Towards High Assurance HTML5 Applications

Devdatta Akhawe

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-56
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-56.html

May 7, 2014

Towards High Assurance HTML5 Applications

by

Devdatta Madhav Akhawe

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Dawn Song, Chair
Professor David Wagner
Professor Brian Carver

Spring 2014

Towards High Assurance HTML5 Applications

Abstract

Towards High Assurance HTML5 Applications

by

Devdatta Madhav Akhawe

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Dawn Song, Chair

Rich client-side applications written in HTML5 proliferate diverse platforms such as mobile devices, commodity PCs, and the web platform. These client-side HTML5 applications are increasingly accessing sensitive data, including users' personal and social data, sensor data, and capability-bearing tokens. Instead of the classic client/server model of web applications, modern HTML5 applications are complex client-side applications that may call some web services, and run with ambient privileges to access sensitive data or sensors. The goal of this work is to enable the creation of higher-assurance HTML5 applications. We propose two major directions: first, we present the use of formal methods to analyze web protocols for errors. Second, we use existing primitives to enable practical privilege separation for HTML5 applications. We also propose a new primitive for complete mediation of HTML5 applications. Our proposed designs considerably ease analysis and improve auditability.

To my parents.

Contents

List of Figures

List of Tables

Acknowledgments

First, I want to thank my advisor Dawn for being such a fantastic advisor and guide through my graduate career. Also, thanks to David Wagner whose advice and guidance I have always sought and received during my graduate career. Thanks also to my committee members Brian Carver and George Necula for their help and guidance.

The research presented in this thesis is a joint effort. A special thanks goes to all my co-authors: Adam Barth, Warren He, Eric Lam, Frank Li, John Mitchell, Prateek Saxena, Dawn Song. Over the course of my graduate life I have co-authored papers with nearly 30 different co-authors. These collaborators, all my friends in the Security group, and all my teachers at Berkeley have directly impacted my research, my work, and my evolution as a researcher and I remain thankful to them all. I am extremely lucky to have been surrounded by and worked with such a tremendously talented group of people over the past five years.

Pursuing graduate studies was in a large part due to all the great mentors and teachers I have had over the years. I would like to particularly thank my undergraduate advisor, Sundar Balasubramaniam, as well as Helen Wang and Xiaofeng Fan for their fantastic mentoring and advice. Without their help and support, it is unlikely I would have even applied to graduate school.

Thanks also to all my friends, from Pilani to Berkeley, who made the stress of graduate life easy to manage. You know who you are and I feel blessed to call such an amazing group of people my friends.

Finally, and most importantly, I want to thank my extended family: my brother, my cousins, my uncles and aunts, and their respective families for their amazing love, care, and guidance over the years. I would like to particularly thank all my four aunts: they ensured I got an education and never lost focus.

Chapter 1

Introduction

Rich client-side HTML5 applications—including packaged browser applications (Chrome Apps) [57], browser extensions [56], Windows 8 Metro applications [98], and applications in new browser operating systems (B2G [105], Tizen [134])—are fast proliferating on diverse computing platforms. These applications run with access to sensitive data such as the user's browsing history, personal and social data, financial documents, and capability-bearing tokens that grant access to these data. A recent study reveals that 58% of the 5,943 Google Chrome browser extensions studied require access to the user's browsing history, and 35% request permissions to the user's data on all websites [34]. In addition, the study found that 67% of 34,370 third-party Facebook applications analyzed have access to the user's personal data [34].[1] HTML5 applications also form a significant chunk of mobile applications; Chin et al. recently found that 70% of smartphone applications they surveyed on Google Play rely on HTML5 code in some form [35]. These HTML5 applications often execute with access to the same sensors available to native mobile applications, including private data from GPS receivers, accelerometers, and cameras.

These trends indicate the evolution of the client-side web from a front-end for servers to a complex application *platform* running privileged applications. Despite immense prior research on detection and mitigation techniques [7, 45, 66, 82, 122], web vulnerabilities are still pervasive in HTML5 applications on emerging platforms such as browser extensions [30]. As the HTML5 platform achieves wider adoption, enabling higher-assurance in the HMTL5 applications is critical to its success. In this thesis, we address this need.

1.1 Towards a Formal Foundation for Web Protocols

First, we present initial work on formal modeling and verification of web protocols. As we discussed above, HTML5 applications on emerging platforms are moving away from the client/server paradigm to a new paradigm of standalone HTML5 applications that

[1] The study measured install-time permissions, which are a lower bound for Facebook applications, since they can request further permissions at runtime.

call diverse web services. The security of protocols used by HTML5 applications to communicate with diverse cloud-based services is just as critical to the security of the platform as the security of the HTML5 application itself. We propose a formal model of web security mechanisms based on an abstraction of the web platform and use this model to analyze the security of five sample web mechanisms and applications.

Web protocols are distinct from network protocols due to the nature of the web: attacker code often runs as part of the user's browser and the attacker can initiate cookie bearing requests. Our work is the first to bring out these issues in a formal setting. We identify three distinct threat models to analyze web applications, ranging from a web attacker who controls malicious web sites and clients, to stronger attackers who can control the network and leverage sites designed to display user-supplied content.

We propose two broadly applicable security goals and study five security mechanisms. In our case studies, which include HTML5 forms, Referer validation, and a single sign-on solution, we use Alloy, a SAT-based model-checking tool, to find two previously known vulnerabilities and three new vulnerabilities. Our case study of a Kerberos-based single sign-on system illustrates the differences between a secure network protocol using custom client software and a similar but vulnerable web protocol that uses cookies, redirects, and embedded links instead.

1.2 Privilege Separation for HTML5 Applications

Next, we present work on improving assurance in HTML5 applications using privilege separation with standardized browser primitives. Current web applications suffer from pervasive over-privileging, which impacts security analysis and audits. One reason for such pervasive over-privileging is the absence of easy to use privilege separation primitives. 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.

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.

1.3 Data Confined HTML5 Applications

Privilege separation only provides isolation of code, it does not, however, provide any control over data flows. HTML5 applications that handle sensitive user data (e.g., password managers, medical record managers) need to securely *confine* data to a whitelist of principals. This is challenging since HTML5 as a platform is designed for sharing and communication, not for restricting the flow of data. An HTML5 application with an injection vulnerability can leak sensitive data, even in the absence of XSS flaws. As HTML5 applications pervade more and more platforms, the absence of high-performance primitives for controlling data-flow severely restrict the ability to reason about data-flow in HTML5 applications.

In Chapter 4, we identify such data-confinement invariants in modern, rich client-side applications. As we discussed above, such HTML5 applications proliferate on diverse platforms, accessing sensitive data. It is critical that the application confine the data to specific principals in a high assurance manner. Unfortunately, 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 over- head 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.

Outline We organize the rest of this thesis as follows: Chapter 2 presents our work on formally modeling web protocols. Chapter 3 discusses our work on privilege separation of HTML5 applications. In Chapter 4, we first present more details of why the HTML5 platform and previous research does not sufficiently address the need for data *confinement* and present a new primitive for data confinement on the HTML5 platform. Finally, we discuss related work in Chapter 5 before concluding in Chapter 6.

Chapter 2

Towards A Formal Foundation for Web Protocols

The research discussed in this chapter was presented at the *23rd Computer Security Foundations Symposium, 2010* at Edinburgh, Scotland. This is joint work with Adam Barth, Eric Lam, John Mitchell, and Dawn Song.

2.1 Introduction

The web, indispensable in modern commerce, entertainment, and social interaction, is a complex delivery platform for sophisticated distributed applications with multifaceted security requirements. Unfortunately, most web browsers, servers, network protocols, browser extensions, and their security mechanisms were designed without analytical foundations. Further complicating matters, the web continues to evolve with new browser features, protocols, and standards added at a rapid pace [72, 74, 77, 87, 107, 127]. The specifications of new features are often complex, lack clear threat models, and involve unstated and unverified assumptions about other components of the web. As a result, new features can introduce new vulnerabilities and break security invariants assumed by web applications [68, 89, 106].

The ad hoc nature and the lack of formalism in the current web protocol design reminds us of the early stage of network security protocol design when subtle flaws were commonplace in widely deployed network security protocols, such as the earlier versions of Kerberos, SSL [90, 100]. To address this problem, researchers have successfully applied formal methods to prove security properties of classes of security protocols such as network authentication and key distribution protocols [101, 126]. This effort has been a fruitful and invaluable research direction—a great volume of research in this area has resulted in a much deeper understanding of how to design network security protocols with security guarantees.

Just as formal models and tools have proven useful in evaluating the security of

network protocols, we believe that abstract yet informed models of the web platform, web applications, and web security mechanisms will be amenable to automation, reveal practical attacks, and support useful evaluation of alternate designs.

In this chapter, we propose a formal model for the web platform, which includes a number of key web concepts, and demonstrate that our model is useful for finding bugs in real-world web security mechanisms. Our model is sufficiently abstract and amenable to formal analysis, yet appears sufficiently detailed to express subtle attacks missed by expert human analysts in several cases. We provide an executable implementation of a subset of our model in Alloy [39] and demonstrate the utility of this subset (and, more generally, our model) via five case studies. Although we imagine our model being used for more than vulnerability discovery, we focus in this work on analyzing existing protocols for design errors. We show that our model can capture two previously known and three previously unknown vulnerabilities.

Our web security model consists of a selection of web concepts, precise threat models, and two broadly applicable security goals. These design choices are informed by previous experience designing and (informally) evaluating web security mechanisms, such as preventing cross-site request forgery [17], securing browser frame communication [17], preventing DNS rebinding [78], and protecting high-security web sites from network attacks [77]. Our experience with these and other suggest that a few central modeling concepts will prove useful for evaluating a wide range of mechanisms.

The central web concepts we formalize in our model include browsers, servers, scripts, HTTP, and DNS, as well as ways they interact. For example, each script context, representing execution of JavaScript within a browser execution environment, is associated with a given "origin" and located in a browser. By making use of browser APIs, such as XMLHttpRequest, these script contexts can direct (restricted forms of) HTTP requests to various DNS names, which resolve to servers. These servers, in turn, respond to these requests and influence the browser's behavior. Although the web security model we describe in Section 2.2 also contains other concepts such as frames, the location bar, and the lock icon, our executable implementation described in Section 2.3 focuses on browsers, servers, scripts, HTTP, and DNS, which form the "backbone" of the model.

We propose three distinct and important threat models: a *web attacker*, an *active network attacker*, and a *gadget attacker*. The most important threat model, at least for mechanisms and studies we are familiar with, is the web attacker. The web attacker operates a malicious web site and may use a browser, but has no visibility into the network beyond requests or responses directed towards the hosts it operates. Many core web security mechanisms are designed to resist the threats we formalize as the web attacker but fail to provide protection against more powerful attackers. An active network attacker has all the abilities of a web attacker plus the ability to eavesdrop, block, and forge network messages. The active network attacker we define is slightly more powerful than the eponymous threat considered when analyzing a traditional network protocol because our active network attacker can make use of browser APIs. Finally, we also consider a web attacker with the ability to inject (limited kinds of) content into otherwise honest web

sites, corresponding to the gadget attacker considered in [17]. This threat lets us consider the robustness of web security mechanisms in the presence of third-party content ranging from comments on a blog to gadgets in a mashup.

The third part of our model formulates two widely applicable security goals that can be evaluated for various mechanisms: (i) new mechanisms should not violate any of the invariants that web sites commonly rely upon for security and (ii) a "session integrity" condition, which states that the attacker is unable to cause honest servers to undertake potentially sensitive actions. There is a wide spectrum of security goals we could investigate, but we focus on these goals because they are generally applicable to many web security mechanism, including those in our case studies.

Concretely, we implement the core components of our model in Alloy [39, 80], an automated tool that translates a declarative object-modeling syntax into propositional input to a SAT solver. Using Alloy's declarative input language, we axiomatize the key concepts, threat models, and security goals of our model. Our axiomatization is incomplete (both because we do not implement all the concepts in our model and because browser implementations might contain bugs) but useful nonetheless. After modeling a specific web security mechanism, the Alloy satisfiability (SAT) solver attempts to find browser and site interactions that violate specified security goals. Typically, we ask not whether a particular web security mechanism is secure, but how powerful an attacker is required to defeat the security mechanism. In this way, we aim to quantify the security of the mechanism.

To demonstrate the utility of our model, we conduct five case studies. We use our model to analyze a proposed cross-site request forgery defense based on the Origin header, Cross-Origin Resource Sharing [87] (the security component of the new cross-origin XMLHttpRequest API in the latest browsers), a proposal [85] to use Referer validation to prevent cross-site scripting, new functionality in the HTML5 form element, and WebAuth, a Kerberos-based single sign-on system used at a number of universities. In each case, our model finds a vulnerability in the mechanism, two of which were previously known and three of which were previously unknown. The Referer validation example, in particular, demonstrates that our model is more sophisticated than previous approaches because [85] analyzed the mechanism with Alloy and concluded that the mechanism was secure. The WebAuth example shows that subtle security issues arise when embedding a well-understood network protocol (Kerberos) in a web security mechanism because of interactions between the assumptions made by the protocol and the behavior of the web platform.

Our study is an initial step that demonstrates that a formal approach to web security is fruitful. Our model is modular and extensible—we can add new web concepts and more detailed models incrementally, letting the model grow over time to encompass a more complete picture of the web platform. As the model grows, the utility of the models grows via a network effect—an extensive model will let us analyze subtle interactions between different components of the web platform and the security mechanism, leading to a deeper and broader understanding of the potential security consequences of any newly introduced

web security mechanisms.

Organization The remainder of this chapter is organized as follows. Section 2.2 presents our formal model. Section 2.3 explains how we implement our model in Alloy. Section 2.4 analyzes five example web security mechanisms using our model. Section 2.5 contains some operational statistics and advice about the model.

2.2 General Model

There are many threats associated with web browsing and web applications, including phishing, drive-by downloads, blog spam, account takeover, and click fraud. Although some of these threats revolve around exploiting implementation vulnerabilities (such as memory safety errors in browsers or tricking the user), we focus on ways in which an attacker can abuse web functionality that exists by design, or flaws in the *web protocol*. For example, an HTML form element lets a malicious web site generate GET and POST requests to arbitrary web sites, leading to security risks like cross-site request forgery (CSRF). Web sites use a number of different strategies to defend themselves against CSRF [17], but we lack a scientifically rigorous methodology for studying these defenses. By formulating an accurate model of the web, we can evaluate the security of these defenses and determine how they interact with extensions to the web platform.

A core idea in our model is to describe what could occur if a user navigates the web and visits sites in the ways that the web is designed to be used. For example, the user could choose to type any web address into the address bar and visit any site, or click on a link provided by one site to visit another. Because browsers support the "back" button, returning the user to a previously visited page, many sites in effect allow a user to click on all of the links presented on a page, not just one. When the user visits a site, the site could serve a page with any number of characteristics, possibly setting a cookie, or redirecting the user to another site. The set of events that could occur, therefore, includes browser requests, responses, cookies, redirects, and so on, transmitted over HTTP or HTTPS.

We believe that examining the set of possible events accurately captures the way that web security mechanisms are designed. For example, the web is designed to allow a user to visit a good site in one window and a potentially malicious site in another. Because the back button is so popular, web security mechanisms are usually designed to be secure even if the user returns to a previously visited page and progresses differently the second (or third or fourth) time.

The model we propose has three main parts: web concepts, threat models, and security goals. The web concepts represent the conceptual universe defined by web standards. Our formalization of these concepts includes a set of browsers, operated by potential victims, each with its user and a browsing history, interacting with an arbitrary number of web servers that receive and send HTTP requests and responses, possibly encrypted using SSL/TLS. Our model considers a spectrum of threats, ranging from a web attacker to a

network attacker to a gadget attacker. For example, a web attacker controls one or more web sites that the user visits, and may operate a browser to visit sites, but does not control the network used to visit other sites. Finally, we regard security goals as predicates that distinguish felicitous outcomes from attacks.

