embedded content [3]. This risk is realized when the web application fails to sufficiently validate or filter end-user input. If an end-user placed the text '</restricted>' somewhere in the content such that the web browser evaluated it, the end result would be that following end-user content would execute without the restriction policies applying. To prevent this scenario from occurring without mandating that policy-controlled content be loaded with a remote reference, this restriction mechanism demands that policy-enforced content be represented using base-64 encoding (a technique inspired by 'data:' URL encoding examples). An unfortunate side-effect of this requirement is that static documents (such as a policy-enforced web page saved to a file) become difficult to manage with normal tools. Further, encoded content might prevent other web security technologies, such as web application firewalls, from detecting or applying policies to such content, unless they were modified to account for the new content encoding.

As part of isolating the untrusted content from the enclosing web page, the restriction mechanism would place the inner content in its own origin by default. Even in cases where no active content is permitted in the restricted block, this can help prevent CSS styles in the parent document from inadvertently applying to the end-user-generated content (and related, to prevent JavaScript in the parent document from unintentionally selecting end-user-generated content due to CSS class name reuse or duplicate element IDs). If the web application author wants to place each restricted block in its own unique origin, nothing further is needed: this is the default behavior. The unique-origin state prevents JavaScript in the restricted-block from accessing cookies of the parent document. If the author wishes a restricted block to be part of the parent page's origin (e.g., to allow easy DOM manipulation from the parent document), the restricted block can be marked to enact this behavior. If the web application author wants to keep the restricted block isolated but communicate with live JavaScript

within it, the standard 'postMessage' mechanism will work (the web application author will need to read an 'origin' property from the restricted-block element to find out what its effective origin is—this origin would be dynamically generated as the content is loaded and will not change as long as the restricted-block element exists).

As the restrictions mechanism requires specialized support in the end-user's web browser, the web browser will need to inform the server of the supported version of the mechanism. By including a version in the support indicator, future updates to the feature's capabilities and/or syntax are simplified. This design choice also permits web applications to detect non-support and return safe content. Since proper support of this mechanism would require web browser involvement, content would need to be heavily filtered or escaped when being returned in this mode. The end result offers a less-enhanced experience for non-supported web browsers while still preserving the safety benefits of full support. A motivated researcher could also conceivably design a web application component to transform restrictions-based HTML into restrictions-free HTML, possibly utilizing some combination of the related work.

In addition to permitting web application authors to integrate multiple policies into policy sets, it would be useful to permit web browsers and end-users to augment policies on a global or web-site specific basis. This variation would behave similarly to CSS 'user style sheets'. Just as policy sets enforce the most-restrictive union of all integrated policies, end-user policies would only serve to further limit functionality available to restricted content (e.g., to block all restricted content from loading plug-ins even if the web site's author chose to permit them). The purpose of this suggestion is not to specify or standardize that behavior, but to urge web browser implementations to consider it.

9. Attack Resilience Analysis

In this section, the policy mechanism is brought to bear on real and imagined attacks. For each

attack, analysis will reveal its nature, enabling factors, the results, and how restrictions would be brought to bear to mitigate or prevent the attack from succeeding.

9.1. Samy Worm (XSS)

This attack is described in [2] and [10]. The 'Samy' MySpace worm took advantage of generous formatting constructs permitted by MySpace in user profiles. MySpace utilized a combination of pattern matching, white-listing, and black-listing to prevent end-users from embedding JavaScript in their user profiles (which would constitute a stored XSS vulnerability). A combination of factors enabled Samy to embed JavaScript in his user profile which was not blocked by MySpace and was executed by many end-users' web browsers: first, MySpace's white-list permitted CSS styles to be specified; second, using a 'javascript:' expression as the URL to a background image caused many popular web browsers to execute the JavaScript; third, an attempt by MySpace to filter out attempts to run JavaScript failed to catch a variation accepted by some web browsers ('java script', where a new-line character separates 'java' and 'script'). Once JavaScript was being executed in end-users' web browsers, the XMLHttpRequest feature and a small amount of logic made it possible to compose and submit a profile-update form on behalf of the end-user.

The XMLHttpRequest feature played a key part in successfully launching the Samy attack. Interestingly, the same-origin policy almost defeated part of the attack, even though MySpace was serving the XSS attack code from its own web site (accordingly, it would be inappropriate to call the attack 'CSRF', as it was not cross-site): the add-friend form interaction required a CSRF token that was served from a different origin from the attack's entry-point origin ('www.myspace.com' instead of 'profile.myspace.com'). The attack was successful, however, because the same XSS content was also served from the desired origin.

