App Isolation: Get the Security of Multiple Browsers with Just One

Eric Y. Chen
Carnegie Mellon University
Mountain View, CA
eric.chen@sv.cmu.edu

Jason Bau
Stanford University
Stanford, CA
jbau@stanford.edu

Charles Reis
Google, Inc.
Seattle, WA
creis@google.com

Adam Barth
Google, Inc.
Mountain View, CA
abarth@google.com

Collin Jackson
Carnegie Mellon University
Mountain View, CA
collin.jackson@sv.cmu.edu

ABSTRACT

Many browser-based attacks can be prevented by using separate browsers for separate web sites. However, most users access the web with only one browser. We explain the security benefits that using multiple browsers provides in terms of two concepts: entry-point restriction and state isolation. We combine these concepts into a general *app isolation* mechanism that can provide the same security benefits in a single browser. While not appropriate for all types of web sites, many sites with high-value user data can opt in to app isolation to gain defenses against a wide variety of browser-based attacks. We implement app isolation in the Chromium browser and verify its security properties using finite-state model checking. We also measure the performance overhead of app isolation and conduct a large-scale study to evaluate its adoption complexity for various types of sites, demonstrating how the app isolation mechanisms are suitable for protecting a number of high-value Web applications, such as online banking.

Categories and Subject Descriptors

H.4.3 [**Information Systems Applications**]: Communications Applications—*Information browsers*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security, Design, Verification

Keywords

Web Browser Architecture, Isolation, Web Application Security, Security Modeling, Cross-Site Request Forgery, Cross-Site Scripting

CCS'11, October 17–21, 2011, Chicago, Illinois, USA.
Copyright 2011 ACM 978-1-4503-0948-6/11/10 ...$10.00.

1. INTRODUCTION

Security experts often advise users to use more than one browser: one for surfing the wild web and others for visiting "sensitive" web sites, such as online banking web sites [1, 2]. This advice raises a number of questions. Can using more than one browser actually improve security? If so, which properties are important? Can we realize these security benefits without resorting to the use of more than one browser?

In this paper, we seek to answer these questions by crystallizing two key security properties of using multiple browsers, which we refer to as *entry-point restriction* and *state isolation.* We find that these two properties are responsible for much of the security benefit of using multiple browsers, and we show how to achieve these security benefits in a single browser by letting web sites opt in to these behaviors.

Consider a user who diligently uses two browsers for security. This user designates one browser as "sensitive" and one as "non-sensitive". She uses the sensitive browser only for accessing her online bank (through known URLs and bookmarks) and refrains from visiting the general Web with the sensitive browser. Meanwhile, she uses only the non-sensitive browser for the rest of the Web and does not use it to visit high-value sites.

Using two browsers in this manner does have security benefits. For example, consider the case of reflected cross-site scripting (XSS). In a reflected XSS attack, the attacker crafts a malicious URL containing an attack string and navigates the user's browser to that URL, tricking the honest web site into echoing back the attack string in a dangerous context. The attack has more difficulty succeeding if the user runs more than one browser because the attack relies on *which* of the user's browsers the attacker navigates. If the attacker navigates the user's non-sensitive browser to a maliciously crafted URL on the user's bank, the attack will have no access to the user's banking-related state, which resides in another browser.

From this discussion, one might conclude that isolation of credentials and other state is the essential property that makes using two browsers more secure. However, another security property provided by using multiple browsers is equally important: entry-point restriction. To illustrate entry-point restriction by its absence, imagine if the attacker could arbitrarily coordinate navigation of the users' two browsers and open an arbitrary bank URL in the sensitive browser. Now, the attacker's maliciously crafted URL and attack string can be transplanted from the non-sensitive browser to the sensitive browser, leading to disaster.

In reality, it is extremely difficult for Web attackers to coordinate the navigation of two different browsers on the users' computer. This isolation between the two browsers provides the entry-point restriction property. Namely, sessions in the sensitive browser with an honest web site always begin with a fixed set of entry points (e.g., the site's home page or a set of bookmarks) and then proceed only to URLs chosen by the web site itself, not those chosen by a third party. Because the bank's entry points are restricted, the attacker is unable to inject the attack string into the user's session with the bank.

State isolation, in turn, augments the security provided by entry-point restriction when using two browsers. State isolation plays a critical role, for example, in preventing history sniffing [3, 4] and cache timing attacks [4, 5] because these attacks do not rely upon the attacker navigating the user's browser to a maliciously crafted URL. State isolation between browsers can even protect a user's high-value session data against exploits of browser vulnerabilities that give the attacker control of the rendering process [6, 7]. In concert, entry-point restriction and state isolation provide the lion's share of the security benefits of using two browsers.

In our example above, we use a single high-value site to illustrate the security benefit of isolation using *two* browsers, but the isolation benefits extend naturally to accessing multiple sites, each in their own browser. In this paper, we show that we can realize these security benefits within a single browser by allowing web sites to whitelist their entry points and request isolated storage. This is not a pinpoint defense against a specific attack but rather a general approach that has benefits in a number of attack scenarios.

The security benefits of our mechanism do come with a compatibility cost for certain types of web sites, as it places some limitations on deep links and third-party cookies. To avoid disrupting existing web sites, we advocate deploying our mechanism as an opt in feature. Furthermore, we hypothesize and experimentally verify the types of web sites that are suitable for our mechanism. Our experiments measured the number of entry points used by popular sites in a study of 10,551 browsers running Mozilla's Test Pilot platform [8]. Over 1 million links were included in our study. We discovered that many security sensitive sites such as online banking applications can easily deploy our mechanisms. However, highly social or content-driven applications such as Facebook and New York Times will have difficulties adopting our proposal.

To evaluate the security benefits of app isolation, we model our proposals in the Alloy language, leveraging previous work on modeling web security concepts in Alloy [9]. We enrich our existing Alloy model with new concepts, such as `EntryPoints` and `RenderingEngines`, to model the essential concepts in our proposal. Our analysis revealed two issues with our initial proposals: one related to HTTP redirects and one related to an unexpected interaction between entry-point restriction and state isolation. We repair these errors and validate that our improved proposals pass muster with Alloy.

We view our main contributions in this paper as follows.

- We crystallize the security benefits of using multiple browsers into two basic concepts.
- We provide a security mechanism that grants a single browser the security benefits of multiple browsers, compatible with certain types of existing sites.
- We validate the security of our mechanism using formal modeling, adjusting our design to patch uncovered vulnerabilities.
- We evaluate the compatibility of our mechanism using Mozilla's Test Pilot platform. We are the first to utilize this platform to conduct an academic study.

