Data-Confined HTML5 Applications

Devdatta Akhawe[1], Frank Li[1], Warren He[1], Prateek Saxena[2], and Dawn Song[1]

[1] University of California, Berkeley, Berkeley, CA, USA
[2] National University of Singapore, Singapore

Abstract. Rich client-side applications written in HTML5 proliferate on diverse platforms, access sensitive data, and need to maintain *data-confinement invariants*. Applications currently enforce these invariants using implicit, ad-hoc mechanisms. We propose a new primitive called a *data-confined sandbox* or DCS. A DCS enables complete mediation of communication channels with a small TCB. Our primitive extends currently standardized primitives and has negligible performance overhead and a modest compatibility cost. We retrofit our design on four real-world HTML5 applications and demonstrate that a small amount of effort enables strong data-confinement guarantees.

1 Introduction

Rich client-side applications written in HTML, CSS, and JS—including browser extensions, packaged browser applications (Chrome Apps) [17], Windows 8 Metro applications [32], and applications in newer browser operating systems (B2G [33], Chrome OS [18])—are fast proliferating on diverse computing platforms. These "HTML5" applications run with access to sensitive user data, such as browsing history, personal and social data, and financial documents, as well as capability bearing tokens that grant access to these data. A recent study of 5,943 Google Chrome browser extensions revealed that 58% required access to the user's browsing history, and 35% requested permissions to the user's data on all websites [10].

Applications handling sensitive data need the ability to *verifiably confine* data to specific principals and to prevent it from leaking to malicious actors. On one hand, the developers want an easy, high-assurance way to confine sensitive data; on the other, platform vendors and security auditors want to verify sensitive data confinement. For example, consider LastPass, a real-world HTML5-based password manager with close to a million users[1]. By design, LastPass only stores an encrypted version of the user's data in the cloud and decrypts it at the client side with the user's master password. It is critical that the decrypted user data (i.e., the clear-text password database) never leave the client. We term this requirement a *data-confinement invariant*. Data-confinement invariants are fundamental security specifications that limit the flow of sensitive data to a trusted set of security principals. These data-confinement invariants are not

[1] `https://www.lastpass.com`

explicitly stated in today's HTML5 applications but are implicitly necessary to preserve their privacy and security guarantees.

We observe two hurdles that hinder practical, high-assurance data confinement in existing client-side HTML5 applications. First, mechanisms to specify and enforce data-confinement invariants are absent in HTML5 platforms as a result, they remain hidden in application designs; raising the TCB. Second, client-side HTML5 applications have numerous channels to communicate with distrusting principals, and no unified monitoring interface like the OS system call interface exists. Due to the number of channels available to HTML5 applications, attackers can violate data confinement invariants even in the absence of code injection vulnerabilities [45,26]. As we explain in Section 3.2, previous research proposals do not offer complete mediation, or have an unacceptably large TCB and compatibility cost.

We introduce the data-confined sandbox (or DCS), a novel security primitive for client-side HTML5 applications. A data-confined sandbox is a unit of execution, such as code executing in an `iframe`, the creator of which explicitly controls all the data imported and exported by the DCS. Our design provides the creator of a DCS a secure reference monitor to interpose on all communications, privileged API accesses, and input/output data exchanges originating from the DCS.

Data-confined sandboxes are a fundamental primitive to enable a data-centric security architecture for emerging HTML5 applications. By moving much of the application code handling sensitive data to data-confined sandboxes, we can enable applications that have better resilience to privacy violating attacks and that are easy to audit by security analysts.

Contributions. We make the following main contributions:

- We introduce the concept of data confinement for client-side HTML5 applications that handle sensitive data (Section 2).
- We identify the limitations of current security primitives in the HTML5 platform that make them insufficient for implementing data-confinement invariants (Section 3.2).
- We design and implement a data-confined sandbox, a novel mechanism in web browsers that provides complete mediation on all explicit data communication channels (Section 4) and discuss how to implement such a new primitive without affecting the security invariants maintained by the HTML5 platform (Section 4.3).
- We demonstrate the practicality of our approach by modifying four applications that handle sensitive data to provide strong data confinement guarantees (Section 6). All our code and case studies are publicly available online [13].

2 Data Confinement in HTML5 Applications

Data confinement is a data-centric property, which limits the flow of sensitive data to an explicitly allowed set of security principals. In this section, we present example data-confinement invariants from real-world applications. Our focus is on modern HTML5 applications that handle sensitive data or tokens with complex client-side logic leading to a large client-side TCB.

2.1 Password Managers

Password managers organize a user's credentials across the web in a centralized store. Consider LastPass, a popular password manager that stores encrypted credential data in the cloud. LastPass decrypts the password database only at the client side (in a 'vault') with a user provided master password. A number of data-confinement invariants are implicit in the design of LastPass.

- First, the user's master password should never be sent to *any* web server (including LastPass servers).
- Second, the password database should only be sent back to the LastPass servers after encryption.
- Third, the decrypted password database on the client-side should not leak to *any* web site.
- Finally, only individual decrypted passwords should be sent only to their corresponding websites: e.g., the credentials for `facebook.com` should only be used on `facebook.com`.

2.2 Client-Side SSO Implementations

Single sign-on (SSO) mechanisms have emerged on the web to manage users' online identities. These mechanisms rely on confining secret tokens to an allowed set of principals. Consider Mozilla's recent SSO mechanism called BrowserID. It has the following data-confinement invariants implicit in its design:

- It aims to share authorization tokens only with specific participants in one run of the protocol.
- Similar to the 'vault' in LastPass, BrowserID provides an interface for managing credentials in a user 'home page.' This home page data should not leak to external websites.
- The user's BrowserID credentials (master password) should never be leaked to a third party: only the authorization credentials should be shared with the intended web principals involved in the particular instance of the protocol flow.

