Privilege Separation in HTML5 Applications

Devdatta Akhawe, Prateek Saxena, Dawn Song
University of California, Berkeley
{devdatta,prateeks,dawnsong}@cs.berkeley.edu

Abstract

The standard approach for privilege separation in web applications is to execute application components in different web origins. This limits the practicality of privilege separation since each web origin has financial and administrative cost. In this paper, we propose a new design for achieving effective privilege separation in HTML5 applications that shows how applications can cheaply create arbitrary number of components. Our approach utilizes standardized abstractions already implemented in modern browsers. We do not advocate any changes to the underlying browser or require learning new high-level languages, which contrasts prior approaches. We empirically show that we can retrofit our design to real-world HTML5 applications (browser extensions and rich client-side applications) and achieve reduction of 6x to 10000x in TCB for our case studies. Our mechanism requires less than 13 lines of application-specific code changes and considerably improves auditability. Our design has influenced the security architecture of upcoming Chrome applications.

1 Introduction

Applications written with JavaScript, HTML5 and CSS constructs (called HTML5 applications) are becoming ubiquitous. Rich web applications and web browser extensions are examples of HTML5 applications that already enjoy massive popularity [25, 30]. The introduction of browser operating systems [4, 16], and support for HTML5 applications in classic operating systems [38, 54] herald the convergence of web and desktop applications. However, web vulnerabilities are still pervasive in emerging web applications and browser extensions [14], despite immense prior research on detection and mitigation techniques [5, 17, 28, 33, 45].

Privilege separation is an established security primitive for providing an important second line of defense [44]. Commodity OSes enable privilege separated applications via isolation mechanisms such as LXC [36], seccomp [46], SysTrace [41]. Traditional applications have utilized these for increased assurance and security. Some well-known examples include OpenSSH [42], QMail [10] and Google Chrome [7]. In contrast, privilege separation in web applications is harder and comes at a cost. If an HTML5 application wishes to separate its functionality into multiple isolated components, the same-origin policy (SOP) mandates that each component execute in a separate web origin.[1] Owning and maintaining multiple web origins has significant practical administrative overheads. [2] As a result, in practice, the number of origins available to a single web application is limited. Web applications cannot use the same-origin policy to isolate every new component they add into the application. At best, web applications can only utilize sub-domains for isolating components, which does *not* provide proper isolation, due to special powers granted to sub-domains in the cookie and document.domain behaviors.

Recent research [15, 33] and modern HTML5 platforms, such as the Google Chrome extension platform (also used for "packaged web applications"), have recognized the need for better privilege separation in HTML5 applications. These systems advocate re-architecting the underlying browser or OS platform to force HTML5 applications to be divided into a fixed number of components. For instance, the Google Chrome extension framework requires that extensions have three components, each of which executes with different privileges [7]. Similarly, recent research proposes to partition HTML5 applications in "N privilege rings", similar to the isolation primitives supported by x86 processors [33]. We observe two problems with these approaches. First, the

[1] Browsers isolate applications based on their origins. An origin is defined as the tuple <scheme, host, port>. In recent browser extension platforms, such as in Google Chrome, each extension is assigned a unique public key as its web origin. These origins are assigned and fixed at the registration time.

[2] To create new origins, the application needs to either create new DNS domains or run services at ports different from port 80 and 443. New domains cost money, need to be registered with DNS servers and are long-lived. Creating new ports for web services does not work: first, network firewalls block atypical ports and Internet Explorer doesn't include the port in determining an application's origin

fixed limit on the number of partitions or components creates an artificial and unnecessary limitation. Different applications require differing number of components, and a "one-size-fits-all" approach does not work. We show that, as a result, HTML5 applications in such platforms have large amounts of code running with unnecessary privileges, which increases the impact from attacks like cross-site scripting. Second, browser re-design has a built-in deployment and adoption cost and it takes significant time before applications can enjoy the benefits of privilege separation.

In this paper, we rethink how to achieve privilege separation in HTML5 applications. In particular, we propose a solution that does not require any platform changes and is orthogonal to privilege separation architectures enforced by the underlying browsers. Our proposal utilizes standardized primitives available in today's web browsers, requires no additional web domains and improves the auditability of HTML5 applications. In our proposal, HTML5 applications can create an arbitrary number of "unprivileged components." Each component executes in its own *temporary origin* isolated from the rest of the components by the SOP. For any privileged call, the unprivileged components communicate with a "privileged" (parent) component, which executes in the main (permanent) origin of the web application. The privileged code is small and we ensure its integrity by enforcing key security invariants, which we define in Section 3. The privileged code mediates all access to the critical resources granted to the web application by the underlying browser platform, and it enforces a fine-grained policy on all accesses that can be easily audited. Our proposal achieves the same security benefits in ensuring application integrity as enjoyed by desktop applications with process isolation and sandboxing primitives available in commodity OSes [36, 41, 46].

We show that our approach is practical for existing HTML5 applications. We retrofit two widely used Google Chrome extensions and a popular HTML5 application for SQL database administration to use our design. In our case studies, we show that the amount of trusted code running with full privileges reduces by a factor of 6 to 10000. Our architecture does not sacrifice any performance as compared to alternative approaches that redesign the underlying web browser. Finally, our migration of existing applications requires minimal changes to code. For example, in porting our case studies to this new design we changed no more than 13 lines of code in any application. Developers do not need to learn new languages or type safety primitives to migrate code to our architecture, in contrast to recent proposals [29]. We also demonstrate strong data confinement policies. To encourage adoption, we have released our core infrastructure code as well as the case studies (where permitted) and made it all freely available online [47]. We are currently collaborating with the Google Chrome team to apply this approach to secure Chrome applications, and our design has influenced the security architecture of upcoming Chrome applications.

In our architecture, HTML5 applications can define more expressive policies than supported by existing HTML5 platforms, namely the Chrome extension platform [7] and the Windows 8 Metro platform [38]. Google Chrome and Windows 8 rely on applications declaring install-time permissions that end users can check [9]. Multiple studies have found permission systems to be inadequate: the bulk of popular applications run with powerful permissions [3, 21] and users rarely check install-time permissions [20]. In our architecture, policy code is explicit and clearly separated, can take into account runtime ordering of privileged accesses, and can be more fine-grained. This design enables expert auditors, such as maintainers of software application galleries, to reason about the security of applications. In our case studies, these policies are typically a small amount of static JavaScript code, which is easily auditable.

2 Problem and Approach Overview

Traditional HTML applications execute with the authority of their "web origin" (protocol, port, and domain). The browser's same origin policy (SOP) isolates different web origins from one another and from the file system. However, applications rarely rely on domains for isolation, due to the costs associated with creating new domains or origins.

In more recent application platforms, such as the Google Chrome extension platform [9], Chrome packaged web application store [25] and Windows 8 Metro applications [38], applications can execute with enhanced privileges. These privileges, such as access to the geo-location, are provided by the underlying platform through *privileged APIs*. Applications utilizing these privileged API explicitly declare their *permissions* to use privileged APIs at install time via manifest files. These applications are authored using the standard HTML5 features and web languages (like JavaScript) that web applications use; we use the term *HTML5 applications* to collectively refer to web applications and the aforementioned class of emerging applications.

Install-time manifests are a step towards better security. However, these platforms still limit the number of application components to a finite few and rely on separate origins to isolate them. For example, each Google Chrome extension has three components. One component executes in the origin of web sites that the extension interacts with. A second component executes with the extension's permanent origin (a unique public key assigned to it at creation time). The third component exe-

cutes in an all-powerful origin having the authority of the web browser. In this section, we show how this limits the degree of privilege separation for HTML5 applications in practice.

2.1 Issues with the Current Architecture

In this section, we point out two artifacts of today's HTML5 applications: *bundling* of privileges and *TCB inflation*. We observe that these issues are rooted in the fact that, in these designs, the ability to create new web origins (or security principals) is severely restricted.