1.1 Organization.

The rest of this paper is organized as follows. Section 2 presents related app isolation work. Section 3 identifies the key security benefits of using multiple browsers. Section 4 discusses how browsers can identify apps that have opted in to our proposal. Section 5 and Section 6 describe our design in detail. Section 7 evaluates our proposal in terms of its security, complexity to adopt, and performance, and we conclude in Section 8.

2. BACKGROUND

In this section, we examine how the security properties of using multiple browsers have surfaced in related work and compare them to our proposal.

2.1 Isolation with multiple browsers

For users who choose to browse the web using multiple browsers, site-specific browsers (SSBs) can make the browsing experience simpler and more convenient. SSBs provide customized browsers that are each dedicated to accessing pages from a single web application. Examples of SSBs include Prism [10] and Fluid [11].

SSBs are simply special-purpose browsers and can provide the security benefits of using multiple general-purpose browsers. However, SSBs can become difficult to manage when users interact with and navigate between a large number of different web applications. We show that a single browser can realize the security benefits of SSBs without the management burden on the user. For example, our proposal allows users to seamlessly and securely follow a link from one app to another, even in a single browser tab.

2.2 Isolation within a single browser

The concept of finer-grained isolation inside a single browser has been explored by many researchers. However, prior work has not identified the essential factors needed for a single browser to achieve the same security benefits as using multiple browsers.

Recent browsers have begun employing sandbox technology that protects the local file system from attacks that exploit browser vulnerabilities. For example, Internet Explorer on Windows Vista introduced Protected Mode [12], which protects the local file system from being modified by a compromised rendering engine. The Google Chrome browser's sandbox additionally protects the local file system from being read by a compromised by a rendering engine [6]. Unfortunately, neither of these sandboxing technologies protect *web application* state, such as cookies and local storage data, from being accessed by a compromised rendering engine.

The OP browser [13] isolates plugins from state associated with other applications by enforcing restrictions on the cross-origin request API exposed to plugins. The Gazelle browser [14] goes a step further by restricting the cross-origin request API for the entire rendering engine. Under the Gazelle approach, a *web application*'s state is only visible to the rendering engine containing it. This prevents a malicious web entity from compromising its own rendering engine to gain access to the state of other web applications. However, because Gazelle denies rendering engines from requesting cross-origin resources unless their MIME type indicates a library format such as JavaScript or Cascading Style Sheets (CSS), it imposes a compatibility cost on many web sites [15].

One approach that can mitigate the compatibility costs of restricting the cross-origin request API is to allow an application to explicitly declare the URLs that compose it. One example of this approach is the Tahoma browser [16], which allows applications to specify a manifest file listing which URLs should be included in the same protection domain. Tahoma uses a separate state container for each application, so state associated with one application will be inaccessible in another. Although Tahoma realized the importance of isolating web application state, it did not incorporate the other benefit of using multiple browsers: restricting non-sensitive web sites from directing the user to a sensitive URL.

OMash [17] only attaches cookies to same-origin requests, effectively isolating state within a particular site. Each new entry into a site creates a new session. This approach mitigates reflected XSS, cross-site request forgery (CSRF), and click-jacking, since another site cannot hi-jack an existing session with a hyperlink or iframe. However, the drawback of OMash lies in its inability to maintain user state across multiple browsing sessions.

Content Security Policy [18] attempts to mitigate XSS by allowing web sites to only execute scripts from whitelisted external JavaScript files. SOMA [19] aims to alleviate XSS and CSRF by making the host of web content mutually approve the content request with the web content embedder. Unfortunately, both of these defenses are geared to counter individual attacks such as XSS and CSRF. They do not achieve the full security benefits as using multiple browsers, such as defenses against rendering engine exploits.

In contrast, our work aims to capture the same underlying properties of using separate browsers for sensitive web apps, gaining the security benefits in a single browser.

3. SECURELY ISOLATING WEB SITES

In this section, we investigate exactly which security benefits can be achieved by visiting sensitive web sites in a different web browser than non-sensitive web sites. We classify many common browser-based attacks and show that a large number of them can be mitigated through the use of multiple browsers. We then introduce two new mechanisms in a single browser that can be used to achieve these same benefits, for particular web sites that choose to opt in and accept the compatibility implications.

3.1 Benefits of Multiple Browsers

Suppose a user wishes to protect certain sensitive web sites from more dangerous ones by using two browsers, A and B. To achieve this, she must abide by the following rules:

1. Only type in passwords for sensitive web sites with Browser A. This rule ensures that user state for the sensitive web sites are stored only in Browser A.

2. Never type in URLs or click on bookmarks to non-sensitive web sites with Browser A, and never type into Browser A URLs received from non-sensitive web sites or other untrusted sources. This rule prevents Browser A from leaking any sensitive information to Browser B and prohibits Browser B from contaminating sensitive states in Browser A.

If the user strictly abides by the rules above, all sensitive state would reside in Browser A, isolated from non-sensitive web sites and unable to leak to Browser B. Furthermore, the integrity of Browser A is maintained because untrusted content in Browser B cannot infect Browser A, preventing attacks such as reflected XSS.

These rules prevent the "cross-origin" versions of the attacks listed in Table 1. We classify attacks as "cross-origin" if the attack is launched from a different origin than the victim origin, as opposed to "same-origin" attacks (such as one Facebook page trying to mount a CSRF attack on another Facebook page). Using a separate browser does not prevent the same-origin versions of these attacks, nor same-origin only attacks such as stored XSS, because the attacker resides in the same browser as the victim.

We provide a more thorough analysis of the cross-origin attacks from Table 1 below. We assume the attacker wishes to attack a victim web site to which the user has authenticated and can lure the user into visiting a malicious web site on a different origin. Furthermore, we assume that the user uses separate browsers according to the rules above.

- **Reflected XSS** – In a reflected XSS attack, the attacker lures the user into visiting a malicious URL inside the non-sensitive browser. This URL will allow the attacker's script to execute inside the victim's origin. However, because the user is authenticated to the victim site in the sensitive browser, the attacker's script will not have access to the user's session.

- **Session fixation** – In a session fixation attack, the attacker includes a known session ID inside a victim URL, then lures the user into visiting this URL and tricks her into logging in. Once the user is logged in, the attacker can freely impersonate the user with the shared session ID. However, because the user only types her password into the sensitive browser, the attack will fail.

- **Cross-origin resource import** – In a cross origin resource import attack, the attacker's page requests a sensitive resource from the victim's origin as a script or style sheet. If the user were authenticated to the victim site in the same browser, this request can leak confidential information to the attacker. However, the user is authenticated instead in the sensitive browser, thereby foiling the attack.

