On the Incoherencies in Web Browser Access Control Policies

Kapil Singh*, Alexander Moshchuk†, Helen J. Wang† and Wenke Lee*
*Georgia Institute of Technology, Atlanta, GA
Email: {ksingh, wenke}@cc.gatech.edu
†Microsoft Research, Redmond, WA
Email: {alexmos, helenw}@microsoft.com

***Abstract*—Web browsers' access control policies have evolved piecemeal in an ad-hoc fashion with the introduction of new browser features. This has resulted in numerous incoherencies. In this paper, we analyze three major access control flaws in today's browsers: (1) principal labeling is different for different resources, raising problems when resources interplay, (2) runtime changes to principal identities are handled inconsistently, and (3) browsers mismanage resources belonging to the user principal. We show that such mishandling of principals leads to many access control incoherencies, presenting hurdles for web developers to construct secure web applications.**

A unique contribution of this paper is to identify the compatibility cost of removing these unsafe browser features. To do this, we have built WebAnalyzer, a crawler-based framework for measuring real-world usage of browser features, and used it to study the top 100,000 popular web sites ranked by Alexa. Our methodology and results serve as a guideline for browser designers to balance security and backward compatibility.

I. Introduction

Web browsers have gradually evolved from an application that views static web pages to a rich application platform on which mutually distrusting web site principals co-exist and interact [1]–[3]. Along the way, the browsers' access control policies have also been evolving, but unfortunately this happened in a piecemeal and ad-hoc fashion alongside the introduction of new browser features (such as AJAX) or resources (such as local storage). There have been no principles or invariants that a new access control policy must follow or maintain. Consequently, numerous incoherencies in browsers' access control policies exist, presenting hurdles for web programmers to build robust web applications.

In this paper, we examine the current state of browser access control policies, uncover and analyze the incoherencies in these policies, and measure the cost of eliminating them in today's web.

An access control policy configures how a principal accesses certain resources. This involves defining how principals are identified, how resources are labeled with principal IDs, and how these labels may be changed and handled at runtime. Unfortunately, browsers often mismanage principals, resulting in access control inconsistencies. We focus on three major sources of these problems: inconsistent principal labeling, inappropriate handling of principal label changes, and disregard of the user principal.

Inconsistent principal labeling. Today's browsers do not have the same principal definition for all browser resources (which include the Document Object Model (DOM), network, cookies, other persistent state, and display). For example, for the DOM (memory) resource, a principal is labeled by the origin defined in the same origin policy (SOP) in the form of <protocol, domain, port> [4]; but for the cookie resource, a principal is labeled by <domain, path>. Different principal definitions for two resources are benign as long as the two resources do not interplay with each other. However, when they do, incoherencies arise. For example, when cookies became accessible through DOM's "document" object, DOM's access control policy, namely the SOP, undermines some of cookie's access control policies (Section II-C1 gives a more detailed analysis).

Inappropriate handling of principal label changes. A web application is allowed to change its principal's label at runtime through the use of the `document.domain` DOM property. Nevertheless, the access control state is often kept static and such "effective" principal IDs set by `document.domain` are disregarded. This leads to access control incoherencies.

Disregard of the user principal. In this paper, we introduce the concept of a *user principal* in the browser setting. The user principal represents the user of a browser. Sometimes, the user principal is disregarded in existing browsers' access control policies. Certain resources should belong to the user principal *exclusively*. They include the user-private state such as clipboard and geolocation, user actions like navigating back and forward in browsing history, and a browser's UI including the current tab. These resources should not be accessible by web applications without user permission; otherwise, a web site could impersonate the user and violate user privacy. Unfortunately, today's DOM APIs expose some of these resources to web applications.

To systematically analyze and uncover the incoherencies created by these three problem areas, we have devised a set of coherency principles and constructed tests to check major browsers (including Internet Explorer, Firefox, and Google Chrome) for violations of these principles and to uncover the incoherencies that ensue.

A major goal of our work is to evaluate the compatibility cost of removing unsafe browser features that contribute to

the incoherencies. To this end, we have built *WebAnalyzer*, a scalable, crawler-based browser-feature measurement framework that can inspect a large number of web pages by rendering them in instrumented browsers. WebAnalyzer captures the DOM interactions of a page by interposing between the JavaScript engine and the DOM renderer, captures the protocol-level behavior through an HTTP proxy, and analyzes the visual appearance of a page by extracting its page layout.

Armed with WebAnalyzer, we have conducted measurements on the prevalence of unsafe browser features over the most popular 100,000 web sites as ranked by Alexa [5]. Our results pinpoint some unsafe features that have little backward compatibility cost and are thus possible to remove from current browsers without breaking many sites. For example, we find that most APIs controlling user-owned resources, descendant navigation, and incoherencies in XMLHttpRequest's principal labeling have low compatibility costs, whereas a substantial number of sites depend on "dangerous" functionality provided by `document.domain` or transparent cross-origin overlapping frames. Overall, we believe that by estimating the prevalence of unsafe features on the web, our measurements can guide future browsers to make better security vs. functionality trade-offs.

In summary, this work makes the following contributions:

- A systematic, principal-driven analysis of access control incoherencies in today's browsers.
- Introduction of the user principal concept for the browser setting.
- A comprehensive, extensible compatibility measurement framework.
- The first large-scale measurements on the compatibility cost of coherent access control policies.

The rest of the paper is organized as follows. Section II presents our systematic analysis of today's browser access control policies and enumerates access control incoherencies. Section III discusses our measurement motivation, tools, and infrastructure. Section IV presents our measurement results and gives recommendations on which unsafe policies can be eliminated with acceptable compatibility cost. Section V discusses limitations of our approach, Section VI presents related work, and Section VII concludes.

II. An analysis of browser access control incoherencies

In this section, we present our systematic analysis of today's browser access control policies and enumerate their incoherencies.

A. Methodology

For a systematic analysis, we establish the following access control coherency principles to guide our search for incoherencies:

1) Each shared browser resource, i.e. a resource shared among multiple principals, should have a principal definition (labeling of principals that share the resource) and have an access control policy.
2) For each non-shared browser resource that is explicitly owned by a single principal, the resource should have an owner principal with a specific label or be globally accessible.
3) When two resources interplay, both resources should have the same principal definition.
   This is because when two resources have different ways of labeling principals and when they interplay, their respective access control policies can be in conflict.
4) All access control policies must consider the runtime label of the principals, namely, the "effective" principal ID.
5) The user principal's resources should not be accessible by web applications.
   This is because when the user principal's resources are accessible by web applications, the user's privacy may be compromised or a web application could act on the user's behalf without the user's knowledge.

We look for violations of these principles and check for incoherencies when violations take place. The pseudocode below illustrates our manual analysis process.

```
0 foreach (browser resources) {
1   if exists (access control) {
2     if !considers (effective principal ID)
3       check improper principal ID changes
4   } else
5     check if lack of policy is appropriate
6 }
7
8 foreach (pairs of resources) {
9   if (they interplay &&
10        the principal/owner labeling differs)
11     check resource interplay incoherencies
12}
```

For each resource, we check whether it has an access control policy. If not, we check whether the lack of policy is appropriate (line 5, for example, Section II-E illustrates on how some resources that belong to the user principal lack access control considerations). If yes, we further check whether the access control policy considers the effective principal ID that sites can change dynamically at render-time. If it does not, then we check for incoherencies there (line 3, Section II-D).

In addition, we go through all pairs of resources; if they interplay and if they have the different principal definitions, we check for incoherencies (line 11, Section II-C). Careful

Shared resources	Principal definition
DOM objects	SOP origin
cookie	domain/path
localStorage	SOP origin
sessionStorage	SOP origin
display	SOP origin and dual ownership *

Table I
SHARED BROWSER RESOURCES AND THEIR RESPECTIVE PRINCIPAL DEFINITIONS. *DISPLAY ACCESS CONTROL IS NOT WELL-DEFINED IN TODAY'S BROWSERS.

Non-shared resources	Owner
XMLHttpRequest	SOP origin
postMessage	SOP origin
clipboard	user*
browser history	user*
geolocation	user

Table II
NON-SHARED BROWSER RESOURCES AND THEIR RESPECTIVE OWNER PRINCIPAL. *ACCESS CONTROL IS NOT WELL-DEFINED IN TODAY'S BROWSERS.

readers may wonder what happens to the interplay of more than two resources. Coherency in this context is a transitive property. That is, if a Resource 1 and Resource 2's access control policies are coherent (namely have the same principal definitions) and that of Resource 2 and Resource 3 are coherent, then the access control policies of Resource 1 and Resource 3 are also coherent since their principal definitions should also be the same.

The enumeration of resources is done by manually browsing through IE's source code (more in Section II-B). Our incoherency checks are done through test programs on major browser versions.