Web Concepts

The central concepts of the web are common to virtually every web security mechanism we wish to analyze. For example, web mechanisms involve a web browser that interacts with one or more web servers via a series of HTTP requests and responses. The browser, server, and network behavior form the "backbone" of the model, much in the same way that cryptographic primitives provide the backbone of network protocols. Many of the surprising behaviors of the web platform, which lead to attacks against security mechanisms, arise from the complex interaction between these concepts. By modeling these concepts precisely, we can check their interactions automatically.

Non-Linear Time We use a branching notion of time because of the browser's "back" button. In other words, we are not concerned with the actual temporal order between unrelated actions. Instead, if a user could click on either of two links, then use the back button to click on the other, we represent this as two actions that are unordered in time. In effect, our temporal order represent necessary "happens before" relations between events (e.g., an HTTP request must happen before the browser can receive the corresponding HTTP response). Instead of regarding these branches as possible futures, we regard them as all having occurred, for example letting an attacker transport knowledge from one branch to another. In addition to conceptual economy, abstracting from the accidental linear order can reduce the number of possible states in need of exploration.

This notion of time leads to a model that is largely *monotonic*. If an attacker can undertake an action at one point in time, we assume that action is thereafter always available to the attacker (because the user can usually return to that state via the back button). In contrast, traditional models of network protocol security are non-monotonic: once the protocol state machine advances to step 3, the attacker can no longer cause the state machine to accept a message expected in step 2. Although we do not exploit this monotonicity directly, we believe this property bears further investigation.

Browser The user's web browser, of course, plays a central role in our model of web security. However, the key question is what level of abstraction to use for the browser. If we model the browser at too low a level (say bytes received over the network being parsed by an HTML parser and rendered via the CSS box model into pixels on the screen), our model will quickly become unwieldy. The HTML5 specification alone is some 45,000 lines of text. Instead, we abstract the browser into three key pieces:

- *Script Context.* A script context represents all the scripts running in the browser on behalf of a single web origin. The browser does not provide any isolation guarantees between content within an origin: all same-origin scripts "share fate." Correspondingly, we group the various scripts running in different web pages within an origin into a single script context and imagine them acting in unison.

- *Security UI.* Some parts of the browser's user interface have security properties. For example, the browser guarantees that the location bar accurately displays the URL of the top-level frame. We include these elements (notably, the location bar, the lock icon, and the extended validation indicator) in our model and imbue them with their security properties. In addition, we model a forest of frames, in which each frame is associated with a script context and each tree of frames is associated with a constellation of security indicators. We assume that each frame can overwrite or display (but not read) the pixels drawn by the frame below it in the hierarchy, modeling (at a high level) how web pages are drawn.

- *State Storage.* Finally, the browser contains some amount of persistent storage, such as a cookie store and a password database. We assume that confidential information contained in these state stores is associated with an origin and can be read by a script context running on behalf of that origin. To keep the model monotonic, we model these state stores as "append-only," which is not entirely accurate but simplifies the model considerably.

Servers We model web servers as existing at network locations (which are an abstraction of IP addresses). Each web server is owned by a single principal, who controls how the server responds to network messages. Servers controlled by "honest" principals follow the specification but a server controlled by a malicious principal might not. Servers have a many-to-many relation to DNS names (e.g., `www.example.com`), which themselves existing in a delegation hierarchy (e.g., `www` delegates to `example`, which delegates to `com`). Holding servers in a many-to-many relation with DNS names is essential for modeling various tricky situations, such as DNS rebinding [78], where the attacker points a malicious DNS name at an honest server.

Network Finally, browsers and servers communicate by way of a network. In contrast to traditional models of network security, our model of the network has significant internal structure. Browsers issue HTTP requests to URLs, which are mapped to servers via DNS. The requests contain a method (e.g., GET, POST, DELETE, or PUT) and a set of HTTP headers. Individual headers carry semantics. For example, the Cookie header contains information retrieved from the browser's cookie store and the Referer header identifies the script context that initiated the request. It is an important part of security mechanisms such as CSRF defenses [17] that the Referer header, for example, is set by the browser and not controlled by content rendered in the browser.

Network requests can be generated by a number of web APIs, including HTML forms, XMLHttpRequest, and HTTP redirects, each of which imposes different security constraints on the network messages. For example, requests generated by XMLHttpRequest can be sent only to the same origin (in the absence of CORS [87]), whereas requests generated by HTML forms can be sent to any origin but can contain only certain methods and headers. These restrictions are essential for understanding the security of the web platform. For example, the Google Web Toolkit relies on the restrictions on custom HTTP headers imposed by the HTML form element to protect against CSRF [131].

Threat Models

When evaluating the security of web applications, we are concerned with a spectrum of threats. The weakest threat is that of a *web attacker*: a malicious principal who operates a web site visited by the user. Starting with the web attacker as a base, we can consider more advanced threats, such as an *active network attacker* and a *gadget attacker*.

Web attacker Although the informal notion of a web attacker has appeared in previous work [14, 17, 76, 77], we articulate the web attacker's abilities precisely.

- *Web Server.* The web attacker controls at least one web server and can respond to HTTP requests with arbitrary content. Intuitively, we imagine the web attacker as having "root access" to these web servers. The web attacker controls some number of DNS names, which the attacker can point to any server. Canonically, we imagine the DNS name `attacker.com` referring to the attacker's main web server. The web attacker can obtain an HTTPS certificate for domains owned by the attacker from certificate authorities trusted by the user's browser. Using these certificates, the attacker can host malicious content at URLs like `https://attacker.com/`.

- *Network.* The web attacker has no special network privileges. The web attacker can respond only to HTTP requests directed at his or her own servers. However, the attacker can *send* HTTP requests to honest servers from attacker-controlled network endpoints. These HTTP requests need not comply with the HTTP specification, nor must the attacker process the responses in the usual way (although the attacker can simulate a browser locally if desired). For example, attacker can send an arbitrary value in the Referer header and need not follow HTTP redirects. Notice that the web attacker's abilities are decidedly *weaker* than the usual network attacker considered in studies of network security because the web attacker can neither eavesdrop on messages to other recipients nor forge messages from other senders.

- *Browser.* When the user visits the attacker's web site, the attacker is "introduced" to the user's browser. Once introduced, the attacker has access to the browser's web APIs. For example, the attacker can create new browser windows by calling the `window.open()` API. We assume the attacker's use of these APIs is constrained by

the browser's security policy (colloquially known as the "same-origin policy" [143]), that is, the attacker uses only the privileges afforded to every web site. One of the most useful browser APIs, from the attacker's point of view, is the ability to generate cross-origin HTTPS requests via hyperlinks or the HTML form element. Attacks often use these APIs in preference to directly sending HTTP requests because (1) the requests contain the user's cookies and (2) the responses are interpreted by the user's browser.

A subtle consequence of these assumptions is that (once introduced) the attacker can maintain a persistent thread of control in the user's browser. This thread of control, easily achieved in practice using a widely known web application programming technique [5], can communicate freely with (and receive instructions from) the attacker's servers. We do not model this thread of control directly. Instead, we abstract these details by imagining a single coherent attacker operating at servers and able to generate specific kinds of events in the user's browser (accurately associated with the attacker's origin).

Network Attacker An active network attacker has all the abilities of a web attacker as well as the ability to read, control, and block the contents of all *unencrypted* network traffic. In particular, the active network attacker need not be present at a network endpoint to send or receive messages at that endpoint. We assume the attacker cannot corrupt HTTPS traffic between honest principals because trusted certificate authorities are unwilling to issue the attacker certificates for honest DNS names, although these certificate authorities are willing to issue the attacker HTTPS certificates for malicious DNS names and the attacker can, of course, always self-sign a certificate for an honest DNS name. Without the appropriate certificates, we assume the attacker cannot read or modify the contents of HTTPS requests or responses.

Gadget Attacker The gadget attacker [18] has all the abilities of a web attacker as well as the ability to inject some limited kinds of content into honest web sites. The exact kind of content the gadget attacker can inject depends on the web application. In many web applications, the attacker can inject a hyperlink (e.g., in email or in blog comments). In some applications, such as forums, the attacker can inject images. In more advanced applications, such as Facebook or iGoogle, the attacker can inject full-blown gadgets with extensive opportunity for misdeeds. We include the gadget attacker to analyze the robustness of security mechanisms to web sites hosting (sanitized) third-party content.

User Behavior The most delicate part of our threat model is how to constrain user behavior. If we do not constrain user behavior at all, the user could simply send his or her password to the attacker, defeating most web security mechanisms. On the other hand, if we constrain the user too much, we risk missing practical attacks.

- *Introduction.* We assume the user might visit any web site, including the attacker's web site. We make this assumption because we believe that an honest user's interaction with an honest site should be secure even if the user separately visits a malicious site in a different browser window. A concerted attacker can always acquire traffic by placing advertisements. For example, in a previous study [78], we mounted a web attack by purchasing over 50,000 impressions for $30. In other words, we believe that this threat model is an accurate abstraction of normal web behavior, *not* an assumption that web users promiscuously visit all possible bad sites in order to tempt fate.

- *Not Confused.* Even though the user visits the attacker's web site, we assume the user does not confuse the attacker's web site with an honest web site. In particular, we assume the user correctly interprets the browser's security indicators, such as the location bar, and enters confidential information into a browser window only if the location bar displays the URL of the intended site. This assumption rules out *phishing* attacks [44, 51], in which the attacker attempts to fool the user by choosing a confusing domain name (e.g., `bankofthevvest.com`) or using other social engineering. In particular, we do *not* assume a user treats `attacker.com` as if it were another site. However, these assumptions could be relaxed or varied in order to study the effectiveness of specific mechanisms when users are presented with deceptive content.

Feasiblity Because our model of a web attacker is relatively weak, attacks that can be mounted by a web attacker can be carried out in practice without any complex or unusual control of the network. In addition, web attacks can also be carried out by a standard man-in-the-middle network attacker because a man-in-the-middle can intercept arbitrary HTTP requests and inject content that will be rendered by the victim's browser.

Anyone can easily achieve the network capabilities of a web attacker. There are several techniques an attacker can use to drive traffic to `attacker.com`. For example, the attacker may place advertisements through advertising networks, display popular content indexed by search engines, and send bulk e-mail attracting users. Moreover, the act of viewing an attacker's advertisement is generally sufficient to mount a web-based attack [78], and a user generally has no control over which advertisements are placed on well-viewed commercial sites.

We believe that a normal but careful web user who reads news and conducts normal banking, investment, and retail transactions, cannot effectively monitor and restrict the source of all content rendered in his or her browser.

Security Goals

Although different web security mechanisms have different security goals, there are two security goals that seem to be fairly common:

- *Security Invariants.* The web contains a large number of existing web applications that make assumptions about web security. For example, some applications assume that a user's browser will never generate a cross-origin HTTP request with DELETE method because that property is ensured by today's browsers (even though cross-origin GET and POST requests are possible). When analyzing the security of new elements of the web platform, it is essential to check that these elements respect these (implicit) security invariants and "don't break the web" (i.e., introduce vulnerabilities into existing applications). We formalize this goal as a set of invariants servers expect to remain true of the web platform. Although we focus at present on invariants relevant to the mechanisms at hand, we believe that future work on web security can fruitfully aim to identify more invariants.

- *Session Integrity.* When a server takes action based on receiving an HTTP request (e.g., transfers money from one bank account to another), the server often wishes to ensure that the request was generated by a trusted principal and not an attacker. For example, a traditional cross-site request forgery vulnerability results from failing to meet this goal. We formalize this goal by recording the "cause" of each HTTP request (be it an API invoked by a script or an HTTP redirect) and checking whether the attacker is in this casual chain.

We are unaware of previous work that recognizes the value of identifying clear web security invariants. However, because many web security mechanisms depend on complementary properties of the browser, web protocols, and user behavior, we believe that these invariants form the core of a comprehensive scientific understanding of web security.

2.3 Implementation in Alloy

We implement a subset of our formal model in the Alloy modeling language. Although incomplete, our implementation [4] contains the bulk of the networking and scripting concepts and is sufficiently powerful to find new and previously known vulnerabilities in our case studies. In this section, we summarize how we implement the key concepts from the model in this language. We first express the base model, containing the web concepts and threats, and then add details of the proposed web mechanism. Finally, we add a constraint that negates the security goal of the mechanism and ask Alloy for a satisfying instance. If Alloy can produce such an instance, that instance represents an attack because the security goal has failed.

Expressing our model in Alloy has several benefits. First, expressing our model in an executable form ensures that our model has precise, testable semantics. In creating the model, we found a number of errors by running simple "sanity checks." Second, Alloy lets us express a model of unbounded size and then later specify a size bound when checking properties. We plan to use this distinction in future work to prove a "small model" theorem bounding the necessary search size (similar to [103]). Finally, Alloy translates our

high-level, declarative, relational expression of the model into a SAT instance that can be solved by state-of-the-art SAT solvers (e.g. [104]), letting us leverage recent advances in the SAT solving community.

An Introduction to Alloy

Alloy [39, 80, 125] is a declarative language based on first order relational logic. All data types are represented as relations and are defined by their *type signatures*; each type signature plays the role of a type or subtype in the type system. A type signature declaration consists of the type name, the declarations of *fields*, and an optional *signature fact* constraining elements of the signature. A *subsignature* is a type signature that extends another, and is represented as a subset of the base signature. The immediate subsignatures of a signature are disjoint. A *top-level* signature is a signature that does not extend any other signature. An *abstract signature*, marked *abstract*, represents a classification of elements that is intended to be refined further by more "concrete" subsignatures.

In Alloy, a *fact* is a constraint that must always hold. A *function* is a named expression with declaration parameters and a declaration expression as a result. A *predicate* is a named logical formula with declaration parameters. An *assertion* is a constraint that is intended to follow from the facts of a model.

The *union* $(+)$, *difference* $(-)$ and *intersection* $(\&)$ operators are the standard set operators. The *join* $(.)$ of two relations is the relation obtained by taking concatenations of a tuple from the first relation and another tuple from the second relation, with the constraint that the last element of the first tuple matches the first element of the second tuple, and omitting the matching elements. For example, the join of $\{(a, b), (b, d)\}$ and $\{(b, c), (a, d), (d, a)\}$ is $\{(a, c), (b, a)\}$. The *transitive closure* $(\hat{})$ of a relation is the smallest enclosing relation that is transitive. The *reflexive-transitive closure* $(*)$ of a relation is the smallest enclosing relation that is both transitive and reflexive.

Alloy Analyzer is a software tool that can be used to analyze models written in Alloy. The Alloy code is first translated into a satisfiability problem. SAT solvers are then invoked to exhaustively search for satisfying models or counterexamples to assertions within a bounded *scope*. The scope is determined jointly by the user and the Alloy Analyzer. More specifically, the user can specify a numeric bound for each type signature, and any type signature not bounded by the user is given a bound computed by the Alloy Analyzer. The bounds limit the number of elements in each set represented by a type signature, hence making the search finite.

Realization of Web Concepts

Many of the concepts in our general model have direct realizations in our implementation. For example, we define types representing `Principals`, `NetworkEndpoints`, and `NetworkEvents`. A `NetworkEvent` represents a type of network message that has a sender (i.e., it is `from` a `NetworkEndpoint`) and a recipient (i.e., it is `to` a `NetworkEndpoint`).

It is a subsignature of `Event`, hence it also inherits all fields of `Event`. A `NetworkEvent` can be either an `HTTPRequest` or an `HTTPResponse`, which include HTTP-specific information such as a `Method` and a set of `HTTPRequestHeaders` or `HTTPResponseHeaders`, respectively:

```
abstract sig NetworkEvent extends Event {
  from: NetworkEndpoint,
  to: NetworkEndpoint
}
abstract sig HTTPEvent extends NetworkEvent {
  host: Origin
}
sig HTTPRequest extends HTTPEvent {
  method: Method,
  path: Path,
  headers: set HTTPRequestHeader
}
sig HTTPResponse extends HTTPEvent {
  statusCode: Status,
  headers: set HTTPResponseHeader
}
```

Figure 2.1 depicts some of the types used in our expression together with the relations between these types. For example, `HTTPRequest` is a subtype of `HTTPEvent`, and contains `path` and `headers` as some of its fields. This metamodel provides a conceptual map of our model. In the remainder of this section, we highlight parts of the model that lend intuition into its construction.