Other SSO mechanisms, like Facebook Connect, often process capability-bearing tokens (such as OAuth tokens). Implementation weaknesses and logic flaws can violate these invariants, as researchers demonstrated in 2010 [24,3], 2011 [43], and 2012 [41].

2.3 Electronic Medical Record Applications

Electronic medical record (EMR) applications provide a central interface for patient data, scheduling, clinical decisions, and billing. Strict compliance regulations, such as HIPAA, require data confinement for these applications, with financial and reputational penalties for violations. OpenEMR is the most popular open-source EMR application [38] and has a strict confinement requirement: an instance of OpenEMR should not leak user data to *any* principal other than hospital servers.

Note the dual requirements in this application: first, OpenEMR's developers want to ensure data confinement to their application; second, hospitals need to verify that OpenEMR is not leaking patient data to any external servers. In the current design, it is difficult for hospitals to verify this: any vulnerability in the client-side software can allow data disclosure.

2.4 Web Interfaces for Sensitive Databases

Web-based database administration interfaces are popular today, because they are easy to use. PhpMyAdmin is one such popular interface with thousands of downloads each week [34]. The following data-confinement invariants are implicit in its design:

- Data received from the database server is not sent to any website.
- User inputs (new values to store) are only sent to the database server's data insertion endpoint.

Currently, a code injection vulnerability in the client-side interface can enable attackers to steal the entire database, as the interface executes with the database user's privileges. Moreover, the application is large and not easily auditable to ensure data-confinement invariants.

Prevalence of Data Confinement. The discussion above only provides exemplars: *any* application handling sensitive data typically has a confinement invariant. Due to space constraints, we have made our analysis of the twenty most popular Google Chrome extensions available online [13]. All applications handling sensitive data (sixteen applications in total) maintained an invariant implicitly.[2] The trusted code base for these extensions varied from 7.5KB to 1.24MB. Sensitive data available to the extensions vary from access to the user's browsing history to the user's social media login credentials.

3 Problem Formulation

Given the prevalence of data confinement in HTML5 applications, we aim to support secure data confinement in HTML5 applications. Due to the increasingly sensitive nature of data handled by modern HTML5 applications, a key

[2] The remaining four extensions dealt mainly with the website style and appearance and did not access sensitive data.

requirement is *high assurance*: small TCB, complete mediation. Further, for ease of adoption, we aim for a mechanism with minimal compatibility costs.

The idea of such high assurance mechanisms is not new, with Saltzer and Schroeder laying it down as a fundamental requirement for secure systems [39]. Our focus is on developing a high assurance mechanism for HTML5 applications. We first discuss the challenges in achieving high assurance data confinement in HTML5 applications, followed by a discussion on why current and proposed primitives do not satisfy all our goals. We discuss our design in Section 4.

3.1 HTML5 and Data Confinement: Challenges

A number of idiosyncrasies of the HTML5 platform make practical data confinement with a small TCB difficult. First, the HTML5 platform lacks mechanisms to explicitly state data-confinement invariants—current ad-hoc mechanisms do not separate policy and enforcement mechanism. Due to the coarse-grained nature of the same origin policy, enforcing these invariants on current HTML5 platforms increases the TCB to the whole application.

Achieving a small TCB is particularly important on the HTML5 platform. The JavaScript language and the DOM interface make modular reasoning about individual components difficult. All code runs with ambient access to the DOM, cookies, localStorage, and the network. Further, techniques like prototype hijacking can violate encapsulation assumptions and allow attackers to leak private variables in other modules. The DOM API makes confinement difficult to ensure even in the absence of code injection vulnerabilities [45,26].

Achieving complete mediation on the HTML5 platform is also difficult. The HTML5 platform has a large number of data disclosure channels, as by design it aims to ease cross-origin resource loading and communication. We categorize these channels as:

- **Network channels**. HTML5 applications can make network requests via HTML elements like `img`, `form`, `script`, and `video`, as well as JavaScript and DOM APIs like `XMLHttpRequest` and `window.open`. Furthermore, CSS stylesheets can issue network requests by referencing images, fonts, and other stylesheets.
- **Client-side cross-origin channels**. Web browsers support a number of channels for client-side cross-origin communication. This includes exceptions to the same-origin policy in JavaScript such as the `window.location` object. Initially, mashups used these cross-origin communication mechanisms for fragment ID messaging (via the `location.hash` property) between cross-origin windows. Current mashups rely on newer channels like `postMessage`, which are also a mechanism for data leaks.
- **Storage Channels**. Another source of data exfiltration are storage channels like localStorage, cookies, and so on. These channels do not cause network requests or communicate with another client-side channel as above; instead, they allow code to exfiltrate data to other code that will run in the future in the same origin (or, in case of cookies, even other related origins). Browsers tie storage channels to the origin of an application.

Given the wide number of channels available for inadvertent data disclosure, we observe that no unified interface exists for ensuring confinement of fine-grained code elements in the HTML5 platform. This is in contrast to system call interposition in commodity operating systems that provides complete mediation. For example, mediation of data communication channels using system call sandboxing techniques is well-studied for modern binary applications [30,19,36]. Previous work also developed techniques to automate identification and isolation of subcomponents that process sensitive data [30,7]. Our work shares these design principles, but targets HTML5 applications.