Despite our effort to be comprehensive, it is possible that we miss some browser resources or miss some interplays among the resources. We hope our work to be a start for a community effort on mapping out the full set of browser access control policies.

B. Browser resources

In this section, we enumerate all types of browser resources. A browser resource may be shared among (some definition of) principals or may not be shared and is explicitly owned by some principal. Table I shows the shared resources and their respective principal definitions. Table II shows non-shared resources and their respective owners. We now describe each resource, their principal or owner definition, and its access control policy in turn.

A *DOM object* is a memory resource shared among principals labeled with SOP origins, namely, <protocol, domain, port>. The access control policy of DOM objects is governed by SOP [4], which mandates that two documents from different origins cannot access each other's HTML documents using the Document Object Model (DOM), which is the platform- and language-neutral interface that allows scripts to dynamically access and update the content, structure and style of a document [6].

A *cookie* is a persistent state resource. The browser ensures that a site can only set its own cookie and that a cookie is attached only to HTTP requests to that site. By default, the principal is labeled with the host name and path, but without the protocol and the port number [7], [8], unlike SOP origins. For example, if the page `a.com/dir/1.html` creates a cookie, then that cookie is accessible to `a.com/dir/2.html` and other pages from that `dir/` directory and its subdirectories, but is not accessible to `a.com/`. Furthermore, `https://a.com/` and `http://a.com/` share the cookie store unless a cookie is marked with a "secure" flag. Non-HTTPS sites can still set "secure" cookies in some implementations, but cannot read them back [9]–[11]. A web programmer can make cookie access less restrictive by setting a cookie's `domain` attribute to a postfix domain or the path name to be a prefix path.

Local storage is the persistent client-side storage shared among principals defined by SOP origins [12].

Session storage is storage for a tab [12]. Each tab has a unique set of session storage areas, one for each SOP origin. The *sessionStorage* values are not shared between tabs. The lifetime of this storage is the same as that of the tab.

Display does not have a well-specified access control policy in today's browsers and standards (corresponding to line 5 in our pseudocode). Our earlier work Gazelle [3] specified an access control policy for display (and Gazelle further advocated that this policy be enforced by the browser kernel, unlike existing browsers). In Gazelle's model, a web site principal delegates its display area to another principal in the form of cross-domain iframes (or objects, images). Such an iframe (window) is co-owned by both the host page's principal, called landlord, and the nested page's principal, called tenant (both labeled with SOP origins). Principals other than the landlord and the tenant have no access permissions for the window. For the top-level window, the user principal owns it and plays the role of its landlord. Gazelle's policy further specifies how landlord and tenant should access the four attributes of a window, namely the position, dimensions, pixels, and URL location. This specification guarantees that the tenant cannot interfere with the landlord's display, and that the tenant's pixels, DOM objects, and navigation history are private to the tenant. Gazelle's policy is coherent with SOP. In Table III, we summarized the access control matrix for Gazelle, IE 8, Firefox 3.5, and Chrome 2. The access control of the URL location attribute corresponds to the navigation policy of a browser. Descendant navigation policy allows navigating a descendant window regardless of its origin; this was advocated and implemented over several browsers [13]. Gazelle's policy is *child navigation*

	Landlord			Tenant		
	Gazelle	IE	FF/Chrome	Gazelle	IE	FF/Chrome
position (x,y,z)	RW	RW	RW		RW	
dimensions (height, width)	RW	RW	RW	R	RW	R
pixels		W*	W*	RW	RW	RW
URL location	W	W	RW*	RW	RW	RW

Table III
ACCESS CONTROL POLICY FOR A WINDOW'S LANDLORD AND TENANT (BEING A DIFFERENT PRINCIPAL FROM THE LANDLORD) ON GAZELLE, IE 8, FIREFOX 3.5, AND CHROME. RW*: THE URL IS READABLE ONLY IF THE LANDLORD SETS IT. IF THE TENANT NAVIGATES TO ANOTHER PAGE, LANDLORD WILL NOT SEE THE NEW URL. W*: THE LANDLORD CAN WRITE PIXELS WHEN THE TENANT IS TRANSPARENTLY OVERLAID ON THE LANDLORD.

policy. (We elaborate in Section II-C3 that the descendant navigation policy is at conflict with DOM's SOP.) Our tests indicate that Firefox 3.5 and Chrome 2 currently support the child policy, while IE 8 supports the descendant policy. All major browsers allow any window to navigate the top-level window, while Gazelle only allows top-level window navigation from the top-level window's tenant and the user.

XMLHttpRequest allows a web site principal to use scripts to access its document origin's remote data store by issuing an asynchronous or synchronous HTTP request to the remote server [14]. XMLHttpRequest2 [15] and XDomainRequest have been recently proposed and implemented in major browsers to allow cross-origin communications with remote servers, where HTTP authentication data and cookies are not sent by default. These networking capabilities are not shared and strictly belongs to a web site principal labeled with a SOP origin.

PostMessage is a recently proposed client-side cross-origin communication mechanism that is now implemented in all major browsers. This is also a web site principal's capability which is not shared with any other principals.

The last three resources in the non-shared resource table, namely clipboard, browser history, and geolocation, all belong to the user principal, and web applications should not be able to access them directly. However, they are all accessible by scripts through the DOM API, causing problems that we describe in Section II-E.

C. The interplay of the resources

From the enumeration of the resources and their respective principal or owner definition in the above section, we derived the following problematic pairs of resources, where the two resources interplay and their principal or owner definitions differ: DOM-cookie, cookie-XMLHttpRequest, and DOM-display. We elaborate on these interplays below.

1) DOM and Cookies: DOM and cookies interplay because scripts are able to create or modify cookies by using the `document.cookie` property in the DOM API.

With no protocol in cookie's principal definition, cookies are vulnerable to information leaks. A cookie intended for a secure HTTPS principal can be passed over HTTP and be exposed to network attackers. This can be prevented

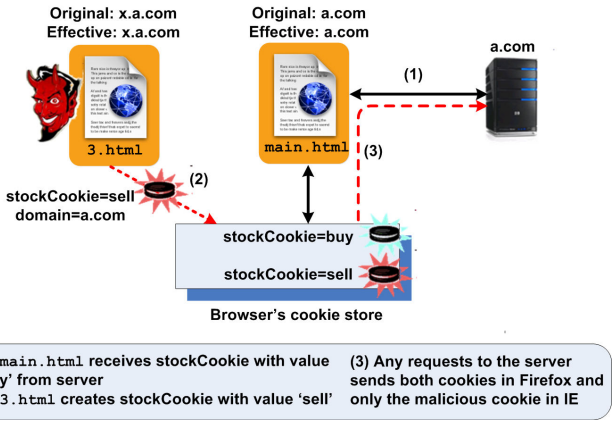

Figure 1. Incoherency arises from the interplay between the access control policies of DOM and cookies

by setting the cookie with the "secure" flag. However, a "secure" cookie can still be set by an HTTP response and be accessed by scripts belonging to an HTTP page as long as their domains are the same. Additionally, different services running on different ports of the same domain can access each other's cookies. Moreover, the path protection of cookies becomes ineffective as a script from a different path can access the cookie based on SOP.

The interplay between DOM and cookies also allows the scripts to set the effective domain of a cookie to any suffix of the original domain by setting the *domain* attribute of the cookie. This can lead to inconsistencies in the current browsers. Figure 1 shows a scenario in which such inconsistencies lead to an undefined behavior in the browsers. In this example, a cookie named "stockCookie" with value "buy" is stored in the cookie store for the domain `a.com`. A script injected into a compromised page belonging to `x.a.com` can create another cookie with the same name but with a different value "sell" while setting its domain attribute to `a.com`.

While this leads to a compromised state in the current browsers, different browsers deviate in their behavior creating further inconsistencies in the web applications supporting multiple browsers. Firefox 3 sets this cookie with

a domain value of `.a.com` resulting in multiple cookies with the same name in browser's cookie store. The browser attaches both cookies (genuine cookie with domain `a.com` and evil cookie with domain `.a.com`) to any server requests to `a.com`. The server only receives the cookie's name-value pair without any information about its corresponding domain. This results in the server receiving two cookies with the same name. Since server-side behavior is not defined in case of duplicate cookies [9], it leads to inconsistent state at `a.com`'s server. In case of IE 8, the original cookie value is overwritten and only the wrong cookie value is received by the server.

2) Cookies and XMLHttpRequest: Cookies and XMLHttpRequest interplay because XMLHttpRequest can set cookie values by manipulating HTTP headers through scripts. XMLHttpRequest's owner principal is labeled by the SOP origin, while cookie has a different principal definition (Section II-B).

If a server flags a cookie as "HttpOnly", the browser prevents any script from accessing (both reading and writing) the cookie using the `document.cookie` property. This effectively prevents cookies being leaked to unintended parties via cross-site scripting attacks [16].