Common vulnerabilities (like XSS and mixed content) today actually translate to powerful gains for attackers in current architectures. Recent findings corroborate the need for better privilege separation—for instance, 27 out of 100 Google Chrome extensions (including the top 50) recently studied have been shown to have exploitable vulnerabilities [14]. These attacks grant powerful privileges like code execution in *all* HTTP and HTTPS web sites and access to the user's browsing history.

As a running example, we introduce a hypothetical extension for Google Chrome called ScreenCap. ScreenCap is an extension for capturing screenshots that also includes a rudimentary image editor to annotate and modify the image before sending to the cloud or saving to a disk.

Bundling. The ScreenCap extension consists of two functionally disjoint components: a screenshot capturing component and an image editor. In the current architecture, both the components run in the same principal (origin), despite requiring disjoint privileges. We call this phenomenon *bundling*. The screenshot component requires the `tabs` and `<all_urls>` permission, while the image editor only requires the `pictureLibrary` permission to save captured images to the user's picture library on the cloud.

Bundling causes over-privileged components. For example, the image editor component runs with the powerful `tabs` and `<all_urls>` permission. In general, if an application's components require privilege sets α_1, α_2..., *all* components of the application run with the privileges $\bigcup \alpha_i$, leading to over-privileging. As we show in Section 5.4, 19 out of the Top 20 extensions for the Google Chrome platform exhibit bundling. As discussed earlier, this problem manifests on the web too.

TCB inflation. Privileges in HTML5 are ambient—all code in a principal runs with full privileges of the principal. In reality, only a small application core needs access to these privileges and rest of the application does not need to be in the trusted computing base (TCB). For example, the image editor in ScreenCap consists of a number of complex and large UI and image manipulation libraries. All this JavaScript code runs with the ambient privilege to write to the user's picture library. Note that this is in addition to it running bundled with the privileges of the screenshot component.

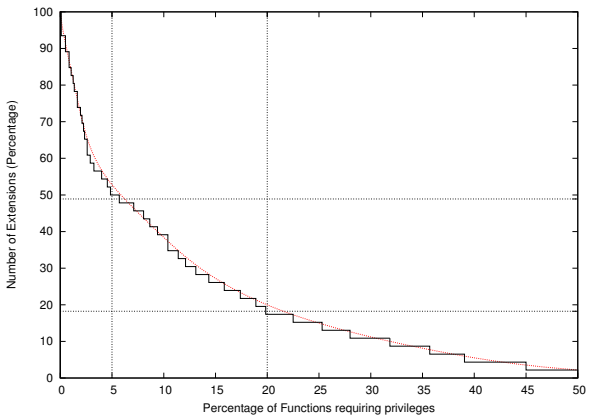

Figure 1: CDF of percentage of functions in an extension that make privileged calls (X axis) vs. the fraction of extensions studied (in percentage) (Y axis). The lines for 50% and 20% of extensions as well as for 5% and 20% of functions are marked.

We measured the TCB inflation for the top 50 Chrome extensions. Figure 1 shows the percentage of total functions in an extension requiring privileges as a fraction of the total number of static functions. In half the extensions studied, less than 5% of the functions actually need any privileges. In 80% of the extensions studied, less than 20% of the functions require any privileges.

Summary. It is clear from our data that HTML5 applications, like Chrome extensions, do not sufficiently isolate their sub-components. The same-origin policy equates web origins and security principals, and web origins are fixed at creation time or tied to the web domain of the application. All code from a given provider runs under a single principal, which forces privileges to be ambient. Allowing applications to cheaply create as many security principals as necessary and to confine them with fine-grained, flexible policies can make privilege separation more practical.

Ideally, we would like to isolate the image editor component from the screenshot component, and give each component exactly the privileges it needs. Moving the complex UI and image manipulation code to an unprivileged component can tremendously aid audit and analysis. Our first case study (Section 5.1) discusses unbundling and TCB reduction on a real world screenshot application. We achieved a 58x TCB reduction.

2.2 Problem Statement

Our goal is to design a new architecture for privilege separation that side steps the problem of scarce web origins and enables the following properties:

Reduced TCB. Given the pervasive nature of code in-

jection vulnerabilities, we are interested, instead, in reducing the TCB.

Ease of Audit. Dynamic code inclusion and use of complex JS constructs is pervasive. An architecture that eases audits, in spite of these issues, is necessary.

Flexible policies. Current manifest mechanisms provide insufficient contextual data for meaningful security policies. A separate flexible policy mechanism can ease audits and analysis.

Reduce Over-privileging. Bundling of disjoint applications in the same origin results in over-privileging. We want an architecture that can isolate applications agnostic of origin.

Ease of Use. For ease of adoption, we also aim for minimal compatibility costs for developers. Mechanisms that would involve writing applications for a new platform are outside scope.

Scope. We focus on the threat of vulnerabilities in *benign* HTML5 application. We aim to enable a privilege separation architecture that benign applications can utilize with ease to provide a strong second line of defense. We consider malicious applications as out of scope, but our design improves auditability and may be applicable to HTML5 malware in the future.

This paper strictly focuses on mechanisms for achieving privilege separation and on mechanisms for expressive policy-based confinement. Facilitating policy development and checking if policies are reasonable is an important issue, but beyond the scope of this paper.

3 Design

We describe our privilege separation architecture in this section. We describe the key security invariants we maintain in Section 3.2 and the mechanisms we use for enforcing them in Section 3.3.

3.1 Approach Overview

We advocate a design that is independent of any privilege separation scheme enforced by the underlying browser. In our design, HTML5 applications have one *privileged parent* component, and can have an arbitrary number of *unprivileged children*. Each child component is spawned by the parent and it executes in its own *temporary* origin. These temporary origins are created on the fly for each execution and are destroyed after the child exits; we detail how temporary origins can be implemented using modern web browsers primitives in Section 3.3. The privileged parent executes in the main (permanent) origin assigned to the HTML5 application, typically the web origin for traditional web application. The same origin policy isolates unprivileged children from one another and from the privileged parent. Figure 2 shows our proposed HTML5 application architecture. In our design,

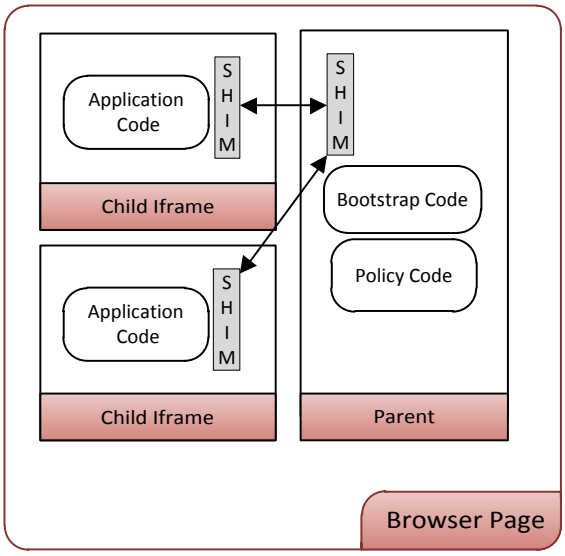

Figure 2: High-level design of our proposed architecture.

applications can continue to be authored in existing web languages like JavaScript, HTML and CSS. As a result, our design maintains compatibility and facilitates adoption.

Parent. Our design ensures the integrity of the privileged parent by maintaining a set of key *security invariants* that we define in Section 3.2. The parent guards access to a powerful API provided by the underlying platform, such as the Google Chrome extension API. For making any privileged call or maintaining persistent data, the unprivileged children communicate with the parent over a thin, well-defined *messaging interface*. The parent component has three components:

- *Bootstrap Code.* When a user first navigates to the HTML5 application, a portion of the parent code called the bootstrap code executes. Bootstrap code is the *unique* entry point for the application. The bootstrap code downloads the application source, spawns the unprivileged children in separate temporary origins, and controls the lifetime of their execution. It also includes boilerplate code to initialize the messaging interface in each child before child code starts executing. Privileges in HTML5 applications are tied to origins; thus, a temporary origin runs with no privileges. We explain temporary origins further in Section 3.3.