3.2 Insufficiency of Existing Mechanisms

None of the primitives available in today's HTML5 platform achieve complete mediation with a small TCB. Browser-supported primitives, such as Content Security Policy (CSP), block some network channels but not all. Current mechanisms in web browsers aim for integrity, not confinement. For example, even the most restrictive CSP policy cannot block data leaks through anchor tags and `window.open`. Similarly, our previous work on privilege separation of HTML5 applications does not provide any confinement guarantees [4]. An unprivileged child can leak data by making a request for an image or including a CSS style from a remote host.

Table 1. Comparison of current solutions for data confinement

System Name	Complete Mediation	Compatibility Cost	Small TCB
HSTS	No: HTTPS pages only	Low	Yes
CSP	No: anchors and `window.open`	High: disables eval	Yes
JS Static Analysis	No: no CSS & DOM	High: disables eval	No
JS IRMs (Cajole, Conscript)	No: no CSS & DOM	High: disables eval	Yes
JSand	No: no CSS	High: SES	No
Treehouse	Yes	High: code change	No
`sandbox` with Temp. Origins	No: all network channels	Low	Yes
Data-confined sandboxes	**Yes**	**Low**	**Yes**

Recent work on information flow and non-interference show promise for ensuring fine-grained data-confinement in JavaScript; unfortunately, these techniques currently have high overhead for modern applications [11]. IBEX proposed writing extensions in a high-level language (FINE) in a language amenable to deep analysis to ensure conformance with specific policies [23]. In contrast, our work does not require significant changes to web applications. Further, as we explain below, these approaches also have a large TCB.

Another approach to interpose on all data communication channels is to do static analysis of the application source code [14,16,31]. Static analysis systems cannot reason about dynamic constructs such as `eval`, which are used pervasively by existing applications [37] and modern JavaScript libraries [1]. As a result, such mechanisms have a high compatibility cost. When combined with rewriting techniques, such as cajoling [16], JS analysis techniques can achieve complete mediation on client-side cross-frame channels; but still do not provide complete mediation over DOM and CSS channels.

JSand [2] introduced a client-side method of sandboxing third-party JavaScript libraries. It does so by encapsulating all Javscript objects in a wrapper that mediates property accesses and assignments, via an application-defined policy. This approach does not protect against scriptless attacks such as those using CSS. Additionally, it relies on the use of Secure EcmaScript 5 (SES), which is not compatible for some JavaScript libraries. JSand does provide a support layer to improve compatibility with legacy JavaScript code, but this is a partial transformation and involves a high performance overhead.

Treehouse uses new primitives, like web workers and EcmaScript5 sealed objects, in the HTML5 platform to ensure better interposition [27]. Treehouse proposes to execute individual components in web workers at the client side. One concern with the Treehouse approach is that web workers also run with some ambient privileges: e.g., workers have access to `XMLHttpRequest`, synchronous file APIs, script imports, and spawning new workers, which attackers can use to leak data. Treehouse relies on the seal/unseal features of ES5 to prevent access to these APIs, but this mechanism requires intrusive changes to existing applications and has a high compatibility cost.

Perhaps the most important limitation of *all* primitives not directly supported by browsers is their large TCB. For example, in the case of Treehouse, application code (running in workers) cannot have direct access to the DOM, since that would break all security guarantees. Instead, application code executes on a virtual DOM in the worker that the parent code copies over to the main web page. As a result, the security of these mechanisms depends on the correctness of the monitor/browser model (e.g., the parent's client side monitor in Treehouse).

Since the DOM, HTML, CSS, and JS are so deeply intertwined in a modern HTML5 platform, such a client side monitor is essentially replicating the core logic of the browser, leading to a massive increase in the TCB. Further, Treehouse implements this complex logic in JavaScript. Corresponding issues plague static analysis systems, new language mechanisms like IBEX, and code rewriting systems like Caja—all of them assume a model of the HTML5 platform to implement their analysis/rewriting logic.

While implementing a model of HTML5 for analysis and monitoring is difficult, the approaches discussed above suffer from another fundamental limitation: they work on a model of HTML5, not the real HTML5 standard implemented in the platform (browser). Any mismatch between the browser and the model can lead to a vulnerability, as observed (repeatedly) for Caja [20,22,21,15] and AdSafe [31,35].

3.3 Threat Model

We focus on *explicit* data communication channels in the HTML5 platform core, as defined above. Ensuring comprehensive mediation on explicit data channels is an important first step in achieving data-confined HTML5 applications. Our proposed primitive does not protect against covert and side channels (such as shared browser caches [28] and timing channels [6]) or self exfiltration channels [9], which are a subject of ongoing research. These channels are important. However, we point out that popular isolation mechanisms on existing systems also do not protect against these [46,8,44]. We believe explicit channels cover a large space of attacks, and we plan to investigate extending our techniques to covert channels in the future.

In addition to focusing on explicit channels, our primitive only targets the core HTML5 platform; our ideas extend to add-ons/plugins, however we exclude them from our present implementation. We defend against the standard web attacker model, in which the attacker cannot tamper with or observe network traffic for other web origins and cannot subvert the integrity of the HTML5 platform itself [3].

4 The Data Confined Sandbox

To draw a parallel with binary applications, current mechanisms for confining HTML5 applications are analogous to analyzing the machine code *before* it executes to decide whether it violates any guarantees. We argued above that such mechanisms cannot provide high assurance. Instead, taking a systems view of the problem of data confinement, we argue for an `strace`-like high assurance monitor for the HTML5 platform.