Principals A `Principal` is an entity that controls a set of `NetworkEndpoints` and owns a set of `DNSLabels`, which represent fully qualified host names:

```
abstract sig Principal {
  servers: set NetworkEndpoint,
  dnslabels: set DNS
}
```

The model contains a hierarchy of subtypes of `Principal`. Each level of the hierarchy imposes more constraints on how the principal can interact with the other objects in the model by adding declarative invariants. For example, the servers owned by principals

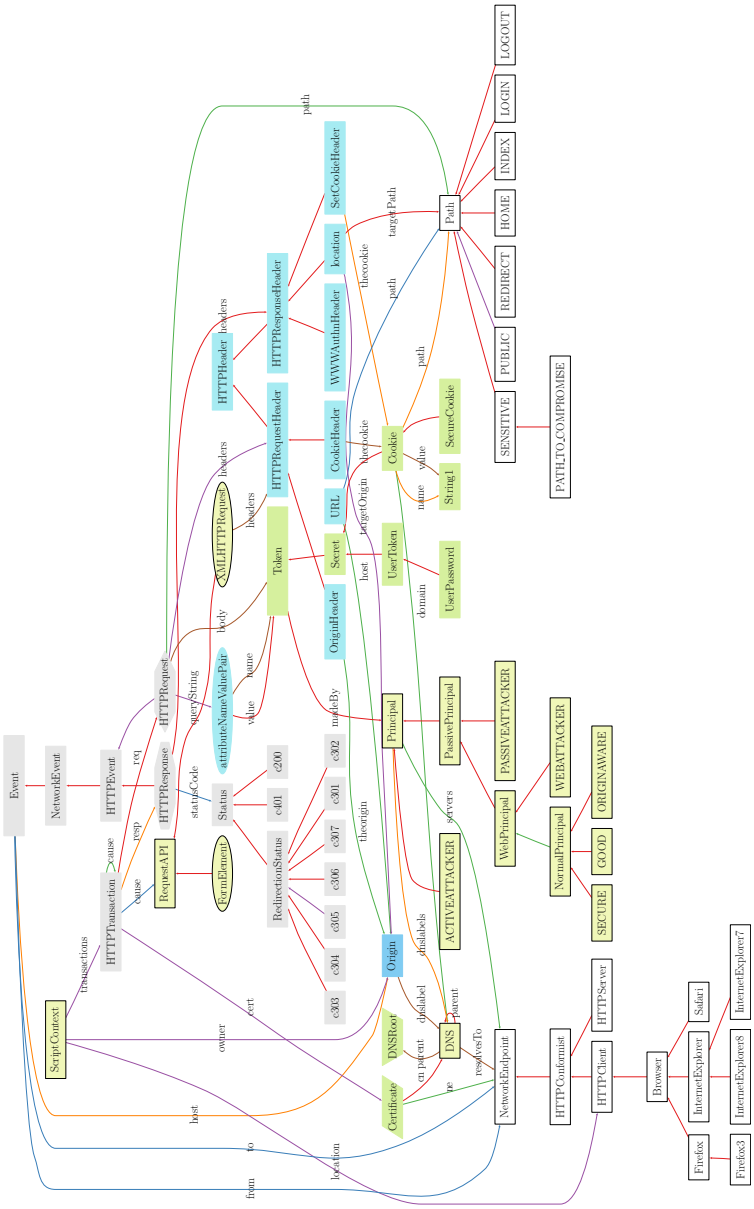

Figure 2.1: The metamodel of our formalization of web security. Red unmarked edges represent the 'extends' relationship.

who obey the network geometry (i.e., every kind of principal other than active network attackers) must conform to the routing rules of HTTP:

```
abstract sig PassivePrincipal
  extends Principal{} {
    servers in HTTPConformist
}
```

Browsers A `Browser` is an `HTTPClient` together with trusted `CertificateAuthorites` and a set of `ScriptContexts`. (For technical convenience, we store the `Browser` as a property of a `ScriptContext`, but the effect is the same.)

```
abstract sig Browser
  extends HTTPClient {
  trustedCA: set CertificateAuthority
}
sig ScriptContext {
  owner: Origin,
  location: Browser,
  transactions: set HTTPTransaction
}
```

The browser uses the `trustedCAs` to validate HTTPS certificates that `NetworkEndpoints` send during the SSL handshake.

In addition to being located in a particular `Browser`, a `ScriptContext` also has an `Origin` and a set of `HTTPTransactions`. The `Origin` is used by various browser APIs to implement the so-called "same-origin" policy. For example, the `XMLHTTPRequest` object (a subtype of `RequestAPI`) prevents the `ScriptContext` from initiating `HTTPRequests` to a foreign origin. The `transactions` property of `ScriptContext` is the set of `HTTPTransactions` (`HTTPRequest`, `HTTPResponse` pairs) generated by the `ScriptContext`.

Facts and Assertions

We find it convenient to model the browser's cookie store using *fact* statements about cookies, rather than declare a new *type signature* for it. We require that `HTTPRequests` from `Browsers` contain only appropriate cookies from previous `SetCookieHeaders`. Selecting the appropriate cookies uses a rule that has a number of cases reflecting the complexity of cookie policies in practice, part of which is shown below. More explicitly, a browser

attaches a cookie to an HTTPRequest only if the cookie was set in a previous HTTPResponse and the servers of the HTTPRequest and HTTPResponse have the same DNS label.

```
fact {
  all areq:HTTPRequest | {
    areq.from in Browser
    hasCookie[areq]
  } implies all acookie: reqCookies[areq]|
    some aresp: getBrowserTrans[areq].resp | {
      aresp.host.dnslabel = areq.host.dnslabel
      acookie in respCookies[aresp]
      happensBeforeOrdering[aresp,areq]
  }
}
```

Causality Every HTTPTransaction has a cause, which is either another HTTPTransaction (e.g., due to a redirect) or a RequestAPI, such as XMLHttpRequest or FormElement. Each RequestAPI imposes constraints on the kinds of HTTPRequests the API can generate. For example, this constraint limits the FormElement to producing GET and POST requests:

```
fact {
  all t:ScriptContext.transactions |
    t.cause in FormElement implies
      t.req.method in GET + POST
}
```

Session Integrity Using the cause relation, we can construct the set of principals involved in generating a given HTTPTransaction. The predicate below is especially useful in checking assertions of session integrity properties because we can ask whether there exists an instantiation of our model in which the attacker caused a network request that induced an honest server to undertake some specific action.

```
fun involvedServers[
    t:HTTPTransaction
    ]:set NetworkEndpoint{
 (t.*cause & HTTPTransaction).resp.from
 + getTransactionOwner[t].servers
}
```

```
pred webAttackerInCausalChain[
  t:HTTPTransaction]{
    some (WEBATTACKER.servers
    & involvedServers[t])
}
```

2.4 Case Studies

In this section, we present a series of case studies of using our model to analyze web security mechanisms. We study five web security mechanisms. For the first two, the Origin header and Cross-Origin Resource Sharing, we show that our model is sufficiently expressive to rediscover known vulnerabilities in the mechanism. For the other three, Referer Validation, HTML5 forms, and WebAuth, we use our model to discover previously unknown vulnerabilities.

Origin Header

Barth et al. proposed that browsers identify the origin of HTTP requests by including an Origin header and that web sites use that header to defend themselves against Cross-Site Request Forgery (CSRF) [17].

Modeling To model the Origin header, we added an `OriginHeader` subtype of `HTTP-RequestHeader` to the base model and required that browsers identify the origin in the header:

```
fun getOrigin[r:HTTPRequest] {
  (r.headers & OriginHeader).theorigin
}
fact BrowsersSendOrigin{
  all t:HTTPTransaction,sc:ScriptContext | {
    t in sc.transactions
  } implies {
    getOrigin[t.req] = sc.owner
  }
}
```

To model the CSRF defense, we added a new type of honest web server that follows the recommendations in the paper (namely rejects "unsafe" methods that have an untrusted Origin header):

```
pred checkTrust[r:HTTPRequest,p:Principal]{
  getOrigin[r].dnslabel in p.dnslabels
}
fact {
  all aResp: HTTPResponse | {
    aResp.from in ORIGINAWARE.servers
    and aResp.statusCode = c200
  } implies {
    let theTrans = getTransaction[aResp] |
      theTrans.req.method in safeMethods or
      checkTrust[theTrans.req,ORIGINAWARE]
  }
}
```

Vulnerability We then checked whether this mechanism satisfies session integrity. Alloy produces a counter example: if the honest server sends a POST request to the attacker's server, the attacker can redirect the request back to the honest server. Because the Origin header comes from the original `ScriptContext`, the honest server will accept the redirected request, violating session integrity. Although this vulnerability was known previously, the bug eluded both the authors and the reviewers of the original paper.

Solution One potential solution is to update the Origin header after each redirect, but this approach fails to protect web sites that contain open redirectors (which are remarkably common). Instead, we recommend naming all the origins involved in the redirect chain in the Origin header. The current Internet-Draft describing the Origin header [16] includes this fix. We have verified that the fixed mechanism enjoys session integrity in our model (for the finite sizes used in our analysis runs).

Cross-Origin Resource Sharing

Cross-Origin Resource Sharing (CORS) lets web sites opt-out of some of the browser's security protections. In particular, by returning various HTTP headers, the site can instruct the browser to share the contents of an HTTP response with specific origins, or to let specific origins issue otherwise forbidden requests. CORS is somewhat complex because it distinguishes between two kinds of requests: simple requests and complex requests. Simple requests are supposedly safe to send cross-origin, whereas complex requests require a *pre-flight* request that asks the server for permission before sending the potentially dangerous request. CORS is a good case study for our model for two reasons: (1) maintaining the web security invariants is a key requirement in the design, driving the distinction between simple and complex requests; (2) complex requests can disrupt session integrity if the pre-flight request is not handled properly.

Modeling We model the `PreflightRequest` as a subtype of `HTTPRequest` imbued with special semantics by the browser. Even though the implementations of CORS in browsers reuse the XMLHttpRequest JavaScript API, we model CORS using a new `RequestAPI`, which we call `XMLHttpRequest2`, making it easier to compare the new and old behavior. CORS involves a number of HTTP headers, which we model as subtypes of `CORSResponseHeader`. Finally, we model servers as `NetworkEndpoints` that never include any CORS headers in HTTP responses.

```
fact {
 all p:PreFlightRequest | {
  p.method = OPTIONS and
  some p.headers & AccessControlRequestMethod
  and some p.headers & OriginHeader and
  some p.headers & AccessControlRequestHeaders
}
fact {
  all t:HTTPTransaction,sc:ScriptContext |{
    t in sc.transactions and
    t.^cause in
      (XMLHTTPRequest2+HTTPTransaction)
  } implies {
    isPreFlightRequestTransaction[t]
    or isSimpleCORSTransaction[t]
    or isComplexCORSTransaction[t]
    or (not isCrossOriginRequest[t.req])
  }
}
```

Vulnerability Alloy produced a (previously known) counter-example that breaks a key web security invariant because a legacy server might redirect the pre-flight request to the attacker's server. According to the current W3C Working Draft [87], browsers follow these redirects transparently, letting the attacker's server return a CORS header that opts the legacy server into receiving new kinds of requests (such as DELETE requests). This attack is fairly practical because many web sites contain open redirectors. Notice that we did not need to model open redirectors explicitly. Instead, a legacy server might redirect requests to arbitrary locations because the model does not forbid these responses.

Solution A simple solution is to ignore redirects for preflight requests. The most recent Editor's Draft [86] (which is more up-to-date) has precisely this behavior. We verify the security of the updated protocol (up to a finite size) in our model by adding the following requirement:

```
fact {
  all first:HTTPTransaction | {
    first.req in PreFlightRequest and
    first.resp.statusCode in RedirectionStatus
  } implies no second:HTTPTransaction |
    second.cause = first
}
```

Referer Validation

A recent paper [85] proposes that web sites should defend against CSRF and Cross-Site Scripting (XSS) by validating the Referer header. The authors claim that these attacks occur "when a user requests a page from a target website s by following a link from another website s' with some input." To defend against these attacks, the web site should reject HTTP requests unless (1) the Referer header is from the site's origin or (2) the request is directed at a "gateway" page, that is carefully vetted for CSRF and XSS vulnerabilities; see Figure 2.2. What makes this security mechanism a particularly interesting case study is that the authors model-check their mechanism using Alloy. However, their model omits essential details like HTTP Redirects and cross-origin hyperlinks. As a result, it is unable to uncover a vulnerability in their mechanism.

Modeling Because the Referer header is already part of the model, we only needed to add a new class of principal: `RefererProtected` that exhibits the required behavior. We then added a constraint that HTTP requests with external Referers are allowed only on the "`LOGIN`" page:

```
fact {
 all aReq:HTTPRequest | {
  (getTransaction[aReq].resp.from
      in RefererProtected.servers )
  and isCrossOrigin[aReq]
  } implies aReq.path = LOGIN
}
```

Vulnerability Alloy produces a counterexample for the session integrity condition because the attacker can mount a CSRF attack against a `RefererProtected` server if the server sends a request to the attacker's server first (see the dashed lines in Figure 2.2). For example, if the attacker can inject a hyperlink into the honest site, the user might follow that hyperlink, generating an HTTP request to the attacker's server with the honest site's URL in the Referer header. The attacker can then redirect that request back to the honest

server. This previously unknown vulnerability in Referer validation is remarkably similar to the vulnerability in the Origin header CSRF defense described above.

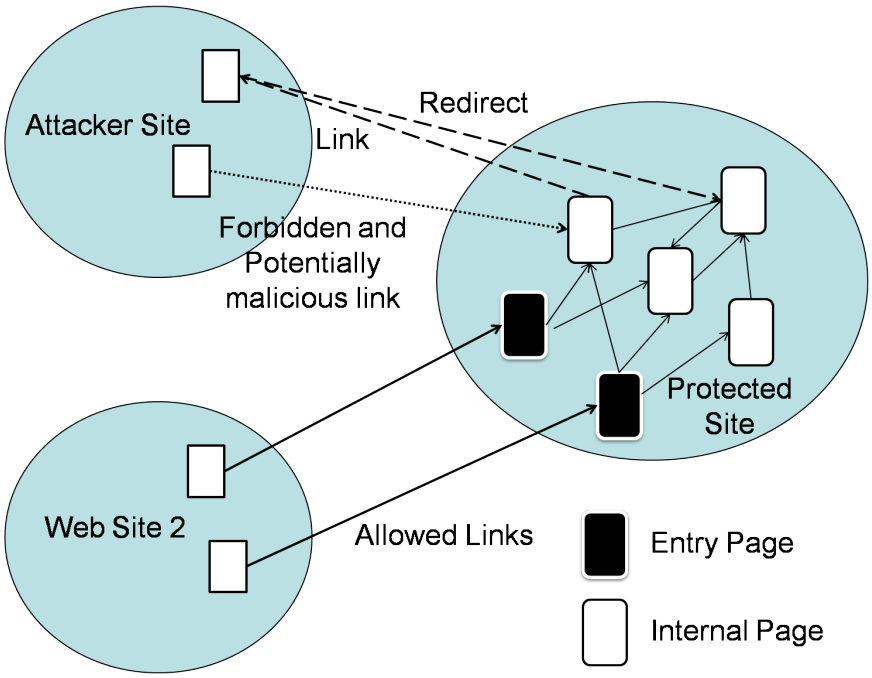

Figure 2.2: Vulnerability in Referer Validation. This figure is adapted from [85], with the attack (dashed line) added.

Solution This vulnerability is difficult to correct on the current web because the Referer header is already widely deployed (and therefore, for all practical purposes, immutable). One possible solution is for the web site to suppress all outgoing Referer headers using, for example, the `noreferrer` relation attribute on hyperlinks.

HTML5 Forms

HTML5, the next iteration of the HyperText Markup Language, adds functionality to the `FormElement` API to generate HTTP requests with PUT and DELETE methods. To avoid introducing security vulnerabilities into existing web sites, the specification restricts these new methods to requests that are sent to a server in the same origin as the document containing the form.

