Improving the Security and Robustness of Modern Web Browsers

Charles Reis
creis@cs.washington.edu
Department of Computer Science and Engineering
University of Washington

Abstract

Despite their popularity, modern web browsers do not offer a secure or robust environment for interacting with untrusted content. Today's web users face a variety of threats, including exploits of browser vulnerabilities, interference between web sites, script injection attacks, and abuse of authentication credentials. To address these threats, I leverage an analogy between operating systems and web browsers, as both must run independent programs from multiple sources. My hypothesis is that mechanisms from OS research can improve the security and robustness of modern web browsers. In this report, I propose abstractions and mechanisms to isolate independent web content within the browser, and I propose two separate interposition techniques to support flexible security policies. Combined, these contributions can improve the safety of web browsers, while preserving backwards compatibility and imposing low overhead.

1 Introduction

Web browsers have become required applications for millions of people, who use them daily to interact with a mix of privacy-critical and untrustworthy content. Web browsers have also experienced a major shift in functionality over the past decade, from simple document renderers to complex runtime environments for code from the web. Because of this combination of trusted and untrusted active input, the security and robustness of web browsers are issues of great importance. Unfortunately, these issues have been addressed incompletely as browsers have evolved. As a result, modern browsers have inadequate security policies and implicit boundaries for isolating content. This leaves them vulnerable to a variety of threats. In this report, I show how ideas from operating systems research can be applied to the runtime environments of browsers, and I show how such ideas can concretely improve the security and robustness of modern web browsers.

Security and robustness problems in current browsers are manifested in a wide variety of threats. I consider four categories of threats in particular, though I discuss additional threats in Section 2. First, implementation flaws in web browsers are frequently discovered, and these flaws can often be exploited by malicious web sites to run arbitrary code on client machines. Browser developers must quickly defend users from such exploits, because malware authors frequently target browser vulnerabilities [41, 55]. Second, code from many different hosts can run concurrently in the browser, contending for resources like CPU and memory. Current browsers are not robust to this resource contention, resulting in interference and poor failure isolation between web sites. Third, script injection or "cross-site scripting" (XSS) attacks have become widespread [7], allowing adversaries to steal private information or disrupt trusted web content. The complexity of web content and browser implementations have made these attacks difficult to prevent. Fourth, "cross-site request forgery" (CSRF) attacks allow an untrusted site to abuse a user's credentials on another site. Adversaries can then perform damaging actions in the user's name.

Many of these threats are exacerbated by the complexity of modern browsers and their policies. For example, most browsers tolerate malformed input, making it difficult to filter attacks that might succeed despite appearing malformed. Browsers also have inconsistent isolation policies for many resources, including cookies, cached objects, scripts from third parties, and communication requests. This results in unexpected vulnerabilities, such as opportunities to covertly track users [3, 28]. These inconsistencies also make it difficult to isolate the effects of visiting a given web site. Unfortunately, current trends are leading to further inconsistencies. Browser plugins for new "rich" content types are becoming more popular (e.g., Flash and Silverlight [39]), and each has its own security policies. Overall, these inconsistencies and complexity present large challenges for reasoning about browser security.

To address the current threats, I leverage an analogy between modern web browsers and operating systems. Both are tasked with running independent code from different sources, while offering a reliable and secure platform. Indeed, many of the above threats have analogs at the OS level. OS vulnerabilities must be patched quickly to protect the system. Resource contention and interference between user programs posed a problem in early desktop operating systems, such as MS-DOS and Mac OS. Trojan horses and code injection attacks are similar to XSS attacks, and CSRF attacks are instances of the "confused deputy" problem observed for OS protection mechanisms [22]. OS researchers have explored many ways to defend systems from these threats, yet relatively few of their ideas have taken root in web browsers.

My hypothesis is that mechanisms from operating systems research can be applied to modern web browsers to demonstrably improve their security and robustness. To show this, I will incorporate analogs of OS isolation and interposition mechanisms into web browsers. The likely contributions of this work include the following:

- A set of abstractions for decomposing the browser and isolating content from different web sites.
- Isolation mechanisms that prevent interference between these abstractions, both in terms of resource contention and authentication credentials.
- A code rewriting framework that allows administrators to enforce flexible policies on untrusted web content.
- An interposition layer in the browser that supports extensible security policies, independent of the type of content or the use of browser extensions.
- A set of security policies that leverage interposition to defend against browser security threats, such as exploitable vulnerabilities and XSS attacks.

I will show how these contributions defend against current threats to web browsers, preserve backwards compatibility, and impose low performance overhead. The contributions will provide explicit boundaries that allow users to have multiple independent browsing sessions in a single browser. They will also provide extension points within the browser, as a platform for future web security research.

The remainder of this paper is organized as follows. Section 2 discusses related work, including foundational operating systems research, relevant browser security work, and related web concerns. I present preliminary work supporting my hypothesis in Section 3. I describe my proposals in Section 4, outlining additional work to improve browser security and robustness. I discuss how I will evaluate the proposed work in Section 5, and I present conclusions and directions for future work in Section 6.

2 Related Work

This section explores foundational OS research that is relevant for modern web browsers, as they have evolved into runtime environments for untrusted code. Web security has recently become a hot topic in its own right, and this section also places my proposal in context with directly related browser research and other relevant concerns for web users and developers.

2.1 Foundational Work

I first discuss foundational work in security and robustness for operating systems and language runtime environments. I focus on the topics of isolation and interposition, showing how OS research on these topics is relevant to my proposed hypothesis.

Isolation and Confinement On any multiprocessing system, the isolation of independent programs is important to prevent unwanted interference. For traditional operating systems, the *process* abstraction has long offered a mechanism for isolating independent programs. Modern OS processes feature separate address spaces and concurrency, along with tools for resource management and accounting. These attractive properties have led to research that incorporates process-like abstractions into other runtime environments as well, such as Java [5, 30]. My work finds that browsers are hampered by a lack of process-like isolation for web content, and I propose using OS processes to prevent unwanted interactions between such content.

Process-based isolation can come at a cost, however, due to the high overhead for inter-process communication. As a result, many researchers have investigated lightweight fault domains to allow intra-process isolation [51, 13, 12, 49]. For example, SFI [51] provides memory safety by confining untrusted code in sandboxes, while Asbestos [12] offers isolated event processes within an OS process. Such approaches often trade slight overhead in the common case for substantially faster cross-domain communication. It is not yet clear whether such a tradeoff is worthwhile in web browsers. If so, lightweight fault domains may offer an alternative to process-based isolation in the browser.