We call our primitive the data confined sandbox, or DCS (Section 4.1). Our key contribution is identifying that the shrewd design of the DCS primitive provides high assurance with minimal compatibility concerns (Section 4.2). Introducing any new primitive on the HTML5 platform brings up security concerns. A primitive like DCS that provides monitoring capabilities to arbitrary code is particularly fraught. We discuss how we ensure that we do not introduce new vulnerabilities due to our primitive in Section 4.3.

4.1 Design of DCS

Figure 1 presents the architecture of an application using the DCS design. Our design extends our previous work on privilege separation [4]. Our key contribution is identifying how to extend the ideas of privilege separation to provide complete mediation on the HTML5 platform. We first recap privilege separated HTML5 applications and then discuss the DCS design.

Modern HTML5 platforms allow applications to run arbitrary code (specified via a `data:/blob:` URI) in a temporary, unprivileged origin [4]. Privilege separated HTML5 applications run most application code in an arbitrary number of unprivileged iframes (children). A small privileged parent iframe, with access to full privileges of the web origin, provides access to privileged APIs, such as cookie access and platform APIs like camera access. Unprivileged children communicate with the parent through a tightly controlled `postMessage` channel (dotted arrows in Figure 1).

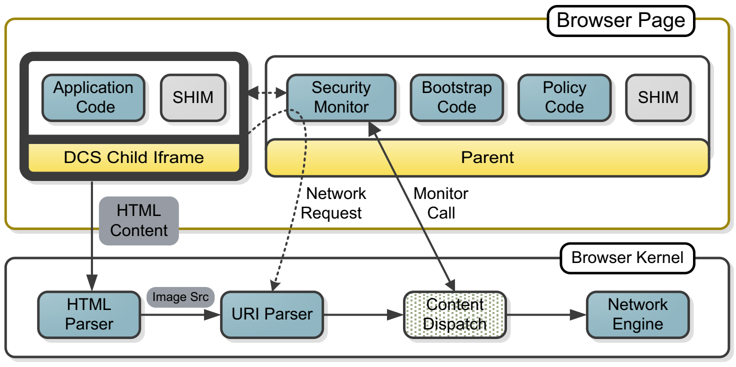

Fig. 1. High-level design of an application running in a DCS. The only component that runs privileged is the parent. The children run in data-confined sandboxes, with no ambient privileges and all communication channels monitored by the parent.

The parent can enforce policies on the requests it receives over this `postMessage` channel from its unprivileged children [4]. The parent uses its privileged interfaces to fulfill approved requests, such as authenticated `XMLHttpRequest` calls (curved dotted arrow in Figure 1). To increase assurance, the parent code enforces a number of security invariants such as disabling all dynamic code evaluation, allowing only a text interface with the children, and setting appropriate MIME types for static code downloaded by the bootstrap code.

Though this privilege separation architecture provides integrity, it does *not* provide data confinement. Any compromised child can make arbitrary requests on the network through the numerous data disclosure channels outlined earlier. We propose a new primitive, the *data-confined sandbox* or DCS, that enforces confinement of data in the child. Our primitive relies on the browser to ensure confinement. Similar to privilege separation, applications only need to switch to using the DCS and write an appropriate policy.

Consider the browser kernel in Figure 1. Any content that a DCS child requests the browser to display passes through the HTML/JS/CSS parser. If the browser encounters a URI that it needs to load, it invokes the URI parser, which then invokes the content dispatch logic in the browser. We modify this code for DCS children to call a security monitor that the parent defines (solid arrow in Figure 1). The security monitor in the parent is transparent to the child. The browser's call to the parent also includes the unique id identifying the child iframe and details about the request. From there, the security monitor can decide whether to grant the request or not.

Example. Consider the 'vault' for the LastPass web application. In our redesign, when the user navigates to the LastPass application, the server returns bootstrap code (the parent) that downloads the original application code and executes it in a data-confined sandbox (the child). The code in the DCS starts executing and makes network requests to include all the complex UI, DOM, and encryption libraries. Finally, the LastPass child code in the DCS makes a request for the encrypted password database and decrypts it with the user provided password.

The parent security monitor can enforce a simple policy such as only allowing network requests to `http://lastpass.com`. Alternatively, the parent can enforce stateful policies: e.g., the monitor function could only allow resource loads (i.e., scripts, images, styles) until the DCS child loads the encrypted password database. After loading the encrypted database, the security monitor disallows all future network requests.

4.2 Achieving High Assurance

Recall our goals of complete mediation, small TCB, and backwards compatibility. We discuss how our DCS design achieves all of them.

Complete Mediation. As discussed Section 3, HTML5 applications only have three channels for data leakage: storage channels tied to the origin, network channels, and client-side cross-origin channels. Since all application code runs in children of temporary origins that only exist for the duration of the application's execution, the application code does not have access to any (storage) channel tied to the origin (e.g., cookies, localStorage).

In a DCS, except for a blessed `postMessage` channel to the parent, the browser disables all client-side communication channels. This includes cross-origin communication channels like `postMessage` and cross-origin window properties (like `location.hash`). The `postMessage` channel is the *only* client-side cross-origin channel available to the data-confined child, and the browser guarantees that the channel only connects to the parent. The `postMessage` channel allows the parent to proxy privileged APIs for the child. Further, the `postMessage` channel also allows the parent to provide a channel to proxy `postMessages` to other client-side `iframes`—our design only enforces complete mediation by the parent.

HTML5 applications can request network resources via markup like scripts, images, links, anchors, and forms and JavaScript APIs like `XMLHttpRequest`. In our design, the children can continue to make these network requests; the DCS transparently interposes on all these network channels. The parent defines a 'monitor' function that the browser executes before dispatching a network request. If the function returns false, the browser will not make the network request.