- *Parent Shim.* During their execution, unprivileged children can make privileged calls to the parent. The parent shim marshals and unmarshals these requests to and from the children. The parent shim also presents a usable interface to the policy code component of the parent.

- *Policy Code.* The policy code enforces an application-specific policy on *all* messages received from children. Policy code decides whether to allow or disallow access to privileged APIs, such as access to the user's browsing history. This mechanism provides complete mediation on access to privileged APIs and supports fine-grained policies, similar to system call monitors in commodity OSes like SysTrace [41]. In addition, as part of the policy code, applications can define additional restrictions on the privileges of the children, such as disabling import of additional code from the web.

Only the policy code is application-specific; the bootstrap and parent shim are the same across all applications. To ease adoption, we have made the application-independent components available online. The application independent components need to be verified once for correctness and can be reused for all application in the future. For new applications using our design, only the application's policy code needs to be audited. In our experimental evaluation, we find that the parent code is typically only a small fraction of the rest of the application and our design invariants make it statically auditable.

Children. Our design moves all functional components of the application to the children. Each child consists of two key components:

- *Application Code*. Application code starts executing in the child after the bootstrap code initializes the messaging interface. All the application logic, including code to handle visual layout of the application, executes in the unprivileged child; the parent controls no visible area on the screen. This implies that all dynamic HTML (and code) rendering operations execute in the child. Children are allowed to include libraries and code from the web and execute them. Vulnerabilities like XSS or mixed content bugs (inclusion of HTTP scripts in HTTPS domains) can arise in child code. In our threat model, we assume that children may be compromised during the application's execution.

- *Child Shim*. The parent includes application independent shim code into the child to seamlessly allow privileged calls to the parent. This is done to keep compatibility with existing code and facilitate porting applications to our design. Shim code in the child defines wrapper functions for privileged APIs (e.g., the Google Chrome extension API [24]). The wrapper functions forward any privileged API calls as messages to the parent. The parent shim unmarshals these messages, checks the integrity of the message and executes the privileged call if allowed by the policy. The return value of the privileged API call is marshaled into messages by the parent shim and returned to the child shim. The child shim unmarshals the result and returns it to the original caller function in the child. Certain privileged API functions take callbacks or structured data objects; in Section 4.1 we outline how our mechanism proxies these transparently. Together, the parent and child shim hide the existence of the privilege boundary from the application code.

3.2 Security Invariants

Our security invariants ensure the integrity and correctness of code running in the parent with full privileges. We do not restrict code running in the child; our threat model assumes that unprivileged children can be compromised any time during their execution. We enforce four security invariants on the parent code:

1. The parent cannot convert any string to code.

2. The parent cannot include external code from the web.

3. The parent code is the only entry point into the privileged origin.

4. Only primitive types (specifically, strings) cross the privilege boundary.

The first two invariants help increase assurance in the parent code. Together, they disable dynamic code execution and import of code from the web, which eliminates the possibility of XSS and mixed content vulnerabilities in parent code. Furthermore, it makes parent code statically auditable and verifiable. Several analysis techniques can verify JavaScript when dynamic code execution constructs like `eval` and `setTimeout` have been syntactically eliminated [5, 17, 26, 28, 37].

Invariant 3 ensures that *only* the trusted parent code executes in the privileged origin; no other application code should execute in the permanent origin. The naive approach of storing the unprivileged (child) code as a HTML file on the server suffers from a subtle but serious vulnerability. An attacker can directly navigate to the unprivileged code. Since it is served from the same origin as the parent, it will execute with full privileges of the parent without going through the parent's bootstrap mechanism. To prevent such escalation, invariant 3 ensures that all entry points into the application are directed only through the bootstrap code in the parent. Similarly, no callbacks to unprivileged code are passed to the privileged API—they are proxied by parent functions to maintain Invariant 3. We detail how we enforce this invariant in Section 3.3.

Privilege separation, in and of itself, is insufficient to improve security. A problem in privilege-separated C applications is the exchange of pointers across the privilege boundary, leading to possible errors [23, 49]. While JavaScript does not have C-style pointers, it has first-class functions. Exchanging functions and objects across the privilege boundary can introduce security vulnerabilities. Invariant 4 eliminates such attacks by requiring that only primitive strings are exchanged across the privilege boundary.

3.3 Mechanisms

We detail how we implement the design and enforce the above invariants in this section. Whenever possible, we rely on browser's mechanisms to declaratively enforce the outlined invariants, thereby minimizing the need for code audits.

Temporary Origins. To isolate components, we execute unprivileged children in separate `iframes` sourced from temporary origins. A temporary origin can be created by assigning a fresh, globally unique identifier that the browser guarantees will never be used again [8]. A temporary origin does not have any privileges, or in other words, it executes with null authority. The globally unique nature means that the browser isolates every temporary origin from another temporary origin, as well as the parent. The temporary origin only lasts as long as the lifetime of the associated `iframe`.

Several mechanisms for implementing temporary origins are available in today's browsers, but these are rarely found in use on the web. In the HTML5 standard, iframes with the `sandbox` directive run in a temporary origin. This primitive is standardized and already supported in shipping versions of Google Chrome/ChromeOS, Safari, Internet Explorer/Windows 8, and a patch for Mozilla Firefox is in the final stages of review [13].

Enforcement of Security Invariants. To enforce security invariants 1 and 2 in the parent, our implementation utilizes the Content Security Policy (CSP) [48]. CSP is a new specification, already supported in Google Chrome and Firefox, that defines browser-enforced restrictions on the resources and execution of application code. In our case studies, it suffices to use the CSP policy directive `default-src 'none'; script-src 'self'`—this disables *all* constructs to convert strings into code (**Invariant 1**) and restricts the source of all scripts included in the page to the origin of the application (**Invariant 2**). We find that application-specific code is typically small (5 KB) and easily auditable in our case studies. On platforms on which CSP is not supported, we point out that disabling code evaluation constructs and external code import is possible by syntactically restricting the application language to a subset of JavaScript [26, 28, 37].

We require that all non-parent code, when requested, is sent back as a text file. Browsers do not execute text files—the code in the text files can only execute if downloaded and executed by the parent, via the bootstrap mechanism. This ensures **Invariant 3**. In case of pure client-side platforms like Chrome, this involves a simple file renaming from `.html` to `.txt`. In case of classic client-server web applications, this involves returning a `Content-Type` header of `text/plain`. To disable mime-sniffing, we also set the `X-Content-Type-Options` HTTP header to `nosniff`.

Messaging Interface. We utilize standard primitives like XMLHttpRequest and the DOM API for downloading the application code and executing it in an iframe. We rely on the `postMessage` API for communication across the privilege boundary. `postMessage` is an asynchronous, cross-domain, purely client-side messaging mechanism. By design, `postMessage` only accepts primitive strings. This ensures **Invariant 4**.

Policy. Privilege separation isolates the policy and the application logic. Policies, in our design, are written in JavaScript devoid of any dynamic evaluation constructs and are separated from the rest of the complex application logic. Permissions on existing browser platforms are granted at install-time. In contrast, our design allows for more expressive and fine-grained policies like granting and revoking privileges at run-time. For example, in the case of ScreenCap, a child can get the ability to capture a screenshot only once and only after the user clicks the 'capture' button. Such fine-grained policies require the policy engine to maintain state, reason about event ordering and have the ability to grant/revoke fine-grained privileges. Our attempt at expressive policies is along the line of active research in this space [29], but in contrast to existing proposals, it does not require developers to specify policies in new high-level languages. Our focus is on mechanisms to support expressive policies; determining what these policies should be for applications is beyond the scope of this paper.