- **Click-jacking** – Click-jacking attacks overlay a transparent iframe from a victim page over a part of the attacker's page where the user is likely to click. This aims to trick the user into clicking somewhere on the

Cross-origin Attacks	Entry-point Restriction	State Isolation	Separate browser
Reflected XSS	✓		✓
Session Fixation	✓		✓
Cross-Origin Resource Import	✓	✓	✓
Click-jacking		✓	✓
CSRF	✓	✓	✓
Visited Link Sniffing		✓	✓
Cache Timing Attack		✓	✓
Rendering Engine Hi-jacking		✓	✓

Table 1: Cross-origin attacks mitigated by entry-point restriction, state isolation, and using separate browsers. Same-origin attacks, such as stored XSS, are not mitigated.

victim's page (e.g., the delete account button) without realizing it. Because the user is authenticated in the sensitive browser, clicking on the transparent victim iframe in the non-sensitive browser will cause no damage.

- **Cross-site request forgery** – In a traditional CSRF attack, the adversary makes subresource requests within a page she owns in an attempt to change the user's state on the victim's server. This attack succeeds because the user's credentials are attached to the attacker's subresource request. However, because the user authenticates only in the sensitive browser, the malicious request will not have a cookie attached, rendering it harmless.

- **Visited link sniffing** – The attacker's web site might attempt to sniff the user's browsing history by drawing visited links in a different color or style than unvisited ones, and then using JavaScript or CSS to discover which have been visited. Although a possible mitigation has been proposed and adopted by several major browsers [20], new attacks have been discovered that can detect browsing history despite the defense [21]. However, if the user uses separate browsers, these browsers have different history databases, so a web site in the non-sensitive browser is unable to discern the browsing history of the sensitive browser.

- **Cache timing attack** – Similar to visited link sniffing attacks, an attacker can measure the time to load a victim resource to determine if the user has visited it [4, 5]. Different browsers have different caches for their web resources, so web sites in the non-sensitive browser cannot detect cache hits or misses in the sensitive browser.

- **Rendering engine hi-jacking** – A powerful attacker might exploit a vulnerability in the browser's rendering engine to hi-jack its execution. For browsers with a single rendering engine instance (e.g., Firefox and Safari), this would let the attacker access all the user's state, such as the victim site's cookies and page contents. These attacks still apply to browsers with multiple rendering engine instances, if they rely on the rendering engine to enforce the Same-Origin Policy (e.g., Chrome and IE8). However, if the user logged in to the victim site with a different browser, the victim's cookies and sensitive pages will reside in an entirely different OS process. Assuming the exploited rendering engine is sandboxed, the attacker's exploit is unable to access this process.

3.2 Site Isolation in a Single Browser

As shown in the previous section, using a dedicated browser to visit certain sites mitigates a significant number of web attacks. This observation raises a question: which properties of browsing with a single browser make it vulnerable to these attacks? We believe the answer to this question can be summarized in three points:

1. Malicious sites are free to make requests to vulnerable parts of victim's site.

2. Malicious sites can make requests that have access to the victim's cookies and session data.

3. Malicious sites can exploit the rendering engine for direct access to in-memory state and to stored data from the victim site.

Our key observation is that these abilities are not fundamental flaws of browsing with a single browser but rather weaknesses of current browsers. We believe that for many types of web sites, it is possible to simulate the behavior of multiple browsers with a single browser by solving each of these weaknesses. These changes come with a compatibility cost, however, because benign third-party sites are also prevented from accessing the user's cookies. We evaluate the complexity that different types of sites face for adopting these changes in Section 7.2.

In the next three sections, we introduce mechanisms for removing these limitations in a single browser. First, we provide a means for web sites to opt in to this protection if they accept the compatibility implications. Second, we prevent untrusted third parties from making requests to vulnerable parts of these web sites. Third, we isolate the persistent and in-memory state of these sites from other sites. Because our approach works best with "app-like" web sites that contain sensitive user data and few cross-site interactions, we refer to this approach as *app isolation*.

4. IDENTIFYING ISOLATED WEB APPS

App isolation can provide a web site with the security benefits of running in a dedicated browser, but it comes at some compatibility cost. Isolating cookies and in-memory state not only prevents malicious web sites from compromising sensitive data, it can also hinder legitimate web sites from sharing information. For example, Facebook Connect[22] lets web sites access visitors' identifying information via Facebook, which would not work if Facebook was isolated in a separate browser. To remain compatible with web sites that desire this sharing, we employ an opt-in policy that lets web developers decide whether to isolate their site or web application from the rest of the browser.

We must choose the opt-in mechanism carefully to avoid introducing new security concerns, and we must consider the granularity at which the isolation should take effect. In this section, we first show the consequences of an inadequate opt-in mechanism using HTTP headers. Then, we describe a viable origin-wide approach with *host-meta*, and refine it to support sub-origin level web applications with manifest files.

Bootstrapping with HTTP headers.

As a straw man, we first consider identifying an isolated app using a custom HTTP header (e.g., X-App-Isolation: 1). If the browser receives this header on an HTTP response, the browser treats all future responses from the origin as belonging to the isolated app.

The primary disadvantage of this approach is that it does not verify that the given response has the privilege to speak for the entire origin. This lack of verification lets owners of portions of an origin (e.g., `foo.com/~username/`) opt the entire origin in to app isolation. A malicious sub-domain owner can use this mechanism to prevent desirable sharing on other parts of the origin, or he can misconfigure the app (e.g., listing a non-existent entry point) to perform a denial-of-service attack. Worse, bootstrapping with a custom HTTP response header might not enforce the policy for the initial request sent to the server, opening a window of vulnerability.

Bootstrapping with Host-meta.

To avoid attacks that grant the privileges of the entire origin to each resource, the browser can instead bootstrap app isolation using a file at a well-known location that can only be accessed by the legitimate owner of the origin. The *host-meta* mechanism is designed for exactly this reason [23]. With *host-meta*, the owner of the origin creates an XML file containing app isolation meta data located at `/.well-known/host-meta`. This meta data can include configuration information, such as a list of acceptable entry points. Because *host-meta* should be controllable only by the legitimate owner of the origin, an adversary controlling only a directory will not be able to influence the app isolation policy for the entire origin.

It is essential to retrieve *host-meta* information through a secure channel such as HTTPS. Otherwise, an active network attacker can replace the host-meta entries with bogus URLs, allowing the attacker to conduct denial-of-service or other misconfiguration attacks.