We rely on an external monitor (i.e., one running in the parent) over an inline one. This ensures that the monitor does not share any state with the unprivileged child, making it easier to reason about its runtime integrity and correctness. As we discuss in Section 5, the security monitor is not hard to implement—most browsers already have an internal API for controlling network access, which they expose to internal browser code as well as popular extensions such as AdBlock and NoScript.

Small TCB. The TCB in any data confinement mechanism includes the policy code and the enforcement code. In our design, this includes the monitor code in the parent as well as our browser modifications to ensure complete mediation for the parent monitor. Relying on the browser allows us to create a data confinement design with a small enforcement code, as evidenced by our 214 line implementation described in Section 5. This small enforcement TCB allows for easier validation and auditing.

Compatibility. Our design for network request mediation is discretionary, as compared to client-side channels that we block outright. An alternative design is to disallow all network requests too, and only permit network access via the `postMessage` channel between the parent and child. Such a design has a significantly higher compatibility cost. HTML5 applications pervasively employ network channels. In contrast, the use of client-side channels is rare—for example, Wang et al. report that cross-origin `window.location` read and writes occur in less than 0.1% of pages [40]. Therefore, we find that it is acceptable to disable cross-origin client-side channels and force the child to use the blessed `postMessage` channel to the parent to access these.

4.3 Security Considerations

Our design of the DCS primitive is careful not to introduce new security vulnerabilities in the browser. We do not want to allow an arbitrary website to learn information or execute actions that it could not already learn or execute. The security policy of the current web platform is the same-origin policy. The introduction of the DCS should not violate any of the existing same-origin policy invariants baked into the platform. We enforce this goal with the following two invariants:

– *Invariant 1*: The parent should *only* be able to monitor application code that it could already monitor on the current web platform (albeit, through more fragile mechanisms).
– *Invariant 2*: The parent should not be able to infer anything about a resource requested by a DCS that is not already possible on the current web platform.

We explain how our design enforces the above invariants. First, in our design, a data-confined sandbox can only apply to `iframe`s with a `data:` URI source, not to arbitrary URIs. Therefore, a malicious site cannot monitor arbitrary web pages. In an iframe with a `data:` URI source, the creator of the iframe (the parent) specifies the source code that executes. This code is under complete control of the parent anyways. The parent can parse the `data:` URI source for static requests and redefine the DOM APIs to monitor dynamic requests [25]. Thus, even in the absence of our primitive, the parent can already monitor any requests a `data:` URI `iframe` makes.

To ensure Invariant 2, we only call the security monitor for the *first* request made for a particular resource. As we noted above, the parent can already monitor this request. Future requests (e.g., redirects) are not in the control of the parent, and we do not call the security monitor for them. While this can cause security issues (particularly, if the parent whitelists an open-redirect), allowing the parent to monitor redirects would cause critical vulnerabilities.

For example, consider a page at `http://socialnetwork.com/home` that redirects to `http://socialnetwork.com/username`. Consider a DCS child created by `attacker.com` parent. If this child creates an iframe with source `http://socialnetwork.com/home`, our modified browser calls the security monitor with this URI before dispatching the request. However, to ensure Invariant 2, the browser does *not* call the security monitor with the redirect URI (i.e., `http://socialnetwork.com/username`). Further, since the iframe is now executing in the security context of `http://socialnetwork.com/`, Invariant 1 ensures that any image or script loads made by the `socialnetwork.com` iframe do not call the security monitor.

5 Implementation

We implemented support for data-confined sandboxes in the Firefox browser. Our modified browser and our case studies (Section 6) are all available online [13]. Our implementation is fewer than 214 lines of code, with only 60 lines being the core functionality. The low implementation cost substantiates our intuition that the monitoring facility is best provided by the browser. Since major browsers already support temporary origins, we only need to add support for mediating client-side and network channels of a DCS child.

First, we restrict cross-origin client-side channels to a blessed postMessage channel. As a fundamental security invariant, the same-origin policy restricts cross-origin JavaScript access to a restrictive white-list of properties. In Firefox, this whitelist is present in `js/xpconnect/wrappers/AccessCheck.cpp`. We modified the `IsPermitted` function to block all cross-origin accesses, except for the blessed `postMessage` channel.

The `NSIContentPolicy` interface is a standard Firefox API used to monitor network requests. Popular security and privacy extensions, such as NoScript, AdBlock, and RequestPolicy, rely on this API, as do security features such as CSP and mixed content blocking. We register a listener to forward requests for monitored DCS children to the parent's security monitor function. We do *not* implement a new mediation infrastructure—any bypass of our mediation infrastructure would also be a critical vulnerability in the Firefox browser, allowing bypass of all the features and extensions discussed above.

Applications can mark an `iframe` as a DCS using the `dcfsandbox` attribute, similar to the `iframe sandbox` attribute. An `iframe` that has this attribute only supports a `data:` or `blob:` URIs for its `src` attribute. Such a DCS `iframe` implements all the restrictions that a `sandboxed iframe` supports, but provides a complete mediation interface to the parent as described above.

To measure the overhead of calling the parent's monitor code, we measured the increase in latency caused by a simple monitor that allows all requests. We measured the time required for script loads from a web server running on the local machine and found that the load time increased from 16.73ms to 16.74ms.

This increase is statistically insignificant, and pales in comparison to the typical latencies of 100ms observed on the web.

Due to the semantics of network requests in HTML5, the monitor function runs synchronously: a long running monitor function could freeze the child. The ability to cause stability problems via long running synchronous tasks is already a problem in browsers and is not an artifact stemming from our design.