The restricted-block mechanism could be successfully used to prevent this attack. The most

basic security policy that would defang this attack would be to deny JavaScript execution (a more appropriate technique would be to white-list the desired functionality, enabling CSS styling and remote resource loading). Given a declared policy to avoid executing JavaScript, the JavaScript expression embedded in the CSS URL would fail to execute. As indicated in the attacker's description of the events, the ability to inject JavaScript to run in end-users' web browsers was essential to carrying out this attack.

9.2. WMF Image Flaw

The WMF flaw is a vulnerability in the application libraries supporting a less-common image format [13]. The flaw impacts web browsers, because web browsers use the flawed application library. The exposure for flaws in image rendering is significant: images will render automatically in web browsers. The image rendering flaw can be reasonably compared to vulnerabilities in popular plug-in formats, except that plug-ins can normally be disabled. This image rendering flaw is capable of being exploited with JavaScript and plug-ins both disabled in the web browser. No specific instance was identified of this flaw being used on web sites with end-user-generated content; however, the possibility is clear, and the attack vector would simply be the ability to embed an image whose URL is provided by the end-user.

The realization of a WMF flaw attack would be to include an IMG tag with a URL pointing to a WMF file. The WMF file could be hosted anywhere. The restricted-block mechanism could block this attack in principle: if remote images are desired, the white-list capability could be utilized to list approved image MIME types. White-listing the most popular image formats (JPEG, GIF, PNG, and BMP) would mitigate the attack vector. Since a web site author is unlikely to anticipate the need to restrict image formats, this mitigation could be considered reactionary in nature. A more realistic variation of restriction-based mitigation would be that the web site author white-lists a certain image

hosting service for remote images embedded in end-user content. Such a hosting service could be independently configured to permit only certain image formats to be uploaded (or such a hosting service could automatically re-encode all uploaded images to a common, safe format).

9.3. General Non-Persistent XSS

The Samy MySpace worm is an example of a persistent XSS vulnerability because it involves the web application serving XSS content to any end-users that visit the web site area containing the XSS. A non-persistent XSS vulnerability, on the other hand, occurs when the XSS content comes from the HTTP request parameters alone [2]. A necessary characteristic of vulnerable web pages is that the web page must include one or more of the HTTP request parameters (which can include URL parameters) into the web page's HTML content without escaping it. To attack an end-user with a non-persistent XSS vulnerability, an attacker must cause the end-user to access the vulnerable web page through a specific link. The impact of a non-persistent XSS attack is the same as with a persistent XSS attack: the end-user's web browser executes JavaScript provided by the attacker.

The restricted-block mechanism would suffice to mitigate non-persistent XSS attacks. In places where HTTP request parameters would be inserted into the web page, the web page author inserts a restricted tag to white-list no functionality (or whatever sub-set of functionality is required, but not executing JavaScript). Within that restricted tag, the web page would include the HTTP request parameter contents in question. As with the WMF vulnerability, this approach to mitigating the attack requires the web application author to already be aware of the need to distrust the HTTP request headers, which is often enough to inspire the use of existing escaping mechanisms. Unlike escaping mechanisms, though, this technique would permit safely including rich content.

9.4. CVE-2008-5249 - Cross-site Scripting (XSS) Vulnerability in MediaWiki

Three different issues are identified as part of this flaw [32, 33]. Each will be visited in turn. The public disclosures did not document the flaws precisely, but the source code patch applying the fixes was small enough to analyze for explanation here (see http://download.wikimedia.org/mediawiki/1.13/mediawiki-1.13.3.patch.gz).

One flaw occurred due to the ability for end-users to upload files and set their MIME types (aside from an already-defined list of unsafe MIME types). Based on comments in the source code, various versions of Internet Explorer could ignore the served MIME type and reinterpret the served file differently (possibly as HTML, in which case it could execute JavaScript in the context of the web site, constituting an XSS attack). The restricted-block proposal would not be able to mitigate this attack, because it applies to document fragments, not whole documents. Content Security Policy headers, however, could mitigate this.

Another XSS vulnerability was that certain error messages could be emitted without proper HTML escaping, resulting in what appears to be a reflected XSS weakness (based on the vulnerability announcement and on additional HTML escaping added to error-handling code). The restricted-block proposal would successfully mitigate this attack as in section 9.3 above.

Another vulnerability was that the 'Special:Import' web page failed to use CSRF token validation, with the impact being that an attacker could trick a logged-in end user into importing an attacker-controlled document into the web site. The restricted-block proposal does not purport to mitigate received CSRF attacks.