Additional Confinement of Child Code. By default, no restrictions are placed on the children beyond those implied by use of temporary origins. Specifically, the child does *not* inherit the parent's CSP policy restrictions. In certain scenarios, the application developer *may* choose to enforce additional restrictions on the child code, via an appropriate CSP policy on the child `iframe` at the time of its creation by the parent code. For example, in the case of ScreenCap, the screenshot component can be run under the `script-src 'self'`. This increases assurance by disabling inline scripts and code included from the web, making XSS and mixed content attacks impossible. The policy code can then grant the

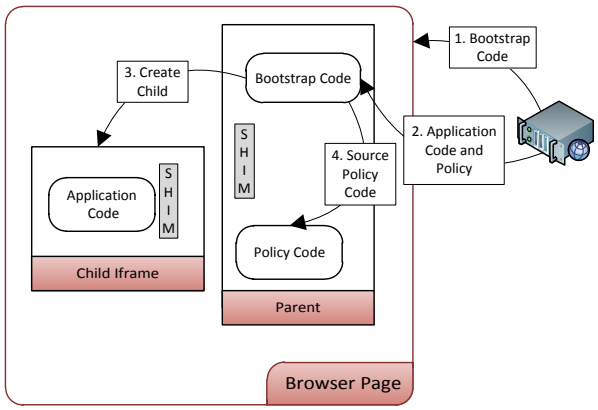

Figure 3: Sequence of events to run application in sandbox. Note that only the bootstrap code is sent to the browser to execute. Application code is sent directly to the parent, which then creates a child with it.

```
var sb_content="<html><head>";
sb_content+="<meta http-equiv='X-WebKit-CSP'";
//csp_policy is defined in downloaded policy
sb_content+="content='"+csp_policy+"'>";
sb_content+="<script src='"+childShimSrc+"'>";
//the baseurl is current window uri
//so that relative URIs work
sb_content+="<base href='"+baseurl+"'>";
//contents of app.txt
sb_content+=application_code;
// attribute values are URI-decoded
// by HTML parser
sb_content=encodeURIComponent(sb_content);
var fr = document.createElement("iframe");
fr.src="data:text/html;charset=utf-8,"+
    sb_content;
//sandboxed frames run in fresh origin
fr.setAttribute('sandbox','allow-scripts');
document.body.appendChild(fr);
```

Listing 1: Bootstrap Code (JavaScript)

powerful privilege of capturing a screenshot of a user's webpage to a high assurance screenshot component.

4 Implementation

As outlined in Section 3, the parent code executes when the user navigates to the application. The bootstrap code is in charge of creating an unprivileged sandbox and executing the unprivileged application code in it. The shim code and policy also run in the parent, but we focus on the bootstrap and shim code implementation in this section. The unprivileged child code and the security policy vary for each application, and we discuss these in our case studies (Section 5).

Figure 3 outlines the steps involved in creating one unprivileged child. First, the user navigates to the application and the parent's bootstrap code starts executing (**Step 1** in Figure 3). In **Step 2**, the parent's bootstrap code retrieves the application HTML code (as plain text files) as well as the security policy of the application. For client-side platforms like Chrome and Windows 8, this is a local file retrieval.

The parent proceeds to create a temporary origin, unprivileged iframe using the downloaded code as the source (**Step 3**, Figure 3). Listing 1 outlines the code to create the unprivileged temporary origin. The parent builds up the child's HTML in the `sb_content` variable. The parent can optionally include content restrictions on the child via a CSP policy, as explained in Section 3.3. Creating multiple children is a simple repetition of the step 3.

The parent also sources the child shim into the child `iframe`. The parent concatenates the child's code (HTML) and URI-encodes it all into a variable called `sb_content`. The parent creates an `iframe` with `sb_content` as the `data:` URI source, sets the sandbox attribute and appends the iframe to the document. The parent code also inserts a `base` HTML tag that enables relative URIs to work seamlessly.

`data:` is a URI scheme that enables references to inline data as if it were an external reference. For example, an iframe with src attribute set to `data:text/html;Hi` is similar to an iframe pointing to an HTML page containing only the text 'Hi'. Recall our enforcement mechanism for Invariant 3: the application code is a text file. The use of `data:` is necessary to convert text to code that the `iframe src` can point to, without storing unprivileged application code as HTML or JavaScript files.

4.1 API Shims

Recall that the child executes in a temporary origin, without the privileges needed for making privileged calls like `chrome.tabs.captureVisibleTab`. Privileged API calls in the original child code would fail when it executes in a temporary origin; our transformation should, therefore, take additional steps to preserve the original functionality of the application. In our design, we propose API shims to proxy calls to privileged API in the child to the parent code safely and transparently.

The child shim defines wrapper objects in the child that proxy a privileged call to the parent. The aim of the parent and child shim is to make the privilege separation boundary transparent. We have implemented shims for all the privileged API functions needed for our case studies. This implementation of the parent shim is 5.46 KB and that of the child shim is 9.1 KB. Note that only the parent shim is in the TCB.

Figure 4 outlines the typical events involved in proxying a privileged call. First, the child shim defines a stub implementation of the privileged APIs (e.g., `chrome.tabs.captureVisibleTab`) that, when called, forwards the call to the parent. On receiving the message, the parent shim checks with the policy and if

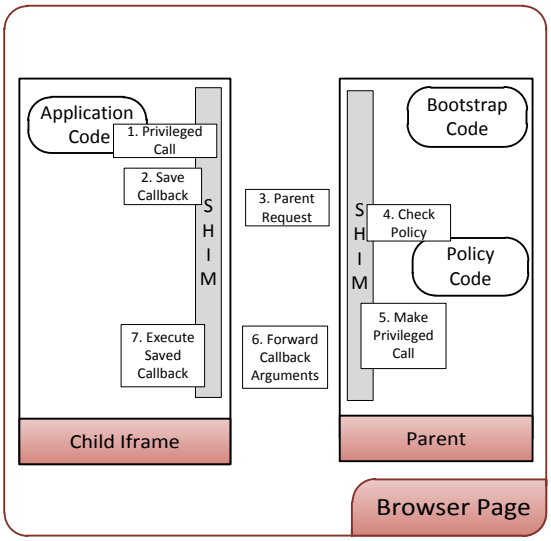

Figure 4: Typical events for proxying a privileged API call. The numbered boxes outline the events. The event boxes span the components involved. For example, event 4 involves the parent shim calling the policy code.

the policy allows, the parent shim makes the call on behalf of the child. On completion of the call, the parent shim forwards the callback arguments (given by the runtime) to the child shim, and the child shim executes the original callback.

Continuing with our running example, we give concrete code examples of the shims for the `chrome.tabs.captureVisibleTab` function, used to capture a screenshot. `captureVisibleTab` takes three arguments: a windowID, an options object, and a callback parameter. On successfully capturing a screenshot of the given window, the chrome runtime executes the callback with the encoded image data as the only argument. Note that the callback parameter is a first-class function; our invariants do not allow exchange of a function across the privilege boundary.

Child Shim. The child shim creates a stub implementation of the privileged API. In the unprivileged child, a privileged call would fail since the child does not have privileges to execute it. Instead, the stub function defined by the child function is called. This stub function marshals all the arguments and sends it to the parent. Listing 2 is the child shim implementation for the `captureVisibleTab` function.

No code is passed across the privilege boundary. Instead, the child saves the callback (Step 2 in Fig. 4) and forwards the rest of the argument list to the parent (Step 3). The callback is stored in a cache and a unique identifier is sent to the parent. The parent uses this identifier later.

```
tabs.captureVisibleTab =
 function(windowid,options,callback){
   var id =callbackctr++;
   cached_callbacks[id] = callback;
  sendToParent({
    "type":"tabs.captureVisibleTab",
    "windowid":windowid,
    "options":options,
    "callbackid":id
     });
};
```

Listing 2: Child shim for captureVisibleTab

```
//m is the argument given to
// sendToParent in the child shim
if(m.type==='tabs.captureVisibleTab')
{//fail if policy does not allow
if(!policy.allowCall(m){ return;}
tabs.captureVisibleTab(
 m.windowid,
 m.options,
 function(imgData){
  sendToChild({
    type:"cb_tabs.captureVisibleTab",
    id:m.callbackid,
    imgData: imgData
    });
 });
}
```

Listing 3: Parent shim for captureVisibleTab

We stress that this process is transparent to the application: the parent code ensures that the child shim is loaded before any application code starts executing. The application can continue calling the privileged API as before.