The purpose of HttpOnly cookies is that such cookies should not be touched by client-side scripts. However, XMLHttpRequests are created and invoked by client-side JavaScript code, and certain methods of the XMLHttpRequest object facilitate access to cookies: `getResponseHeader` and `getAllResponseHeaders` allow reading of the "Set-cookie" header, and this header includes the value of HttpOnly cookies. Another method, `setRequestHeader`, enables modification of this header to allow writing to HttpOnly cookies.

Some of the latest browsers have tried to resolve this issue with varied success. IE 8 currently prevents both read and write to cookies via "Set-cookie" header, but still allows access via "Set-cookie2" header [17]. Firefox has also recognized and fixed the issue for cookie reads: their fix prevents XMLHttpRequest from accessing cookie headers of any response, whether or not the HttpOnly flag was set for those cookies [18]. This is a bold step taken by Firefox, as our results show that a considerable number of web pages still read cookie headers from XMLHttpRequest (Section IV). However, we have still observed the writing issue with HttpOnly cookies using Firefox 3.5. A script can set a cookie with the same name as the HttpOnly cookie and can have a different value set using the `setRequestHeader` method. This results in a duplicate cookie being sent to the server, thus creating an inconsistent state on the server side.

3) DOM and Display: One incoherence takes place on URL location of a window. The descendant navigation policy (Section II-B) is at conflict with DOM's SOP. Descendant navigation policy allows a landlord to navigate a window, a resource created by its descendant through a DOM API, even if the landlord and the descendant are different principals. This gives a malicious landlord more powerful ways to manipulate a nested, legitimate sites than just overdrawing: with overdrawing, a malicious landlord can imitate a tenant's content, but the landlord cannot send messages to the tenant's backend in the name of the tenant. As an example attack, imagine that an attacker site nests a legitimate trading site as its tenant. The trading site further nests an advisory site and uses a script to interact with the advisory window to issue trades to the trading site backend (e.g., making a particular trade based on the advisory's recommendation shown in the URL fragment). With just one line of JavaScript, the attacker could navigate the advisory window (which is a descendant) and create unintended trades.

Another conflict lies in the access control on the pixels of a window. DOM objects are ultimately rendered into the pixels on the screen. SOP demands non-interference between the DOM objects of different origins. However, existing browsers allow intermingling the landlord's and tenant's pixels by overlaying transparent tenant iframes on the landlord, deviating from the non-interference goal of SOP. This enables an easy form of clickjacking attacks [19]. In contrast, Gazelle advocates cross-principal pixel isolation in accordance with SOP (Table III, row "pixels").

D. Effective Principal ID

Browsers allow cross-principal sharing for "related" sites by allowing sites to change their principal ID via the `document.domain` property [4]. This property can be set to suffixes of a page's domain to allow sharing of pages across frames. For example, a page in one frame from `x.a.com` and a page from `www.a.com` initially cannot communicate with each other due to SOP restrictions. This is one of the few methods for cross-origin frames to communicate before the advent of postMessage [20]. However, changing `document.domain` violates the principle of least privilege: once a subdomain sets its domain to its suffix, there is no control over which other subdomains can access it.

Furthermore, almost no existing access control policies of today's browsers take such "effective" principal IDs into consideration. In the following subsections, we examine how the disregard of effective principal IDs leads to dual identities and incoherencies exploitable by attackers. In our attack model, an attacker owns a subdomain (through third-party content hosting as in iGoogle or by exploiting a site vulnerability). As we will show in the following sections, the attacker can leverage `document.domain` to penetrate the base domain and its other subdomains.

1) Cookie: Any change of origin using `document.domain` only modifies the effective principal ID for DOM access and does not impact the domain for

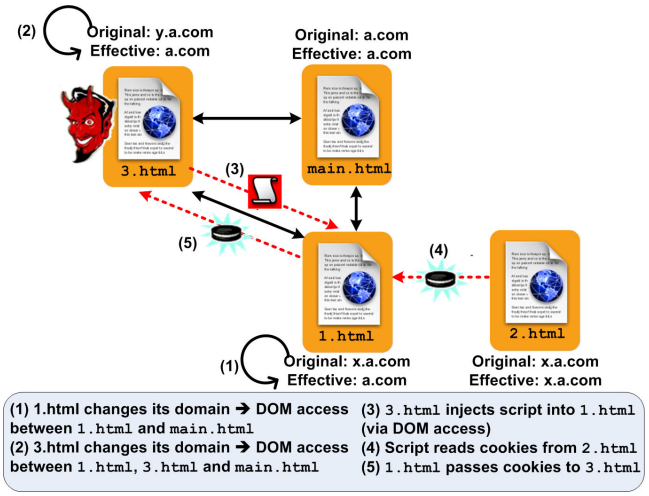

Figure 2. Lack of effective principal ID consideration in cookie's access control policy

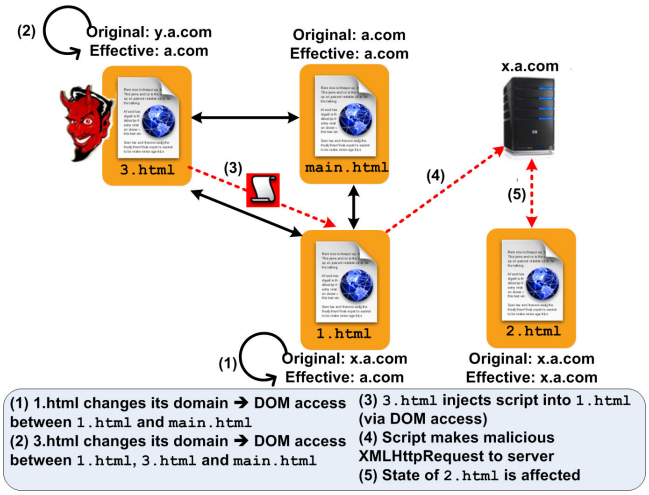

Figure 3. Lack of effective principal ID consideration in XMLHttpRequest's access control policy

cookie access. Figure 2 shows an attack to exploit this inconsistent behavior of browser policy design. In this scenario, a page `1.html` in domain `x.a.com` changes it effective domain to `a.com`. As a result, it can access the DOM properties of other pages belonging to `a.com`, but it can no longer access the pages of its original domain `x.a.com`. However, since the effective domain does not change for cookie access, the page still maintains access to the cookies belonging to its original domain. This inconsistent dual identity possessed by the page acts as a bridge to access cookies from both the original domain and the effective domain.

In order to launch the attack, an attacker (after owning a subdomain page) first assumes the identity of `a.com` and subsequently injects a script into the page `1.html`. This injected script can now read and write the cookies belonging to `x.a.com` including any cookies created later. Effectively, if the attacker can compromise a page in one of the subdomains, he can access the cookies of any other subdomains that change their effective origin to the base domain.

2) XMLHttpRequest: Change of origin for scripts does not change the effective principal ID for XMLHttpRequest usage. This enables a (malicious) script in a (compromised) subdomain to issue XMLHttpRequest to the servers belonging to the base domain and its other subdomains. The attack scenario is illustrated in Figure 3. Page `1.html` has changed its effective domain value to `a.com` from the original value of `x.a.com`. With no effect on XMLHttpRequest usage, scripts in `1.html` can still make requests to the server belonging to `x.a.com`. This again gives a script a dual identity – one for DOM access (`a.com`) and another for XMLHttpRequest (`x.a.com`). Therefore, an attacker compromising any subdomain can inject a script into `1.html` via DOM access, and this script can then make XMLHttpRequest calls to the original domain of the page. Since a well-crafted XMLHttpRequest can change the server-side state for the web application, and this state might be shared between other pages within the domain `x.a.com`, such attack can possibly impact all pages belonging to `x.a.com`.

3) postMessage: `postMessage` also ignores any `document.domain` changes: if `x.a.com` changes domain to `a.com` and sends a message to `y.b.com`, `y.b.com` still sees the message's origin as `x.a.com`. Also, if `y.b.com` changes its domain to `b.com`, `x.a.com` still has to address messages to `y.b.com` for them to be delivered. This gives the attacker (with a compromised subdomain) an opportunity to send messages while masquerading under the identity of another subdomain (Figure 4).

4) Storage: Based on our tests, IE 8 does not take any `document.domain` changes into consideration for both local storage and session storage. Firefox 3.5 also ignores effective principal ID for local storage. However, for session storage, any domain changes via `document.domain` are considered: the old session storage is lost for the original domain and a new session storage is created for the effective principal.