9.5. CVE-2008-0971 - Barracuda Networks Products Multiple Cross-Site Scripting Vulnerabilities

In this vulnerability, a management console web page is susceptible to reflected XSS [34, 35],

where a HTTP request parameter ('auth_type') is embedded in the resulting web page without filtering or escaping. The restricted-block proposal would mitigate this attack if the affected form were wrapped with a security policy that blocked execution of scripts.

9.6. CVE-2006-4308 - Multiple Cross-site Scripting (XSS) Vulnerabilities in Blackboard Learning System

The Blackboard system was found to permit JavaScript event handlers to be provided for certain types of tags in the discussion forum feature [36, 37]. Further, certain HTML tags that could cause the web browser to load and display external content were permitted. The restricted-block proposal would mitigate these weaknesses: a restrictive policy that permitted only styling and remote images with approved MIME types and to approved destinations would prevent the web browser from executing the malicious content.

10. Review of Complementary Solutions

In the web security space, solutions do not have to operate in isolation. It is reasonable for a web application author to utilize and deploy multiple approaches to achieve the end-goal of protecting end-users and their own web sites from attack. In this section, related work is discussed in the context of how it inter-operates with the current restricted-block proposal.

The Content Security Policy feature shares a good amount of similarity with the intent this proposal. One benefit of CSP compared to RB is that CSP only requires changes within the HTTP response headers. For this reason, it can deployed for a web site without necessarily modifying the underlying web application: a proxy could add the headers as appropriate. It could be useful to define a mapping from RB policy to CSP policy to maximize coverage. A benefit of CSP compared to RB is that CSP can cope in situations where end-users may upload whole documents (where a document-wide security policy makes clear sense).

The existing 'X-Frame-Options' HTTP response header provides a useful complementary capability to this proposal. In its current state, this restricted-block proposal does not address click-jacking attacks by other parties (though it can be used to prevent embedded content from being used to launch click-jacking attacks). While no deliberate interactions are suggested for this feature, there is no apparent negative interaction between it and restricted-block. It could conceivably be useful —as future work—to expand the policy to impose requirements on observed 'X-Frame-Options' responses to requests triggered by the nested restricted content.

Existing CSRF mitigation proposals and solutions all still make sense in the context of this feature. CSRF tokens, CSRF token HTTP request headers, and origin HTTP request headers are all still important aspects of self-protection for a web application, since this proposal does not address being targeted by CSRF attacks. As with 'X-Frame-Options', this feature provides a complementary technique for CSRF mitigation, by preventing suitably-restricted untrusted embedded content from carrying out CSRF attacks (and indeed, same-site request forgery).

Serving untrusted user content from separate origins is also a technique that combines well with this one. As a defense in depth technique, it supports a web site in providing security for legacy (lacking restricted-block functionality) web browsers. Proper support for legacy browsers, as suggested earlier, would still require the web application to serve different content (lacking 'restricted' tags) to legacy browsers, but both restricted and legacy content could be served from a different origin to separate sensitive web site functions from end-user-generated content.

11.Conclusions

The popularity of end-user-generated content has continued to grow without a corresponding improved mechanism to constrain the same content. Mash-up style portals (such as iGoogle) exist in the same context and with the same limitations. It is in the interest of the wider web application

developer ecosystem for a convenient mechanism to take untrusted end-user content and semi-trusted third-party content and safely embed it into a web site. If such a mechanism existed and were widely supported by web browsers, the web industry could perhaps manage to get the recent XSS vulnerabilities list (see 'http://web.nvd.nist.gov/view/vuln/search-results?query=xss&search_type=last3months&cves=on') to go empty. Based on the prevalence of XSS vulnerability reports, it is apparent that software engineering practices such as careful handling of input and output are either too much to ask from web application programmers or are too difficult to apply in the face of modern web application implementations.

The current proposal of restricted blocks addresses some gaps by current proposals and currently-deployed web browser functionality. A number of security vulnerabilities have been shown to be mitigated by or to be out of scope of restricted blocks. This proposal and other existing mechanisms should be able to mitigate current weaknesses if applied correctly and in concert (except for end-user actions resulting from phishing or visual confusion, which are best served by education campaigns), though some important 'existing' mechanisms are still not widely implemented enough [21] or perhaps are not well-understood enough by enough web developers.

12. Unanswered Questions and Future Work

One important weakness this proposal only coarsely controls is plug-in content actions. No suggestion actively prohibits applying declarative permissions to plug-in content, but the anticipated coarse and fine-grained controls do not necessarily account for functionality available to plug-in content. Future work that would be helpful to close this loop-hole would include defining policy actions that make sense for plug-in content as well as proposing mechanisms for enforcing such policies around plug-ins that are not written by web-browser vendors.