Parent Shim. On receiving the message, the parent's shim first checks with the policy (Step 4 in Fig. 4 and line 5 in Listing 3) and if the policy allows it, the parent shim makes the requested privileged call.

In case of ScreenCap, a simple policy could disallow `captureVisibleTab` call if the request came from the image editor, and allow the call if the request came from the screenshot component. Such a policy unbundles the two components. If a network attacker compromises one of the two components in ScreenCap, then it only gains the ability to make request already granted to that component. As another example, the application can enforce a policy to only allow one `captureVisibleTab` call after a user clicks the 'capture' button. All future requests during that execution of the application are denied until the user clicks the 'capture' button again.

Note that the privileged call is syntactically the same as what the child would have made, except for the callback. The modified callback (lines 9-14 in Listing 3) forwards the returned image data to the child (Step 6), the original callback still executes in the child.

Child Callback The message handler on the child receives the forwarded arguments from the parent and exe-

```
if(
 m.type==='cb_tabs.captureVisibleTab'
 ){
 var cb_id = m.callbackid;
 var savedCb = cached_callbacks[cb_id
     ];
 savedCb.call(window,m.imgData);
 delete cached_callbacks[cb_id];
 }
```

Listing 4: Child shim for captureVisibleTab: Part 2

cutes the saved callback with the arguments provided by the parent. (Step 7 in Figure 4 and line 6 in Listing 4). The saved callback is then deleted from the cache (Line 7).

Persistent State. We take a different approach to data persistence APIs like `window.localStorage` and `document.cookie`. It is necessary that the data stored using these APIs is also stored in the parent since the next time a child is created, it will run in a fresh origin and the previous data will be lost. We point out that enabling persistent storage while maintaining compatibility requires some changes to code. Persistent storage APIs (like `window.localStorage`) in today's platforms are synchronous; our proxy mechanism uses `postMessage` to pass persistent data, but `postMessage` is asynchronous. To facilitate compatibility, we implement a wrapper for these synchronous API calls in the child shim code and asynchronously update the parent via `postMessage` underneath. For example, a part of the localStorage child shim is presented in Listing 5. The shim creates a wrapper for the `localStorage` API using an associative array (viz., `data`). On every update, the new associative array is sent to the parent. On receiving the `localStorage_save` message, the parent can save the data or discard it per policy.

We observe that in our transformation, calls to API that access persistent state become asynchronous which contrasts the synchronous API calls in the original code. To preserve the application's intended behavior, in principle, it may be necessary to re-design parts of the code that depend on the synchronous semantics of persistent storage APIs—for example, when more than one unprivileged children are sharing data via persistent state simultaneously. In our case studies so far, however, we find that the application behavior does not depend on such semantics. In future work, we plan to investigate transformation mechanisms that can provide reasonable memory consistency properties in accessing persistent local storage.

5 Case Studies

We retrofit our design onto three HTML5 applications to demonstrate that our architecture can be adopted by applications today:

```
  setItem: function (key, value) {
     data[key] = value+'';
     saveToMainCache(data);
   },

  saveToMainCache: function(data){
     sendToParent({
     "type":"localStorage_save",
     "value":data
     });
       },
```

Listing 5: localStorage Shim in the Child Frame

- As an example of browser extensions, we retrofit our design to Awesome Screenshot, a widely used chrome extension (802,526 users) similar to ScreenCap.
- As an example of emerging packaged HTML5 web applications, we retrofit our design to SourceKit, a full-fledged text editor available as a Chrome packaged web application. SourceKit's design is similar to editors often bundled with online word processors and web email clients. These editors typically run with the full privileges of the larger application they accompany.
- As an example of traditional HTML5 web applications, we retrofit our design to SQL Buddy, a PHP web application for database administration. Web interfaces for database administration (notably, PHPMyAdmin) are pervasive and run with the full privileges of the web application they administer.

Our goal in this evaluation is to measure (a) the reduction in TCB our architecture achieves, (b) the amount of code changes necessary to retrofit our design, and (c) performance overheads (user latency, CPU overheads and memory footprint impact) compared to platform redesign approaches. Table 1 lists our case studies and summarizes our results. First, we find that the TCB reduction achieved by our redesign ranges from 6x to 10000x. Due to the prevalence of minification, we believe LOC is not a useful metric for JavaScript code and, instead, we report the size of the code in KB. Second, we find that we require minimal changes, ranging from 0 to 13 lines, to port the case studies to our design. This is in addition to the application independent shim and bootstrap code that we added.

We also demonstrate examples of expressive policies that these applications can utilize. The focus of this paper is on mechanisms, not policies, and we do not discuss alternative policies in this work.

Finally, we also quantify the reduction in privileges we would achieve in the 50 most popular Chrome extensions with our architecture. We also find that in half the extensions studied, we can move 80% of the functions

out of the TCB. This quantifies the large gap between the privileges granted by Chrome extensions today and what is necessary. In addition, we also analyze the top 20 Chrome extensions to determine the number of components bundled in each. We find that 19 out of the top 20 extension exhibit bundling, and estimate that we can separate these between 2 to 4 components, in addition to the three components that Chrome enforces.

To facilitate further research and adoption of our techniques, we make all the application independent components of the architecture and the SQL Buddy case study available online [47]. Due to licensing restrictions, we are unable to release the other case studies publicly.

Table 1: Overview of case studies. The TCB sizes are in KB. The lines changed column only counts changes to application code, and not application independent shims and parent code.

Application	Number of users	Initial TCB (KB)	New TCB (KB)	Lines Changed
Awesome Screenshot	802,526	580	16.4	0
SourceKit	14,344	15,000	5.38	13
SQL Buddy	45,419	100	2.67	11

5.1 Awesome Screenshot

The Awesome Screenshot extension allows a user to capture a screenshot of a webpage similar to our running example [18]. A rudimentary image editor, included in the extension, allows the user to annotate and modify the captured image as he sees fit. Awesome Screenshot has over 800,000 users.[3]

The extension consists of three components: `background.html`, `popup.html`, and `editor.html`. A typical interaction involves the user clicking the Awesome Screenshot button, which opens `popup.html`. The user selects her desired action; `popup.html` forwards the choice to `background.html`, which captures a screenshot and sends it to the image editor (`editor.html`) for post-processing. All components communicate with each other using the `sendRequest` API call.

Privilege Separation. We redesigned Awesome Screenshot following the model laid out in Section 3 (Figure 2). Each component runs in an unprivileged temporary origin. The parent mediates access to privileged APIs, and the policy keeps this access to the minimum required by the component in question.

[3]Due to a bug in Chrome, the current Awesome Screenshot extension uses a NPAPI binary to save big ($>$ 2MB) images. We used the HTML5 version (which doesn't allow saving large files) for the purposes of this work. This is just a temporary limitation.

Code Changes. Apart from the application independent code, we required no changes to the code. The parent and child shims make the redesign seamless. We manually tested the application functionality thoroughly and did not observe any incompatibilities.

Unbundling. In the original version of Awesome Screenshot, the image editor (`editor.html`) accepts the image from `background.html` and allows the user to edit it, but runs with the full privileges of the extension—an example of bundling. Similarly, the `popup.html` only needs to forward the user's choice to `background.html` but runs with all of the extension's privileges.

In our privilege-separated implementation of Awesome Screenshot, the editor code, stored in `editor.txt` now, runs within a temporary origin. The policy only gives it access to the `sendRequest` API to send the `exit` and `ready` messages as well as receive the image data message from the background page.