Inconsistency arises when `document.domain` changes are ignored (for both session storage and local storage in IE; for only local storage in Firefox). An attacker (being able to inject a script into one of the pages of any subdomain, say `x.a.com`) can change its origin to the base domain `a.com` and can successfully inject a script into the DOM of the base domain or any other origins (e.g., `y.a.com`) that change identity to the base domain. Since access control checks on storage rely on original domain (i.e., `y.a.com`), the malicious script can now freely access the storage belonging

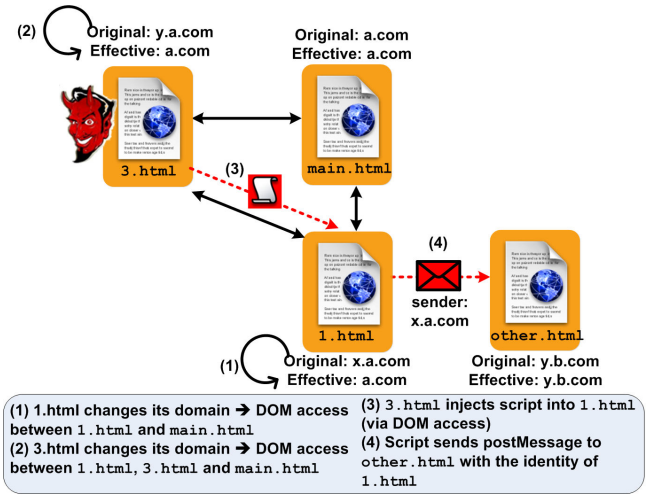

Figure 4. Lack of effective principal ID consideration in postMessage

to `y.a.com`.

E. The User Principal

In this paper, we introduce the concept of the *user principal* in the browser setting. The user principal represents the user of the browser. Unfortunately, it has often been neglected in browser access control policies.

While a web application does manage the user's data and experience for that particular application (e.g., a user's banking data at a banking site), certain browser resources or data belong to the user exclusively and should not be accessible by any web site without user permissions. Such resources include: user's private data, such as clipboard data and geolocation; user actions, such as clicking on the forward and back button; devices, such as camera and microphone; and browser UI, including the current tab window (top-level window).

Unfortunately, in today's browsers, some of these resources are directly exposed to web applications through the DOM API. This breaks the fundamental rule of protecting resources belonging to different principals from one another, as the user principal's resources can be accessed and manipulated by site principals. This can result in privacy compromises, information leaks, and attacks that trick users into performing unintended actions. In this section, we examine the user principal resources and describe our findings on how they may be accessed improperly by web applications.

1) User actions: The `focus` and `blur` properties of the `window` object allow web sites to change focus between the windows that they opened irrespective of the origins. This enables an attacker site to steal focus or cause the user to act on a window unintentionally.

The `window` object has a `history` property with an array of user-visited URLs. Browsers have been denying any site's access to this array to protect user privacy, but they do allow a site to navigate the browser back and forward in history through the `back()` and `forward()` methods [8]. Worse, our tests indicate that Firefox 3 and Google Chrome 2 allow any child window to navigate the top-level window back or forward in history *irrespective* of the origin. In many cases this is just a nuisance, but some properly-crafted history navigation by a malicious application can lead to more severe damage. For example, the user might be tricked to make multiple purchases of the same product.

We have also investigated synthetic event creation. The DOM API allows a site to generate synthetic mouse or keyboard events through the `document.createEvent()` method (or `document.createEventObject()` in IE). In IE, a programmer could directly invoke a `click()` method on any HTML element to simulate user clicks. These techniques are useful for debugging purposes. To our delight, all major browsers are careful not to let a web site to manipulate another site's user experience with these synthetic user events. Note that it is benign for a site to simulate the user's actions for itself, since loading and rendering site content can by itself achieve any effects of simulating user actions (e.g., simulating a mouse click is equivalent of calling the `onclick` function on the corresponding element).

2) Browser UI: An important part of the browser UI is the current tab window, or top-level window. In today's browsers, any web site loaded in any window is able to reposition and resize a top-level window through the `moveTo`, `moveBy`, `resizeTo`, and `resizeBy` properties of the top-level window. Resizing the currently active top-level window effectively resizes the browser window. Firefox 3 allows an application to resize a browser window even in the presence of multiple tabs, while IE 8 and Chrome 2 do not allow this. A site can also open and close a top-level window using `open` and `close` methods. The use of `open` method has been mitigated through built-in popup blockers. IE 8 allows any frame to close a top-level window irrespective of the origin, while Firefox 3 and Chrome 2 prevent this from happening. These capabilities allow an attacker site (even when deeply nested in the DOM hierarchy, say a malicious ad) to directly interfere with the user's experience with the browser UI.

Some of the other loopholes in browser UI have already been fixed. For example, the status bar can no longer be set by a web site.

3) User-private state: Jackson et al. have shown that a user's browsing history can be exposed by inspecting the color of a visited hyperlink [21], raising privacy concerns. The hyperlink's color is intended for the user, and it is not necessary for web sites to be able to read it.

The clipboard data also belongs exclusively to the user principal. All versions of IE since 5.0 support

APIs to access clipboard data. A web site can get contents of a user's clipboard by successfully calling `window.clipboardData.getData("Text")`. Depending on the default Internet security settings, the browser may prompt user before getting the data. However, the prompt does not identify the principal making the request (simply using the term "this site"). As a result, a malicious script embedded on a third-party frame may trick the user into giving away his clipboard because he thinks that such access is being requested by the trusted top-level site.

Geolocation is one of the latest browser features that allows a site to determine the client's location by using the `navigator.geolocation` [12] interface. At the time of writing, Firefox 3.5 is the only stable production browser supporting this HTML5 feature. Geolocation is user-private data. Today's browsers do ask user permission before accessing it. However, issues arise when a site embeds content from multiple principals (i.e., in frames), and more than one origin needs access to geolocation information. The geolocation dialog is active for only one origin at a time; if there is a request to access geolocation from `b.com` while the dialog for `a.com` is still active, it is ignored — the principal that succeeds in invoking the geolocation dialog first wins. Therefore, if a malicious party manages to embed a script (or a frame) on the page, it can prevent the main site from triggering the geolocation dialog by invoking the dialog first. As a result, the malicious party can create denial-of-service against the main site, preventing it from retrieving a user's geolocation information. Additionally, it could trick the user into giving away location to itself rather than the main site (e.g., using phishing domain names like `www.gooogle.com`).

Changing `document.domain` also generates inconsistencies. The geolocation prompt is designed to work only with the original principals, and even if a site changes identity, the prompt still displays the original domain as the requesting domain. For an example site `good.a.com` that changes its document.domain to `a.com`, this causes the following problems:

- If an attacker site `evil.a.com` changes its document.domain to `a.com`, it can steal position information from `good.a.com`, if `good.a.com` has stored or displayed this information in a place that is accessible via the DOM (e.g., using `parent.document.getElementById("coords").innerHTML`).
- If another site `evil.a.com` also changes its domain to `a.com`, it could impersonate `good.a.com`, by using `parent.navigator.geolocation.getCurrentPosition`, which would trigger the access prompt using `good.a.com`, instead of `evil.a.com`.

III. The WebAnalyzer Measurement Framework

To achieve consistent browser access control policies, browser vendors need to remove or modify the features that contribute to incoherencies. For example, disallowing domain-setting for cookies, eliminating `document.domain`, and removing support for accessing user principal resources are steps towards secure new browsers. However, this begs the question of what the cost of these feature removals is and how many web sites will break as a result. In today's highly competitive browser market, backward compatibility with the existing web is paramount.

To help browser vendors balance security and compatibility, we set off to build a measurement system to measure the cost of security. Many previous web compatibility studies have been *browser-centric*: they have evaluated the degree to which a given browser supports various web standards or is vulnerable to attacks [22], [23]. In contrast, we take a *web-centric* perspective and actively crawl the web to look for prevalence of unsafe browser features on existing web pages. Compared to existing crawlers, however, static web page inspection is insufficient. Dynamic features such as AJAX or post-render script events require us to actively render a web page to analyze its behavior at run time. Moreover, the incoherencies we identified in Section II require analysis of not just a page's JavaScript execution [24], but also DOM interactions, display layout, and protocol-layer data.

To address these challenges, we have constructed a scalable, execution-based crawling platform, called WebAnalyzer, that can inspect a large number of web pages by rendering them in an instrumented browser. The platform consumes a list of URLs (defined by a human operator or generated by a traditional web crawler), and distributes them among virtual machine workers, which renders them using IE_{WA}, a specially instrumented version of Internet Explorer. IE_{WA} provides dynamic mediation for all browser resources, and detects when a resource invocation matches one of preset policy rules. Even though our framework is extensible to a large variety of browser policies, we concentrate on "unsafe feature" rules derived from our analysis in Section II.