One downside of bootstrapping with *host-meta* is that is has poor performance because an additional round trip is required to fetch a resource if the host-meta file is not in the cache.

Bootstrapping with Manifest File.

The above proposals work at the granularity of an origin. However, it is also possible to isolate web apps at a finer granularity without violating the security concerns of "finer-grained origins" [24].

In the Chrome Web Store, web application developers package their applications using a manifest file [25]. This method of packaging web applications is becoming common; for example, Mozilla's Open Web Applications are also packaged using a such file [26]. The file includes a list of URL patterns that comprise the application, together with other meta data such as requested permissions. The manifest file provides extra context to the browser for how to treat the app, allowing it to enforce policies that might break ordinary web content. The Chrome Web Store also supports "verified apps [27]," in which the manifest file's author demonstrates that she has control over all origins included in the application's URL patterns.

We use additional syntax in the manifest to let applications in the Chrome Web Store opt in to app isolation. The URL patterns in the manifest might or might not span an entire origin, which would allow a site like Google Maps (e.g., `http://www.google.com/maps`) to opt into isolation features without affecting the rest of the origin. The Chrome Web Store already provides a mechanism for verifying that the manifest is provided by the web site author, which we leverage to prevent a malicious manifest file from bundling attacker URLs in the same app as a victim site.

The reason this does not run afoul of the typical security concerns of finer-grained origins is that our state isolation effectively separates the application's pages from the rest of the web, including non-application pages in the same origin. Origin contamination via scripts or cookies is blocked because an application page and a non-application page do not share the same renderer process or cookie store.

Both origin-level isolation using *host-meta* and application-level isolation using manifest files are viable opt in mechanisms. We leave it to browser vendors to decide on which method they deem appropriate. In the remainder of this paper, we refer to the unit of isolation as an *app*, whether designated as an origin or a collection of URLs in a manifest.

5. ENTRY-POINT RESTRICTION

Using multiple browsers securely requires the user to refrain from visiting a sensitive app at a URL that could be constructed by an attacker. Instead, the user always visits the app in the sensitive browser from a known starting point. Simulating this behavior with a single browser requires an intuitive way of visiting URLs of sensitive apps without compromising security. Our proposal for *Entry-point Restriction* provides a way to safely transition between sensitive and non-sensitive pages in a single browser, without altering the user's behavior.

In this section, we present the rules for entry-point restriction and discuss the challenges for selecting appropriate entry points. Table 1 lists attacks that are prevented by entry-point restriction.

5.1 Design

Under the entry-point restriction policy, we define an *entry point* to be a landing page of an app designated by the app's owner. Any app may choose to opt into entry-point restriction by providing at least one entry point.

Once an app opts into entry-point restriction, the browser may load a resource from the app if and only if at least one of the following statements holds true.

- The resource is requested by a page inside the app.
- The URL of the resource is a valid entry point for the app.

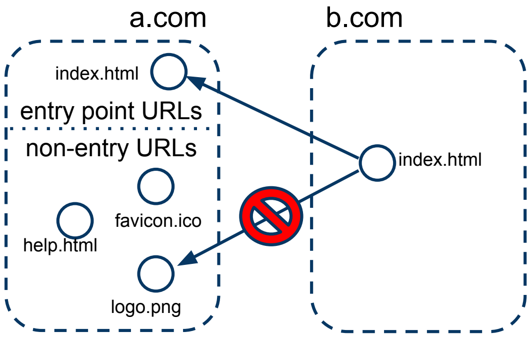

Figure 1: Entry-point Restriction

Figure 1 illustrates how entry-point restriction works in practice. Suppose `a.com` hosts its company logo at `a.com/logo.png`, and specifies an entry point at `a.com/index.html`. Meanwhile, `b.com/index.html` includes a hyperlink to `a.com/index.html` and embeds an image with the source attribute of `a.com/logo.png`. In this scenario, the user is able to follow the hyperlink because it points to a valid entry point. However, the logo will fail to load because it is a cross origin request to a non-entry-point URL.

Choosing entry points.

Entry points are whitelisted URLs or URL patterns that the app's owner trusts to load safely even when requested by an attacker's page. Therefore, it is crucial to choose these URLs with care. The goal is to select common landing pages that do not present opportunities for the attacker to exploit the user's credentials. We offer the following guidelines for selecting entry points.

- An entry point URL should cause no sensitive state changes, because an attacker may request it with the user's credentials.
- An entry point URL pattern should be as tightly constrained as possible, reducing the opportunity for attackers to place malicious code in the URL.
- An entry point URL should not return confidential information in a form that could be accessed on a cross-origin page, such as JSON.

In most cases, the default landing page for an app meets these guidelines. Many sites expect their users to arrive via multiple landing pages, creating a need to support multiple entry points. For example, an online banking site may have an English login page and a French login page. For additional flexibility, we also allow web sites to use wildcard characters in their entry point URLs (e.g., www.a.com/*/ where * represents any number of non-'/' characters). Allowing wildcards trades off some security benefits for compatibility because the number of wildcard characters and entry points is directly correlated with the size of the app's attack surface. However, this is still an improvement over having no such policy. We recommend that app owners restrict their policies as tightly as possible and use wildcards only when necessary.

It is important to note that not all web sites are well-suited for entry-point restriction. Some sites depend heavily on deep links to content, such as socially integrated sites like Facebook or content-oriented sites like New York Times. These sites will have a difficult time adopting the policy, because it is extremely difficult to identify all the URLs that legitimate sites may link to. We show in Section 7.2 that other types of sites, such as online banks, are amenable to these restrictions and can benefit from entry-point restriction.

Sub-resource restrictions.

By default, entry-point restriction must deny sub-resources at non-entry-point URLs from loading. This will prevent vulnerabilities such as reflected XSS and CSRF; however, it may also affect legitimate web pages. For example, loading a non-entry-point image will fail despite being typically a safe action.

Fortunately, this usability constraint can be alleviated. Entry-point restriction is only necessary because malicious requests will have the user's authentication tokens attached to them. Section 6 describes how State Isolation can be used to isolate these authentication tokens from sub-resource requests. When used in conjunction with state isolation, entry-point restriction can safely allow sub-resources at non-entry-point URLs.

5.2 Implementation

We implemented a proof-of-concept entry-point restriction mechanism in the Chromium browser. The entire system consists of less than 100 lines of C++ code. Our implementation enforces entry-point restriction inside Chromium's WebKit rendering engine. More specifically, we modified the *CanDisplay()* function of *SecurityOrigin*, which gets called before every web resource request. If the URL of a web resource violates the entry-point restriction policy, WebKit will not issue the network request.