Lampson discusses more general concerns for confining untrusted code, including explicit and covert channels for communication [36]. In practice, it may be difficult to enumerate all such channels, and while many are well-known for web browsers (e.g., network communication or cache timing [16]), others are still being discovered (e.g., visited link history [28]). In confining untrusted web content, I aim to provide effective isolation of explicit communication channels while supporting existing proposals for blocking known covert channels [16, 28].

Closer to the web, Borenstein [6] and Ousterhout et al. [42] explore the design of Safe-Tcl for supporting untrusted code in email messages. Safe-Tcl confines scripts using two interpreters that are analogous to kernel and user levels in the OS. Notably, it offers an easy way to extend the abilities of particular scripts on the fly. While scripted email has not become popular, JavaScript has filled a similar role for the web. However, JavaScript is confined by a fixed and complex security policy, which has evolved over time and applies to various browser resources in different ways. Like Safe-Tcl, I aim to provide a cleaner interface for confining scripts and other active web content in an extensible way.

Interposition and Filtering Interposition is a fundamental tool in operating system design: using a level of indirection to allow or deny access to particular resources, based on a policy. OS protection mechanisms are perhaps the most basic example, interposing on resources such as the filesystem or network. Saltzer and Schroeder outline basic principles of protection [45], including design principles and mechanisms such as ACLs and capabilities. Unsurprisingly, many of their concepts translate directly to browsers and web content. Interposing between web content and the browser can support flexible security policies to govern the content's behavior.

The need for extensible security policies in systems is well recognized. At the OS level, system call interposition can enforce policies to restrict the behavior of untrusted programs. For example, Janus intercepts particular system calls with policy modules [20]. These modules abstract away the specific filtering that must be done for each system call, allowing policy authors to focus on higher level abstractions like filesystem paths. Garfinkel identifies challenges in interposing on OS behavior with a filtering approach [18], and he proposes a delegation architecture to avoid these pitfalls [19]. Nooks [49] takes a different approach, using wrapper code to interpose on calls between the kernel and its extensions. These various mechanisms are attractive for web browsers, to support configurable security policies on web content. I explore browser analogs to system call interposition in Section 4.2.

At a higher level than the OS, Wallach et al. explore extensible security architectures for Java [53]. They discuss interposition mechanisms for the JVM that enforce flexible policies on untrusted code. Their work supports such policies for Java applets in web browsers, but it does not apply to resources within the browser itself, such as the Document Object Model (DOM). I propose an extensible security architecture within the browser, independent of the type of active content. In this way, policies can apply uniformly to all types of content, including JavaScript, Java applets, Flash, and newer formats such as Silverlight [39].

Interposing at the level of program code can also be attractive, as it requires no changes to the underlying platform [14, 15]. For example, Erlingsson and Schneider use code rewriting to support inline reference monitors [14]. These monitors enforce extensible security policies within the code of an untrusted application. I leverage similar techniques for web content, since this content can often be rewritten before reaching the browser.

Shield [54] uses a filtering technique related to interposition, intercepting network packets bound for an application. Shield filters this traffic for exploits of known vulnerabilities. Shield's network-based approach shares deployment advantages with code rewriting, as neither requires changes to the underlying platform. Web content can also be filtered in such a way, but because web content can include executable code, the Shield approach must be combined with code rewriting to be effective on the web.

2.2 Browser Security

More recent work offers directly relevant proposals to address threats for web browsers. These include efforts to better isolate web content from different sources, handle malware on the web, mitigate script injection attacks, and limit abuses of client authentication. My proposed work will build on these efforts, offering more comprehensive isolation and interposition mechanisms for the web browser as an application platform.

Isolation of Web Content Tahoma isolates web applications on the client, both from each other and from the client's operating system [8]. It uses separate virtual machines (VMs) for each web application, running an independent browser instance for each. These instances are managed using a "browser operating system" (BOS): a common software layer outside the browser. The BOS can confine web applications based on their self-provided manifest files.

Like Tahoma, my work aims to isolate independent web content to prevent interference. However, our assumptions differ: unlike Tahoma, I treat the browser as trusted software. I argue this is necessary: as the runtime environment for web content, the browser must be trusted not to falsify content or abuse the user's credentials. Instead of fully sandboxing the entire browser, I aim to isolate web content *within* the browser. By trusting the browser, I can leverage lighter-weight isolation mechanisms than Tahoma, such as processes. To protect the underlying OS from exploits of browser flaws, I propose vulnerability filtering techniques similar to Shield [54].

Compared with Tahoma, I also place greater emphasis on backwards compatibility. Tahoma requires manifest files that specify confinement policies for each web site. These manifests would need to be deployed or inferred, both of which would be difficult for many reasons. Web sites may have little incentive to confine their own content with manifests, and inferring confinement policies is difficult from outside the browser. For example, the BOS would be unable to distinguish between HTTP requests for web pages from those for XML data, but these request types should be subject to different security policies. Tahoma also places a greater burden on the user to manage and understand the manifest for each web application before approving it. In contrast, I aim to isolate web content without requiring manifests or other changes to web sites, and without placing new management burdens on the user. These goals can be accomplished with new isolation abstractions within the browser itself.

Other work addresses isolation within the browser. Anupam and Mayer discuss vulnerabilities that result from poorly defined JavaScript security policies, and they offer a more formal model of how browsers, scripts, and interpreters should interact [3]. Their model describes when objects should be isolated or shared between contexts. I concur that browsers demand thorough isolation and security policies, but I propose them at a deeper level than the scripting engine. Instead, I offer process-based isolation between carefully defined abstractions within the browser, and I propose an interposition layer to enforce policies at the level of the DOM. As a result, policies can be applied uniformly to JavaScript code and other forms of web content.

As browsers have evolved into environments for executing code, researchers have proposed other refinements to the isolation of web content. For example, the discoveries of cache timing attacks and other storage channels [16, 28] have led researchers to extend the "same-origin" policy of current browsers [44] to a broader set of resources. I support these proposals, and I aim to incorporate such isolation between content without losing backwards compatibility. I also hope to offer a flexible architecture to better support such research in the future.