To build IE_{WA}, the central piece of our measurement platform, we leverage public COM interfaces and extensibility APIs exported by Internet Explorer 8. Figure 5 shows the architecture of IE_{WA}, which centers around three major interposition modules: (1) a script engine proxy, which provides JavaScript and DOM interposition, (2) a network proxy based on Fiddler [25], and (3) display dumper, which enables custom analysis of a page's layout as it is visible to the user. Next, we discuss each module in turn.

Script engine proxy. We build on our earlier system in MashupOS [1] to implement a JavaScript engine proxy (called *script engine proxy* (SEP)): SEP is installed between IE's rendering and script engines, and it mediates and customizes DOM object interactions. SEP exports the

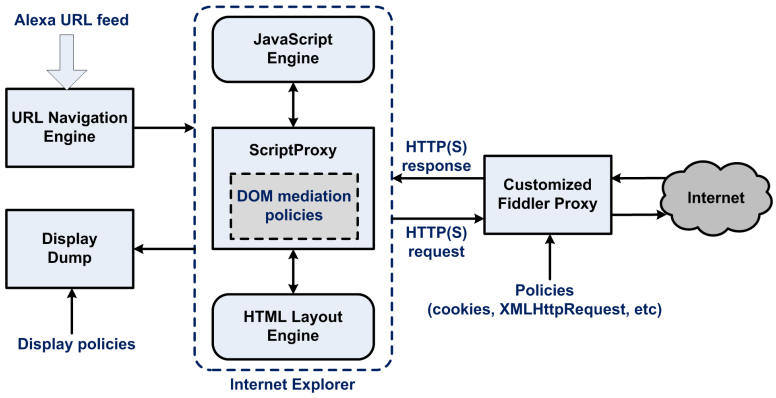

Figure 5. High-Level Architecture of IE_{WA}.

script engine API to IE's renderer, and it exports the DOM and rendering interfaces to IE's script engine. Each DOM object is interposed by a corresponding object wrapper. When IE's script engine asks for a DOM object from the rendering engine, SEP intercepts the request, retrieves the corresponding DOM object, associates the DOM object with its wrapper object inside SEP, and then passes the wrapper object back to the original script engine. Any subsequent invocation of wrapper object methods from the original script engine passes through SEP. SEP is implemented as a COM object and is installed into IE by modifying IE's JavaScript engine ID in the Windows registry.

Network interposition. In addition to SEP, we route the browser's network traffic through a proxy to monitor all HTTP/HTTPS requests and analyze cookie transfers as well as network APIs like XMLHttpRequest. Our network proxy is implemented using the *FiddlerCore* interfaces provided by the public-domain Fiddler web debugging proxy [25], [26].

Display analysis. In order to evaluate display policies, it is necessary to analyze a browser's visual output as seen by the user. For this purpose, we use a customized version of IE's rendering engine that exposes COM interfaces to extract a textual representation of a particular page's visual layout at any stage of rendering. In our current evaluation, we use these COM interfaces to save a snapshot log of IE's display after a page has fully loaded. Because some pages have post-render events that alter layout, we wait an additional 5 seconds before taking a display snapshot. Snapshot logs provide a mapping between a page's objects and their layout properties, such as position, dimensions, or transparency. They can be analyzed offline for the presence of unsafe frame overlapping behavior or other dangerous page layouts.

Navigation. To facilitate automatic analysis for a large number of URLs, IE_{WA} includes a URL navigation engine, which utilizes IE's extensibility interfaces, such as `IWebBrowser2`, to completely automate the browser's navigation. In addition to pointing the browser to new URLs, this module also cleans up state such as pop-ups between consecutive URLs, detects when sites fail to render (e.g., 404 errors), and recovers from any browser crashes.

Visiting a site's home page is sometimes insufficient to invoke the site's core functionality. For example, a feature may be accessed only when the user clicks on a link, types search queries, or causes mouse event handlers to run.

It is difficult and time-consuming to fully automate a site's analysis to study all possible features and pages that could be invoked using all combinations of user input. Instead of aiming for complete coverage within a particular site, we enhanced our navigation engine with simple heuristics that simulate some user interaction. After rendering a site's home page, IE_{WA} will find and simulate a click on at most five random links, producing five random navigation events. In addition, IE_{WA} will check for presence of a search form, fill it with random keywords, and submit it. We restrict all simulated navigations to stay within the same origin as a site's home page.

These simple enhancements maintain our ability to examine a large number of sites while adding the ability to properly handle many (but not all) sites with home pages that do not invoke the site's main functionality. For example, we can navigate to a random article on Wikipedia, a random video on YouTube, a random profile on MySpace, a random Twitter feed, and a random search query on Google. We evaluate the success of this methodology against a user-driven browsing study in Section IV-G and discuss its limitations in Section V.

Performance. We deployed our system on several desktop machines, each with an Intel 2.4 GHz quad-core CPU and 4 GB of RAM. Our IE_{WA} workers run inside a Windows Vista VMware virtual machine to prevent malware infection. We executed multiple workers in each VM, isolating them from one another using different UIDs and different remote desktop sessions.

On such a setup, one IE_{WA} worker is able to analyze about 115 typical web sites per hour. Each site's processing time includes the home page, five random link clicks, and one

form submission, as well as overheads introduced by IE_{WA}'s three interposition modules. We found that we could execute up to eight parallel workers in one VM, for a throughput of 900 sites per VM, before saturating the CPU. Optimizing this infrastructure for performance was not a goal of this paper and is left as future work.

IV. Experimental Results

Our analysis in Section II provides an understanding of the security characteristics of the current access control policies in browsers. In this section, we complete the other half of the equilibrium by using the measurement infrastructure presented in Section III to study the prevalence of unsafe browser features (analyzed in Section II) on a large set of popular web sites. By presenting both sides, we enable the browser vendors to make more informed decisions about whether or not to continue supporting a particular unsafe feature based on its real-world usage.

A. Experimental overview

1) Choosing the sites for analysis: Instead of randomly crawling the web and looking for unsafe features, we decided to focus our attention on the "interesting" parts of the web that people tend to visit often. Accordingly, to seed our analysis, we take the set of 100,000 most popular web sites ranked by *Alexa* [5], as seen on November 9, 2009, as our representative data set. The data collection and analysis were completed in the last week of February 2010.

2) Defining the compatibility cost: We define the cost of removing a feature to be the number of Alexa-ranked, top 100,000 sites that use the feature.

We conservatively assume that disallowing a feature will significantly hinder a site's functionality, whereas it could simply cause a visual nuisance. A more detailed analysis on the effect of policy changes on page behavior is promising but is left as future work.

3) High-level results: We obtained our results by rendering each of the 100,000 seed links using WebAnalyzer, saving all interposition logs for offline analysis. This way, we were able to obtain data for 89,222 of the 100,000 sites. There are several reasons why no data was produced for the rest of sites. First, some sites could not be accessed at the time of our analysis due to failed DNS lookups, "404 Not Found" errors, and other similar access problems. Second, some sites timed out within our chosen threshold interval of 2 minutes, due to their slow or continuous rendering. We decided to drop any such sites from our analysis. Finally, some sites did not contain any JavaScript code, and as a result they did not trigger our event filters. Nonetheless, we believe that we have been able to analyze a sufficiently large set of sites with a reasonable success ratio, and our data set and the scope of measurement is much larger than that used by earlier related studies [24].

Tables IV, V, and VI present the results of our analysis, showing how frequently each feature we analyzed earlier is encountered. Next, we organize our findings according to our discussion in Section II and discuss their implications on future browser security policies.

B. The interplay of browser resources

1) DOM and Cookies: Cookie usage is extremely popular, and so is their programmatic DOM access via `document.cookie`, which we found on 81% web sites for reading and 76% of web sites for writing cookie values, respectively. The use of the cookie's `domain` attribute is also widespread (67% of sites), with about 46% of sites using it to actually change the domain value of the cookie. As a result, the issues described in Section II-C1 cannot be solved by simply deprecating the usage of this attribute and changing the principal definition of cookies. One possible approach to solve the inconsistency issue with cookie handling is to tag the cookie with the origin of the page setting the cookie. This information should be passed to the server to allow the server to differentiate between duplicate cookies.

Section II-C1 also identified inconsistencies pertaining to cookies and HTTP/HTTPS, which we now support with measurements. First, 0.07% of sites alarmingly send `secure` cookies over HTTP. This effectively tampers with the integrity of cookies that may have been intended for HTTPS sessions [10]. Fortunately, it appears that this functionality can be disallowed with little cost. Surprisingly, a much larger number of sites (5.48%) sent HTTP cookies over HTTPS. The HTTP cookies cannot be kept confidential and are accessible to HTTP sessions. Our recommended solution to this problem is that the "secure" flag should be enforced for any cookies passed over an HTTPS connection even if the web developer fails to set the flag. This would still enable the HTTPS site to access the cookie for its own functionality and any sharing with the HTTP site should be done explicitly.