Modeling Modeling this extension to the web platform was trivial: we added PUT and DELETE to the whitelist of methods for `FormElement` and added a requirement that these requests be sent to the same origin:

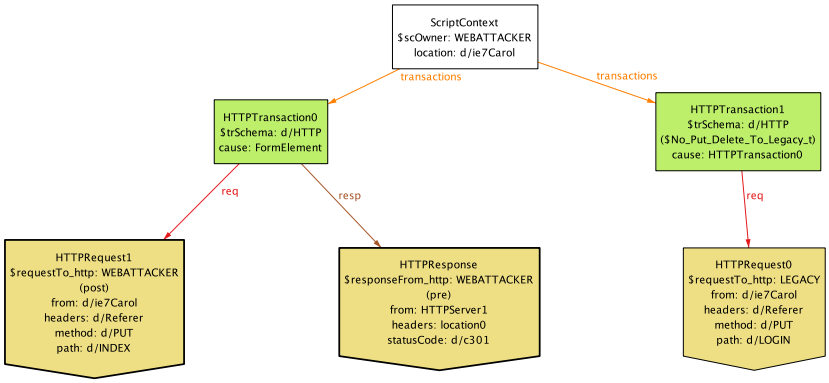

Figure 2.3: Counterexample generated by Alloy for the HTML5 form vulnerability.

```
t.req.method in PUT+DELETE implies
  not isCrossOriginRequest[t.req]
```

Vulnerability Alloy produces a counterexample that breaks a web security invariant. An attacker can generate a PUT request to `attacker.com` and then redirect that request to an honest server, causing the server to receive an unexpected PUT request. Figure 2.3 depicts part of the counterexample produced by Alloy. Although apparently simple, this vulnerability had not been previously detected in HTML5 despite extensive review by hundreds of experts. (To be fair, HTML5 is enormous and difficult to review in its entirety.)

Solution The easiest solution is to refuse to follow redirects of PUT or DELETE requests generated from HTML forms. We have verified this fix (up to a finite size) using our model. We have contributed our findings and recommendation to the working group [11] and the working group has adopted our solution.

WebAuth

In our most extensive case study, we analyze WebAuth [123], a web-based authentication protocol based on Kerberos. WebAuth is deployed at a number of universities, including Stanford University. WebAuth is similar to Central Authentication Service (CAS) [96], which was originally developed at Yale University and has been deployed at over eighty universities [81], including UC Berkeley. Although we analyze WebAuth specifically, we have verified that the same vulnerability exists in CAS.

Protocol Of all our case studies, WebAuth most resembles a traditional network protocol. However, new security issues arise when embedding the protocol in web primitives because

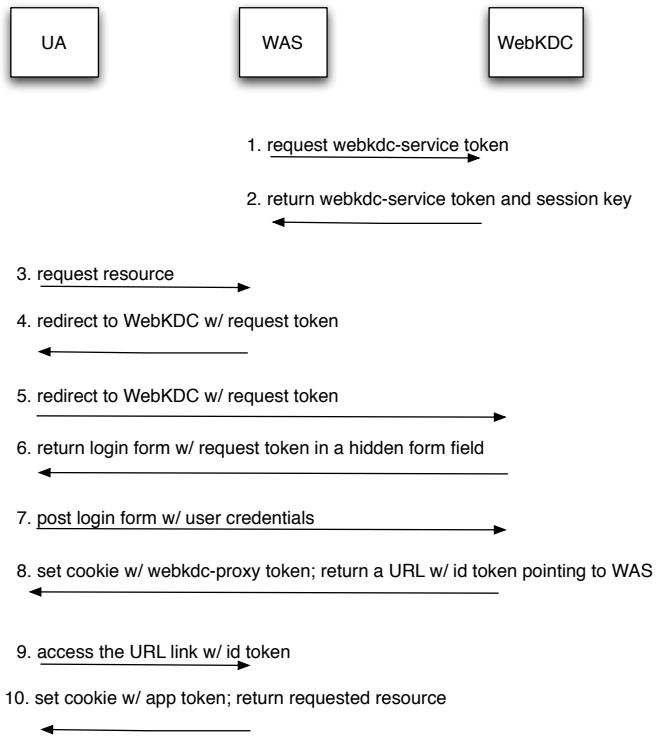

Figure 2.4: The WebAuth protocol

the web attacker can interact with the protocol in different ways than a traditional network protocol attacker can interact with, say, Kerberos. The WebAuth protocol involves three roles:

1. User-Agent (UA), the user's browser,
2. WebAuth-enabled Application Server (WAS), a web server that is integrated with WebAuth, and
3. WebKDC, the web login server.

Although WebAuth supports multiple authentication schemes, its use of tokens and keys closely resembles Kerberos. The WebKDC shares a private key with each WAS, authenticates the user, and passes the user's identity to the WAS via an encrypted token (i.e., ticket). WebAuth uses so-called "Secure" cookies to store its state and HTTPS to transmit its messages, ostensibly protecting the protocol from network attackers.

The main steps of the protocol are depicted in Figure 2.4. We describe the steps below:

- *WAS Initialization (Steps 1–2).* At startup, the WAS authenticates itself to the WebKDC using its private Kerberos key, and receives a *webkdc-service token* and a session key.

- *Login (Steps 3–10).* When the user wishes to authenticate to the WAS, the WAS creates a *request token* and redirects the UA to the WebKDC, passing the token in the URL. The WebKDC authenticates the user (e.g., via a user name and password), stores a cookie in the UA (to authenticate the user during subsequent interactions without requiring the user to type his or her password again), and redirects the user back to the WAS, passing an *id token* identifying the user in the URL. The WAS then verifies various cryptographic properties of the token to authenticate the user. Finally, the WAS stores a cookie in the UA to authenticate the user for the remainder of the session.

Modeling To model WebAuth, we added a number of type signatures for the WebAuth tokens, and predicates for HTTP message validation. For example, the `WAPossessToken-ViaLogin` predicate tests whether the WAS has received a proper *id token*:

```
pred WAPossessTokenViaLogin[httpClient:
HTTPClient, token:WAIdToken, usageEvent:Event]
{ some t1:HTTPTransaction|{
    happensBeforeOrdering[t1.req, usageEvent]
    and t1.req.path = LOGIN and
    t1.req.to in WAWebKDC and
    t1.req.from in httpClient and
    t1.resp.statusCode in RedirectionStatus and
    WAContainsIdToken[t1.resp, token] and
    token.creationTime = t1.resp.post and
    token.username in httpClient.owner
  }
}
```

The confidentiality of tokens is key to modeling the security properties of the WebAuth protocol. As with cookies, we require an HTTPClient to have received a token in a previous HTTP request before including the token in another HTTP request:

```
fact {
  all httpClient:HTTPClient,req:HTTPRequest,
  token:WAIdToken | {
    req.from in httpClient and
    WAContainsIdToken [req, token]
  } implies
    WAPossessToken[httpClient, token, req]
}
```

Notice that we permit the attacker to be registered as a user at the WebKDC and to send and receive HTTP requests and responses to the WAS and the WebKDC both directly and via the browser.

Vulnerability Alloy produces a counterexample showing that the WebAuth protocol does not enjoy session integrity because the attacker can force the user's browser to complete the login procedure. Worse, the attacker can actually force the user's browser to complete the login procedure *with the attacker's credentials*. This previously unknown vulnerability is a variation of login CSRF [17], a vulnerability in which the attacker can confuse a web site into authenticating the honest user as the attacker. We confirmed this attack on the Stanford WebAuth implementation by embedding a link containing an *id token* of the attacker in an email, and verifying that a user who clicks on the link is logged in as the attacker to the system. The same attack works against the CAS deployment at U.C. Berkeley. Login CSRF vulnerabilities have a number of subtle security consequences [17]. One reason that Stanford or Berkeley might be concerned about these vulnerabilities is that if some information (such as a download) is available only to registered students, a registered student could log in and then export a link that allows others to access protected information, without revealing the password they used to authenticate.

The vulnerability arises because the WAS lacks sufficient context when deciding whether to send message 10. In particular, the WAS does not determine whether it receives message 3 and message 9 from the same UA. An attacker can run the first eight steps of the protocol with the WAS and WebKDC directly, but splice in the UA by forwarding the URL from message 8 to the user's browser. In a traditional network protocol, this attack might not succeed because the UA would not accept message 10 without previously sending message 3, but the attack succeeds in the web setting because (1) the UA is largely stateless, and (2) the attacker can induce the UA to send message 9 by showing the user a hyperlink on `attacker.com`.

Solution We suggest repairing this vulnerability by binding messages 3 and 9 to the same UA via a cookie. Essentially, the WAS should store a nonce in a cookie at the UA with message 4 and should include this nonce in the *request token*. In message 8, the WebKDC includes this nonce in the *id token*, which the UA forwards to the WAS. The WAS, then, should complete the protocol only if the cookie received along with message 9 matches the nonce in the *id token*. We have verified that the fixed mechanism enjoys session integrity in our model (up to a finite size).

The security of this scheme is somewhat subtle and relies on a number of security properties of cookies. In particular, this solution is not able to protect against active network attackers because cookies do not provide integrity: an active network attacker can overwrite the WAS cookie with his or her own nonce (even if the cookie is marked "Secure"). However, active network attackers can mount these sorts of attacks against virtually all web applications because an active network attacker can just overwrite the final session

Case Study	Lines of new code	No. of clauses	CNF gen. time (sec)	CNF solve time (sec)
Origin Header	25	977,829	26.45	19.47
CORS	80	584,158	24.07	82.76
Referer Validation	35	974,924	30.75	9.06
HTML5 Forms	20	976,174	27.67	73.54
WebAuth	214	355,093	602.4	35.44

Table 2.1: Statistics for each case study

cookie used by the application, regardless of the WebAuth protocol. Nonetheless, our proposed solution improves the security of the protocol against web attackers.

2.5 Measurement

We implemented the model in the Alloy Analyzer 4.1.10. We wrote our security invariants as assertions and asked Alloy to search for a counterexample that violates these assertions, bounding the search to a finite size for each top-level signature. This bound is also called the *scope* of the search, and for our experiments was set at 8. Alloy also allows us to specify finer-grained bounds on each type, but we do not use this feature in our experiments.

For each case study, we counted the number of lines of new Alloy code we had to add to the base model (some 2,000 lines of Alloy code) to discover a vulnerability and measured the time taken by the analyzer to generate the conjunctive normal form (CNF) as well as the time taken by the SAT solver (minisat) to find a solution (see Table 2.1). All tests were performed on an Intel Core 2 Duo CPU 3.16Ghz with 3.2GB of memory. As is common in other SAT solving applications, we observe no clear correlation between the number of lines of new code, the number of clauses generated, and the CNF generation and solving times.

The SAT solver is able to find a counterexample (if one exists) in a few minutes. In the absence of a counterexample, the time taken by the SAT solver increases exponentially as the scope is increased. To quantify this behavior, we measured the time taken to analyze the HTML5 form vulnerability before and after we implemented the fix in the model (see Figure 2.5). Recall that, after the fix, Alloy is not able to find a counterexample.

2.6 Summary of Results

We presented several steps toward a formal foundation for web security. The model described comprises key web security concepts, such as browsers, HTTP, cookies, and script contexts, as well as the security properties of these concepts. We have a clearly defined threat model, together with a spectrum of threats ranging from web attackers to

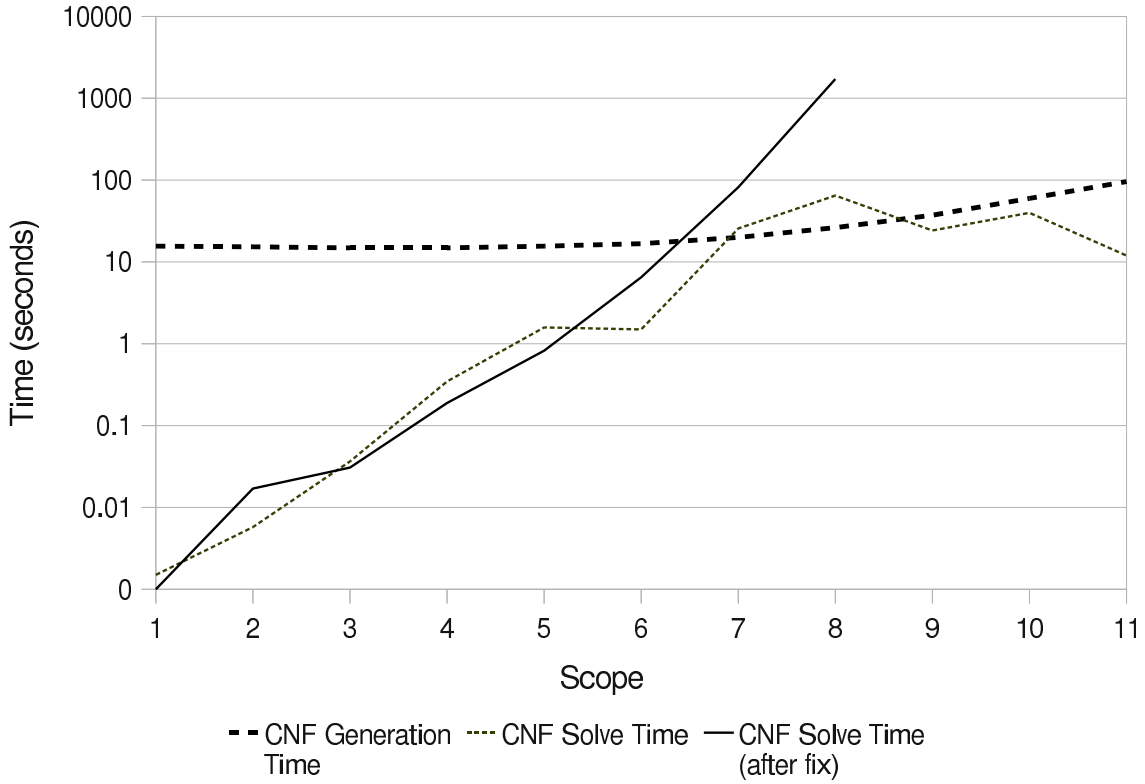

Figure 2.5: Log-scale graph of analysis time for increasing scopes. The SAT solver ran out of memory for scopes greater than eight after the fix.

network attackers to gadget attackers. In this model, we have also included two high-level security properties that appear to be commonly required of web security mechanisms.

We have implemented core portions of our model in Alloy, which lets us execute the model and check whether various web security mechanisms actually have the security properties their designers desire. We have used this implementation to study five examples, ranging in complexity from a small tweak to the behavior of the HTML form element in HTML5 to a full-blown web single sign-on protocol based on Kerberos. In each case, we found vulnerabilities, two previously known, three previously unknown. These vulnerabilities arise because of the complex interaction between different components of the web platform.

As the web platform continues to grow, automated tools for reasoning about the security of the platform will increase in importance. Already web security is sufficiently complex that a working group of experts miss "simple" vulnerabilities in web platform features. These vulnerabilities appear simple in retrospect because only a tiny subset of the platform is required to demonstrate the *insecurity* of a mechanism, whereas knowledge of the entire platform is required to demonstrate its *security*. Of course, our model (and

our implementation) does not capture the entire web platform. However, we believe our model is an important first step towards creating a formal foundation for web security.

Chapter 3

Privilege Separation for HTML5 Applications

In the previous chapter, we examined how to secure the protocols complex HTML5 applications rely on to securely communicate with the cloud. Next, we present techniques to secure the HTML5 application itself. This work was presented at the *21st Usenix Security Symposium, 2012* at Bellevue, Washington. This is joint work with Prateek Saxena and Dawn Song.

3.1 Introduction

Privilege separation is an established security primitive for providing an important second line of defense [121]. Commodity OSes enable privilege separated applications via isolation mechanisms such as LXC [93], seccomp [58], SysTrace [113]. Traditional applications have utilized these for increased assurance and security. Some well-known examples include OpenSSH [114], QMail [20] and Google Chrome [10]. 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

[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

components, which does *not* provide proper isolation, due to special powers granted to sub-domains in the cookie and document.domain behaviors.

Recent research [32, 82] 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 [10]. Similarly, recent research proposes to partition HTML5 applications in "N privilege rings", similar to the isolation primitives supported by x86 processors [82]. We observe two problems with these approaches. First, the 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 chapter, we rethink how to achieve privilege separation in HTML5 applications. In particular, we present 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 same-origin policy. 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 3.3. 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 [58, 93, 113].