Drive-by Downloads Researchers have observed a large number of pages on the web that perform "drive-by downloads," installing malware by exploiting browser vulnerabilities [41, 55]. A wide range of defenses have been explored for this problem. Strider HoneyMonkey uses data about offending sites to pursue legal action and blacklists [55], while SpyProxy tests content in a VM for safety before sending it to the client [40]. Tahoma instead sandboxes the entire browser in a VM to contain any damage from malware [8]. In contrast, I explore vulnerability-specific filtering similar to Shield [54]. This technique cannot block zero-day exploits, but it can efficiently block all exploits of known browser vulnerabilities before patches are applied.

Script Injection Script injection (also known as XSS) attacks occur when adversaries place script code on an otherwise trusted web site. This script code runs when users visit the page, allowing adversaries to steal the users' cookies or private data, arbitrarily modify the victim web site, or launch distributed denial of service attacks against servers of their choice [21]. Web developers must carefully sanitize any input they receive from users, to prevent such scripts from appearing on their page. Unfortunately, comprehensively filtering script code from user input is a non-trivial problem, as demonstrated by the success of the Samy and Yamanner worms on MySpace and Yahoo Mail [1, 4].

Many server-side techniques have been proposed to block XSS attacks [47, 25, 26, 57], requiring each web site to carefully protect itself. Some client-side proposals attempt to mitigate the damage of XSS attacks, such as taint analysis in the browser [50] and proxies or firewalls for blocking certain attack behavior [27, 34]. These techniques are generally fail-open, where false negatives allow an attack to succeed. In my preliminary work, I propose a whitelist mechanism that offers a fail-closed solution. Concurrent work on BEEP proposed a similar mechanism [31]. While whitelists require changes to both web sites and browsers to achieve protection, they are backwards compatible with existing content and browsers.

Authentication Abuse Compared to XSS attacks, cross-site request forgery (CSRF) attacks exploit a very different weakness in browsers. CSRF attacks are an instance of the "confused deputy" problem [22]: an adversary fools the browser into abusing the user's credentials. For example, an adversary may post an image to a public forum similar to the following:

```
<img src="http://auction.com/
          bid.cgi?item=12&price=150">
```

Any browser that visits the forum will automatically request the above URL. If the user happens to be logged into `auction.com` in another window, the browser will send the user's credentials (in the form of a cookie) with the request, thus bidding on the adversary's chosen item. Unlike XSS attacks, this attack does not require placing content on the target web site.

Such attacks can succeed because browsers append authentication information (e.g., cookies and HTTP authentication credentials) on *all* requests to a given site, regardless of the origin of the request. CSRF attacks have been hypothesized for several years [56], and many recent vulnerabilities have surfaced, including ones affecting Netflix, Google Mail, and several open source web applications [17, 52, 33].

Like XSS attacks, many proposed solutions for CSRF require careful development practices on web servers [46]. A typical defense requires fresh tokens to accompany any authenticated request, to ensure the request came from a freshly generated page in the intended application. Jovanovic et al. propose a server-side proxy to automate the inclusion of such tokens [33]. On the other hand, client-side defenses for CSRF attacks are attractive, as they do not require changes to all web servers. Johns and Winter propose RequestRodeo [32], a client-side proxy that identifies suspicious, or "unentitled," requests between sites and strips authentication information from them. Subsequent requests to the site would continue to include authentication information. However, this approach is likely confuse users on legitimate links across sites: the first page view will not be authenticated, but subsequent page views will be. I propose a more principled and intuitive defense against CSRF attacks within the browser, using the session abstraction described in Section 4.1.

2.3 Related Web Concerns

Beyond the above topics of browser security, researchers have explored related aspects of security and robustness on the web. I offer a brief discussion to place my proposed work in context.

Phishing Phishing has proven to be a significant threat for web users. Phishing attacks lure users to a malicious web site that is similar in appearance to a trusted web site, for the purpose of harvesting private information like credit card numbers or passwords. Researchers have investigated phishing practices [11] and proposed mechanisms to help users detect attacks [10, 23]. For example, Dhamija and Tygar propose a mechanism that relies on users to match an image in the browser's user interface with an image on a web page to ensure the page is not spoofed [10]. Like most phishing defenses (including those offered by Tahoma [8] and Ye and Smith [58]), this work relies on having trusted portions of the UI that attackers cannot easily spoof.

While phishing attacks do pose a threat to web users, I view them as largely outside the scope of this work. My hypothesis addresses the system-level guarantees that users should expect of the browser as a runtime environment. Thus, I address phishing attacks conducted via XSS attacks, since these attacks violate the integrity of a legitimate page. However, I do not address any disconnect between the site a human believes he is visiting and the actual site he visits, nor do I discuss using trusted portions of the browser UI.

Server-side Security Many researchers have considered ways to improve the security of web applications on the server-side. Proposals include security gateways for policy enforcement [47], fault injection tools for testing [25], and static or dynamic analyses to eliminate particular threats [26, 57]. For example, Xie and Aiken suggest the use of static analysis of PHP programs to prevent SQL injection attacks [57]. They rely on developers to define black box sanitization functions, and their analysis can then track which strings need to be sanitized. While they claim the approach is amenable to XSS defense, the sanitization functions are much harder to implement for blocking JavaScript (as shown by the Samy worm [1]).

I view server-side security for web applications as complementary to my work. Clearly, web servers must be carefully protected from security threats, and flaws on servers present a risk for their clients. However, users cannot depend on *all* web sites to uniformly follow best security practices. Thus, any browser mechanisms to defuse attacks on the web have great value from a user's perspective. Improving browser security is also appealing due to the relatively small number of browser implementations to modify.

Cross-origin Communication In recent years, "mashup" web sites have become increasingly popular. Mashups compose content and script code from multiple hosts to provide a new and valuable service, such as overlaying bus routes on a map service [48]. Unfortunately, current web developers must trade security for functionality when building such sites, because the same-origin policy prevents data communication between a mashup and the sites it composes. Instead, mashups must either inefficiently proxy the desired data through their own server, or execute remote JavaScript files that provide the data (e.g., using JSON [2]). Malicious data providers could include arbitrary script code in such files, taking full control of the mashup site. No controlled communication mechanisms for this task currently exist.

Recent proposals have sought to improve data exchange between partially trusted principals in the browser. JSONRequest [9] offers a cross-origin RPC primitive that sanitizes JSON data before using it. Subspace [29] shows how to use embedded frames in current browsers to share a JavaScript object between otherwise isolated pages, allowing controlled communication at a slight setup cost. For a more principled approach, MashupOS [24] proposes abstractions in the browser to isolate content while supporting controlled communication and various trust relationships.