We found a large number of sites (16.2%) using HttpOnly cookies, which is an encouraging sign — many sites appear to be tightening up their cookie usage to better resist XSS attacks.

2) Cookies and XMLHttpRequest: Our measurements show that the issues arising from undesirable interplay of `XMLHttpRequest` and HttpOnly cookies (Section II-C2) can possibly be eliminated, since very few sites (0.30%) manipulate cookie headers in XMLHttpRequest responses.

3) DOM and Display: Section II-C3 argued that the descendant navigation policy is at conflict with SOP for DOM. We observe iframe navigations on 7.7% of sites and all of them are child navigation (regardless of the origin). The absence of descendant navigation in the top 100,000 sites indicates a potentially very low cost to remove it.

Measurement Criteria	Total instances (count)	Unique sites	
		Count	Percentage
document.cookie (read)	5656310	72587	81.36%
document.cookie (write)	2313359	68230	76.47%
document.cookie domain usage (read)	2032522	59631	66.83%
document.cookie domain usage (write)	1226800	41327	46.32%
Secure cookies over HTTP	259	62	0.07%
Non-secure cookies over HTTPS	15589	4893	5.48%
Use of "HttpOnly" cookies	33180	14474	16.22%
Frequency of duplicate cookies	159755	4955	5.55%
Use of XMLHttpRequest	19717	4631	5.2%
Cookie read in response of XMLHttpRequest	1261	265	0.30%
Cross-origin descendant navigation (reading descendant's location)	6043	61	0.07%
Cross-origin descendant navigation (changing descendant's location)	0	0	0.00%
Child navigation (parent navigating direct child)	22572	6874	7.7%
document.domain (read)	1253274	63602	71.29%
document.domain (write)	8640	1693	1.90%
Use of cookies after change of effective domain	295960	1569	1.76%
Use of XMLHttpRequest after change of effective domain	225	87	0.10%
Use of postMessage after change of effective domain	0	0	0.00%
Use of localStorage after change of effective domain	42	10	0.01%
Use of local storage	1227	169	0.19%
Use of session storage	0	0	0.00%
Use of fragment identifier for communication	5192	3386	3.80%
Use of postMessage	6523	845	0.95%
Use of postMessage (with no specified target)	0	0	0.00%
Use of XDomainRequest	527	125	0.14%
Presence of JavaScript within CSS	224266	4508	5.05%

Table IV
USAGE OF VARIOUS BROWSER FEATURES ON POPULAR WEB SITES (FEBRUARY 2010). ANALYSIS INCLUDES 89,222 SITES.

Sites containing at least one `<iframe>`	36549 (40.8%)
Average number of `<iframe>`'s per site	3.2
Sites with at least one pair of overlapping frames	5544 (6.2%)
Sites with at least one pair of overlapping cross-origin frames	3786 (4.2%)
Sites with at least one pair of *transparent* overlapping frames	1616 (1.8%)
Sites with at least one pair of *transparent* overlapping cross-origin frames	1085 (1.2%)

Table V
SUMMARY OF DISPLAY LAYOUTS OBSERVED FOR THE TOP 100,000 ALEXA WEB SITES (DECEMBER 2009). 89,483 SITES WERE RENDERED SUCCESSFULLY AND ARE INCLUDED IN THIS ANALYSIS.

In addition, we have analyzed the visual layouts of all sites to determine whether there are dangerous pixel interplays between windows of different principals (Section II-C3). Our results are summarized in Table V[1]. We found that 41% of sites embed at least one `iframe`, and the average number of iframes embedded on a particular page is 3.2. Overlapping iframes appear to be common — 6.2% of sites contained at least one overlapping pair of iframes — but only 29% of these overlaps involved transparent iframes. Most (68%) overlapping scenarios involve different principals.

The most dangerous situations occur when a transparent frame is overlaid on top of a frame belonging to a different principal (Section II-C3). We identified 1,085 sites (1.2%) that contained at least one pair of *transparent*, cross-origin overlapping iframes. We observed that most of these overlaps involved domains serving ad banners, so the main site functionality might remain unaffected if the dangerous transparency is disallowed.

[1]Our display analysis was performed in December 2009, separately from script engine and network analysis that we performed in February 2010, causing a slight difference in the number of successfully rendered sites in Tables V and IV.

Measurement Criteria	Total instances (count)	Unique sites	
		Count	Percentage
Setting top-level window's location	55759	2851	3.20%
Change focus of window	5221	2314	2.59%
Reading color of hyperlinks	82587	1560	1.75%
Accessing browser's history	1910	721	0.81%
Use of `defaultStatus` (write)	1576	241	0.27%
Reading user's Geolocation	251	149	0.17%
Use of `resizeTo`	339	134	0.15%
Use of `defaultStatus` (read)	528	108	0.12%
Use of `moveTo`	258	100	0.11%
Close a window	130	86	0.10%
Access to user's clipboard	24	17	0.02%
Blur a window	54	13	0.01%
Use of `resizeBy`	13	8	0.01%
Use of `moveBy`	4	1	0.00%
Use of `outerWidth`	2	1	0.00%
Use of `outerHeight`	4	1	0.00%

Table VI
PREVALENCE OF RESOURCES BELONGING TO THE USER PRINCIPAL ON POPULAR WEB SITES. ANALYSIS INCLUDES 89,222 SITES.

Summary. We found that interplays between DOM and cookies have a high compatibility impact, while removing the interplays between cookies and XMLHttpRequest would affect only 0.30% of sites. For interplays related to display, we found that descendant navigation can be disallowed with no cost, while disallowing overlaps between transparent cross-origin frames would affect 1.2% of sites.

C. Changing effective Principal ID

In Section II-D, we showed that `document.domain` is an unsafe and undesirable part of today's web, as observed by others as well [9]. Unfortunately, we found its usage on the web to be significant: 1.9% of sites change their effective domain via `document.domain`.

We mentioned certain features which become incoherent when combined with `document.domain`. Cookies are accessed by about 1.76% of the sites after a change in effective domain, making it difficult to enforce a unified effective domain for cookie access (Section II-D1). Only 0.08% of sites use XMLHttpRequest after an effective UID change (Section II-D2), so it appears possible to make XMLHttpRequest respect effective domain with little cost. The same holds true for postMessage — we found no sites using postMessage after an effective UID change. The new local storage abstractions are not widespread — only 0.19% of the sites were using `localStorage` (0.01% after an effective domain change), and no sites were using `sessionStorage` — so we anticipate that origin-changing weaknesses that we outlined in Section II-D4 can be removed with little compatibility cost.

Summary. Overall, while disallowing `document.domain` completely carries a substantial cost (1.9% of sites), browsers can eliminate its impact on XMLHttpRequest, local storage, and postMessage at a much lower cost (0.19% of sites total). On the flip side, browser vendors have to make a much tougher choice (affecting 1.76% of sites) to prevent effective UID inconsistencies pertaining to cookies.

D. Resources belonging to the user principal

Table VI shows the results of our analysis for the cost of protecting user-owned resources discussed in Section II-E. The cost of tightening access control for user resources appears to be low with the exceptions of link-color access (1.8%), the focus-changing functions (2.6%), and setting top-level window location (3.2%).

Interestingly, 149 sites (0.17%) already use the new Geolocation primitives [12]. This number seems low enough for browsers to take actions to tighten its access control.

Overall, we found that 12 of the 16 user-principal APIs we examined can be removed while collectively affecting only 0.80% of unique sites.

E. Other noteworthy measurements

We measured prevalence of some primitives for cross-frame and cross-window communication, which are critical for cross-principal security. Fragment identifier messaging is most popular, being found at 3.8% of sites. A non-negligible number (0.95%) of sites have already adopted `postMessage`, and all sites use its newer definition that requires specifying the target window [13]. Another safer alternative for cross-domain communication, XDomainRequest, is also being slowly adopted (0.14%).

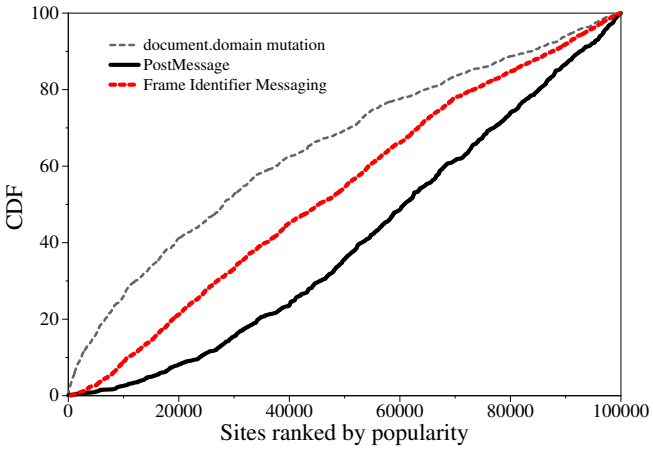

Figure 6. A CDF for prevalence of cross-frame communication mechanisms according to the ranking of sites that use them.