TCB Reduction. The image editor in the original Awesome Screenshot extension uses UI and image manipulation libraries (more than 500KB of complex code), which run within the same origin as the extension. As a result, these libraries run with the ambient privileges to take screenshots of any page, log the user's browsing history, and access the user's data on any website. While some functions in the extension do need these privileges, the complete codebase does not need to run with these privileges.

In our privilege-separated implementation of Awesome Screenshot, the amount of code running with full privileges (TCB) decreased by a factor of 58. We found the UI and image manipulation libraries, specifically jQuery UI, used dynamic constructs like `innerHTML` and `eval`. Our design moves these potentially vulnerable constructs to an unprivileged child.

The code in the child can still request privileged function calls via the interface provided by the parent. However, this interface is thin, well defined and easily auditable. In contrast, in the non-privilege separated design, the UI and image libraries run with ambient privileges. In contrast, in the original extension *all* the code needs to be audited.

Example Policy. In addition to unbundling the image editor from the screenshot component, the parent can enforce stronger, temporal policies on the application. In particular, the parent can require that the `captureVisibleTab` function is only called once after the user clicks the `capture` button. Any subsequent calls have to be preceded by another button click. Such temporal policies are impossible to express and enforce in current permission-based systems.

5.2 SourceKit Text Editor

The SourceKit text editor is an HTML5 text editor for a user's documents stored on the Dropbox cloud service [19]. It uses open source components like the Ajax.org cloud editor [1] and Dojo toolkit [50], in conjunction with the Dropbox REST APIs [19].

SourceKit is a powerful text editor. It includes a file-browser pane and can open multiple files at the same time. The text editor component supports themes and syntax highlighting. The application consists of 15MB of JavaScript code, all of which runs with full privileges.

Privilege Separation. In our least privilege design, the whole application runs in a single child. Redesigning SourceKit to move code to an unprivileged temporary origin was seamless because of the library shims (Section 4.1). One key change was replacing the included Dojo toolkit with its asynchronous version. The included Dojo toolkit uses synchronous `XMLHttpRequest` calls, which the asynchronous `postMessage` cannot proxy. The asynchronous version of Dojo is freely available on the Dojo website. We do not include this change in the number of lines modified in Table 1.

Unbundling. Functionally, SourceKit is a single Chrome application, and no bundling has occurred in its design. Popular Web sites (like GitHub [31]), use the text editor module as an online text editor [1]. In such cases, the text editor runs *bundled* with the main application, inheriting the application's privileges and increasing its attack surface. While we focus only on SourceKit for this case study, our redesign directly applies to these online text editors.

TCB Reduction. In our privilege separated SourceKit, the amount of code running with full privileges reduced from 15MB to 5KB. A large part of this reduction is due to moving the Dojo Toolkit, the syntax highlighting code and other UI libraries to an unprivileged principal. Again, we found the included libraries, specifically the Dojo Toolkit, relying on dangerous, dynamic constructs like `eval`, string arguments to `setInterval`, and `innerHTML`. In our redesign, this code executes unprivileged.

Code Change. In addition to the switch to asynchronous APIs, we also had to modify one internal function in SourceKit to use asynchronous APIs. In particular, SourceKit relied on synchronous requests to load files from the `dropbox.com` server. We modified SourceKit to use an asynchronous mechanism instead. The change was minor; only 13 lines of code were changed.

Example Policy. In the original application, all code runs with the tabs permission, which allows access to the user's browsing history, and permission to access dropbox.com. In our privilege-separated design, the policy only allows the child access to the `tabs.open` and `tabs.close` Chrome APIs for accessing `dropbox.com`. Similarly, it only forwards tab events for `dropbox.com` URIs. Thus, after the redesign, the child has access to the user's browsing history *only* for `dropbox.com`, and not for all websites. Implementing this policy requires only two lines of code—an if condition that forwards events only for `dropbox.com` domains suffices.

SourceKit accesses Dropbox using the Dropbox OAuth APIs [19]. At first run, SourceKit opens Dropbox in a new tab, where the user can grant SourceKit the requisite OAuth access token [39]. The parent can only allow access to the tabs privileges at first run, and disable it once the child receives the OAuth token. Such temporal policies cannot be expressed by install-time permissions implemented in existing platforms.

We can also enforce stronger policies to provide a form of data separation [12]. By default, the Dropbox JS API [34] stores the OAuth access token in localStorage, accessible by all the code in the application. Instead, the policy code can store the OAuth token in the parent and append it to all `dropbox.com` requests. This mitigates data exfiltration attacks where the attacker can steal the OAuth token to bypass the parent's policy.[4] Such application-specific data-separation policies cannot be expressed in present permission systems.

5.3 SQL Buddy

SQL Buddy is an open source tool to administer the MySQL database using a Web browser. Written in PHP, SQL Buddy is functionally similar to phpMyAdmin and supports creating, modifying, or deleting databases, tables, fields, or rows; SQL queries; and user management.

SQL Buddy uses the MooTools JS library to create an AJAX front-end for MySQL administration. It uses the MySQL user table for authentication and logged-in users maintain authentication via PHP session cookies.

Privilege Separation. We modified SQL Buddy to execute all its code in an unprivileged child. To ensure that no code is interpreted by the browser, we required all PHP files to return a Content-Type header of `text/plain`, as discussed in Section 3.3. Only two new files: `buddy.html` and `login.html` execute in the browser; these are initialized by the bootstrap code.

Unbundling. A typical SQL Buddy installation runs at `www.example.net/sqlbuddy`, and helps ease database management for the application at `www.example.net`. Classic operating system mechanisms can isolate SQL Buddy and the main application on the server side. But SQL Buddy runs with the full privileges of the applica-

[4] For example, to prevent malware, the parent can require that all files accessed using SourceKit have non-binary file extensions.

tion on the client-side. In particular, an XSS vulnerability in SQL Buddy is equivalent to an XSS vulnerability on the main application: it is not isolated from the application at the client-side. SQL Buddy inherits all the privileges of the application, including special client-side privileges such as access to camera, geolocation, and ambient privileges granted to the web origin such as the ability to do cross-origin XMLHttpRequests [51].

In our privilege-separated redesign, a restrictive policy on the child mitigates SQL Buddy bundling. The parent allows the child `XMLHttpRequest` access to only `/sqlbuddy/<filename>.php` URIs. No other privilege is available to SQL Buddy code, including `document.cookie`, `localStorage`, or `XMLHttpRequest` to the main application's pages. This policy isolates SQL Buddy from any other application executing on the same domain, a hitherto unavailable option.

Code Change. The key change we made to the SQL Buddy client side code was to convert the login script at the server. The original SQL Buddy system returned a new login page on a failed login. Instead, we changed it to only return an error code over `XMLHttpRequest`. The client-side code utilized this response to show the user the new login page, thereby preserving the application behavior. This change required modification of only 11 lines of code.

TCB Reduction. SQL Buddy utilizes the MooTools JavaScript library, which runs with the full privileges of the application site (e.g., `www.example.net`). Over 100KB of JavaScript code runs with full privileges of the `www.example.net` origin. This code uses dangerous, dynamic constructs such as `innerHTML` and `eval`. In our design, the total amount of code running in the `www.example.net` origin is 2.5KB, with the JavaScript code utilizing dynamic constructs running in an unprivileged temporary origin

Example Policy. Privilege separation reduces the ambient authority from these libraries. For example, the session cookie for `www.example.net`, is never sent to the child: all HTTP traffic requiring the cookie needs to go through the parent. Note that the cookie for the `www.example.net` principal includes both, the SQL Buddy session cookie as well as the cookie for the main `www.example.net` application. In case of successful code injection, the attacker cannot exfiltrate this cookie. Furthermore, the policy strictly limits privileged API access to those calls required by SQL Buddy. The SQL Buddy code does not have ambient authority to make privileged calls in the `www.example.net` principal. Again, implementing this policy requires two lines of JavaScript code in our architecture.