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 [67]. 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 [116]. 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 [10] and the Windows 8 Metro platform [98]. Google Chrome and Windows 8 rely on applications declaring install-time permissions that end users can check [15]. Multiple studies have found permission systems to be inadequate: the bulk of popular applications run with powerful permissions [6, 48] and users rarely check install-time permissions [49]. 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 or security teams, 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.

3.2 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 [15], Chrome packaged web application store [63] and Windows 8 Metro applications [98], 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 executes 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.

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 [30]. 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 (hypothetical) `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 $\alpha_1, \alpha_2 ...$, *all* components of the application run with the privileges $\bigcup \alpha_i$, leading to over-privileging. As we show in Section 3.5, 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.

We measured the TCB inflation for the top 50 Chrome extensions. Figure 3.1 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.

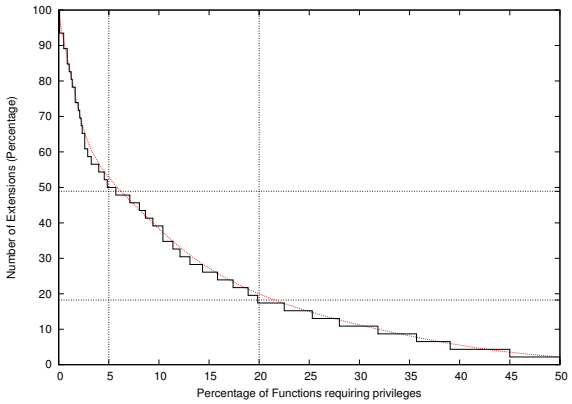

Figure 3.1: CDF of percentage of functions in an extension that make privileged calls (X axis) vs. the fraction of extensions studied (in percentage) (Y axis). The lines for 50% and 20% of extensions as well as for 5% and 20% of functions are marked.

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 3.5) discusses unbundling and TCB reduction on a real world screenshot application. We achieved a 58x TCB reduction.

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 injection vulnerabilities, we are interested, instead, in reducing the TCB. Reducing the TCB also helps our second goal of easier audits.

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 use 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 work 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 work.

3.3 Design

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

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 3.2 shows our proposed HTML5 application architecture. In our design, 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 3.3. The parent guards access to a

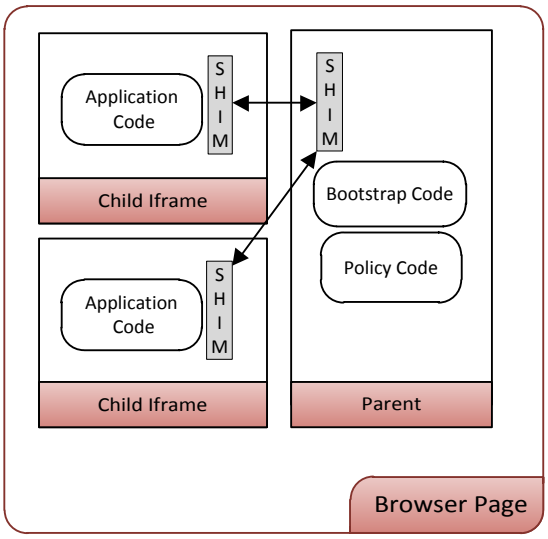

Figure 3.2: High-level design of our proposed architecture.

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 [113]. 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 [116]. 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 [62]). 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 3.4 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.

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 [7, 45, 55, 66, 94].

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. For example, a problem in privilege-separated applications written in C is the exchange of pointers across the privilege boundary, leading to possible errors [53, 130]. 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.

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 `iframe`s 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 [12]. 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 current (as of writing) versions of all browsers [43].

Modern browsers also support setting the sandbox attribute on a page via HTTP headers, as part of the Content Security Policy (CSP) header. We do not focus on this primitive in this chapter but it is easy to adapt our code to use the CSP sandbox instead. We caution the reader that the CSP sandbox is not widely supported. For example, Mozilla Firefox currently does not support this mechanism (although, developers are working on it [26]). Unlike our design, older browsers that do not support the sandbox via the CSP header would run child code with full privileges, defeating our security invariants. Instead, relying on bootstrap code (as we do) means that the parent code can check for sandbox support before executing child code.

Enforcement of Security Invariants. To enforce security invariants 1 and 2 in the parent, our implementation utilizes the Content Security Policy (CSP) [127]. 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 [55, 66, 94].

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 rely on 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 [67], 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 work.

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

3.4 Implementation

As outlined in Section 3.3, 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 3.5).

Figure 3.3 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 3.3). 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.

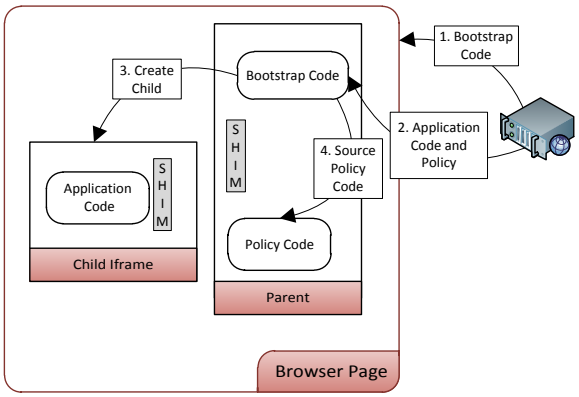

Figure 3.3: Sequence of events to run application in sandbox. Note that only the bootstrap code is sent to the browser to execute. Application code is sent directly to the parent, which then creates a child with it.

```
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 3.1: Bootstrap Code (JavaScript)

The parent proceeds to create a temporary origin, unprivileged iframe using the downloaded code as the source (**Step 3**, Figure 3.3). Listing 3.1 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.

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 rely on 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 3.4 outlines the typical events involved in proxying a privileged call. First, the child shim defines a stub implementation of the privileged APIs (for example, `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 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. `chrome.tabs.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

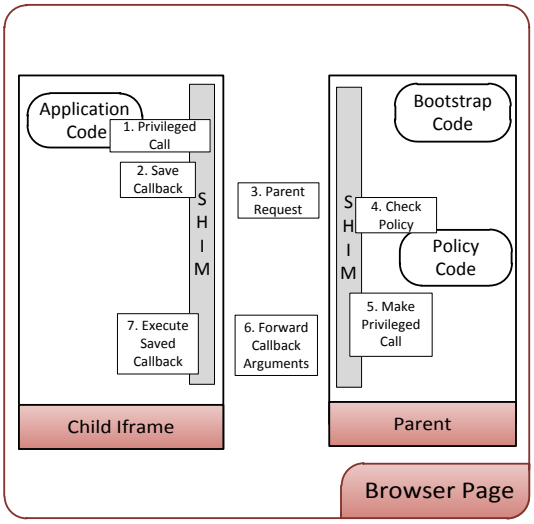

Figure 3.4: Typical events for proxying a privileged API call. The numbered boxes outline the events. The event boxes span the components involved. For example, event 4 involves the parent shim calling the policy code.

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

Listing 3.2: Child shim for captureVisibleTab

function marshals all the arguments and sends it to the parent. Listing 3.2 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. 3.4) 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.

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.

```
//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 3.3: Parent shim for captureVisibleTab

Parent Shim. On receiving the message, the parent's shim first checks with the policy (Step 4 in Fig. 3.4 and line 5 in Listing 3.3) 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 3.3) 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 executes the saved callback with the arguments provided by the parent. (Step 7 in Figure 3.4 and line 6 in Listing 3.4). The saved callback is then deleted from the cache (Line 7).

Persistent State. We take a different approach to data persistence APIs like `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

```
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 3.4: Child shim for captureVisibleTab: Part 2

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

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

Listing 3.5: localStorage Shim in the Child Frame

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 3.5. 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.

3.5 Case Studies

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

- 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 3.1 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 workr 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 [116]. Due to licensing restrictions, we are unable to release the other case studies publicly.

Table 3.1: 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

Awesome Screenshot

The Awesome Screenshot extension allows a user to capture a screenshot of a webpage similar to our running example [46]. 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 Awesome Screenshot 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 3.3 (Figure 3.2). 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.

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.

[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.

Unbundling. In the original version of Awesome Screenshot, `editor.html`, the image editor 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.

SourceKit Text Editor

The SourceKit text editor is an HTML5 text editor for a user's documents stored on the Dropbox cloud service [47]. It uses open source components like the Ajax.org cloud editor [2] and Dojo toolkit [133], in conjunction with the Dropbox REST APIs [47].

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 3.4). 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 3.1.

Unbundling. Functionally, SourceKit is a single Chrome application, and no bundling has occurred in its design. Popular Web sites (like GitHub [73]), use the text editor module as an online text editor [2]. 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 [47]. At first run, SourceKit opens Dropbox in a new tab, where the user can grant SourceKit the requisite OAuth

access token [107]. 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 [25]. By default, the Dropbox JS API [83] 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.

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 application 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 [87].

In our privilege-separated redesign, a restrictive policy on the child mitigates SQL Buddy bundling. The parent allows the child `XMLHttpRequest` access only to URIs of the form `/sqlbuddy/<filename>.php`. No other privilege is available to SQL Buddy code, including `document.cookie`, `localStorage`, or `XMLHttpRequest` to the main application's

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

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.

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 [115] 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 3.1. We believe that our analysis is overly conservative, being syntactic, so these numbers represent only an undercount of the over-privileging in these applications.

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

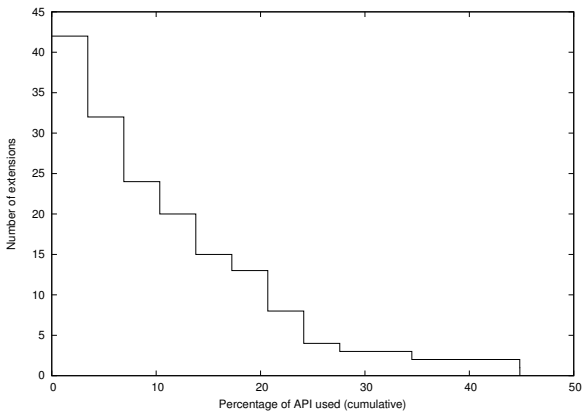

Figure 3.5: Frequency distribution of event listeners and API calls used by the top 42 extensions requiring the `tabs` permission.

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 3.5 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.

3.6 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.

3.7 Summary of Results

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 use 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. In the next chapter, we propose extending this primitive to enforce stronger data confinement properties.

Chapter 4

Data-Confined HTML5 Applications

This work was presented at the *18th European Symposium on Research in Computer Security, 2013* at Egham, United Kingdom. This is joint work with Frank Li, Warren He, Prateek Saxena, and Dawn Song.

4.1 Introduction

In the previous chapter, we discussed how privilege separation allows us to reduce the trusted computing base (TCB) of HTML5 applications by breaking them up into an arbitrary number of unprivileged components. While privilege separation enables TCB reduction, it does not give any guarantees on the flow of data available to an unprivileged child. On modern and upcoming platforms, 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 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

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

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 [71, 142]. As we explain in Section 4.3, 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 4.2).
- We identify the limitations of current security primitives in the HTML5 platform that make them insufficient for implementing data-confinement invariants (Section 4.3).
- 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 4.4) and discuss how to implement such a new primitive without affecting the security invariants maintained by the HTML5 platform (Section 4.4).
- We demonstrate the practicality of our approach by modifying four applications that handle sensitive data to provide strong data confinement guarantees (Section 4.6). All our code and case studies are publicly available online [117].

4.2 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.

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`.

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 [69], 2011 [137], and 2012 [129].

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 [120] 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.

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 [110]. 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. As we present in Section 4.6, we studied the top twenty most popular extension on the Google Chrome platform and found that 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.

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

4.3 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 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 [121]. 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 4.4.

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 [71, 142].

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

Table 4.1: 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**

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 [58, 88, 113]. Previous work also developed techniques to automate identification and isolation of subcomponents that process sensitive data [25, 88]. Our work shares these design principles, but targets HTML5 applications.

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, privilege separation (Chapter 3) of HTML5 applications does not provide any confinement guarantees. An unprivileged child can leak data by making a request for an image or including a CSS style from a remote host.

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 [36]. IBEX proposed writing extensions in a high-level language (FINE) amenable to deep analysis to ensure conformance with specific

policies [67]. 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 [38, 55, 94]. Static analysis systems cannot reason about dynamic constructs such as `eval`, which are used pervasively by existing applications [118] and modern JavaScript libraries [1]. As a result, such mechanisms have a high compatibility cost. When combined with rewriting techniques, such as Cajoling [55], 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 [3] 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 [75]. 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 a primitive not directly supported by browsers is its 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 [54, 59, 60, 61] and AdSafe [94, 111].

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 [79] and timing channels [13]) or self exfiltration channels [33], 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 [28, 140, 144]. 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, which we formalized in Chapter 2. To recap, the web attacker cannot tamper with or observe network traffic for other web origins and cannot subvert the integrity of the HTML5 platform itself. None the less, defending against a network attacker is easier with a data-confined sandbox—the privileged parent script need only restrict all communication to a secure (https) channel.

4.4 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 4.4). Our key contribution is identifying that the shrewd design of the DCS primitive provides high assurance with minimal compatibility concerns (Section 4.4). 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 4.4.

Design of DCS

Figure 4.1 presents the architecture of an application using the DCS design. Our design extends the privilege separated design from Chapter 3. 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.

As discussed in Chapter 3, modern HTML5 platforms allow applications to run arbitrary code (specified via a `data:`/`blob:` URI) in a temporary, unprivileged origin. Privilege

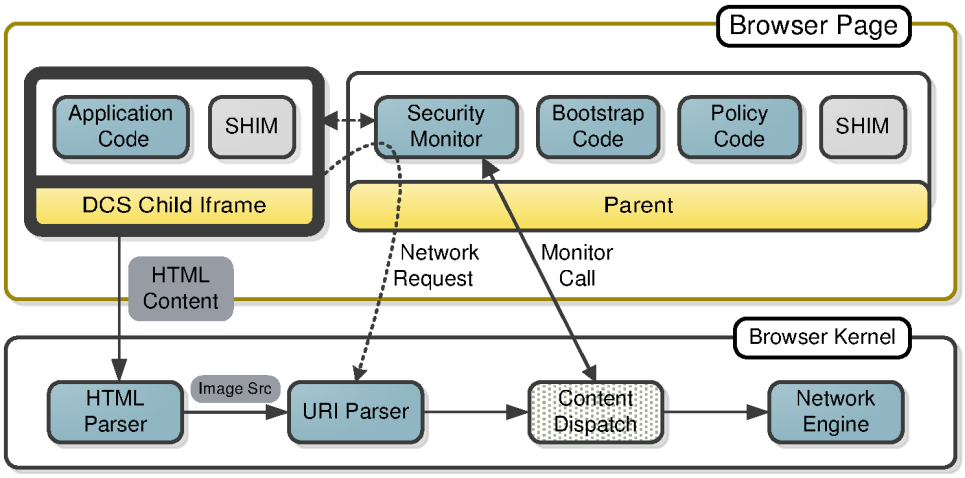

Figure 4.1: High-level design of an application running in a DCS. The only component that runs privileged is the parent. The children run in data-confined sandboxes, with no ambient privileges and all communication channels monitored by the parent.

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 4.1).

The parent can enforce policies on the requests it receives over this `postMessage` channel from its unprivileged children. The parent uses its privileged interfaces to fulfill approved requests, such as authenticated `XMLHttpRequest` calls (curved dotted arrow in Figure 4.1). 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 4.1. 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 4.1). 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 `https://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.

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 4.3, 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 4.5, 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 4.5. 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 [124]. 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. With the powerful metaprogramming capabilities of JavaScript, it is also possible to write shim code that transparently intercepts postMessage calls and forwards messages to the parent, which in turn can forward messages to the original target if allowed. We have not implemented this.

Requests made by the DCS have an empty `Referer` and `Origin` header. Resource requests that require the application's origin in these headers will fail. This design is intentional: the ability to make requests with the application's URI in the `Referer` and `Origin` headers is an authority not available to the unprivileged children.

The lack of the correct `Referer` and `Origin` headers did not affect any of our case studies. Currently, only browsers based on WebKit send the `Origin` header, and web applications do not rely on these headers, as privacy conscious users often turn them off. To maintain compatibility with servers that rely on these features, the DCS can send a message to the parent, requesting it to make the appropriate request. An alternate design,

which we have not investigated, would be to insert the parent's URI in the `Referer` and its origin in the `Origin` headers automatically.

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 [70]. 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.

4.5 Implementation

We implemented support for data-confined sandboxes in the Firefox browser. Our modified browser and our case studies (Section 4.6) are all available online [117]. 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 inside the Firefox source code at `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 `sandbox`ed `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.

4.6 Case Studies

We retrofit our application architecture to four web applications to demonstrate the practicality of our approach. All our case studies, like our browser modifications, are open-source and freely available [117].

- As an example of a password manager, we retrofit the Clipperz password management web application [37]. Similar to LastPass, sensitive user data is stored encrypted in the cloud and decrypted based on the user password.
- As an example of an SSO service, we modified the BrowserID implementation to use our architecture. BrowserID is Mozilla' implementation of the verified email protocol for single sign-on, and is used by Mozilla services like Bugzilla, MDN, and a number of other third-party sites [24]. Users can also download and run a local instance of BrowserID.
- We also demonstrate the practicality of our approach on the OpenEMR electronic medical record system. OpenEMR is a popular open-source electronic medical record management software [108]. A number of federal and state laws have strict requirements for medical data confinement and patient privacy: we demonstrate how our architecture can enforce useful policies in the OpenEMR client-side application.
- As an example of a database administration interface, we modify the SQL Buddy web application to ensure data-confinement invariants for the (possibly sensitive) database. Our redesign allows us to give strong confidentiality guarantees for the SQL Buddy interface.
- Finally, we demonstrate the wide-spread need for data confinement by studying the twenty most popular Chrome extensions, and identifying their data-confinement requirements. We find that 16 of the extensions we studied maintain a data-confinement invariant.

Table 4.2 lists our case studies and summarizes our results. We find that our redesigns are minimally intrusive (fewer than 184 lines changed in each of our case studies) and achieve significant TCB reduction. We evaluate our design on these case studies by measuring (a) the TCB reduction, (b) the lines of code changed to implement our redesign, and (c) the invariants we are able to enforce on the redesigned applications.

Clipperz

Clipperz is an open-source HTML5 password manager that allows a user to store a variety of sensitive data, such as website logins, bank account credentials, and credit card information [37]. Sensitive data is stored encrypted in the cloud and is decrypted 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.

The application relies on open-source components including the MochiKit library [102] and the YUI library [141]. In sum, Clipperz consists of 1.4MB of JavaScript code, all of which runs in a single security principal, with access to all sensitive data. The Clipperz

Table 4.2: List of our case studies, as well as the individual components and policies in our redesign.

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

application uses inline scripts and `data:` URIs extensively. We found that enforcing strong CSP restrictions to protect against XSS breaks several subcomponents of the Clipperz application.

Privilege Separation We modified Clipperz to execute its application code in an unprivileged DCS. We reused existing shim code (Chapter 3) to achieve seamless privilege separation. The one key change was to proxy handling links (such as Clipperz help page) and 'direct logins,' to the parent, since our design does not grant a DCS the privilege to open pop-up windows. Privilege separating the Clipperz application required changing 67 lines of code. Note that privilege separation in and of itself does not ensure data confinement: if an attacker compromises the code in the child, it can send data to an attacker website, for example, by loading an image.

Two application specific changes were necessary. The first change was in the implementation of the direct login feature. Originally, the feature relies on the MochiKit API to manipulate the DOM of a separate window, which is disallowed from within the DCS. In our redesign, we replace the MochiKit usage within the direct login implementation with standard JavaScript window and DOM manipulation. These privileged calls can be proxied to the parent through the child shim. The other change was in anchor tag links. The application's JavaScript code generates a number of anchor elements, typically in association with a stored form or login information. Navigation via the link is also disallowed within a DCS. To allow navigation, the "onClick" anchor tag flag are set upon element creation to a function that postMessages the parent requesting the link be open.

The changes were minor, with 20 lines of Javascript code changed and 9 hard-coded HTML anchor tags manually modified to include the "onClick" flag. We manually tested

the application functionality thoroughly and did not observe any incompatibilities.

Data-Confinement Invariants Our privilege-separated design executes the Clipperz application code in a data-confined child, which allows the parent to enforce a confinement policy. One implemented policy was simple, like the security monitor in Listing 4.1, which allows the DCS child access only to `postMessage` and a whitelist of images and JavaScript libraries. The flexibility of our primitive also allows for powerful, stateful policies. In this case, our more expressive monitor function allows the Clipperz application to make network requests only until it downloads the password database; once the DCS child downloads the password database, the monitor function disallows further network access.[3] Relying on a whitelist of network resources means that we can protect against inadvertent disclosure of the user entered master password via a developer mistake or scriptless injection attacks. On the current HTML5 platform (without DCS) this requires an auditor or the security engineer to trust a much larger TCB.

Although our redesign makes data theft significantly harder, a compromised instance of Clipperz still has one (self) exfiltration channel. Clipperz's 'direct login' functionality navigates to a saved webpage and auto-fills the login credentials. A malicious script, executing in the compromised DCS, can request the parent to 'direct login' to an attacker controlled webpage, and provide the username and password for (say) facebook.com. This would allow the attacker controlled webpage to learn the user credentials for facebook.com.

In our redesign, we mitigate the above attack by creating two children: the UI component and the non-UI component. The UI component does not have direct access to the 'direct login' feature. Instead, a direct login requires sending a message to the non-UI component. The new component retrieves the associated credentials and completes the direct login process. In contrast to the vault page, this component does not need complex UI code and other supporting JavaScript libraries. In our implementation, this component executes with a strong CSP, providing higher assurance.

The DCS approach affords us the flexibility of enforcing a different policy on each child. The security monitor allows images and scripts to be loaded in the UI component from a set of whitelisted URLs. The direct login component has no UI and the security monitor disallows image loads in that component. Both components are always allowed access to the parent via `postMessage`. Again, the policy is temporal in nature, where upon database access, the security monitor blocks all communication in both components except to the parent.

Monitor Code We use the code in Listing 4.1 for the security reference monitor. Separate confidentiality policies can be enforced on each child component. The UI component, identified by the iframe id `mainframe`, allows images to be loaded from a set of whitelisted image URLs. The secondary component, identified by iframe id `secondframe`, has no UI and needs no image loads. Both components are always allowed access to

[3]Except for navigation to pages like the help page.