Using JavaScript within CSS has long been considered dangerous [9]. We found this pattern in use on about 5% of the sites.

F. Correlating unsafe features and site popularity

Next, we consider how the popularity of sites correlates with prevalence of unsafe features. A policy is more costly to correct if it is used by very highly ranked sites, since more people would visit them and encounter broken functionality. Fortunately, we found that most features do not exhibit a significant popularity bias, behaving uniformly with no regard to a site's popularity. Nevertheless, we found some exceptions. Figure 6 shows a CDF of the usage of various mechanisms that could be used for cross-frame communication according to the sites' ranking. Interestingly, fragment identifier messaging has little dependence on popularity, `document.domain` tends to be used more by higher-ranked sites, and `postMessage` is found more on lower-ranked sites, with very little use in the top 2000 sites. This went against our hypothesis that higher-ranked, high-profile sites would likely be written using the latest and safest web standards. A possible explanation could be that the top sites are motivated to use features compatible with the largest number of browsers and client platforms.

As another example, Figure 7 diagrams the prevalence of resources belonging to the user principal according to the ranking of the sites that use them (a dot is displayed for every site using a particular feature). Some features, such as `resizeBy` or clipboard access, are only found on very low-ranked sites and are thus good candidates to remove with little impact. Only a handful of features appear in the top 100 sites, where compatibility cost is very high for any site.

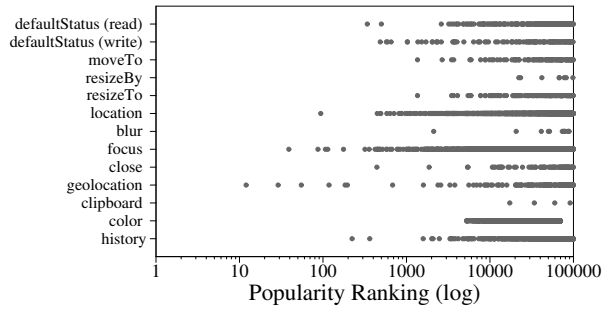

Figure 7. A CDF for prevalence of user-owned resources according to the ranking of sites that use them.

G. Methodology validation using user-driven analysis

In the previous sections, we examined sites by visiting their home pages and relying on WebAnalyzer's heuristics (see Section III) to simulate a few basic user actions to invoke additional functionality that may be hidden behind "splash" home pages. However, our methodology may miss site functionality that requires user login forms (e.g., on Facebook), other more sophisticated user event handlers (e.g., mouse movements), or following many links away from the home page. In general, it is very difficult, if not impossible, to simulate user actions that open access to representative features of an arbitrary site.

To evaluate the limitations of our heuristics-driven approach, we conducted a user-driven examination of the top 100 Alexa sites. To do this, one of the authors manually visited these sites with IE_{WA} and used his best judgement to invoke the site's representative functionality. For example, for analyzing Facebook, the author logged into his Facebook account, browsed through several profiles, and invoked several applications such as photo viewing or messaging.

We then compared the results obtained through this manual analysis to those obtained using WebAnalyzer for the same sites. Table VII summarizes the results of our comparison. We observe that the numbers of sites using a particular feature are mostly comparable, providing confidence that our heuristic-driven navigation engine in WebAnalyzer works well in practice. Some features have higher prevalence with the user-driven analysis, as expected, but there are only a couple of outliers. For example, Geolocation was found on nine sites, all found on multilingual versions of `maps.google.com`. In manual analysis, the user invoked maps on each of the nine versions of the Google site, where WebAnalyzer randomly picked and followed the link to `Maps` on three of these sites. On the other hand, on several occasions, WebAnalyzer also found features that were missed by manual analysis, as can be seen in higher prevalence for features like reading `document.domain`. This can happen when WebAnalyzer navigates to a link that the user did not examine as part of representative

Measurement Criteria	Number of sites	
	WebAnalyzer	Manual
document.cookie (read)	93	86
document.cookie (write)	86	76
document.cookie domain usage (read)	78	70
document.cookie domain usage (write)	59	59
Secure cookies over HTTP	0	2
Non-secure cookies over HTTPS	11	8
Use of "HttpOnly" cookies	27	30
Frequency of duplicate cookies	17	8
Use of XMLHttpRequest	32	28
Cookie read in response of XMLHttpRequest	0	0
Cross-origin descendant-navigation (reading descendant's location)	0	0
Cross-origin descendant-navigation (changing descendant's location)	0	0
Child navigation (parent navigating direct child)	1	2
document.domain (read)	78	59
document.domain (write)	18	19
Use of cookies after change of effective domain	18	19
Use of XMLHttpRequest after change of effective domain	4	2
Use of localStorage after change of effective domain	2	1
Use of session storage	0	0
Use of local storage	4	3
Use of fragment identifier for communication	0	1
Use of postMessage	1	1
Use of XDomainRequest	1	2
Presence of JavaScript within CSS	16	27
Setting top-level window's location	1	2
Change focus of window	2	2
Reading user's Geolocation	3	9

Table VII
COMPARISON OF USER-DRIVEN ANALYSIS VS. WEBANALYZER FOR THE TOP 100 ALEXA SITES. FEATURES NOT SHOWN HERE WERE USED BY ZERO SITES FOR BOTH USER-DRIVEN AND WEBANALYZER STUDIES.

functionality on a given site. Overall, we felt our heuristics-driven approach achieved good coverage, though larger-scale user-driven measurements would still be very valuable in complementing WebAnalyzer measurements.

V. DISCUSSION AND LIMITATIONS

Benefits of heuristics-driven automated crawling. In our original design, WebAnalyzer visited only the top-level page of each site we studied. We quickly realized that this analysis failed for sites that hide much of their functionality behind "splash" home pages. This became most apparent when studying the original results for Table VII. We observed that for many sites, clicking on a link or filling out a search form on the home page would expose a noticeably larger (though still not complete) set of functionality. Thus, we augmented WebAnalyzer with simple heuristics that imitate this user behavior (see Section III).

As an example, our original system saw XMLHttpRequest calls on only 13 pages of the top 100 pages, whereas the new one identified 32 such pages (see Table VII). One of the reasons is that many search sites use XMLHttpRequest to auto-complete the search string as users type it; our old system did not trigger this behavior, whereas our new system triggered it when auto-filling the search textbox. Many other features showed a similarly dramatic jump in prevalence.

Limits of automated crawler-based measurements. Although we believe that our resulting measurements provide a good representation of the use of browser features on popular web sites, it is likely that we missed certain features because the code path to invoke them was not triggered in our analysis. For example, sites like Facebook or banks require a user to sign in, game sites require particular mouse gestures to invoke certain behavior, and numerous sites require appropriate text (such as stock symbols or user's personal data) to be entered into forms. Even if we could solve some of these problems, for example by enumerating all events registered on a page or using a database of dummy usernames and passwords [27], automatically invoking certain features, such

as buying products on shopping sites, is inappropriate. This ultimately limits our ability to explore *all* features invoked on today's web.

We also did not try to exhaustively crawl each site. Even in our user-driven analysis (Section IV-G), we did not attempt to enumerate and invoke all gadgets on every page of each site. Thus, the results we collect for a particular site cannot be used as a list of *all* features the site might have. Our aim was to favor breadth over depth and obtain good coverage for the representative features of 100,000 sites we tested. While our infrastructure could also be used for exhaustively crawling each site, we would need to dramatically scale up our current infrastructure to cover a comparable number of sites, and we leave this as future work.

Picking the right browser. Some sites check the client's browser version (using the user-agent header) before deciding to invoke a particular code path. Although not a base requirement, we developed WebAnalyzer with IE as the underlying browser. This could prevent code invocations that are intended for non-IE browsers, thereby leading to missed features. For example, XMLHttpRequest2 [15] is currently not supported by IE, and it would be missed by WebAnalyzer if the site invokes it only after verifying browser support.

A related problem is fallback code that invokes an alternative implementation of a feature that a browser doesn't support. For example, a site could first check whether the browser supports `postMessage` for cross-frame communication, and fall back on fragment identifier messaging if it does not. Because we use IE 8, we will log that this site uses `postMessage`, but older browsers would utilize fragment identifier messaging.