5.4 Top 50 Google Chrome extensions

Finally, we measure the opportunity available to our technique by quantifying the extent of TCB inflation and bundling in Chrome extensions. To perform this analysis, we developed a syntactic static analysis engine for JavaScript using an existing JavaScript engine called Pynarcissus [43] and performed a manual review for additional confidence. We report our results on 46 out of the top 50 extensions we study.[5] In our analysis, we (conservatively) identify all calls to privileged APIs (i.e., calls to the `chrome` object) and list them in Figure 1. We believe that our analysis is overly conservative, being syntactic, so these numbers represent only an undercount of the over-privileging in these applications.[6]

TCB Reduction. We show the distribution of the number of functions requiring any privileges as a percentage of the total number of functions. TCB inflation is pervasive in the extensions studied. In half the extensions, less than 5% of the total functions require any ambient privileges. In the current architecture the remaining 95% run with full privileges, inflating the TCB.

Bundling. We manually analyzed the 20 most popular Google Chrome extensions, and found 19 of them exhibited bundling. The most common form of bundling occurred when the options page or popup window of an extension runs with full privileges, in spite of not requiring any privileges at all. While the Google Chrome architecture does enable privilege separation between content scripts and extension code, running all code in an extension with the same privileges is unnecessary.

Another form of over-privileging occurs due to the bundling of privileges in Chrome's permission system. Google Chrome's extension system bundles multiple privileges into one coarse-grained install-time permission. For example, the `tabs` permission in Chrome extension API, required by 42 of the 46 extensions analyzed, bundles together a number of related, powerful privileges. This install-time permission includes the ability to listen to eight events related to `tabs` and windows, access users' browsing history, and call 20 other miscellaneous functions. Figure 5 measures the percentage of the `tabs` API actually used by extensions as a percentage of the total API granted by `tabs` for the 42 extensions analyzed. As can be seen, no extension requires the full privileges granted by the `tabs` permission, with one extension requiring 44.83% of the permitted API being the highest. More than half of the extensions require only 6.9% of the API available, which indicates over-privileging. In our design, the policy acts on fine-grained function calls and replaces coarse-grained permissions.

[5] Due to limitations of Pynarcissus, it was unable to completely parse code in 4 out of the top 50 extensions.

[6] More precise analysis can be used in the future.

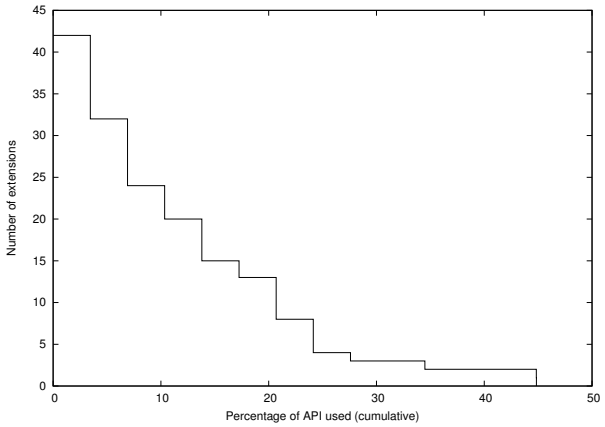

Figure 5: Frequency distribution of event listeners and API calls used by the top 42 extensions requiring the `tabs` permission.

6 Performance Benchmarks

Our approach has two possible overheads: run-time overhead caused by the parent's mediation on privileged APIs and the memory consumption of the new DOM and JavaScript heap created for each iframe. We measure the impact of each below.

Performance Overhead. First, as a micro-benchmark, we measured the run-time overhead caused by the parent's mediation on privileged APIs. We created a function that measures the total time taken to open a tab and then close it. This involves four crossings of the privilege boundary.

We performed the experiment 100 times with and without privilege separation. The median time with and without privilege-separation was 140ms and 80ms respectively. This implies an overhead of 15ms on each call crossing the sandbox.

As a macro-benchmark, we measured the amount of time required to load an image in the Awesome Screenshot image editor. Recall that the image editor receives the image data from the background page. We took a screenshot of `www.google.com` and measured the time taken for the image to load in the image editor, once the background sends it. We repeated the experiment 20 times each for the privilege separated and the original versions. The average (median) amount of time taken for the image load was 72.5ms (77.3ms) for the image load in the original Awesome Screenshot extension, and 78.5ms (80.1ms) for the image load in the privilege separated version—an overhead of 8.2% (3.6%). In our testing, we have not noticed any user-perceivable increase in latency after our redesign.

Memory Consumption. We measured the increase in memory consumption caused by creating a new temporary origin `iframe`, and found no noticeable increase in memory consumption.

On the Google Chrome platform, an alternate mechanism to get additional principals is creating a new extension. For example, Awesome Screenshot could be broken up into two extensions: a screenshot extension and an image editor extension. In addition to requiring two install decisions from the user, each additional extension runs in its own process on the Chrome platform. We measured the memory consumption of creating two extensions over a single extension and found an increase in memory consumption of 20MB. This demonstrates that our approach has no memory overhead as opposed to the 20MB overhead of creating a new extension.

7 Related Work

The concept of privilege separation was first formalized by Saltzer and Schroeder [44]. Several have used privilege separation for increased security. We discuss the most closely related works in the space.

Privilege Separation in Commodity OS Platforms. Notable examples of user-level applications utilizing privilege separation include QMail [10], OpenSSH [42] and Google Chrome [7]. Brumley and Song investigated automatic privilege separation of programmer annotated C programs and implemented data separation as well [12]. More recently, architectures like Wedge [11] identified subtleties in privilege separating binary applications and enforcing a default-deny model. Our work shows how to achieve privilege separation in emerging HTML5 applications, which are fuelling a convergence between commodity OS applications and web applications, without requiring any changes to the browser platform.

Re-architecting Browser Platforms. Several previous works on compartmentalizing web applications have suggested re-structuring the browser or the underlying execution platform altogether. Some examples include the Google Chrome extension platform [9], Escudo [33], MashupOS [52], Gazelle [53], OP [27], IPC Inspection [22], and CLAMP [40]. Our work advocates that we can achieve strong privilege separation using abstractions provided by modern browsers. This obviates the need for further changes to underlying platforms. We point out that temporary origins is similar to MashupOS's "null-principal `SERVICEINSTANCE`" proposal; therefore, the alternative line of research into new browser primitives has indeed been fruitful. Our work demonstrates how we can utilize these advancements by combining deployed primitives (like temporary origins and CSP [48]) to achieve effective privilege separation, without requiring any further changes to the platform.

Carlini et al. [14] study the effectiveness of privilege separation in the Chrome extension architecture and find

that in 4 (19) out of 61 cases, insufficient validation of messages exchanged over the privilege boundary allowed for full (partial) privilege escalation. In our design, we explicitly prohibit the parent from using incoming messages in a way that can lead to code execution. Furthermore, Chrome extensions today tend to have inflated TCB in the privileged component as we show in Section 5.4. This is in contrast to our proposed design.

Mashup & Advertisement Isolation. The problem of isolating code in web applications, especially in mashups [6, 52] and malicious advertisements [35], has received much attention in research. Our work has similarities with these works in that it uses isolation primitives like `iframe`s. However, one key difference is that we advocate the use of temporary origins, which are now available in most browsers, as a basis for creating arbitrary number of components.

In concurrent work, Treehouse [32] provides similar properties, but relies on isolated web workers with a virtual DOM implementation for backwards compatibility. A virtual DOM allows Treehouse to interpose on all DOM events, providing stronger security and resource isolation properties, but at a higher performance cost.

Language-based Isolation of web applications. Recent work has focused on language-based analysis of web application code, especially JavaScript, for confinement. IBEX proposed writing extensions in a high-level language (FINE) that can later by analyzed to conform to specific policies [29]. In contrast, our work does not require developers to learn new language, and thus maintains compatibility with existing code. Systems like IBEX are orthogonal to our approach and can be supported on top of our architecture; if necessary, the parent's policy component can be written in a high-level language and subject to automated analysis.