My proposed isolation abstractions are similar in purpose to those in MashupOS, but I focus on lower-level resource management and authentication isolation. There are important distinctions between the abstractions I propose and those in MashupOS, but the proposals are not necessarily incompatible. Supporting both low-level isolation and effective communication models between web applications is a valuable direction for future work.

3 Preliminary Work

My preliminary work has taken promising steps toward improving isolation and interposition in web browsers, as well as leveraging interposition for new security policies. I have isolated independent web content in the browser with OS processes, which

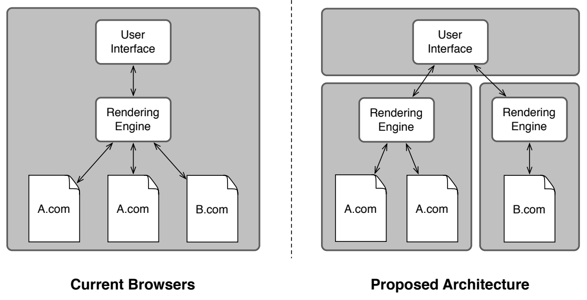

Figure 1: Current and proposed browser architectures. Gray boxes indicate process boundaries.

prevents interference due to resource contention. I have built a code rewriting framework to interpose on JavaScript and HTML content, which can defend browsers from exploits of known vulnerabilities. I have also implemented a XSS defense mechanism, which interposes on code passed to the browser's JavaScript engine. These efforts support my hypothesis that web browser security and robustness can be enhanced using mechanisms from OS research, without sacrificing backwards compatibility. In this section, I present strengths and weaknesses of these efforts, to explore how they can be extended to confirm my hypothesis.

3.1 Process-based Browser Isolation

Measurements of web content indicate a clear rise in the amount and complexity of JavaScript code being used on popular web sites. These sites are competing for resources like CPU and memory within the browser, and current web browsers do a poor job of isolating these sites from each other. Browser flaws can also lead to crashes when visiting a web site, and this can cause failures for many open web sites. I have observed test pages in the lab and actual pages on the web that trigger concrete problems, such as unresponsiveness and browser crashes. As web content continues to grow in complexity, these problems are likely to worsen.

My solution is to modify the web browser to isolate independent web sites from each other. In preliminary work with my advisors, I separate documents based on the hostnames (i.e., second-level domain names, such as washington.edu) from which they originate. This granularity is backwards compatible with the current same-origin policy in web browsers [44], so that documents that communicate via the DOM can continue to do so. I use OS processes as an isolation mechanism, so that documents from different hostnames are rendered in separate address spaces, as shown in Figure 1. To implement this, I built a prototype that places instances of KDE's KHTML rendering engine in separate processes. It then maps their displays into the correct windows, tabs, and frames of the Konqueror web browser.

The primary strengths of this work are the extra safety it provides for visiting web content from multiple hosts, backwards compatibility with existing web sites, and acceptable overhead for process-based isolation. I have demonstrated that the prototype can prevent content from one hostname from interfering with other content, in ways such as rendering engine crashes, unresponsiveness, and memory contention. The current prototype has incomplete support for cross-document communication, but it is otherwise backwards compatible with the 100 most popular web sites. I have also shown the overhead for starting rendering engine processes and visiting pages to be acceptably low.

However, the current work leaves room for improvement. The choice of hostname granularity for isolating content is simple and intuitive, but it can be further refined to protect users from additional threats, as I discuss in Section 4. It is also important to extend the prototype to support cross-document communication, to demonstrate backwards compatibility and measure overhead on a wider collection of real web sites. Such a performance evaluation should also reveal whether OS processes impose undue overhead for common browsing tasks, and thus whether lightweight fault domains should be considered.

3.2 BrowserShield

As complex runtime environments, web browsers periodically have flaws that can be exploited by malicious web sites. These flaws may allow adversaries to run arbitrary code on visitor's machines, presenting a critical threat to users. Browser developers work quickly to provide patches for such vulnerabilities when they are discovered, but these patches can be delayed for many reasons, such as extensive testing in enterprise environments.

With colleagues at Microsoft Research, I developed an interposition framework called BrowserShield that can enforce flexible security policies on web content [43]. These policies include *vulnerability shields* that can be used to detect and block exploits of known browser vulnerabilities, acting as a first line of defense until patches are applied. BrowserShield interposes on HTML and JavaScript content using code rewriting. Policies are written in JavaScript and provided to the framework to enforce.

BrowserShield has a number of key strengths. Because the framework is flexible and policy based, it can be useful for many applications, from security to comprehensive link rewriting. Our evaluation shows that BrowserShield can provide defense for actual browser vulnerabilities. Additionally, because BrowserShield operates on web content directly, no changes to the browser itself are required. Instead, the framework can be deployed in numerous ways, including on client proxies, firewalls, or web servers.

BrowserShield faces a few drawbacks, however, which may present challenges for effectively deploying security policies. First, it must perfectly match the tokenizing and parsing logic of the browsers it protects. This logic is complex, poorly specified, and different between browsers. As a result, attackers may use malformed input to bypass BrowserShield's filters and successfully run unfiltered code in a client's browser.[1] Second, the fact that BrowserShield's policies are expressed in JavaScript is both a strength and a weakness. While this provides a familiar and flexible way to specify policy logic, it offers only an indirect way to express some desired policies. For example, it is difficult to guard all the ways that JavaScript can insert HTML into a document, and it can be difficult to distinguish specific types of DOM objects, because JavaScript lacks the notion of a class. Third, BrowserShield currently only protects content containing HTML or JavaScript. Other formats like Flash also have access to the DOM, and they could be used to evade BrowserShield's policies. Some such formats may not be amenable to code rewriting techniques. Fourth, BrowserShield is unable to rewrite content encrypted with SSL. Finally, BrowserShield may face a prohibitive performance hurdle on pages with substantial JavaScript code, since JavaScript is a relatively slow language. The additional code introduced by BrowserShield's comprehensive rewriting may cause such pages to run unacceptably slowly.

3.3 Script Whitelists

As an example of leveraging interposition to improve security, I have modified Firefox to support script whitelists as a defense against XSS attacks. As discussed in Section 2.2, most existing solutions to XSS attacks are fail-open, allowing attacks to succeed if the mechanism or policy is incomplete. Attackers have proven capable of finding flaws in current defenses, so a fail-closed defense is desirable.