The compatibility cost of features invoked in browser-dependent code paths depends not only on the number of web sites using a feature, but also on the number of visitors utilizing a particular browser that relies on such code. Evaluating the second part of this cost is orthogonal to our goals in this paper: rather than exploring prevalence of features on web sites, it asks how many of a web site's clients rely on a particular browser. Web server operators can easily answer this question by profiling "user-agent" strings in incoming HTTP requests. As future work, we can integrate other browsers into WebAnalyzer, or we can modify IE_{WA} to render a site with a set of user-agent strings representing other browsers; this would capture a more complete set of the site's code.

Studying other web segments. Our focus on the top 100,000 sites represents a particular segment of the web with a good balance of the very top sites and some of the less popular "tail". However, this still covers only a tiny fraction of the billions of pages on today's web. In addition, our analysis excluded intranet sites, which are hidden from traditional crawlers, and which can influence backwards compatibility decisions for a browser. We leave exploration of these other segments of the web as important future work.

VI. Related Work

We are not the first to find and analyze flaws in browser security policies. Previous work has looked at weaknesses in cross-frame communication mechanisms [13], frame navigation policies [3], [13], client-side browser state [21], cookie path protection [28], protection among documents within same origin [2], display protection [3], and other issues. Zalewski [9] documents the security design in browsers including some loopholes. This work complements these efforts by identifying incoherencies in browser's access control policies. To our knowledge, this is the first principal-driven analysis on browsers' access control policies.

DOM access checker [22] is a tool designed to automatically validate numerous aspects of domain security policy enforcement (cross-domain DOM access, JavaScript cookies, XMLHttpRequest calls, event and transition handling) to detect common security attacks or information disclosure vectors. Browserscope [29] is a community-driven project for tracking browser functionality. Its security test suite [23] checks whether new browser security features are implemented by a browser. In our analysis of access control policies, we uncovered incoherencies by examining the interplay between resources, runtime identity changes, and the user principal's resource access control. This focus and methodology differ from this previous or ongoing work, and our analysis not only touches on DOM, but also on the HTTP network layer and display. Nevertheless, we plan to contribute our test programs to one of these test suites.

Compared to previous work, a unique aspect of this work is our extensive evaluation of the cost of removing unsafe policies from the current web by actively crawling and executing web content. Yue et al. [24] also used a crawling-based, execution-based approach to measure the prevalence of unsafe JavaScript features on 6805 popular web sites. They used a JavaScript interposition technique that is similar to IE_{WA}'s script engine proxy, but they lack IE_{WA}'s network and display interposition capabilities, limiting the policies they can monitor. As well, we present results from a significantly larger dataset.

Our active crawling infrastructure builds on previous efforts that have analyzed safety of web pages by rendering them in real browsers running within virtual machines [30]–[34]. We extend these frameworks with additional browser interposition support to monitor unsafe browser security policies.

VII. Conclusions

In this paper, we have examined the current state of browser access control policies and analyzed the incoherencies that arise when browsers mishandle their principals by (1) inconsistently labeling resources with principal IDs, (2) inappropriately handling principal identity changes via

`document.domain`, and (3) neglecting access control for certain resources belonging to the user principal. In addition to pointing out these incoherencies, we have developed a web compatibility analysis infrastructure and measured the cost of removing many unsafe policies we identified for a large set of popular web sites. Overall, this work contributes to the community's understanding of browser access control policies, and it provides the much-needed answer to the browsers' *compatibility vs. security* dilemma by identifying unsafe policies that can be removed with little compatibility cost.

ACKNOWLEDGEMENT

We would like to thank Xiaofeng Fan, Yutaka Suzue, and Carl Edlund for their valuable help during the implementation of this work. We would also like to acknowledge Collin Jackson and David Wagner for their helpful discussions. We also thank the anonymous reviewers and our shepherd Michael Locasto for their valuable comments.

REFERENCES

[1] H. J. Wang, X. Fan, J. Howell, and C. Jackson, "Protection and Communication Abstractions for Web Browsers in MashupOS," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, Oct. 2007.

[2] C. Jackson and A. Barth, "Beware of Finer-Grained Origins," in *Web 2.0 Security and Privacy (W2SP)*, Oakland, CA, May 2008. [Online]. Available: http://seclab.stanford.edu/websec/origins/fgo.pdf

[3] 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 *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, Aug. 2009.

[4] J. Ruderman, "Same Origin Policy for JavaScript," http://www.mozilla.org/projects/security/components/same-origin.html. Accessed on Nov. 14, 2009.

[5] "Alexa," http://www.alexa.com/.

[6] "Document Object Model," http://www.w3.org/DOM/. Accessed on Nov. 14, 2009.

[7] D. Kristol and L. Montulli, "HTTP State Management Mechanism," in *IETF RFC 2965*, Oct. 2000.

[8] D. Flanagan, *Javascript: The Definitive Guide*. O'Reilly Media Inc., 2006.

[9] M. Zalewski, "Browser Security Handbook," 2008, http://code.google.com/p/browsersec/wiki/Main. Accessed on Nov. 14, 2009.

[10] A. Barth, "HTTP State Management Mechanism," IETF Draft 2109, Feb 2010, http://tools.ietf.org/html/draft-ietf-httpstate-cookie-03.

[11] C. Jackson and A. Barth, "ForceHTTPS: Protecting High-Security Web Sites from Network Attacks," in *WWW*, 2008.

[12] "HTML 5 Editor's Draft," October 2008, http://www.w3.org/html/wg/html5/.

[13] A. Barth, C. Jackson, and J. C. Mitchell, "Securing Frame Communication in Browsers," in *Proceedings of the 17th USENIX Security Symposium*, San Jose, CA, Jul. 2008.

[14] "XMLHttpRequest," http://www.w3.org/TR/XMLHttpRequest/. Accessed on Nov. 14, 2009.

[15] "XMLHttpRequest Level 2," http://www.w3.org/TR/XMLHttpRequest2/. Accessed on Nov. 14, 2009.

[16] "Mitigating Cross-site Scripting With HTTP-only Cookies," http://msdn2.microsoft.com/en-us/library/ms533046.aspx. Accessed on Nov. 14, 2009.

[17] "HttpOnly," http://www.owasp.org/index.php/HTTPOnly. Accessed on Nov. 14, 2009.

[18] "Mozilla Foundation Security Advisory 2009-05: XMLHttpRequest allows reading HTTPOnly cookies," http://www.mozilla.org/security/announce/2009/mfsa2009-05.html. Accessed on Nov. 14, 2009.

[19] "Clickjacking," http://en.wikipedia.org/wiki/Clickjacking.

[20] "Whats New in Internet Explorer 8," 2008, http://msdn.microsoft.com/en-us/library/cc288472.aspx. Accessed on Nov. 14, 2009.

[21] 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 (WWW)*, Edinburgh, Scotland, May 2006.

[22] M. Zalewski and F. Almeida, "Browser DOM Access Checker 1.01," http://lcamtuf.coredump.cx/dom_checker/. Accessed on Nov. 14, 2009.

[23] C. Jackson and A. Barth, "Browserscope Security Test Suite," http://mayscript.com/blog/collinj/browserscope-security-test-suite. Accessed on Nov. 14, 2009.

[24] C. Yue and H. Wang, "Characterizing Insecure JavaScript Practices on the Web," in *Proceedings of the 18th International Conference on World Wide Web (WWW)*, Madrid, Spain, Apr. 2009.

[25] E. Lawrence, "Fiddler web debugging tool," http://www.fiddler2.com/fiddler2/. Accessed on Nov. 14, 2009.

[26] "FiddlerCore," http://fiddler.wikidot.com/fiddlercore. Accessed on Nov. 14, 2009.

[27] "BugMeNot," http://www.bugmenot.com/. Accessed on Mar. 1, 2010.

[28] M. O'Neal, "Cookie Path Best Practice," http://research.corsaire.com/whitepapers/040323-cookie-path-best-practice.pdf. Accessed on Nov. 14, 2009.

[29] "Browserscope," http://www.browserscope.org/. Accessed on Nov. 14, 2009.

[30] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy, "A Crawler-based Study of Spyware on the Web," in *Proceedings of the 13th Annual Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, Feb. 2006.

[31] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King, "Automated Web Patrol with Strider HoneyMonkeys," in *Proceedings of the 13th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2006.

[32] N. Provos, P. Mavrommatis, M. Rajab, and F. Monrose, "All Your iFrames Point to Us," in *Proceedings of the 17th USENIX Security Symposium*, San Jose, CA, Jul. 2008.

[33] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu, "The Ghost in the Browser: Analysis of Web-Based Malware," in *Proceedings of the 1st Workshop on Hot Topics in Understanding Botnets (HotBots)*, Berkeley, CA, USA, 2007.

[34] A. Moshchuk, T. Bragin, D. Deville, S. D. Gribble, and H. M. Levy, "SpyProxy: Execution-based Detection of Malicious Web Content," in *Proceedings of the 16th USENIX Security Symposium*, Boston, MA, Aug. 2007.