6 Case Studies

We retrofit our application architecture to four web applications to demonstrate the practicality of our approach. We focus on open-source software for our case studies, since that allows us to share our results freely online [13]. Table 2 summarizes all our case studies. Similar to previous work, we use TCB size instead of lines of code as a metric due to the prevalence of JavaScript minification.

We find that we needed minimal changes (at most 184 lines of code) to port existing applications to our design, mirroring our previous experience with privilege separation. In this section, we focus on the policies we implemented for each application; the accompanying technical report provides full details of our experience porting these applications to run under a DCS [5],

Table 2. List of our case studies, as well as the individual components and policies

Application	Initial TCB	New TCB	Lines Changed	Component	Confinement Policy	Other Policies
Clipperz	1.4MB	6.3KB	67	Vault UI	Only to Clipperz server & Direct Login Child	None
				Direct Login	Open arbitrary websites	CSP Policy disabling dynamic code
BrowserID	206.9KB	5.7KB	184	Management	Only to BrowserID server	None
				Dialog	Only to BrowserID server, secure password input	API requests must match state machine
OpenEMR	149.1KB	6.1KB	51	Patient Information	Whitelist of necessary request signatures	None
SQL Buddy	100KB	2.97KB	11	Admin UI	Only to MySQL server	User confirmation for database writes

6.1 The Clipperz Password Manager

Clipperz is an open-source HTML5 password manager that allows a user to store sensitive data, such as website logins, bank account credentials, and credit card information encrypted in the cloud [12]. Clipperz decrypts it at the client side with the user provided password. Users access their data in a single 'vault' page. Users can also click on 'direct login' links that load a site's login page, fill in the user name/password, and submit the login form. All of Clipperz's code and libraries run in a single security principal, with access to all sensitive data. The Clipperz application uses inline scripts and `data:` URIs extensively. We found that enforcing strong CSP restrictions to protect against XSS breaks the Clipperz application.

```
1  var doneLoading_mainframe = false, doneLoading_secondframe = false;
2  function monitor(params) {
3      if (params.id === "mainframe") { /*Policy for UI child*/
4          if (params.url === base_uri) { /*base_uri is the installation
               directory*/
5              return true;
6          } else if (params.type == "IMAGE") {
7              return check_img_whitelist(params.url);
8          } else if (params.type == "SCRIPT") {
9              if (!doneLoading_mainframe && params.url === base_uri + "/shim1.
                   js") {
10                 doneLoading_mainframe = true;
11                 return true;
12             } else if (!doneLoading_mainframe) {
13                 return check_script_whitelist(params.url);
14             }
15         }
16     } else if (params.id == "secondframe") { /*Policy for non-UI child*/
17         if (params.url === base_uri) { return true;}
18         else if (params.type == "SCRIPT") {
19             if (!doneLoading_secondframe && params.url === base_uri + "/shim2
                   .js") {
20                 doneLoading_secondframe = true;
21                 return true;
22             } else if (!doneLoading_secondframe) {
23                 return check_script_whitelist(url);
24             }
25         }
26     }
27     return false;}
```

Listing 1.1. A basic policy for Clipperz

Data-Confinement. We modified Clipperz to run in a pair of data-confined sandboxes: one for the UI and another for the non-UI functionality. Our modifications required minimal effort (67 lines changed) and reduced the TCB from 1.4MB to 6.3KB. This new TCB includes 42 lines of policy code.

Invariants. We apply a temporal policy for each sandbox as shown in Listing 1.1. For both sandboxes, the monitor code in our modified Clipperz applications only allows the DCS access to `postMessage` and a whitelist of images and JavaScript files (lines 7, 13, and 23). We also enforce a temporal policy: we allow network requests only until the sandbox downloads the password database (lines 10 and 20). Once the DCS sandbox downloads the password database, our policy disallows further network access save for navigation to pages like the help page. Relying on a whitelist of network resources means that we can guarantee the secrecy of the user's master password, which is impossible in the current HTML5 platform.

We do not allow the UI code to execute direct logins, since it presents a possible self-exfiltration channel [9]. Instead, it must send a message telling the non-UI component to do a direct login. The non-UI component retrieves the appropriate credentials and completes the direct login process. The non-UI component does not need complex UI code and executes with a restrictive CSP, providing higher assurance.

6.2 The BrowserID SSO Mechanism

BrowserID is a new authentication service by Mozilla. Similar to other single sign-on mechanisms like Facebook Connect and OpenID, BrowserID enables websites (termed Relying Parties) to authenticate a user using Mozilla's centralized service. Users set up a single "master" email/password to sign in to the trusted BrowserID service and can have the service authenticate the user to a Relying Party. Other single sign-on mechanisms share similar designs, and our results are more generally applicable to other single sign-on systems.

BrowserID uses the EJS templating system [29], which loads template files from the server and converts them to code at runtime using `eval`. A number of modern JavaScript templating languages use this technique [1]. The use of `eval` limits the applicability of CSP and static analysis techniques.

Data-Confinement. We modified BrowserID to execute in a data-confined sandbox. We required minimal effort to port BrowserID—the majority of the changes (184 lines) were to switch the EJS library from synchronous XmlHttpRequests to asynchronous requests supported by privilege separated HTML5 applications. Our modifications reduced the TCB from 206.9KB to 5.7KB, which includes 81 lines of policy code.

Invariants. Running BrowserID in a DCS we were able to implement a policy to provide two key guarantees. First, the login and credential managers (management component) do not communicate with any servers other than the BrowserID servers. This allows us to enforce secrecy on the master BrowserID username/password.