A second non-trivial omission is how to deal with content served over non-HTTP mechanisms,

such as FTP or from local files. Off-line resources are problematic in the face of origin-based policies (including same-origin policy) because, taken literally, either all files have the same origin ('file', no server, and no port) or each file has a unique origin (the full path to the file). For now, this area is deferred for the web browser to implement in the same spirit as its current file-based origin policy. If file-based resource usage becomes more popular, it could also be a useful area of expansion to extrapolate the 'remote resource' controls to include local file references. A variation of this issue arises when marked-up HTML occurs in HTML emails.

One particularly useful feature found in Content Security Policy is web site owner alerting. Support for this capability would be useful, both for alerting purposes (for a site owner to discover when an attack is occurring against the web site) and for debugging purposes (to discover when a policy is incorrectly specified or is unintentionally over-restrictive).

Another area for potential expansion would be to incorporate more non-security content filtering needs identified in web applications. An example of content that could be restricted by white-list or prohibited would be links. If links were prohibited in end-user-generated content (or if they were restricted to subject-relevant web sites or to SPAM-checking relay services), it could have the desirable effect of disincentivizing SPAM forum posting (for example).

More obvious avenues for future work would be to implement this proposal in a popular web browser and web application framework. Implementing the web browser side would provide an opportunity to evaluate the performance impact and actual efficacy of this technique, and to suggest how architecturally-compatible web browsers are to this proposal. Implementing the web application side would provide insight into how well the proposed permission controls map to the real structure of existing web applications and web sites. A reference implementation would also prove to be a helpful platform to propose further expansions or changes to the proposal.

13. Bibliography

[1] M. Ter Louw, P. Bisht, and V. Venkatakrishnan, “Analysis of hypertext isolation techniques for XSS prevention,” *Web 2.0 Security and Privacy 2008*, 2008.

[2] J. Grossman, “Cross-site scripting worms and viruses,” *Whitehat Security*, vol. 2007, 2007.

[3] T. Jim, N. Swamy, and M. Hicks, “Defeating script injection attacks with browser-enforced embedded policies,” in *Proceedings of the 16th international conference on World Wide Web*, 2007, pp. 601–610.

[4] B. Eich, “JavaScript: Mobility & ubiquity (two out of three ain’t bad),” in *Dagstuhl Seminar*, vol. 7091.

[5] H. J. Wang, X. Fan, J. Howell, and C. Jackson, “Protection and communication abstractions for web browsers in MashupOS,” in *ACM SIGOPS Operating Systems Review*, 2007, vol. 41, pp. 1–16.

[6] 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*, 2010, pp. 921–930.

[7] A. Barth, C. Jackson, and J. C. Mitchell, “Robust defenses for cross-site request forgery,” in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, pp. 75–88.

[8] A. Barth, C. Jackson, and J. C. Mitchell, “Securing frame communication in browsers,” *Communications of the ACM*, vol. 52, no. 6, pp. 83–91, 2009.

[9] A. Felt, P. Hooimeijer, D. Evans, and W. Weimer, “Talking to strangers without taking their candy: isolating proxied content,” in *Proceedings of the 1st Workshop on Social Network Systems*, 2008, pp. 25–30.

[10] S. Kamkar, *Technical explanation of the myspace worm*. Technical report, http://namb.la, 2005 [Online]. Available: http://namb.la/popular/tech.html

[11] A. K. Sood and R. J. Enbody, “The conundrum of declarative security HTTP response headers:

lessons learned," in *Proceedings of the 2010 international conference on Collaborative methods for security and privacy*, 2010, pp. 8–8.

[12] A. Barth. "The web origin concept," 2011.

[13] A. Gunn. "WMF FAQ: What you need to know," in *ComputerWorld*, 2006 [Online]. Available: http://www.computerworld.com/s/article/107482/WMF_FAQ_What_you_need_to_know

[14] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson, "Busting frame busting: a study of clickjacking vulnerabilities at popular sites," *IEEE Oakland Web*, vol. 2, 2010. Available: http://w2spconf.com/2010/papers/p27.pdf

[15] T. Oda, G. Wurster, P. C. van Oorschot, and A. Somayaji, "SOMA: Mutual approval for included content in web pages," in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, pp. 89–98. Available: https://ccsl.carleton.ca/software/soma/soma-acm-ccs-2008.pdf

[16] D. Wheeler, "Secure programmer: Minimizing privileges," 20-May-2004. [Online]. Available: http://www.ibm.com/developerworks/linux/library/l-sppriv/index.html. [Accessed: 03-Dec-2012].