Heavyweight language-based analyses and rewriting systems have been used for isolating untrusted code, such as advertisements [2,26,37]. Our approach instead relies on a lighter weight mechanism based on built-in browser primitives like `iframe`s and temporary origins.

8 Conclusion

Privilege separation is an important second line of defense. However, achieving privilege separation in web applications has been harder than on the commodity OS platform. We observe that the central reason for this stems in the same origin policy (SOP), which mandates use of separate origins to isolate multiple components, but creating new origins on the fly comes at a cost. As a result, web applications in practice bundle disjoint components and run them in one monolithic authority. We propose a new design that uses standardized primitives already available in modern browsers and enables partitioning web applications into an arbitrary number of temporary origins. This design contrasts with previous approaches that advocate re-designing the browser or require adoption of new languages. We empirically show that we can apply our new architecture to widely used HTML5 applications right away; achieving drastic reduction in TCB with no more than thirteen lines of change for the applications we studied.

9 Acknowledgements

We thank Erik Kay, David Wagner, Adrienne Felt, Adrian Mettler, the anonymous reviewers, and our shepherd, William Enck for their insightful comments. This material is based upon work partially supported by the NSF under the TRUST grant CCF-0424422, by the Air Force Office of Scientific Research (AFOSR) under MURI awards FA9550-09-1-0539 and FA9550-08-1-0352 and by Intel through the ISTC for Secure Computing. The second author is supported by the Symantec Research Labs Graduate Fellowship.

References

[1] Ace - ajax.org cloud9 editor. `http://ace.ajax.org/`.

[2] AdSafe : Making JavaScript Safe for Advertising. `http://www.adsafe.org/`.

[3] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, Phillipa Gill, and David Lie. Short paper: a look at smartphone permission models. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 63–68, New York, NY, USA, 2011. ACM.

[4] Mozilla boot2gecko. `https://wiki.mozilla.org/B2G`.

[5] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. VEX: Vetting browser extensions for security vulnerabilities, 2010.

[6] A. Barth, C. Jackson, and W. Li. Attacks on javascript mashup communication. In *Proceedings of the Web*, volume 2. Citeseer, 2009.

[7] A. Barth, C. Jackson, C. Reis, and TGC Team. The security architecture of the chromium browser, 2008.

[8] Adam Barth. The web origin concept. `http://tools.ietf.org/html/rfc6454`.

[9] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities, 2009.

[10] Daniel J. Bernstein. Some thoughts on security after ten years of qmail 1.0. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, CSAW '07, pages 1–10, New York, NY, USA, 2007. ACM.

[11] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[12] David Brumley and Dawn Song. Privtrans: automatically partitioning programs for privilege separation. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[13] Bugzilla@Mozilla. Bug 341604 - (framesandbox) implement html5 sandbox attribute for iframes. `https://bugzilla.mozilla.org/show_bug.cgi?id=341604`.

[14] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. An evaluation of the google chrome extension security architecture). In *Porceedings of the 21st USENIX Conference on Security*, USENIX Security'12, Berkeley, CA, 2012. Usenix Association.

[15] Eric Yawei Chen, Jason Bau, Charles Reis, Adam Barth, and Collin Jackson. App isolation: get the security of multiple browsers with just one. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 227–238, New York, NY, USA, 2011. ACM.

[16] Chromium os. `http://www.chromium.org/chromium-os`.

[17] M. Dhawan and V. Ganapathy. Analyzing information flow in javascript-based browser extensions. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 382–391. IEEE, 2009.

[18] diigo.com. Awesome screenshot : Capture annotate share. `http://www.awesomescreenshot.com/`.

[19] Dropbox Inc. Dropbox developer reference. `http://www.dropbox.com/developers/reference`.

[20] Adrienne Porter Felt. Advertising and android permissions, Nov 2011. `http://www.adrienneporterfelt.com/blog/?p=357`.

[21] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX conference on Web application development*, WebApps'11, pages 7–7, Berkeley, CA, USA, 2011. USENIX Association.

[22] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission re-delegation: attacks and defenses. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[23] Matthew Finifter, Joel Weinberger, and Adam Barth. Preventing capability leaks in secure JavaScript subsets. In *Proc. of Network and Distributed System Security Symposium, 2010*, 2010.

[24] Google Inc. Google chrome extensions: chrome.* apis. `http://code.google.com/chrome/extensions/api_index.html`.

[25] Google Inc. Google chrome webstore. `https://chrome.google.com/webstore/`.

[26] Google Inc. Issues: google-caja: A source-to-source translator for securing Javascript-based web content. `http://code.google.com/p/google-caja`.

[27] Chris Grier, Shuo Tang, and Samuel T. King. Designing and implementing the op and op2 web browsers. *ACM Trans. Web*, 5:11:1–11:35, May 2011.

[28] Salvatore Guarnieri and Benjamin Livshits. Gatekeeper: mostly static enforcement of security and reliability policies for JavaScript code. In *Usenix Security*, 2009.

[29] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 115–130. IEEE, 2011.

[30] HTTP Archive. Js transfer size and js requests. `http://httparchive.org/trends.php#bytesJS&reqJS`.

[31] GitHub Inc. Edit like an ace. `https://github.com/blog/905-edit-like-an-ace`.

[32] Lon Ingram and Michael Walfish. Treehouse: Javascript sandboxes to help web developers help themselves. In *Proceedings of the 2012 USENIX conference on USENIX annual technical conference*, Berkeley, CA, USA, 2012. USENIX Association.

[33] K. Jayaraman, W. Du, B. Rajagopalan, and S.J. Chapin. Escudo: A fine-grained protection model for web browsers. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 231–240. IEEE, 2010.

[34] Peter Josling. dropbox-js: A javascript library for the dropbox api. `http://code.google.com/p/dropbox-js/`.

[35] Mike Ter Louw, Karthik Thotta Ganesh, and V. N. Venkatakrishnan. Adjail: practical enforcement of confidentiality and integrity policies on web advertisements. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 24–24, Berkeley, CA, USA, 2010. USENIX Association.

[36] lxc linux containers. `http://lxc.sourceforge.net/`.

[37] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 125–140, Washington, DC, USA, 2010. IEEE Computer Society.

[38] Microsoft. Metro style app development, 2012. `http://msdn.microsoft.com/en-us/windows/apps/`.

[39] Oauth. `http://oauth.net/`.

[40] Bryan Parno, Jonathan M. McCune, Dan Wendlandt, David G. Andersen, and Adrian Perrig. Clamp: Practical prevention of large-scale data leaks. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 154–169, Washington, DC, USA, 2009. IEEE Computer Society.

[41] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, pages 18–18, Berkeley, CA, USA, 2003. USENIX Association.

[42] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.

[43] pynarcissus : The narcissus javascript interpreter ported to python. `http://code.google.com/p/pynarcissus/`.

[44] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[45] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. Technical Report UCB/EECS-2010-26, EECS Department, University of California, Berkeley, 2010.

[46] Google seccomp sandbox for linux. `http://code.google.com/p/seccompsandbox/`.

[47] Source code release. `http://github.com/devd/html5privsep`.

[48] Brandon Sterne and Adam Barth. Content security policy: W3c editor's draft, 2012. `https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html`.

[49] G. Tan and J. Croft. An empirical security study of the native code in the jdk. In *Proceedings of the 17th Conference on Security*, pages 365–377. USENIX Association, 2008.

[50] The Dojo Foundation. The dojo toolkit. `http://dojotoolkit.org/`.

[51] Anne van Kesteren (Ed.). Cross-origin resource sharing. `http://www.w3.org/TR/cors/`.

[52] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and communication abstractions for web browsers in MashupOS. In *SOSP*, 2007.

[53] 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 conference on USENIX security symposium*, pages 417–432. USENIX Association, 2009.

[54] H.J. Wang, A. Moshchuk, and A. Bush. Convergence of desktop and web applications on a multi-service os. In *Proceedings of the 4th USENIX conference on Hot topics in security*, pages 11–11. USENIX Association, 2009.