[1]The Samy worm used this technique to bypass MySpace's JavaScript filters [1].

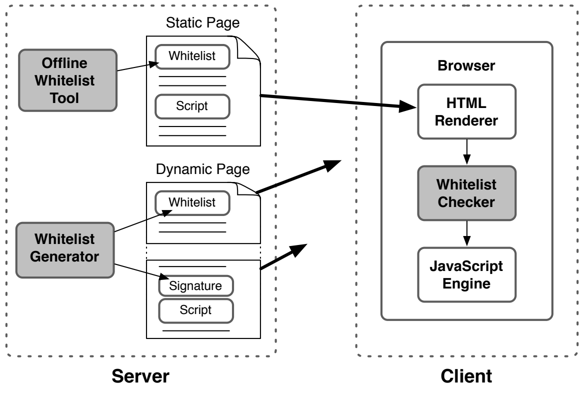

Figure 2: Script whitelist architecture. Whitelists can be added to static pages offline, and they can be added to dynamic pages with an online tool that signs scripts. Browsers check each script fragment against the whitelist before running it.

My solution relies on web developers to preface each page with a comment containing a *script whitelist*: a list of cryptographic digests of each script fragment on the page. My modified build of Firefox detects this comment if it is present, and it interposes on all code strings passed to the browser's JavaScript engine. If the digest of a given fragment does not appear in the whitelist, the browser does not run it. Some extensions to this basic approach are necessary to support real web sites in practice, such as pages for which all script code may not be known in advance. I support flags for blindly trusting code generated by other scripts or from particular third parties, as well as a signature mechanism for scripts that are generated after the page header has been sent to the client. This architecture can support both static and dynamic pages, as shown in Figure 2.

The main benefit of this work is that it gives web developers a practical means to defend themselves against script injection attacks. Regardless of how attackers might find subtle ways to insert content into the page, developers must only ensure the whitelist is protected to prevent attack scripts from running. While this approach requires modifications to both web sites and browsers to prevent attacks, it is backwards compatible with existing browsers and sites. Finally, the extensions to the basic whitelist architecture ensure that whitelists can be effectively deployed on real sites, as my tests with the phpBB application demonstrate.

Unfortunately, the use of signatures for dynamic scripts adversely impacts throughput on the server. Also, presenting developers with an all-or-nothing trust model for generated code and third party scripts is unattractive in many cases. Building upon the trust models proposed by MashupOS [24] may provide one valuable direction for future work. As a final drawback, the prototype requires changes to the Firefox source code and cannot be distributed as an extension, making it more difficult to distribute.

4 Proposed Work

I propose to extend this preliminary work in two directions: isolation and interposition. The proposed work will resolve weaknesses in my preliminary work and provide defenses against additional security threats. It will support my principal hypothesis by employing mechanisms from OS research to improve the security and robustness of web browsers. It will also preserve backwards compatibility with existing web content and offer a flexible platform for future web security research.

First, I will refine the boundaries for isolating web content in the browser. From the current implicit boundaries in the browser, I define two new isolation abstractions: *sessions* and *hosts*. These abstractions allow users to treat a single web browser as if it were multiple separate browsers. I further divide the browser between *user interface* and *runtime environment* abstractions, and I propose using an independent runtime environment for each session. The process-based isolation from my preliminary work will prevent unwanted interference between runtime environments. In addition, the new session and host abstractions will defend against CSRF attacks at natural boundaries.

Second, I will incorporate an interposition layer within the browser itself, supporting BrowserShield-like security policies for the browser's runtime environments. This layer will enforce policies at several useful levels of abstraction, including at the levels of HTML, the DOM, and JavaScript. Compared to BrowserShield, this approach will interpose more consistently on all web content regardless of format, and it will reduce the overhead that accompanies code rewriting. Importantly, the interposition layer will act as an extensible security architecture for the entire browser, serving as a platform for future security research. It should allow new defense mechanisms like script whitelists to be distributed as policies or browser extensions rather than browser source code modifications.

Together, these isolation and interposition proposals will complement each other. Policies may be universal to the browser or specific to a session, such that different policies may be applied to different browser runtime environments. The proposed isolation between runtime environments provides natural boundaries for policies, creating a simple and flexible mechanism for enhancing browser security.

4.1 Isolation Abstractions

To refine my preliminary work on isolating web content within the browser, I note that many security and robustness problems in current browsers could be avoided if users ran multiple web browsers. Specifically, *if a different browser were used for each instance of each web site, many problems would go away.* Resource contention and fatal errors would be handled by the OS and not the browser, which would alleviate CPU contention between sites and isolate crashes. CSRF attacks would be defeated, because credentials from one browser would not be sent when following a link in another browser. Users would gain additional power as well, as they could log into the same site with different accounts concurrently.

This literal approach is not generally feasible or attractive because there are too few independent web browsers for users to run, and because users expect a consistent set of bookmarks, preferences, and plugins across sites. Also, the naive approach of using a separate browser for each site instance is not desirable, because some resources *should* be shared between instances of the same site (e.g., cached objects).

My proposal captures the essence of using multiple web browsers with only a single browser, while clearly defining boundaries for isolating and sharing browser resources. I propose four abstractions within the web browser: *user interface*, *runtime environment*, *host*, and *session*. The web browser will isolate resources and authentication information between these abstractions. Unlike current browsers, a single web browser could then support multiple independent browsing sessions.

User Interface and Runtime Environments Most modern web browsers consist of two largely disjoint sets of functionality: the browser's user interface and its HTML rendering engine. For example, Firefox has a separate rendering engine called Gecko, while KDE's KHTML rendering engine is used in both the Konqueror browser and Apple's Safari browser. The browser's *user interface* consists of features for navigating across sites and within the browser's history, specifying preferences, storing bookmarks, and so on. In contrast, the rendering engine is responsible for displaying each web site and allowing users to interact with it. Because this engine must also support JavaScript execution and plugins like Flash and Java applets, I refer to the rendering engine as the browser's *runtime environment*. For purposes of isolation, I will draw a clear boundary between user interface and runtime environment, allowing a single user interface to support multiple independent runtime environments. One challenge here is how best to share certain resources, like preferences, between the UI and each runtime environment.