```
var doneLoading_mainframe=false;
var doneLoading_secondframe = false;
function monitor(params){
 var url = params.url;
 if(params.id==="mainframe"){
   //Policy for UI child
   if(url===base_uri){
     return true;
   }else if(params.type=="IMAGE"){
     return check_img_whitelist(url);
   }else if(params.type=="SCRIPT"){
     if(!doneLoading_mainframe &&
       url===base_uri+"/shim1.js"){
       doneLoading_mainframe=true;
       return true;
     }else if(!doneLoading_mainframe){
       return check_script_whitelist(url);
     }
   }
 }else if(params.id=="secondframe"){
   //Policy for non-UI child
   if(url===base_uri){
    return true;
   }else if(params.type=="SCRIPT"){
     if(!doneLoading_secondframe &&
       url===base_uri+"/shim2.js"){
       doneLoading_secondframe=true;
       return true;
     }else if(!doneLoading_secondframe){
       return check_script_whitelist(url);
     }
   }
 }
 return false;
}
```

Listing 4.1: Monitor code for Clipperz, where base-uri, is the installation directory

the parent pointed to by `base_uri`. The monitor can also enforce a temporal policy on each component. The application initially loads several Javascript libraries from a set of whitelisted script URLs. In our implementation, the final script loaded is the shim code, and marks the end of the application startup. Post-startup, the monitor blocks all communication except to the parent (and images for `mainframe`).

BrowserID

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 the BrowserID centralized service. Users create a single username/password to log in to the trusted BrowserID service and can register any number of email addresses as identities. Other single sign-on mechanisms share similar designs, and our results are more generally applicable to other single sign-on systems.

The implementation has the following components, typically hosted on the `https://login.persona.org` origin:

- A dialog window that is opened by the Relying Party when the user chooses to login using BrowserID. This window prompts asking the user to sign in using pre-registered email ids. We call this the *dialog* page.
- Other pages that contain public information materials and account management options for the authenticated user. We call these pages the *management* component.

When a user uses Persona to sign in to a website, the Relying Party's code requests an assertion from the `navigator.id` object. The Persona script on the Relying Party's page handles this request by opening the dialog and sending it some of the request's parameters via `postMessage`. The dialog presents an interaction flow where the user authenticates herself to the Persona service (if not already authenticated; an authenticated session persists with a cookie) and picks an email address to identify as. The dialog uses `XMLHttpRequest` to send authentication credentials to the Persona server and to receive the assertion therefrom. Finally, the dialog uses `postMessage` to return the assertion to the Persona script in the Relying Party's document, which returns it to the Relying Party's code.

The production BrowserID front end includes 101.1KB and 105.8KB of JavaScript code in the management and dialog components respectively. The actual TCB is larger, since BrowserID uses the EJS templating system [84]. Similar to a number of modern JavaScript templating languages [1], EJS loads template files from the server and converts them to code at runtime using `eval`. Incidentally, all modern templating languages rely on `eval` (in particular, the function constructor), which limits the applicability of CSP and static analysis techniques.

Privilege Separation We moved all the application code to an unprivileged DCS. Minor changes were required for compatibility. In particular, we modified code that reads the window location, and added a base tag to ensure that links navigate the parent window. The EJS library uses synchronous `XMLHttpRequests` to download the templates. Since the same-origin policy restricts `XMLHttpRequests` to the same origin, the shim code proxies requests in the parent via the asynchronous `postMessage` channel. We modified

the EJS templating code to download the templates asynchronously. The authentication component uses `postMessage` to communicate with the Relying Party: shim code enforces parent mediation on this exchange. In total, 184 lines of code were modified.

Data-Confinement Invariants Executing in a data-confined sandbox, we are able to provide two key guarantees as part of our implementation:

- The login and credential managers (management component) do not communicate with any servers other than the BrowserID servers. This allows us to protect against inadvertent disclosure of the main BrowserID username/password via developer mistake or scriptless HTML injection attacks.
- In one instance of the BrowserID protocol, only 3 specific web principals interact. Our design guarantees sensitive tokens are never leaked to parties outside these three participants. In particular, the parent ensures that the child executes the whole protocol with the same principal and same Relying Party window. In the past, single sign-on mechanisms have had implementation bugs that allowed a MITM of an authentication flow [129, 137]; our design prevents such bugs.

For further hardening, we modified the dialog's login process to move the password entry to the trusted parent. The parent prompts the user for her password and sends it to the server. This way, a compromised dialog will never see the user's password. We also implemented a state machine in the security monitor policy based on the intended dialog behavior. In particular, this state machine ensures that the dialog component performs a series of requests consistent with transitions possible in the state machine. This prevents a compromised dialog from making arbitrary requests in the user's session.

OpenEMR

OpenEMR is the most popular open-source electronic medical record system [120]. 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 setting a session variable, namely the *patient id*. Once the patient id is set, all future requests, such as 'demographic data,' 'notes,' and so on, are returned for the particular patient. If the user wants to navigate to another patient, the user has to use the search interface to reset the patient id.

The OpenEMR design is vulnerable to a self-exfiltration attack [31]. Setting the patient id for a particular session just requires a GET request with a `set_pid` parameter. An attacker, Mallory, can do this via a simple (scriptless) HTML injection (e.g., an `img` or

`link` tag). Mallory can time this injection at the right time (e.g., right before a doctor or nurse adds information about Alice's personal data). As a result of this injected request, the nurse or doctor would end up inserting all of Alice's information to Mallory's file, which Mallory can later read. This is an example of a self-exfiltration attack that coarse-grained policies such as CSP cannot prevent, since the injected request still points to the OpenEMR domain. As we demonstrate below, the fine-grained monitor approach we adopt mitigates this attack.

Privilege Separation We focus on `demographics.php` which presents patient data. It loads with a `set_pid` parameter in the URL and the server sets the patient id accordingly. Scripts on the page then use `XMLHttpRequest` to download patient details, such as history and notes. We modified this page to serve its content as plain text, and a loader page requests the code and runs it in a DCS. The loader page proxies the `XMLHttpRequest` and cross-frame procedure calls through the parent using `postMessage`.

Data-Confinement Invariants By running the page in a DCS, we can provide a strong confidentiality guarantee for the sensitive medical data; namely, the page can make no network requests to origins other than the OpenEMR origin. This same restriction can ensure that all network requests are only over HTTPS (since protocol is part of the origin). Currently, no primitives are available in HTML5 to enable this invariant.[4]

First, the DCS verifiably ensures that sensitive medical data does not inadvertently leak to untrusted principals (modulo covert channels). The DCS can also prevent the page from making arbitrary calls to the large, feature-rich application. In our case, we programmed the security monitor 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.

SQL Buddy

SQL Buddy is an open-source web-based application to handle the administration of MySQL databases. Written in PHP, it allows browsing of possibly sensitive data stored in a MySQL DBMS and supports standard database operations, including SQL queries and the creation, modification, and deletion of databases, tables, fields, and rows. It also allows for management of MySQL users.

Privilege Separation We re-architected SQL Buddy to execute all code in a DCS. We re-used most of the privilege separation code from Chapter 3, only adding a monitor function in the parent. The key change was in the script that logged a user into MySQL. The

[4]HSTS can ensure that requests to OpenEMR server are always over SSL, but it is still possible for the page to include a cross-origin image over HTTP.

original implementation returned a new login page upon a failed login attempt, an action disallowed within a DCS. In our redesign, we return an error code over `XMLHttpRequest`. The client-side code utilizes this code to display the new login page. This modification required changes to only 11 lines of code. The SQL Buddy code does not use any of the client-side communication channels we blocked in a DCS: as a result, modifying it to run in a DCS is essentially the same as privilege separating it.

Data-Confinement Invariants By executing the SQL Buddy application code in a DCS, the parent can enforce strong confidentiality policies. The application runs in two logical stages; the flexibility of the DCS monitor allows us to enforce a policy for each stage.

- Initially, the monitor function restricts communication to only SQL Buddy resources. The monitor allows the application to load a number of whitelisted JavaScript libraries and stylesheets.
- After loading the code and stylesheets, the application no longer requires network access except for loading SQL Buddy resource images and making `XMLHttpRequests` to SQL Buddy PHP code, which are proxied at the parent via `postMessage`. Our monitor code now locks down communication to these two channels.

Our monitoring function restricts all explicit communication channels: if the SQL Buddy code gets compromised, it still cannot send data to arbitrary servers. Separating out a small, trusted parent allows us to enforce finer grained policies. For example, our implementation also limits writes to the database. Any writes to the database require the user to explicitly confirm the write with a simple confirmation prompt created by the parent. Compromised code can not modify the database in the background; the user needs to confirm that she wants to modify the database.

Monitor Code We use the code in Listing 4.2 for the security reference monitor.

Chrome Extensions

To demonstrate the prevalence of data-confinement needs, we also studied the top twenty most popular extensions for the Google Chrome platform and identified their data confinement invariants. Table 4.3 presents the results. Our analysis indicates that data confinement is a widely prevalent requirement; with 16 of the twenty extensions we studied maintaining an invariant implicitly. The trusted code base for the extensions varies 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. The remaining four extensions without an invariant, such as custom-styling extension Stylish [128], dealt with the UI appearance of websites, and did not access sensitive data and made no network