[17] M. Corporation, "HTTP access control (CORS)," Mozilla Developer Network, 08-Oct-2012. [Online]. Available: https://developer.mozilla.org/en-US/docs/HTTP_access_control. [Accessed: 03-Dec-2012].

[18] B. Sterne and A. Barth, "Content Security Policy 1.0," 15-Nov-2012. [Online]. Available: http://www.w3.org/TR/2012/CR-CSP-20121115/. [Accessed: 03-Dec-2012].

[19] T. OWASP, "Top 10 2007-Cross Site Request Forgery - OWASP." [Online]. Available: https://www.owasp.org/index.php?title=Top_10_2007-Cross_Site_Request_Forgery&oldid=81714. [Accessed: 03-Dec-2012].

[20] S. Ford, M. Cova, C. Kruegel, and G. Vigna, "Analyzing and detecting malicious flash advertisements," in *Computer Security Applications Conference*, 2009. ACSAC'09. Annual, 2009,

pp. 363–372.

[21] T. W3C, “Content Security Policy - Web Security,” 04-May-2012. [Online]. Available: http://www.w3.org/Security/wiki/index.php?title=Content_Security_Policy&oldid=517. [Accessed: 03-Dec-2012].

[22] B. Bos, Ç. Tantek, I. Hickson, and H. Wium Lie, “Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification,” 07-Jun-2011. [Online]. Available: http://www.w3.org/TR/2011/REC-CSS2-20110607/. [Accessed: 03-Dec-2012].

[23] D. Raggett, A. Le Hors, and I. Jacobs, “HTML 4.01 Specification,” 24-Dec-1999. [Online]. Available: http://www.w3.org/TR/1999/REC-html401-19991224/. [Accessed: 03-Dec-2012].

[24] J. Stenback, P. Le Hégaret, and A. Le Hors, “Document Object Model (DOM) Level 2 HTML Specification,” 09-Jan-2003. [Online]. Available: http://www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109/. [Accessed: 03-Dec-2012].

[25] A. Grosskurth and M. W. Godfrey, “A reference architecture for web browsers,” in *Software Maintenance, 2005. ICSM’05. Proceedings of the 21st IEEE International Conference on*, 2005, pp. 661–664.

[26] R. Dhamija, J. D. Tygar, and M. Hearst, “Why phishing works,” in *Proceedings of the SIGCHI conference on Human Factors in computing systems*, 2006, pp. 581–590.

[27] M. Corporation, “Same origin policy for JavaScript,” Mozilla Developer Network, 03-Nov-2011. [Online]. Available: https://developer.mozilla.org/en-US/docs/Same_origin_policy_for_JavaScript. [Accessed: 03-Dec-2012].

[28] S. Christey, “Unforgivable vulnerabilities,” *The MITRE Corpotarion*, 2007.

[29] MITRE, “CAPEC - CAPEC-71: Using Unicode Encoding to Bypass Validation Logic (Release 1.7.1),” 18-May-2012. [Online]. Available: http://capec.mitre.org/data/definitions/71.html. [Accessed: 03-Dec-2012].

[30] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh, “Protecting browsers from DNS rebinding attacks,” *ACM Transactions on the Web (TWEB)*, vol. 3, no. 1, p. 2, 2009.

[31] T. Holgers, D. E. Watson, and S. D. Gribble, “Cutting through the confusion: A measurement study of homograph attacks,” in *Proceedings of the 24rd Annual USENIX Technical Conference*, 2006, pp. 261–266.

[32] MITRE, “CVE - CVE-2008-5249,” 2008. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-5249. [Accessed: 03-Dec-2012].

[33] T. Starling, “[MediaWiki-announce] MediaWiki 1.13.3, 1.12.2, 1.6.11 security update,” 15-Dec-2008. [Online]. Available: http://lists.wikimedia.org/pipermail/mediawiki-announce/2008-December/000080.html. [Accessed: 03-Dec-2012].

[34] MITRE, “CVE - CVE-2008-0971,” 2008. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0971. [Accessed: 03-Dec-2012].

[35] DCSL, “Barracuda Networks products - Multiple Cross-Site Scripting Vulnerabilities,” 2008. [Online]. Available: http://dcsl.ul.ie/advisories/03.htm. [Accessed: 03-Dec-2012].

[36] MITRE, “CVE - CVE-2006-4308,” 2006. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4308. [Accessed: 03-Dec-2012].

[37] Pro7on, “BlackBoard Multiple Vulnerabilities (XSS),” *Bugtraq*. [Online]. Available: http://www.securityfocus.com/archive/1/archive/1/444062/100/0/threaded. [Accessed: 03-Dec-2012].