Second, the parent ensures that in one instance of the authentication protocol, the DCS executes the whole protocol with the same BrowserID and Relying Party window. Our design guarantees that sensitive tokens are never leaked to parties outside these participants. In the past, single sign-on mechanisms have had implementation bugs that allowed a MITM of an authentication flow [43,41]; our design prevents such bugs.

For further hardening, we implemented a state machine in the security policy based on the intended dialog behavior, which ensures that the dialog (which asks for passwords and other user input) component performs a series of requests consistent with transitions possible in the state machine. This prevents a compromised dialog DCS from making arbitrary requests in the user's session, such as deleting her account.

6.3 The OpenEMR Patient Information Pages

OpenEMR is the most popular open-source electronic medical record system [38]. With support for a variety of records like patients, billing, prescriptions, medical reports amongst others, OpenEMR is a comprehensive and complex web application. Patient records, prescriptions, and medical reports are highly sensitive data, with most jurisdictions having laws regulating their access and distribution, possibly with penalties for inadvertent disclosure.

We focus on the patient information component of the OpenEMR application. OpenEMR accesses the patient details by using a session variable named `pid` (patient id). Once the user sets the patient id, all future requests, such as 'demographic data,' 'notes,' and so on, can only refer to the particular patient. To navigate to another patient, the user uses the search interface to reset the patient id.

Setting the patient id for a particular session just requires a GET request with a `set_pid` parameter. An attacker can accomplish this with any content injection. For example, an attacker could inject a specially crafted image tag that causes a user to make such a request. As a result, after a user loads the image, the OpenEMR server will return medical records for the attacker-specified user. Note that this is not an XSS attack, but a content injection attack.

Data-Confinement. We modified the patient information component to run under a DCS. This required modifications to 51 lines of code and reduced the TCB from 149.1KB to 6.1KB, which includes a 38-line policy.

Invariants. First, the DCS verifiably ensures that sensitive medical data does not leak to untrusted principals. The DCS can also prevent the page from making arbitrary calls to the large, feature-rich application. In our case, we programmed the security policy to allow only a short whitelist of (method, URL) pairs necessary for the page to function. For example, the monitor denies any request with a `set_pid` parameter. This protects against the content injection attack discussed above. This would not be possible with an origin-based whitelist.

6.4 The SQL Buddy Database Administration Interface

SQL Buddy is an open-source web-based application to handle the administration of MySQL databases. It allows database administrators to browse data stored in a MySQL database and to execute SQL queries and manage database users. SQL Buddy does not use any of the client-side communication channels we block in a DCS.

Data-Confinement. We reused previous work on privilege separation of SQL Buddy, which required only 11 lines of change to the 100KB SQL Buddy application. Our data-confined SQL Buddy application has a TCB of 2.97KB, which includes a 124-line policy.

Invariants. By executing SQL Buddy in a DCS, we can enforce strong confinement policies. The application runs in two logical stages. Initially, the policy restricts communication to only static SQL Buddy resources. Our first-stage policy allows the application to load only these whitelisted JavaScript and CSS files. After loading the scripts and stylesheets, the application only accesses the network to load static images and to make `XMLHttpRequests` to a special endpoint.

Our second-stage policy locks down communication to these two channels. The flexibility of the DCS policy interface is key to enforcing a different policy for each stage.

Our policy restricts all explicit communication channels: if the SQL Buddy DCS is compromised, it cannot send data to arbitrary servers. Our design also allows us to enforce finer grained policies. For example, we have the secure parent show confirmation prompt for database writes. This prevents compromised code in the DCS from surreptitiously modifying the database.

7 Related Work

A number of previous works share our goals of improving assurance in web applications. We gave a detailed comparison to closely related works in Section 3.2. Data confinement has been investigated in native binary applications as well [30]. Zalewski [45] and Heidrich et al. [26] point at a number of attacks that violate data-confinement invariants even in the absence of code injection. IceShield demonstrated the efficacy of modern ES5 features to create a tamper-resistant mediation layer for JavaScript in modern browsers [25]; these may be used a basis for implementing data confinement policy checkers in the future.

8 Conclusion

Modern HTML5 applications handle increasingly sensitive personal data, and require strong data-confinement guarantees. However, current approaches to ensure confinement are ad-hoc and do not provide high assurance. We presented a new design for achieving data-confinement that guarantees complete mediation with a small TCB. Our design is practical, has negligible performance overhead, and does not require intrusive changes to the HTML5 platform. We empirically show that our new design can enable data-confinement in a number of applications handling sensitive data and achieve a drastic reduction in TCB. Future work includes investigating and mitigating covert channels.

Acknowledgements. We thank the anonymous reviewers for their valuable feedback. This research is partially supported by research grant R-252-000-495-133 from Ministry of Education, Singapore. This work was supported by Intel through the ISTC for Secure Computing; by the National Science Foundation under grant numbers CCF-0424422, 0842695, and 0831501 CT-L; by the Air Force Office of Scientific Research under MURI awards FA9550-08-1-0352 and FA9550-09-1-0539; by the Office of Naval Research under MURI grant no. N000140911081.