Host Abstraction I propose a first-class *host* abstraction within the browser, which comprises all documents that originate from a given hostname. This abstraction acts as a superset of the "origin" used in the browser's current same-origin policy [44], such that any two documents that can currently communicate via the DOM must be part of the same host.[2]

This abstraction matches the granularity of process-based isolation from my preliminary work. In this proposal, however, the host abstraction will correspond to a *partition of storage* in the browser, rather than an individual process. Building upon the privacy work of Jackson et al. [28], the host abstraction will consist of a partition of the browser's cache, visited link history, and certain cookies. The host abstraction is only used to partition *persistent* cookies, which are those HTTP cookies for which the server specifies an expiration date [35]. These cookies are commonly used for remembering users and their preferences, but typically not for authentication. In contrast, *session* cookies specify no expiration date and are currently deleted by web browsers only when the browser exits. Such cookies are typically used for authentication purposes, and I partition them at a finer granularity than hosts.

The challenges for building this abstraction include identifying additional browser resources to isolate between hosts, and determining how best to partition these resources, since they may be shared across multiple processes.

Session Abstraction I propose a first-class *session* abstraction within the browser, as a subdivision of the host abstraction. A session consists of all documents from a given host that share either a navigational or a parent-child relationship in the browser. A *navigational relationship* exists between two documents when a user follows a link from one to the

[2] "Origins" correspond to the port, protocol, and full domain name of an object. However, scripts can truncate the domain name used by the policy to a hostname (e.g., maps.google.com to google.com). For this reason, I choose the hostname as a basic unit of isolation.

other, either in the same window or in a new window. Note that following a link from one host to another constitutes the creation of a new session. A *parent-child relationship* is created by embedding a child document as a frame in a parent document, or by programmatically opening a child document in a new window. This relationship corresponds to the set of documents that can communicate via the DOM: a document has no way of naming another document with which it does not share a parent-child relationship. An important note is that a user may have multiple concurrent and independent sessions with the same host.

Sessions will be isolated from each other in two ways. First, each session will correspond to exactly one instance of a browser runtime environment. Runtime environments will be separated by OS processes or other lightweight fault domains to support resource isolation between sessions. Using shared libraries for the runtime environments will help keep memory overhead low. Second, authentication information will be isolated between sessions. This information includes both session cookies and HTTP authentication credentials [32]. Because following a link between hosts creates a new session, CSRF attacks cannot be launched across host boundaries. Instead, a user's authentication credentials remain tied to the session in which she logged in. This feature also allows web sites to easily support concurrent logins from a single user, if they rely on session cookies to distinguish users.

Challenges for implementing the session abstraction include keeping overhead acceptably low and helping users distinguish between sessions in the browser. Identifying sessions is important for accounting (e.g., tracking down a session with a memory leak), terminating misbehaved sessions, or keeping track of authenticated sessions.

Relationship with the Browser In this proposal, a web browser uses a single user interface to display content from multiple runtime environments, one per session, as shown in Figure 3. The abstractions of the UI (windows, tabs, and frames) are independent of the abstractions of the runtime environments (sessions and hosts). I consider *frames* as the basic unit for the UI: each tab consists of a top-level frame and optionally some number of embedded frames, and each window consists of one or more tabs. Each web document is associated with exactly one frame and exactly one session; each frame is associated with a list of documents that form its history. The browser's UI is responsible for *(1)* mapping documents from each session to their respective frames,

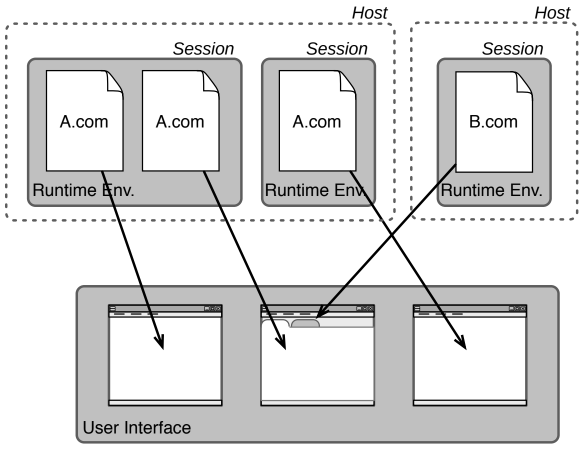

Figure 3: The browser's user interface displays documents from multiple runtime environments. Each runtime environment corresponds to a session.

(2) handling cross-session navigation, and *(3)* managing bookmarks and preferences.

On the whole, this proposal will create a robust environment for interacting with web content from independent hosts. It will isolate resource usage and authentication information between browsing sessions, and it will aim to prevent privacy leaks between hosts. These benefits can be achieved without losing backwards compatibility with existing web content, as I will demonstrate in my evaluation.

4.2 Browser Interposition Layer

BrowserShield demonstrated the utility of using flexible security policies to protect web browsers. However, as discussed in Section 3.2, providing complete interposition from outside the browser can be problematic. Similarly, related work on browser interposition is limited to particular forms of content, such as JavaScript [3, 31] or Java [53].

In practice, many browsers also support extensions to modify their behavior, but the frameworks for such extensions are not designed with security in mind. For example, Firefox extensions appear to have no access to a page before it has been parsed or even before scripts begin to execute.[3] Thus, it can be difficult to improve browser security with current extension frameworks.

Supporting Direct Interposition For these reasons, I propose implementing a secure interposition

[3]For instance, the NoScript extension indirectly blocks scripts using Firefox's principal-checking logic [37, 38].

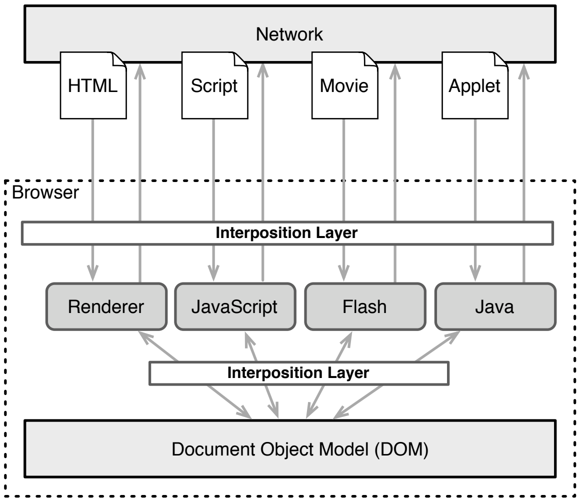

Figure 4: The proposed browser interposition layer will mediate access to the DOM and network I/O, regardless of content type.

layer within the browser itself. The layer will consist of a set of hooks for registering policy functions within the browser. The hooks will give policies direct access to several key aspects of the browser, including HTML, DOM elements, communication, and JavaScript, as shown in Figure 4. More details on the proposed hooks are specified below.