```
var doneInit=false;
function monitor(params){
 var url = params.url;
 if(!doneInit){
   switch(params.type){
     case "SCRIPT":
       return check_script_whitelist(url);
     case "STYLESHEET":
       return check_css_whitelist(url);
     case "IMAGE":
       doneInit=true;
       return check_img_whitelist(url);
   }
 }else{
   if(params.type==="IMAGE"){
     return check_img_whitelist(url);
   }
 }
 return false;
}
```

Listing 4.2: Monitor code for SQL Buddy, where base-uri, is the installation directory

communications. We conservatively label these as not having data-confinement invariants, although they do have the permission to access sensitive data and a compromised application could rely on this access to steal sensitive user data.

4.7 Summary of Results

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.

Table 4.3: Confidentiality Invariants in the Top 20 Google Chrome Extensions

Extension Name	Brief Description	Confidentiality Invariants (if any)	# Users	Code Size
Adblock	Blocks ads on websites	None	8,125,379	335.0KB
Adblock Plus	Blocks ads on websites	None	4,423,859	350.1KB
Google Mail Checker	Signals new emails and displays number of unread emails	No personal Gmail data, such as login credential and email content, should be leaked	3,174,574	7.5KB
Evernote Web Clipper	Uploads web content to user's Evernote account	Uploaded content should only be sent to Evernote. User data, such as login credentials, should remain confidential	2,031,257	1242.2KB
Facebook Photo Zoom	Magnifies Facebook photos when mouse hovers over	No personal Facebook data should leak, including pictures viewed	1,972,261	42.9KB
Turn Off the Light	Dims page except for videos	None	1,915,080	168.2KB
Google Translate	Translates entire pages	Page text only sent to Google	1,749,503	211.9KB
Google Chrome to Phone	Pushes links, maps, and phone numbers to Android device	Content should only be transmitted to Android device, and device information should remain confidential	1,504,709	104.8KB
Stylish	Applies custom styles to sites	None	1,360,817	41.6KB
Google Dictionary	Provides definitions/translations of selected text and popup with Google search	Selected text should only be sent to Google. No party should learn location where word was found	1,285,849	57.3KB
TweetDeck	Tracks user's Facebook/Twitter, providing real-time updates	Personal Faceboook/Twitter data should not be leaked, including login credentials and updates	1,252,169	621.8KB
Screen Capture	Capture and edit screenshots of web pages. Allows sharing to social media.	Images should remain local or be sent only to designated social media sites. No personal social media data, including login credentials, should be leaked.	1,109,465	197.9KB
Fastest Chrome	Auto-loads next pages, provides quick search of highlighted text, and displays popularity of links on social media	Personal social media data should not be leaked. Search queries and highlighted text should only be sent to designated sites.	1,043,820	328.0KB
Add to Amazon Wish List	Add products seen on websites to user's Amazon Wish List	Personal Amazon information, including account credentials, should not be leaked. Product data should only be sent to Amazon.	1,037,475	54.3KB
Awesome Screenshot	Capture, edit, and share screenshots of web pages.	Images should only be sent to Awesome Screenshot servers	1,007,922	492.0KB
Shopping Assistant	Shows similar products from Amazon/eBay when searching retailer websites	Searched product information should only be sent to designated shopping sites	957,562	409.5KB
Facebook Notifications	Signals new Facebook notifications	Personal Facebook data should not be leaked, such as login credentials and notification details	929,612	14.5KB
Speed Dial	Fills new tab page with user-chosen bookmark icons	User bookmarks and history should remain confidential.	816,907	219.3KB
Webpage Screenshot	Capture and edit screenshots of web pages. No image uploads.	Images should remain local	789,482	209.9KB
AddThis	Allows sharing of web content to social media	Shared content should only be sent to appropriate social media site.	783,685	740.7KB

Chapter 5

Related Work

A large body of work shares our goal of securing the client-side HTML5 platform. In this chapter, we discuss previous related work. We also discuss concurrent and follow-on research and industry standards published after our research.

5.1 Formal Verification of Security Protocols

There is a large body of work on formally verifying security properties of network protocols, including model checking using a variety of tools [100, 101, 119, 126], constraint-based methods [99], and formal and automated proof methods [19, 27, 40, 41], but we are not aware of any previous work formalizing web protocols. Barth et al. hint at formal analysis by showing the existence of a frame communication bug that is apparent as soon as the protocol is written down formally [18]. There has also been some work on formal verification of web service security [21, 64]. A number of the notions we formalize have been used informally in the past. For example, MashupOS [135] contemplates web attacker-like threats and the gadget attacker makes an explicit appearance in [95]. The designers of the OP browser [65] use formal methods to verify some security properties of their design (whether or not the security indicators behave as expected). Formal methods have also been used to verify code-level properties of the status bar in Internet Explorer [97]. The most closely related work [85] uses Alloy to verify the security properties of a particular cross-site scripting defense (which we analyze further). However, none of these works attempt to formulate a general model of web security applicable beyond a single mechanism.

Following our work, Bohanon and Pierce formalize the core security policies of the Firefox browser [23]. While our work focuses on web protocol formalization, Bohanon and Pierce focus on the browser: their work has a richer model of the browser but is less applicable for reasoning about web protocols. Bansal et al. share our goals of formal analysis of web protocols and present WebSPI, a variant of the SPI calculus to analyze web protocols [8]. Lerner et al. formalize a subset of the DOM and the DOM event mechanism [91]. Fett et al. present a far more comprehensive model of the web security

platform and analyze the BrowserID SSO service [52]. Their work finds a number of critical security flaws in the BrowserID SSO system, some of which our formalism cannot express. Unfortunately, in its current form, the model cannot be automatically analyzed.

We open-sourced our Alloy code and a number of researchers extended our model to analyze existing or proposed protocols. Chen et al. extend our model to verify the security of a new multiple cookie-store mechanism in Google Chrome [32]. Similarly, Ryck et al. extend our model to verify a anti-CSRF proposal [42]. Cao et al. extend our model to analyze a new configurable origins proposal for the web [29]. Telikicherla et al. extend our model to analyze their proposed cross-origin request protocol [132].

5.2 Privilege Separation for Web Applications

The concept of privilege separation was first formalized by Saltzer and Schroeder [121]. Several proposed and deployed systems have used privilege separation for increased security. Below, we discuss the most closely related works.

Privilege Separation in Commodity OS Platforms. Notable examples of user-level applications utilizing privilege separation include QMail [20], OpenSSH [114] and Google Chrome [10]. Brumley and Song investigated automatic privilege separation of programmer annotated C programs and implemented data separation as well [25]. More recently, architectures like Wedge [22] 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 [15], Escudo [82], MashupOS [135], Gazelle [136], OP [65], IPC Inspection [50], and CLAMP [109]. 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 [127]) to achieve effective privilege separation, without requiring any further changes to the platform.

Carlini et al. [30] 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 an inflated TCB in the privileged component as we show in Section 3.5. This is in contrast to our proposed design.

Mashup & Advertisement Isolation. The problem of isolating code in web applications, especially in mashups [9, 135] and malicious advertisements [92], 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 [75] 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 be analyzed to conform to specific policies [67]. 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 [38, 55, 94]. Our approach instead relies on a lighter weight mechanism based on built-in browser primitives like `iframe`s and temporary origins.

Weinberger, a Google Chrome engineer, extends our idea as the sub-origin proposal for Content Security Policy [138, 139], including an initial implementation for Google Chrome. A key addition in the sub-origin proposal is the ability to *name* unprivileged principals and serialize these named, unprivileged principals into origin strings. In our privilege separated design, code running unprivileged gets a fresh, unpredictable origin. This makes it harder to communicate with the unprivileged principal via `postMessage` and `XMLHttpRequest` (via CORS [87]). Instead, in the sub-origin proposal unprivileged principals get predictable, server-provided labels. For example, Google security engineers could label a F.A.Q. page in `https://mail.google.com` as 'faq.' In addition to dropping privileges, the F.A.Q. page would also get a new, predictable origin (serialized as `suborigin://faq@https://mail.google.com`), which can be used for communication over `postMessage` and `XMLHttpRequest`.

5.3 Data-confined HTML5 Applications

Data confinement has been investigated in native binary applications as well [88], but we focus our discussion on web applications. 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 4.3, but discuss a few other works here.

Zalewski [142] and Heidrich et al. [71] point at a number of attacks that violate data-confinement invariants in web applications even in the absence of code injection. Zalewski's attacks involve subverting the application HTML logic to cause requests (containing capability tokens) to the attacker's website. Zalewski demonstrates multiple techniques such as injecting `base` tags, rerouting `form` action targets, and accessing cross-origin pointers such as `window.name`. Executing the application in a DCS would mitigate these attacks since the DCS whitelist will not allow requests to an attacker website and would block attacker access to cross-origin pointers in the DCS. Zalewski also points out self-exfiltration attacks (see below) and attacks that rely on subverting the logic of client-side browser extensions; the DCS design do not protect against these attacks.

Heidrich et al. demonstrate content-exfiltration attacks via side-channels, a class of attacks that we do not protect against. The DCS design still provides some mitigation against these attacks, since the attacks rely on loading attacker controlled fonts that the DCS parent can block. Finally, both Zalewski and Heidrich present attacks that rely on tricking the user to drag-and-drop secret values across origins. While we do not protect against these attacks, the DCS design increases the difficulty of executing such UI-based attacks. First, the parent monitor can disallow loads of untrusted fonts and other sub-resources in the DCS. Loading attacker controlled fonts and images (via scriptless HTML injection) in the vulnerable application is typically a key component of UI-level attacks since these attacks require confusing the user. Second, the DCS design, with its low TCB parent, prevent the attacker from getting a pointer to the main application frame or even disable embedding the application by untrusted websites.

Chen et al. argue that protecting against web information flow based on destination servers is insufficient [31]. Whitelisted servers might have a database accessible to the attacker. For example, restricting dataflow to Facebook servers is insufficient since the attacker could have a Facebook account. Such self-exfiltration attacks are a result of the coarse-grained nature of previous confinement primitives. As we discussed in our OpenEMR case study, our monitor based design is fine-grained and provides stronger protection against self exfiltration attacks. Depending on the particular application, a self exfiltration attack might still be possible, but we do not investigate further protections.

Popa et al. present Mylar, an extension of the Meteor JavaScript framework for building applications that encrypt all their data sent to the server [112]. Developers need to write their applications in Meteor (affecting backwards compatibility) and tell Mylar what data needs encryption. Similar to our discussion on LastPass, the enforcement of this data-confinement invariant is ad-hoc and a data-confined sandbox could ease reasoning about Mylar's security properties.

Chapter 6

Conclusion

The HTML5 application platform is by far the most popular and widely available application platform today. Modern HTML5 applications can run privileged with access to sensitive data in the cloud or privileged sensors (or any other data source) on novel devices. It is critical that these complex, privileged applications are amenable to audit and analysis. In this thesis, we presented our work on improving the current state of the art by formalizing our understanding and analysis of the web attacker threat model and by developing novel architectures for web application to reduce their trusted computing base and limit data flow.

Impact In collaboration with Google security engineers, we are currently working on standardizing the basic ideas of privilege separation into a robust "sub-origin" primitive for HTML5 applications as part of the Content Security Policy header. We have already had preliminary discussions towards standardization at the W3C Web Application Security Working Group and an initial implementation in Google Chrome (by a Google Chrome Engineer) is already available.

Bibliography

[1] Issue-107538: Comment 35. http://crbug.com/107538.

[2] *ACE - Ajax.org Cloud9 Editor*. http://ace.ajax.org/.

[3] P. Agten, S. V. Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. "JSand: Complete Client-side Sandboxing of Third-Party Javascript without Browser Modifications". In: *ACSAC* (2012).

[4] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John C. Mitchell, and Dawn Song. *Web Security Model Implementation*. 2010. URL: http://code.google.com/p/websecmodel.

[5] Apple Inc. *Remote Scripting with IFRAME*. 2010. URL: http://developer.apple.com/internet/webcontent/iframe.html.

[6] 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. Chicago, Illinois, USA: ACM, 2011, pp. 63–68. ISBN: 978-1-4503-1000-0. DOI: http://doi.acm.org/10.1145/2046614.2046626. URL: http://doi.acm.org/10.1145/2046614.2046626.

[7] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. *VEX: Vetting Browser Extensions For Security Vulnerabilities*. 2010.

[8] Chetan Bansal, Karthikeyan Bhargavan, and Sergio Maffeis. "Discovering concrete attacks on website authorization by formal analysis". In: *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*. IEEE. 2012, pp. 247–262.

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

[10] A. Barth, C. Jackson, C. Reis, and TGC Team. *The security architecture of the Chromium browser*. 2008.

[11] Adam Barth. *<form method="DELETE"> and* 307 *redirects*. 2009. URL: http://www.mail-archive.com/whatwg@lists.whatwg.org/msg19379.html.

[12] Adam Barth. *The Web Origin Concept*. http://tools.ietf.org/html/rfc6454.

[13] Adam Barth. "Timing Attacks on CSS Shaders". `http://www.schemehostport.com/2011/12/timing-attacks-on-css-shaders.html`. 2011.

[14] Adam Barth, Juan Caballero, and Dawn Song. "Secure Content Sniffing for Web Browsers *or* How to Stop Papers from Reviewing Themselves". In: *Proceedings of the 30th IEEE Symposium on Security and Privacy*. Oakland, CA, May 2009.

[15] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. *Protecting Browsers from Extension Vulnerabilities*. 2009.

[16] Adam Barth, Collin Jackson, and Ian Hickson. *The HTTP Origin Header*. 2009. URL: `http://tools.ietf.org/html/draft-abarth-origin`.

[17] Adam Barth, Collin Jackson, and John C. Mitchell. "Robust Defenses for Cross-Site Request Forgery". In: *CCS*. 2008.

[18] Adam Barth, Collin Jackson, and John C. Mitchell. "Securing Frame Communication in Browsers". In: *Proceedings of the 17th USENIX Security Symposium (USENIX Security 2008)*. 2008.

[19] Giampaolo Bella and Lawrence C. Paulson. "Kerberos Version IV: Inductive Analysis of the Secrecy Goals". In: *Proceedings of the 5th European Symposium on Research in Computer Security*. Ed. by J.-J. Quisquater. Springer-Verlag LNCS 1485, 1998, pp. 361–375.

[20] 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. Fairfax, Virginia, USA: ACM, 2007, pp. 1–10. ISBN: 978-1-59593-890-9. DOI: `http://doi.acm.org/10.1145/1314466.1314467`. URL: `http://doi.acm.org/10.1145/1314466.1314467`.