Storing entry points.

Like most web resources, it is desirable for the browser to cache entry-point restriction information for performance. The exact caching method may differ depending on how apps opt into entry-point restriction.

If apps use host-meta to bootstrap app isolation, browsers could cache this information like conventional web resources. Users should be able to clear their app isolation information the same way they clear cookies or browsing histories. Most modern browsers offer users with private browsing features that allow them to browse without persistent storage [28]. To be compatible with these private browsing features, the host-meta information in such modes must be treated like cookies or browsing history, not being written to disk.

If apps instead use app manifests (such as the installed apps from the Chrome Web Store), the policies are stored in the browser's persistent app meta data. This essentially permits the browser to permanently cache the app isolation policies as long as the application is installed. To update the policy, developers can use the standard app update process.

6. STATE ISOLATION

The remaining security benefits of using multiple browsers shown in Table 1 result from isolating an app's state from other web sites. In traditional browsers, attackers can try to take advantage of persistent state, such as using an app's cookies in a CSRF attack. They can also try to directly access in-memory state by exploiting the browser's rendering engine and then inspecting memory.

We can simulate the state benefits of using a separate browser for an app with a single multi-process browser. This requires isolating both the in-memory and persistent state of the app from other web sites, using the process and storage architectures of the browser.

6.1 Design

Once the browser identifies the URLs comprising an isolated app (as discussed in Section 4), it can ensure instances of those pages are isolated in memory and on disk.

In-Memory State.

Any top-level page loaded from a URL in the app's manifest must be loaded in a renderer process dedicated to that app. Any sub-resource requests are then made from the same process as the parent page, even if they target URLs outside the app's manifest.

We treat sub-frames in the same manner as sub-resources. While this may open the app's process to attacks from a non-app iframe, the app does have some control over which iframes are present. Similarly, an app URL may be requested as an iframe or sub-resource outside the app process. The potential risk of this approach, that of framing attacks, is mitigated by persistent state isolation as described below, which ensures that such requests do not carry the user's credentials. This approach has the same security properties as loading the app in a separate browser.

Top-level pages from all other URLs are not loaded in the app's renderer process. Combined with an effective sandbox mechanism [6], this helps prevent an exploited non-app renderer process from accessing the in-memory state present in the app's process.

The browser kernel process can then take advantage of the process isolation between apps and other sites to enforce stricter controls on accessing credentials and other resources. HTTP Auth credentials, session cookies, and other in-memory state stored in the browser kernel is only revealed to the app's process.

Persistent State.

The browser kernel also creates a separate storage partition for all persistent state in the isolated app. Any requests from the app's renderer process use only this partition, and requests from other renderer processes do not have access to it. The partition includes all cookies, localStorage data, and other local state.

As a result, a user's session within an app process is not visible in other renderer processes, even if a URL from the app is loaded in an iframe outside the app process.

The storage partition can also isolate the browser history and cache for an app from that for other web sites. This can help protect against visited link and cache timing attacks, in which an attacker tries to infer a user's specific navigations within an app.

Combining with Entry-point Restriction.

When both entry-point restriction and state isolation are used together, the mechanisms complement each other and we can relax one of the restrictions for entry-point restriction. Specifically, a non-app page can be permitted to request non-entry-point URLs for sub-resources and iframes. This mimics the behavior when using a separate browser for the app, and it still protects the user because credentials are safely restricted to the app process.

6.2 Implementation

We implemented state isolation for apps in Chromium with roughly 1400 lines of code. For in-memory state isolation, Chromium already offers stricter process separation between installed web apps from the Chrome Web Store than most web sites. Pages from URLs in an app manifest are loaded in a dedicated app process. In the general case, Chromium avoids putting pages from different origins in the same process when possible, but cross-origin pages can share a process in many cases to avoid compatibility concerns [29].

However, we needed to strengthen Chromium's process isolation to more thoroughly prevent non-app pages from loading in the app's process. First, we needed to ensure that apps are not placed in general renderer processes if the browser's process limit is reached. Second, we needed to ensure navigations from an app URL to a non-app URL always exit the app's process.

For persistent state isolation, we changed Chromium to create a new URL context (a subset of the user's profile data) for each isolated app. The cookies, localStorage data, and other persistent information is stored on disk in a separate directory than the persistent data for general web sites. The browser process can ensure that this data is only provided to renderer processes associated with the app, and not to general renderer processes.

7. EVALUATION

In this section, we evaluate state isolation and entry-point restriction in three ways. First, we perform a formal analysis for the security properties of these mechanisms. Second, we experimentally assess the feasibility of various web sites adopting these mechanisms. Finally, we quantify their performance overhead relative to using one or multiple browsers.

7.1 Security

We used model-checking to evaluate the combined security characteristics of app isolation using both state isolation and entry-point restriction. Our approach consists of defining the security goals of app isolation, then modeling our implementation, its security goals, and attacker behavior in the web security framework described in [9] using Alloy [30, 31], a declarative modeling language based on first-order relational logic. We then analyze whether the expressed goals were met with the help of the Alloy analyzer software.

Security Goals.

The broad security goal of both our mechanisms are *isolation*. Isolation protects sensitive resources belonging to the app, such as non-entry URLs, scripts, and user credentials, against unauthorized use by web pages or scripts not belonging to the app. We distill two *isolation* goals which, if met, will provide the app with defenses against the attacks described in Section 3.1. (This property holds because the attacks either require an attacker to gain access to exploitable URLs within the app or use sensitive state from the app, or both.)

These goals are modeled by Alloy *assertions* (logical predicates whose consistency with the model may be checked) analogous to the following statements:

1. Browser contexts (pages or scripts) originating outside an app will not read or overwrite state issued within the app, such as credential cookies.

2. Browser contexts (pages or scripts) originating outside an app will not obtain a non-entry resource within the app.

Isolation Mechanisms.

We model entry-point restriction as an Alloy *fact* (a logic constraint which always holds), reproduced below. The *fact* states that the *browser* will not issue any cross-origin requests for a non-entry resource in an entry-restricting origin.

```
fact StrictEntryEnforcement {
  all sc:ScriptContext |
    sc.location=StrictEntryBrowser implies
      no areq:sc.transactions.req |
        areq.path=NON_ENTRY and
        isCrossOriginRequest[areq] and
        isRequestToStrictEntryOrigin[areq]
}}
```

To model state isolation, we refined the browser model of [9] by adding a set of `RenderingEngine`s associated with each `Browser`. Each `RenderingEngine` then runs a set of `ScriptContext`s, as shown in the Alloy *signatures* below:

```
sig Browser extends HTTPClient {
    engines: set RenderingEngine }

sig RenderingEngine {
   contexts: set ScriptContext,
   inBrowser : one Browser }
```

The actual state isolation is modeled by Alloy *facts*. The first fact states that each `cookie` in the model is tagged with the `RenderingEngine` of the `ScriptContext` in which it was first received. The next states that access to `cookie`s are restricted to only `ScriptContext`s from an origin matching the domain setting of the `cookie` executing in a `RenderingEngine` matching the `cookie` tag.