References

1. Chromium Bug Tracker: `http://crbug.com/107538`
2. Agten, P., Acker, S.V., Brondsema, Y., Phung, P.H., Desmet, L., Piessens, F.: JSand: Complete client-side sandboxing of third-party javascript without browser modifications. In: ACSAC (2012)
3. Akhawe, D., Barth, A., Lam, P., Mitchell, J., Song, D.: Towards a Formal Foundation of Web Security. In: CSF (2010)
4. Akhawe, D., Saxena, P., Song, D.: Privilege Separation in HTML5 Applications. In: USENIX Security (2012)
5. Akhawe, D., Li, F., He, W., Saxena, P., Song, D.: Data-confined html5 applications. Technical Report UCB/EECS-2013-20, EECS Department, University of California, Berkeley (March 2013)
6. Barth, A.: Timing Attacks on CSS Shaders (2011), `http://goo.gl/Mos4a`
7. Brumley, D., Song, D.: Privtrans: Automatically Partitioning Programs for Privilege Separation. In: USENIX Security (2004)
8. Cabuk, S., Brodley, C.E., Shields, C.: Ip covert timing channels: design and detection. In: CCS (2004)
9. Chen, E., Gorbaty, S., Singhal, A., Jackson, C.: Self-exfiltration: The dangers of browser-enforced information flow control. In: W2SP (2012)
10. Chia, P.H., Yamamoto, Y., Asokan, N.: Is this app safe?: A large scale study on application permissions and risk signals. In: WWW (2012)
11. Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged information flow for JavaScript. In: PLDI (2009)
12. Clipperz: `http://www.clipperz.com/`
13. Code Release: `https://github.com/devd/data-confined-html5-applications`
14. Crockford, D.: AdSafe, `http://www.adsafe.org/`
15. Hayes, G.: Hacking caja part 2, `http://www.thespanner.co.uk/2012/09/18/hacking-caja-part-2/`
16. Google: Caja, `http://developers.google.com/caja/`
17. Google: Chrome web store, `https://chrome.google.com/webstore`
18. Google: Chromium os, `http://www.chromium.org/chromium-os`
19. Google: Seccomp sandbox for linux, `http://code.google.com/p/seccompsandbox/`
20. Google Caja Bug 51: `http://code.google.com/p/google-caja/issues/detail?id=51`
21. Google Caja Bug 1093: `http://code.google.com/p/google-caja/issues/detail?id=1093`
22. Google Caja: `http://code.google.com/p/google-caja/issues/detail?id=520`
23. Guha, A., Fredrikson, M., Livshits, B., Swamy, N.: Verified security for browser extensions. In: IEEE S&P (2011)
24. Hanna, S., Shin, E., Akhawe, D., Boehm, A., Saxena, P., Song, D.: The emperor's new apis: On the (in) secure usage of new client-side primitives. In: W2SP (2010)
25. Heiderich, M., Frosch, T., Holz, T.: IceShield: Detection and mitigation of malicious websites with a frozen DOM. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 281–300. Springer, Heidelberg (2011)
26. Heiderich, M., Niemietz, M., Schuster, F., Holz, T., Schwenk, J.: Scriptless attacks: stealing the pie without touching the sill. In: CCS (2012)
27. Ingram, L., Walfish, M.: Treehouse: Javascript sandboxes to help web developers help themselves. In: USENIX ATC (2012)

28. Jackson, C., Bortz, A., Boneh, D., Mitchell, J.C.: Protecting browser state from web privacy attacks. In: WWW (2006)
29. Jupiter-IT: EJS Javascript Templates, `http://embeddedjs.com/`
30. Khatiwala, T., Swaminathan, R., Venkatakrishnan, V.: Data Sandboxing: A Technique for Enforcing Confidentiality Policies. In: ACSAC (2006)
31. Maffeis, S., Mitchell, J.C., Taly, A.: Object capabilities and isolation of untrusted web applications. In: IEEE S&P (2010)
32. Microsoft: Metro Apps, `http://msdn.microsoft.com/en-us/windows/apps/`
33. Mozilla: Boot2gecko, `https://wiki.mozilla.org/B2G`
34. phpMyAdmin: `http://www.phpmyadmin.net/`
35. Politz, J.G., Eliopoulos, S.A., Guha, A., Krishnamurthi, S.: ADsafety: type-based verification of javascriptsandboxing. In: USENIX Security (2011)
36. Provos, N.: Improving host security with system call policies. In: Proceedings of the 12th Conference on USENIX Security Symposium, vol. 12, p. 18. USENIX Association, Berkeley (2003)
37. Richards, G., Lebresne, S., Burg, B., Vitek, J.: An analysis of the dynamic behavior of javascript programs. ACM SIGPLAN Notices (2010)
38. Riley, S.: 5 OpenSource EMRs worth reviewing (2011), `http://bit.ly/hUa6l1`
39. Saltzer, J., Schroeder, M.: The protection of information in computer systems. Proceedings of the IEEE 63(9), 1278–1308 (1975)
40. Singh, K., Moshchuk, A., Wang, H., Lee, W.: On the incoherencies in web browser access control policies. In: IEEE S&P (2010)
41. Sun, S., Hawkey, K., Beznosov, K.: Systematically breaking and fixing openid security: Formal analysis, semi-automated empirical evaluation, and practical countermeasures. Computers & Security (2012)
42. Tizen: `https://www.tizen.org/`
43. Wang, R., Chen, S., Wang, X.: Signing me onto your accounts through facebook and google: a traffic-guided security study of commercially deployed single-sign-on web services. In: IEEE S&P (2012)
44. Xu, Y., Bailey, M., Jahanian, F., Joshi, K., Hiltunen, M., Schlichting, R.: An exploration of l2 cache covert channels in virtualized environments. In: CCSW (2011)
45. Zalewski, M.: Postcards from the post-xss world, `http://lcamtuf.coredump.cx/postxss/`
46. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-VM side channels and their use to extract private keys. In: CCS (2012)