[21] K. Bhargavan, C. Fournet, and A.D. Gordon. "Verified reference implementations of WS-Security protocols". In: *Lecture Notes in Computer Science* 4184 (2006), p. 88.

[22] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. "Wedge: Splitting Applications into Reduced-Privilege Compartments". In: *NSDI* (2008).

[23] Aaron Bohannon and Benjamin C Pierce. "Featherweight Firefox: Formalizing the core of a web browser". In: *Proceedings of the 2010 USENIX conference on Web application development*. USENIX Association. 2010, pp. 11–11.

[24] *BrowserID Users*. `https://wiki.mozilla.org/Identity/BrowserID/InTheWild`.

[25] David Brumley and Dawn Song. "Privtrans: Automatically Partitioning Programs for Privilege Separation". In: *USENIX Security* (2004).

[26] Mozilla Bugzilla. *Implement CSP sandbox directive*. `https://bugzilla.mozilla.org/show_bug.cgi?id=671389`.

[27] M. Burrows, M. Abadi, and R. Needham. "A logic of authentication". In: *ACM Transactions on Computer Systems* 8.1 (1990), pp. 18–36.

[28] Serdar Cabuk, Carla E. Brodley, and Clay Shields. "IP covert timing channels: design and detection". In: *CCS* (2004).

[29] Yinzhi Cao, Vaibhav Rastogi, Zhichun Li, Yan Chen, and Alexander Moshchuk. "Redefining web browser principals with a configurable origin policy". In: *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*. IEEE. 2013, pp. 1–12.

[30] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. "An Evaluation of the Google Chrome Extension Security Architecture". In: (2012).

[31] Eric Y Chen, Sergey Gorbaty, Astha Singhal, and Collin Jackson. "Self-exfiltration: The dangers of browser-enforced information flow control". In: *Web 2.0 Security and Privacy Workshop*.

[32] 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. Chicago, Illinois, USA: ACM, 2011, pp. 227–238. ISBN: 978-1-4503-0948-6. DOI: 10.1145/2046707.2046734. URL: http://doi.acm.org/10.1145/2046707.2046734.

[33] E.Y. Chen, S. Gorbaty, A. Singhal, and C. Jackson. "Self-Exfiltration: The Dangers of Browser-Enforced Information Flow Control". In: *W2SP* (2012).

[34] Pern Hui Chia, Yusuke Yamamoto, and N. Asokan. "Is this app safe?: a large scale study on application permissions and risk signals". In: *WWW* (2012).

[35] Erika Chin and David Wagner. "Bifocals: Analyzing WebView Vulnerabilities in Android Applications". In: (2013).

[36] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. "Staged information flow for JavaScript". In: *PLDI* (2009).

[37] Clipperz. http://www.clipperz.com/.

[38] Douglas Crockford. *AdSafe*. http://www.adsafe.org/.

[39] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006. ISBN: 0262101149.

[40] Anupam Datta, Ante Derek, John C. Mitchell, and Arnab Roy. "Protocol Composition Logic (PCL)". In: *Electronic Notes in Theoretical Computer Science* 172 (2007), pp. 311–358.

[41] Anupam Datta, Ante Derek, John C. Mitchell, Vitaly Shmatikov, and Mathieu Turuani. "Probabilistic polynomial-time semantics for a protocol security logic." In: *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP '05)*. Lecture Notes in Computer Science. Springer-Verlag, 2005, pp. 16–29.

[42] Philippe De Ryck, Lieven Desmet, Wouter Joosen, and Frank Piessens. "Automatic and precise client-side protection against CSRF attacks". In: *Computer Security–ESORICS 2011*. Springer, 2011, pp. 100–116.

[43] Alexis Deveria. *Can I use sandbox attribute for iframes*. `http://caniuse.com/\#feat=iframe-sandbox`. 2013.

[44] R. Dhamija, JD Tygar, and M. Hearst. "Why phishing works". In: *Proceedings of the SIGCHI conference on Human Factors in computing systems*. ACM. 2006, p. 590.

[45] M. Dhawan and V. Ganapathy. "Analyzing information flow in JavaScript-based browser extensions". In: *Annual Computer Security Applications Conference*. IEEE. 2009, pp. 382–391.

[46] diigo.com. *Awesome Screenshot*. `http://www.awesomescreenshot.com/`.

[47] *Dropbox Developer Reference*. `http://www.dropbox.com/developers/reference`.

[48] 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. Portland, OR: USENIX Association, 2011, pp. 7–7. URL: `http://dl.acm.org/citation.cfm?id=2002168.2002175`.

[49] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. "Android permissions: User attention, comprehension, and behavior". In: *Proceedings of the Eighth Symposium on Usable Privacy and Security*. ACM. 2012, p. 3.

[50] 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. San Francisco, CA: USENIX Association, 2011, pp. 22–22. URL: `http://dl.acm.org/citation.cfm?id=2028067.2028089`.

[51] E.W. Felten, D. Balfanz, D. Dean, and D.S. Wallach. "Web spoofing: An internet con game". In: *Software World* 28.2 (1997), pp. 6–8.

[52] Daniel Fett, Ralf Küsters, and Guido Schmitz. "An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System". In: *CoRR* abs/1403.1866 (2014).

[53] 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.

[54] Garteh Hayes. *Hacking caja part 2*. `www.thespanner.co.uk/2012/09/18/hacking-caja-part-2/`.

[55] Google. *Caja*. `http://developers.google.com/caja/`.

[56] Google. *Chrome Extensions*. `https://chrome.google.com/webstore/category/extensions`.

[57] Google. *Chrome Web Store*. `https://chrome.google.com/webstore`.

[58] Google. *Seccomp Sandbox*. `http://code.google.com/p/seccompsandbox/`.

[59] Google Caja. *_gel('foo').bar fails due to firefox 2.x hasOwnProperty bug*. `http://code.google.com/p/google-caja/issues/detail?id=51`.

[60] Google Caja. *Negative indices on many Firefox host objects expose static properties*. `http://code.google.com/p/google-caja/issues/detail?id=1093`.

[61] Google Caja. *Static RegExp properties have bizarre mutability properties on Firefox*. `http://code.google.com/p/google-caja/issues/detail?id=520`.

[62] Google Inc. *Google Chrome Extensions: chrome.* APIs*. `http://code.google.com/chrome/extensions/api_index.html`.

[63] Google Inc. *Google Chrome Webstore*. `https://chrome.google.com/webstore/`.

[64] A.D. Gordon and R. Pucella. "Validating a web service security abstraction by typing". In: *Formal Aspects of Computing* 17.3 (2005), pp. 277–318.

[65] Chris Grier, Shuo Tang, and Samuel T. King. "Designing and Implementing the OP and OP2 Web Browsers". In: *ACM Trans. Web* 5 (2 May 2011), 11:1–11:35. ISSN: 1559-1131. DOI: `http://doi.acm.org/10.1145/1961659.1961665`. URL: `http://doi.acm.org/10.1145/1961659.1961665`.

[66] Salvatore Guarnieri and Benjamin Livshits. "Gatekeeper: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code". In: *Usenix Security*. 2009.

[67] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. "Verified security for browser extensions". In: *IEEE S&P* (2011).

[68] Eran Hammer-Lahav. *Acknowledgement Of The Oauth Security Issue*. 2009. URL: `http://blog.oauth.net/2009/04/22/acknowledgement-of-the-oauth-security-issue/`.

[69] S. Hanna, E.C.R. Shin, D. Akhawe, A. Boehm, P. Saxena, and D. Song. "The emperor's new APIs: On the (in) secure usage of new client-side primitives". In: *W2SP* (2010).

[70] Mario Heiderich, Tilman Frosch, and Thorsten Holz. "IceShield: detection and mitigation of malicious websites with a frozen DOM". In: *RAID* (2011).

[71] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. "Scriptless attacks: stealing the pie without touching the sill". In: *CCS* (2012).

[72] Adobe Inc. *Cross-domain policy file specification*. 2008. URL: `http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html`.

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

[74] Microsoft Inc. *XDomainRequest Object*. 2009. URL: `http://msdn.microsoft.com/en-us/library/cc288060%28VS.85%29.aspx`.

[75] L. Ingram and M. Walfish. "Treehouse: Javascript sandboxes to help web developers help themselves". In: *USENIX ATC* (2012).

[76] Collin Jackson and Adam Barth. "Beware of Finer-Grained Origins". In: *In Web 2.0 Security and Privacy (W2SP 2008)*. 2008. URL: `http://seclab.stanford.edu/websec/origins/fgo.pdf`.

[77] Collin Jackson and Adam Barth. "Forcehttps: protecting high-security web sites from network attacks". In: *WWW* (2008).

[78] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. "Protecting browsers from DNS rebinding attacks". In: *ACM Trans. Web* 3.1 (2009), pp. 1–26. ISSN: 1559-1131. DOI: `http://doi.acm.org/10.1145/1462148.1462150`.

[79] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. "Protecting browser state from web privacy attacks". In: *WWW* (2006).

[80] D. Jackson. "Alloy: a lightweight object modelling notation". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11.2 (2002), pp. 256–290.

[81] JASIG. *CAS Deployment*. 2010. URL: `http://www.jasig.org/cas/deployments`.

[82] 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*. IEEE. 2010, pp. 231–240.

[83] Peter Josling. *dropbox-js: A JavaScript library for the Dropbox API*. `http://code.google.com/p/dropbox-js/`.

[84] Jupiter-IT. *EJS JavaScript Templates*. `http://embeddedjs.com/`.

[85] Florian Kerschbaum. "Simple Cross-Site Attack Prevention". In: *Proceedings of the Third international workshop on Security and Privacy in Communication networks*. Nice, France, 2007.

[86] Anne van Kesteren. *Cross-Origin Resource Sharing (Editors Draft)*. 2009. URL: `http://dev.w3.org/2006/waf/access-control`.

[87] Anne van Kesteren (Ed.) *Cross-Origin Resource Sharing*. `http://www.w3.org/TR/cors/`.

[88] Tejas Khatiwala, Raj Swaminathan, and V.N. Venkatakrishnan. "Data Sandboxing: A Technique for Enforcing Confidentiality Policies". In: *ACSAC* (2006).

[89] Tyler Klose. *Confused Deputy Attack on CORS*. 2009. URL: `http://lists.w3.org/Archives/Public/public-webapps/2009AprJun/1324.html`.

[90] J.T. Kohl and B.C. Neuman. *The Kerberos network authentication service (version 5)*. IETF RFC 1510. Sept. 1993.

[91] Benjamin S Lerner, Matthew J Carroll, Dan P Kimmel, Hannah Quay-De La Vallee, and Shriram Krishnamurthi. "Modeling and reasoning about DOM events". In: *Proceedings of the 3rd USENIX conference on Web Application Development*. USENIX Association. 2012, pp. 1–1.

[92] 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. Washington, DC: USENIX Association, 2010, pp. 24–24. ISBN: 888-7-6666-5555-4. URL: http://dl.acm.org/citation.cfm?id=1929820.1929852.

[93] *lxc Linux Containers*. http://lxc.sourceforge.net/.

[94] Sergio Maffeis, John C. Mitchell, and Ankur Taly. "Object Capabilities and Isolation of Untrusted Web Applications". In: *IEEE S&P* (2010).

[95] Jonas Magazinius, Aslan Askarov, and Andrei Sabelfeld. "A Lattice-based Approach to Mashup Security". In: *In Proc. of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2010)*. Beijing, China: ACM, 2010.

[96] Drew Mazurek. *CAS Protocol*. 2005. URL: http://www.jasig.org/cas/protocol.

[97] Jose Meseguer, Ralf Sasse, Helen J. Wang, and Yi-Min Wang. "A Systematic Approach to Uncover Security Flaws in GUI Logic". In: *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2007, pp. 71–85. ISBN: 0-7695-2848-1. DOI: http://dx.doi.org/10.1109/SP.2007.6.

[98] Microsoft. *Metro style app development*. http://msdn.microsoft.com/en-us/windows/apps/.

[99] Jonathan Millen and Vitaly Shmatikov. "Constraint solving for bounded-process cryptographic protocol analysis". In: *CCS '01: Proceedings of the 8th ACM conference on Computer and Communications Security*. Philadelphia, PA, USA: ACM, 2001, pp. 166–175. ISBN: 1-58113-385-5. DOI: http://doi.acm.org/10.1145/501983.502007.

[100] J. C. Mitchell, V. Shmatikov, and U. Stern. "Finite-State Analysis of SSL 3.0". In: *Proceedings of the Seventh USENIX Security Symposium*. 1998, pp. 201–216.

[101] John C. Mitchell, Mark Mitchell, and Ulrich Stern. "Automated Analysis of Cryptographic Protocols Using Murphi". In: *Proceedings of the IEEE Symposium on Security and Privacy*. 1997, pp. 141–151.

[102] *MochiKit*. http://mochi.github.com/mochikit/.

[103] Lee Momtahan. *A Simple Small Model Theorem for Alloy*. Tech. rep. RR-04-11. Oxford University Computing Laboratory, June 2004.

[104] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. "Chaff: engineering an efficient SAT solver". In: *Proceedings of the 38th annual Design Automation Conference*. Las Vegas, NV, United States: ACM, 2001, pp. 530–535. ISBN: 1-58113-297-2. DOI: http://doi.acm.org/10.1145/378239.379017.

[105] Mozilla. *Boot2Gecko*. https://wiki.mozilla.org/B2G.

[106] E.V. Nava and D. Lindsay. "Abusing Internet Explorer 8's XSS Filters". In: *BlackHat Europe*. 2010. URL: http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf.

[107] *OAuth*. http://oauth.net/.

[108] *OEMR*. http://www.oemr.org/wiki/OEMR_Organization_Wiki_Home_Page.

[109] 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*. IEEE Computer Society, 2009, pp. 154–169. ISBN: 978-0-7695-3633-0. DOI: 10.1109/SP.2009.21. URL: http://dl.acm.org/citation.cfm?id=1607723.1608131.

[110] phpMyAdmin. http://www.phpmyadmin.net/.

[111] Joe G. Politz, Spiridon A. Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. "ADsafety: type-based verification of JavaScriptSandboxing". In: *USENIX Security* (2011).

[112] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan. "Securing web applications by blindfolding the server". In: *NDSI* (2014).

[113] Niels Provos. "Improving host security with system call policies". In: *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*. Washington, DC: USENIX Association, 2003, pp. 18–18. URL: http://dl.acm.org/citation.cfm?id=1251353.1251371.

[114] Niels Provos, Markus Friedl, and Peter Honeyman. "Preventing privilege escalation". In: *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*. Washington, DC: USENIX Association, 2003, pp. 16–16. URL: http://dl.acm.org/citation.cfm?id=1251353.1251369.

[115] *pynarcissus : The Narcissus Javascript interpreter ported to Python*. http://code.google.com/p/pynarcissus/.

[116] Code Release. https://github.com/devd/html5privsep.

[117] Code Release. https://github.com/devd/data-confined-html5-applications.

[118] G. Richards, S. Lebresne, B. Burg, and J. Vitek. "An analysis of the dynamic behavior of JavaScript programs". In: *ACM SIGPLAN Notices* (2010).

[119] A. W. Roscoe. "Modelling and verifying Key-exchange protocols using CSP and FDR". In: *8th IEEE Computer Security Foundations Workshop*. IEEE Computer Soc Press, 1995, pp. 98–107.

[120] S. Riley. *5 OpenSource EMRs worth reviewing*. `http://bit.ly/hUa6l1`. 2011.

[121] J.H. Saltzer and M.D. Schroeder. "The protection of information in computer systems". In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308.

[122] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. *A Symbolic Execution Framework for JavaScript*. Tech. rep. UCB/EECS-2010-26. EECS Department, University of California, Berkeley, 2010.

[123] Roland Schemers and Russ Allbery. *WebAuth V3 Technical Specification*. 2009. URL: `http://webauth.stanford.edu/protocol.html`.

[124] K. Singh, A. Moshchuk, H.J. Wang, and W. Lee. "On the incoherencies in web browser access control policies". In: *IEEE S&P* (2010).

[125] Software Design Group, MIT. *Alloy Analyzer 4*. 2010. URL: `http://alloy.mit.edu/alloy4/`.

[126] Dawn Xiaodong Song. "Athena: a New Efficient Automatic Checker for Security Protocol Analysis". In: *Proceedings of the Twelfth IEEE Computer Security Foundations Workshop*. June 1999, pp. 192–202.

[127] Brandon Sterne and Adam Barth. *Content Security Policy: W3C Editor's Draft*. `http://bit.ly/foq8vf`. 2012.

[128] *Stylish*. `http://goo.gl/k1LVVT`.

[129] S.T. Sun, K. Hawkey, and K. Beznosov. "Systematically breaking and fixing OpenID security: Formal analysis, semi-automated empirical evaluation, and practical countermeasures". In: *Computers & Security* (2012).

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

[131] "GWT Team". *Security for GWT Applications*. 2008. URL: `http://groups.google.com/group/Google-Web-Toolkit/web/security-for-gwt-applications`.

[132] Krishna Chaitanya Telikicherla and Venkatesh Choppella. "Alloy model for Cross Origin Request Policy (CORP)". In: IIIT/TR/2013/-1 (2013).

[133] The Dojo Foundation. *The Dojo Toolkit*. `http://dojotoolkit.org/`.

[134] Tizen. `https://www.tizen.org/`.

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

[136] 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*. USENIX Association. 2009, pp. 417–432.

[137] R. Wang, S. Chen, and X.F. Wang. "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).

[138] Joel Weinberger. *Suborigins for Privilege Separation in Web Applications*. `http://blog.joelweinberger.us/2013/08/suborigins-for-privilege-separation-in.html`. Aug. 2013.

[139] Joel Weinberger and The Chromium Authors. *Per-page Suborigins*. `http://www.chromium.org/developers/design-documents/per-page-suborigins`.

[140] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting. "An exploration of L2 cache covert channels in virtualized environments". In: *CCSW* (2011).

[141] *YUI Library*. `http://yuilibrary.com/`.

[142] Michal Zalewski. "Postcards from the post-XSS world". `http://lcamtuf.coredump.cx/postxss/`.

[143] Michal Zawelski. *Browser Security Handbook*. 2009. URL: `http://code.google.com/p/browsersec/wiki/Main`.

[144] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. "Cross-VM side channels and their use to extract private keys". In: *CCS* (2012).