Our app container model also includes a browser behavior relevant to app isolation, as described by Section 6.1: it associates a newly opened `ScriptContext` with the existing `RenderingEngine` of an app if the top-level URL of the new `ScriptContext` is within the app.

Finally, our modeling assumes that users will behave conservatively within an isolated app window, meaning attackers cannot get their `ScriptContext`s in the same `RenderingEngine` as an app when separate `RenderingEngine`s exist.

Web and Rendering Engine Attacker.

We then modeled the abilities of the attacker. As described in [9], the abilities of web attackers include ownership of a web server by which they can introduce `ScriptContexts` under their control into the user's `browser`. Our modeled "rendering engine attackers" can additionally create scripts that compromise the `RenderingEngine` of the user's browser, giving them arbitrary control over other `ScriptContexts` on the same `RenderingEngine`, such as reading cookies, creating new `ScriptContext`s, sending requests, etc. However, our model assumes that storage isolation is enforced by an entity outside the rendering engines, like a browser kernel. Thus, the "rendering engine attacker" cannot compromise the storage isolation mechanism.

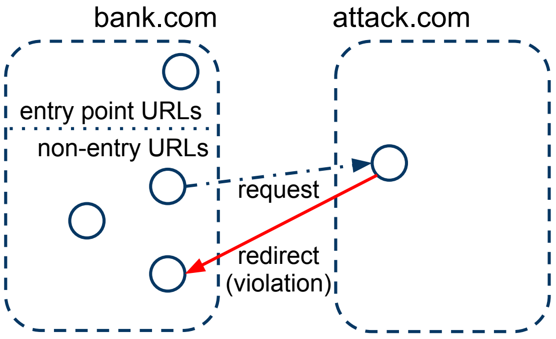

Figure 2: Entry-point restriction violation found by Alloy model.

Entry Restriction Results.

We first checked our implementation of entry restriction against the stated security goals and found that assertion 2 above was violated. We confirmed that this violation also existed in our implementation at that time and note (with some sheepishness) that the implementation bug resembles ones previously found by [9] in Referer validation defenses proposed by [32].

The violation, illustrated in Figure 2, occurs because of HTTP redirects. Suppose origin *bank.com* is a victim origin that uses entry-point restriction, and origin *attack.com* is a external origin that does **not** use entry-point restriction. A page created by *bank.com* is allowed by the browser to cause a request for a non-entry resource in *attack.com*, since *attack.com* does not use entry-point restriction. *attack.com* may then issue a redirect to the browser telling it to find the requested resource back at *bank.com*. The browser will then re-issue the request, now to *bank.com*, which will be granted by *bank.com* because the request was initiated by a context owned by *bank.com*. This violates the integrity goal because the external origin *attack.com* plays a role in redirecting the request back to *bank.com*, thus "requesting" the non-entry resource.

To fix this violation, we updated our implementation to keep track of all redirects experienced by a request and to refrain from sending a request for a non-entry resource to an entry-isolating domain if any external domains are recorded in the request's redirects. We verified that the model containing this fix now upholds the previously violated *integrity* assertion, up to the finite size we tried (up to 10 `NetworkEvent`s, which are either requests or responses).

App Isolation Results.

We then used the model to check both app mechanisms (entry-point restriction and state isolation) and found that neither mechanism individually was able to uphold the security goals in the presence of the rendering engine attacker. For

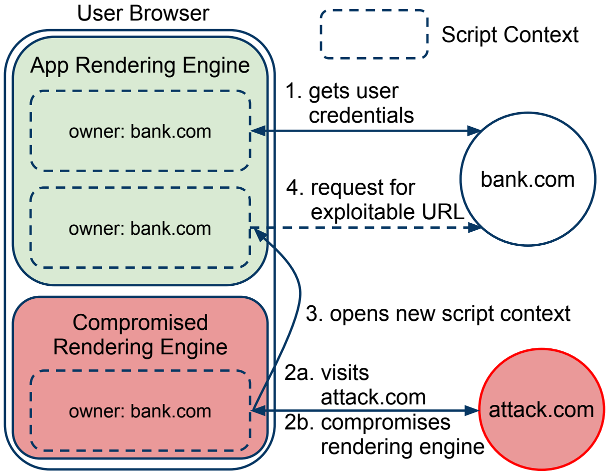

Figure 3: Isolation violation found by Alloy model. Request 4 to a non-entry URL, containing credentials issued within the app, is granted if entry-point restriction is absent or incorrect.

entry-point restriction without state isolation, there is only a single `RenderingEngine` for all `ScriptContexts`, letting the attacker trivially violate assertion 1 above by compromising the `RenderingEngine` and accessing cookies issued by the app. For state isolation without entry-point restriction, the Alloy analyzer found a violation of assertion 2 above that is very similar to the violation we found in the model with both mechanisms. We will therefore describe both results together in the next paragraphs.

When checking the model with both mechanisms, the Alloy analyzer found no violations to assertion 1 above but did find a violation for assertion 2. This violation occurs because our implementation at the time did not consider the app container as part of its notion of "same-origin" when applied to entry-point restriction. Figure 3 illustrates this violation, as well as the violation found for state isolation without entry-point restriction. The scenario is as follows:

Alice opens a session with *bank.com* as an app with a dedicated *app* renderer and receives credentials. Alice also visits *attack.com* with the browser's *ordinary* renderer, causing *attack.com* to send a script which compromises the *ordinary* renderer. The attacker creates a *bank.com* script context in the *ordinary* renderer. Then the attacker causes the *bank.com* script context to open a new window with top-level URL pointed at an exploitable non-entry URL within the *bank.com* app. This new window will open in the *app* renderer, because its initial URL is within the app, and its request for the non-entry URL will pass entry point checks, because the script context which caused the request is "same-origin" (owned by *bank.com*). Similarly, the request will also be sent if entry-point restriction is absent. This last request thus causes the attack to succeed.