- *HTML hooks* allow policies to intercept HTML strings before the browser parses and renders them. These hooks are necessary for implementing vulnerability shields.
- *DOM hooks* allow security policies to regulate how particular DOM elements can be accessed by any web content, regardless of format. Such policies could govern how browser extensions access the DOM as well.
- *Communication hooks* allow policies to restrict or allow cross-document communication or HTTP requests in flexible ways. Such hooks may be useful for defining confinement policies on particular sites, similar to Tahoma [8], or MashupOS-style sandboxes within documents [24].
- *JavaScript hooks* allow policies to intercept JavaScript code strings before they are parsed or executed, which will allow the construction of script whitelist mechanisms as policies. Hooks inside the JavaScript engine will also allow interposition on individual JavaScript functions and object manipulations, as supported by BrowserShield. Similar hooks could be provided for other web content types.

The appropriate way to specify policies remains an open challenge. It may be the case that the JavaScript language is familiar and expressive enough to define policies, if the appropriate hooks are made available. Alternatively, the policy modules used in Janus [20] could provide inspiration: modules could read in a high-level policy specification and enforce it on the relevant set of hooks.

Janus also demonstrates how multiple policies can be registered on a single hook, allowing some to take precedence over others. This would support hierarchical sets of policies. Policies defined within the browser would take precedence, but additional policies could be provided in browser extensions or with individual web documents.

Thus, I consider multiple deployment scenarios for browser policies. The browser itself will include a set of basic policies to confine web content. Vulnerability shields could be distributed by system administrators like virus signatures, without requiring browser restarts. Users could also install their own policies to add browser functionality or enhance security, as a form of browser extension. Web sites could also define their own policies to apply to their pages, as supported by BEEP only for JavaScript policies [31]. Note that such site-specific policies are easy to isolate from other web sites, given the isolation of runtime environments described in Section 4.1.

Anupam and Mayer note that the composition of security policies presents a challenging open question [3]. For example, if JavaScript code under one policy can communicate with an applet under a different policy, both policies may be violated. My proposed interposition layer can possibly address this challenge in browsers by uniformly enforcing basic policies independent of the type of web content.

Finally, note that the set of hooks in the interposition layer may be largely browser agnostic, because it applies primarily to content-specific issues. This suggests that some policies could be browser independent, while others could specify a list of specific browsers and versions to which they apply.

Exploring Desirable Policies It is important to demonstrate that the browser interposition layer is expressive enough to support useful security policies. Thus, I will explore the space of desirable policies and implement a representative set. Clear candidates include vulnerability shields like those in BrowserShield, as well as a script whitelist mechanism similar to that presented in preliminary work. I will pursue enhancements to the whitelist mechanism to improve its ability to handle dynamic scripts efficiently and partly trusted scripts safely. Such enhancements could include consolidating signed code and sandboxing scripts with unknown content.

I will also explore to what extent browser extensions and plugins can be confined by policies, to apply the principle of least privilege [45]. How easily filesystem and network access can be mediated for extensions and plugins remains an open question. Finally, I will show how policies for the interposition layer can support more powerful manipulations of web content than current browser extensions.

4.3 Summary

The two principal aspects of my proposal combine isolation of web content and interposition using flexible security policies. These enhancements will make visiting untrusted content in the browser demonstrably safer. It will also become possible for researchers to easily explore new browser security mechanisms, by authoring policies rather than changing browser source code.

5 Methodology and Evaluation

My hypothesis states that OS research can be used to improve the security and robustness of modern web browsers. To evaluate whether this is correct, I must show that the isolation and interposition mechanisms that I propose for browsers *(1)* create a demonstrably safer environment for interacting with web content, *(2)* do not disrupt existing web content, and *(3)* offer benefits at an acceptable performance cost. My evaluation will show that implementations of these mechanisms can defend against existing security threats and allow the browser to operate reliably in the face of untrusted content. I will incorporate measurements of popular web sites to show both that the stated problems exist in practice and that my proposals offer practical solutions.

5.1 Isolation Abstractions

I will implement the proposed session and host isolation abstractions by extending my browser prototype from preliminary work. The current prototype isolates instances of Konqueror's KHTML runtime environment in separate processes.[4] It must be modified to *(1)* assign documents to KHTML instances based on sessions, *(2)* support cross-document communication within a session, *(3)* partition session cookies between sessions, and *(4)* partition other persistent state between hosts. I will implement a pre-fork optimization to reduce session startup time, and I will implement a session management tool to allow users to view each session's resource utilization, as well as terminate misbehaving sessions.

[4]I chose Konqueror over the more popular Firefox browser because it offers better support for concurrent processes.

Once these features are implemented, I will evaluate the prototype's safety, backwards compatibility, and efficiency. For safety, I will leverage my earlier resource interference tests to show effective isolation of crashes, CPU contention, and memory leaks. I will augment these tests with further examples of real world sites that cause interference with other sites. In addition, I will show that the prototype can defeat observed CSRF attacks by isolating authentication information between sessions.

For backwards compatibility, I will test a representative sample of popular web content in the prototype browser. Unlike the tests in my preliminary work, these tests will use recorded copies of web content to facilitate repeatability. I am primarily concerned with preserving the functionality of existing sites and do not expect to impact visual layout. Thus, these tests will compare the sets of JavaScript errors and the sets of loaded objects between an unmodified browser and the prototype browser. I will also characterize the use of session cookies and persistent cookies for authentication on popular web sites. This will show whether the prototype can defend current sites from CSRF attacks, or whether some sites will need to switch from persistent cookies to session cookies for authentication purposes.

For efficiency, I will measure the time and memory costs for starting a new session and for navigating intra-session and inter-session links. I will also evaluate whether process-based isolation should be replaced with lightweight fault domains like XFI [13]. To do this, I will measure the number of cross domain calls during typical browsing operations to estimate the communication cost for using processes. These metrics will help reveal whether the time and space costs of using processes for isolating browser sessions are acceptable.