The discovery of this vulnerability underscores two points regarding app containers. The first is that **both** entry-point restriction and state isolation mechanisms are necessary to stop a rendering engine attacker. The second is that the same-origin policy must be extended to include app containers. In effect, app containers divide a previously atomic origin into two new origins, one inside and one outside the container. As such, the entry-point restriction policy should have rejected the last request in our scenario, because its source was outside the app container and its target was inside. We implemented this updated notion of same-origin in our model and verified that both *assertions* were now upheld, up to the finite sizes we tried, again up to 10 `NetworkEvents`.

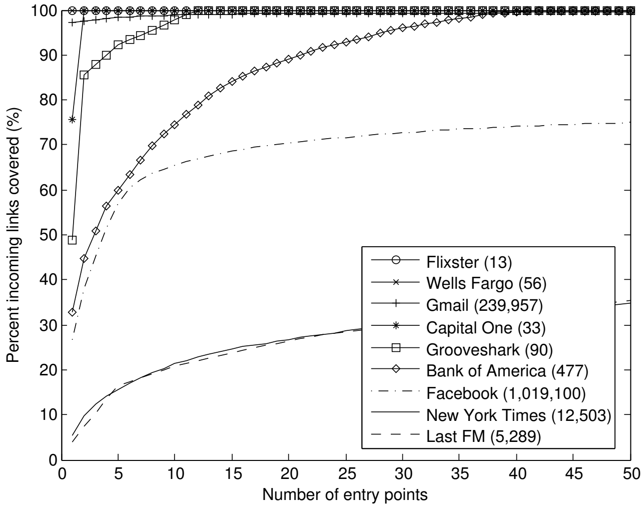

Figure 4: Entry points to popular sites observed by 10,551 Mozilla Test Pilot browsers. Numbers in parentheses indicate the total number of incoming links observed.

7.2 Complexity of Adoption

Entry-point restriction requires web sites to identify all URLs that they wish make public to other web sites. Highly socially integrated sites like Facebook or content-oriented sites like New York Times will have difficulty adopting entry-point restriction due to the inherent complexity in capturing all the possible entry points. On the other hand, we believe non-social web sites such as online banking applications will have an easier time identifying valid entry-points, making it practical to deploy entry-point restriction.

To gain additional insight into the effort required for different web sites to adopt entry-point restriction, we used the Mozilla Test Pilot platform [8]. Mozilla Test Pilot is a Firefox extension installed on more than 3 million active Firefox browsers. Our evaluation was performed on 10,551 of those browsers over a period of 3 days. In our study, we simulated entry-point restriction for 9 web sites shown in Figure 4. These sites consist of popular email and web applications, online banking pages of major financial institutions, news and social networks, and popular Chrome Web Store applications.

For each site, we gathered URL hashes of all *incoming links* to the site that appear on all pages that Test Pilot Browsers visited. Each incoming link represents either an actual HTML link or an embedded resource pointing to the site (e.g., `a.com/logo.png` would be an incoming link to `a.com`). Our results demonstrate several ideas. First, they verify our hypothesis that web sites that encourage the sharing of content will have a difficult time opting in to entry-point restriction (e.g., New York Times, Facebook, and Last.fm). Second, web sites with no intention of sharing content can opt in to entry-point restriction with relatively few entry points (e.g., Wells Fargo, Capital One and Flixster, a Chrome Web Store application). For certain sites such as Gmail and Bank of America, the compatibility with entry-point restriction is less clear. While 10 entry points can cover up to 95 percent of the incoming links, fully covering all incoming links appears to be a non-trivial task. For Gmail, we suspect this to be due to Gmail widgets, multiple login URLs (e.g., `mail.google.com/mail/u/0/`), and web mail portals from numerous organizations hosted by Gmail (e.g., `mail.google.com/a/west.cmu.edu/`). It is difficult to confirm this hypothesis from the data set, as we only collected hashes of the URLs to protect user privacy. For the online banking application for Bank of America, each entry point *path* is valid at a number of regional load-balancing *domain*s, thus accounting for the large number of total entry point URLs.

Web applications that consist of multiple subdomains sometimes face an interesting challenge, if some subdomains are more amenable to app isolation than others. For example, a bank may have some of its login-guarded functionality on a `online` subdomain, while also having a separate `creditcards` subdomain with significant numbers of entry points. If pages on the `creditcards` subdomain can also recognize when a user is logged in, then it is difficult to isolate the two subdomains from each other. Such apps may either face difficulties adopting app isolation or be forced to specify less precise entry points.

To assist web site owners with identifying valid entry points and determining whether app isolation is suitable for their site, we propose a *report-only* mode similar to that of Content Security Policy [33]. Instead of enforcing a policy violation, report-only mode will send a violation report to the app's server. Report-only mode can thus be used to generate a suitable policy file that avoids false positives.

Web developers interested in adopting app isolation should consider the specific feature trade-offs they will be making. Their apps should have limited deep incoming links as entry points. They should not rely on authenticated resources from third parties, and they should not depend on their own authenticated resources being loaded on third party sites. Overall, we found that certain types of sites, including several online banks, do appear to be good candidates for adopting app isolation.

7.3 Performance

In this section, we evaluate the performance overhead of app isolation due to entry-point restriction and state isolation. While extra disk space is required for isolated caches, the overhead is generally far less than using multiple browsers.

7.3.1 Navigation Latency

In an entry-point restriction enabled browser, every web resource load for an app is preceded by an entry-point check. This check determines whether the URL of the web resource matches one of the known entry-points. Entry-point lookup can be made efficient using a hash table, imposing negligible cost on navigation latency. We measured the load times of the Alexa Top 100 Web sites with and without entry-point restriction enabled. For an artificially high list of 10000 entry points, the overhead incurred from hash table lookups was small enough to be lost in the noise (less than 0.1 ms per page load).

Besides entry-point lookup, policy files must also be fetched. The fetch of the policy file is done only once at app installation time, and thus we do not include it in the performance overhead.

7.3.2 Storage and Memory Overhead

To see the impact of state isolation, we measured the disk and memory space required for visiting 12 popular sites in their own tabs, similar to the sites used in Figure 4. Chromium stores a user's persistent state in a configurable profile directory, so we compared three conditions: all sites in a single Chromium profile, all in a single Chromium profile as isolated apps, and each in a separate Chromium profile. For sites that did not require HTTPS, we used pre-recorded network data to reduce variability. For Gmail, Bank of America, and Chase Bank, we logged into an account. We report the average of three trials.

Visiting all sites in a single profile required 19 MB of disk space. Using isolated apps required 86 MB, while multiple browsers required 117 MB. Each of these profiles includes a partial download of Chromium's Safe Browsing database (2.6 MB), which is a source of overhead for each additional browser profile.

We were surprised that isolated apps required over 4 times the space of a single profile. This is because Chromium aggressively allocates disk space for each cache. This behavior could be modified to be less aggressive for isolated apps. Users could also opt for an in-memory cache for isolated apps, which retains the security benefits and lowers the disk space required to 9.6 MB.

The total resident memory required for visiting all sites in separate tabs of a single profile was 729 MB.[1] We found that using isolated apps used a comparable 730 MB, while using a separate browser for each site used an aggregate of 1.83 GB memory.

These results show that by using isolated apps rather than multiple browsers, we can reduce the performance trade-off required for our security benefits.

8. CONCLUSION

We have shown that a single browser can achieve the security benefits of using multiple browsers, by implementing *entry-point restriction* and *state isolation* to isolate sensitive apps. These mechanisms might not be appropriate for every web site, but they can be effective for many high-value web sites, such as online banks. Using this approach, these high-value web sites can help protect themselves and their users from a broad spectrum of attacks with minimal effort.

Acknowledgements

We thank John Mitchell for his helpful suggestions and feedback. We thank David Chan, Sid Stamm, Jono Xia, Jinghua Zhang and the entire Mozilla Test Pilot team for giving us the opportunity to use Mozilla's Test Pilot platform. This work was supported by a Google Focused Research Award on the security of cloud and Web clients.

[1] Chromium over counts memory usage by not fully accounting for shared memory, but this is consistent across our three conditions.

9. REFERENCES

[1] R. Cook, "The Next Big Browser Exploit," *CSO Magazine*, p. 15, Feb 2008.

[2] E. Iverson, "Two Web Browsers can be More Secure than One." http://www.blueridgenetworks.com/securitynowblog/dual-web-browsers-can-avoid-information-disclosures

[3] D. Jang, R. Jhala, S. Lerner, and H. Shacham, "An empirical study of privacy-violating information flows in JavaScript web applications," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*. ACM, 2010, pp. 270–283.

[4] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell, "Protecting browser state from web privacy attacks," in *Proceedings of the 15th International Conference on World Wide Web*, ser. WWW '06. New York, NY, USA: ACM, 2006, pp. 737–744. [Online]. Available: http://doi.acm.org/10.1145/1135777.1135884

[5] E. Felten and M. Schneider, "Timing attacks on web privacy," in *Proceedings of the 7th ACM Conference on Computer and Communications Security*. ACM, 2000, pp. 25–32.

[6] A. Barth, C. Jackson, and C. Reis, "The Security Architecture of the Chromium Browser," 2008 Technical Report.

[7] Mozilla Foundation Security Advisory 2009-29, "Arbitrary code execution using event listeners." http://www.mozilla.org/security/announce/2009/mfsa2009-29.html

[8] Mozilla, "Test Pilot," https://testpilot.mozillalabs.com/.

[9] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song, "Towards a Formal Foundation of Web Security," *Computer Security Foundations Symposium, Proceedings of, IEEE*, vol. 0, pp. 290–304, 2010.

[10] Mozilla, "Prism," http://prism.mozillalabs.com/.

[11] T. Ditchendorf, "Fluid," http://fluidapp.com/.

[12] M. Silbey and P. Brundrett, "Understanding and working in Protected Mode Internet Explorer," 2006, http://msdn.microsoft.com/en-us/library/bb250462.aspx.

[13] C. Grier, S. Tang, and S. T. King, "Secure Web Browsing with the OP Web Browser," in *IEEE Symposium on Security and Privacy*, 2008, pp. 402–416.

[14] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter, "The Multi-Principal OS Construction of the Gazelle Web Browser," in *USENIX Security Symposium*, 2009, pp. 417–432.

[15] L.-S. Huang, Z. Weinberg, C. Evans, and C. Jackson, "Protecting Browsers from Cross-Origin CSS Attacks," in *ACM Conference on Computer and Communications Security)*, 2010.

[16] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen, "A Safety-Oriented Platform for Web Applications," in *IEEE Symposium on Security and Privacy*, 2006, pp. 350–364.

[17] S. Crites, F. Hsu, and H. Chen, "OMash: enabling secure web mashups via object abstractions," in *ACM Conference on Computer and Communications Security*, 2008, pp. 99–108.

[18] S. Stamm, B. Sterne, and G. Markham, "Reining in the Web with Content Security Policy," in *International Conference on World Wide Web (WWW)*, 2010.

[19] T. Oda, G. Wurster, P. V. Oorschot, and A. Somayaji, "SOMA: Mutual Approval for Included Content in Web Pages," in *ACM Conference on Computer and Communications Security*, 2008.

[20] L. D. Baron. (2010) Preventing attacks on a user's history through CSS `:visited` selectors. http://dbaron.org/mozilla/visited-privacy

[21] Z. Weinberg, E. Y. Chen, P. Jayaraman, and C. Jackson, "I Still Know What You Visited Last Summer: Leaking browsing history via user interaction and side channel attacks," in *IEEE Symposium on Security and Privacy*, 2011.

[22] D. Morin, "Announcing Facebook Connect," 2008, https://developers.facebook.com/blog/post/108/.

[23] E. Hammer-Lahav, "Web Host Metadata," 2010, http://tools.ietf.org/html/draft-hammer-hostmeta-13.

[24] C. Jackson and A. Barth, "Beware of Finer-Grained Origins," in *Web 2.0 Security and Privacy*, 2008.

[25] Google, "Packaged Apps," http://code.google.com/chrome/extensions/apps.html.

[26] Mozilla, "Manifest File," https://developer.mozilla.org/en/OpenWebApps/The_Manifest.

[27] Google, "Verified Author," http://www.google.com/support/chrome_webstore/bin/answer.py?hl=en&answer=173657.

[28] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh, "An Analysis of Private Browsing Modes in Modern Browsers," in *USENIX Security Symposium*, 2010, pp. 79–94.

[29] C. Reis and S. D. Gribble, "Isolating Web Programs in Modern Browser Architectures," in *ACM European Conference on Computer Systems (EuroSys)*, 2009.

[30] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.

[31] ——, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[32] F. Kerschbaum, "Simple cross-site attack prevention," in *Security and Privacy in Communications Networks and the Workshops, 2007. SecureComm 2007. Third International Conference on*, Sept. 2007, pp. 464 –472.

[33] Mozilla, "CSP specification," 2011, https://wiki.mozilla.org/Security/CSP/Specification#Report-Only_mode.