5.2 Browser Interposition Layer

To evaluate the proposed interposition layer, I will implement a prototype based on the Firefox browser. I will define an appropriate set of hooks for registering policies, and I will make the hooks available via the API for Firefox extensions. Modifying Firefox is attractive because of its wide adoption and popular support for extensions. I will also consider the challenges of implementing the interposition layer in

Konqueror, to combine my two proposals in a single browser implementation.

The interposition layer improves safety only via the policies it supports, so I will evaluate its expressiveness and the safety offered by example policies. I will build a representative set of security policies that are not easily expressible in current browsers. These will include vulnerability shields to defend known Firefox vulnerabilities. I will show that such shields can block observed exploits of these vulnerabilities. I will also implement an improved script whitelist mechanism as a policy, for which I will show effective defense against observed XSS attacks. Finally, I will explore the interactions between multiple policies, showing how browser-level policies can take precedence over policies provided by extensions and web content.

The interposition layer itself should not impact backwards compatibility with existing web content. However, some policies may impact the way existing web content behaves. I will build a default set of policies that enforce current browser semantics, and I will demonstrate that popular content is not disrupted by these policies. Similarly, I will evaluate whether vulnerability shields create false positives for existing popular content. To achieve this, I will use the same methodology proposed for evaluating the isolation abstractions in Section 5.1. I will also investigate improvements to the practicality of script whitelists (e.g., consolidation of signed scripts), and I will incorporate whitelists into additional web applications.

For efficiency, I will evaluate the impact of both the interposition layer and various security policies on browser performance. I will construct microbenchmarks that exercise each of the implemented hooks, as well as macrobenchmarks based on observed content from popular sites. I will then compare the running times of these benchmarks on an unmodified copy of Firefox, a copy of Firefox with the interposition layer but no policies, and copies of Firefox with various security policies. I will also show the performance impact of composing multiple policies. In addition, I will evaluate script whitelist performance in this architecture, measuring both client rendering times and server throughput. I anticipate that the overhead of the interposition layer can be kept substantially lower than that of BrowserShield, and that improvements to the script whitelist proposal may reduce its impact on server throughput for highly dynamic sites.

6 Conclusion and Future Work

Modern web browsers face a large number of threats to their security and robustness. In particular, four prominent threats include exploitable vulnerabilities, interference from resource contention, XSS attacks, and CSRF attacks. My hypothesis states that isolation and interposition mechanisms from operating systems research can be applied to web browsers to effectively defend against these threats. This hypothesis leverages the analogy between operating systems and web browsers, both of which are responsible for executing and confining programs from different origins. My proposed work offers principled abstractions for isolating untrusted content in the browser, and it evaluates mechanisms for enforcing this isolation on existing content. My proposal also supports extensible security policies with two types of interposition layers. First, BrowserShield uses code rewriting and can be deployed independent of clients. Second, an interposition layer within the browser can support policies independent of content type, at a lower cost than code rewriting.

While this work addresses several current threats on the web, there are many directions open for future exploration. First, the proposed isolation and interposition mechanisms may be helpful for exploring how web content from different sources can communicate or otherwise be integrated safely. Such work could build on the sandboxes proposed by MashupOS [24] and BEEP [31], potentially defining an API to allow documents in different sessions to communicate with each other. Second, the isolation of session and host abstractions may play a useful role in defending against phishing attacks. For example, visual indicators of sessions in the browser's user interface could reduce opportunities for spoofing attacks. Finally, I hope for the browser interposition layer to act as a platform for future web browser security research, as it will support the distribution of security policies as browser extensions rather than modifications to browser source code.

Overall, the preliminary, proposed, and future directions discussed in this report support the hypothesis that web browser security and robustness can be demonstrably improved through the use of ideas and mechanisms from operating systems research.

Acknowledgments

The preliminary work described in this report was joint work with my advisors, Steven D. Gribble, Henry M. Levy, and Brian Bershad. BrowserShield was a joint project with colleagues at Microsoft Re-

search, including John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. Finally, Steve Balensiefer provided useful thoughts about CSRF attacks and browsing sessions.

References

[1] Technical explanation of the myspace worm. `http://namb.la/popular/tech.html`, 2005.

[2] Introducing json. `http://www.json.org/`, 2006.

[3] V. Anupam and A. Mayer. Security of web browser scripting languages: Vulnerabilities, attacks, and remedies. In *USENIX Security Symposium*, Jan. 1998.

[4] C. Babcock. Yahoo mail worm may be first of many as ajax proliferates. `http://www.informationweek.com/security/showArticle.jhtml?articleID=189400799`, June 2006.

[5] G. Back, W. C. Hsieh, and J. Lepreau. Processes in kaffeos: Isolation, resource management, and sharing in java. In *OSDI*, Oct. 2000.

[6] N. S. Borenstein. Email with a mind of its own: The safe-tcl language for enabled mail. In *IFIP Conference on Upper Layer Protocols, Architectures, and Applications*, 1994.

[7] S. M. Christey. Vulnerability type distribution in cve. `http://www.attrition.org/pipermail/vim/2006-September/001032.html`, Sept. 2006.

[8] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A safety-oriented platform for web applications. In *IEEE Symposium on Security and Privacy*, 2006.

[9] D. Crockford. Jsonrequest. `http://www.json.org/JSONRequest.html`, 2006.

[10] R. Dhamija and J. D. Tygar. The battle against phishing: Dynamic security skins. In *Symposium on Usable Privacy and Security (SOUPS 2005)*, July 2005.

[11] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *CHI*, Apr. 2006.

[12] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *SOSP*, 2005.

[13] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *OSDI*, 2006.

[14] Ú. Erlingsson and F. B. Schneider. IRM enforcement of java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

[15] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, pages 32–45, 1999.

[16] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *ACM Conference on Computer and Communications Security*, pages 25–32, 2000.

[17] D. Ferguson. Netflix.com xsrf vuln. `http://www.webappsec.org/lists/websecurity/archive/2006-10/msg00063.html`, Oct. 2006.

[18] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *NDSS*, 2003.

[19] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *NDSS*, 2004.

[20] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *USENIX Security Symposium*, July 1996.

[21] J. Grossman. Cross-site scripting worms and viruses. `http://www.whitehatsec.com/downloads/WHXSSThreats.pdf`, Apr. 2006.

[22] N. Hardy. The confused deputy (or why capabilities might have been invented). *Operating Systems Review*, 22(4):36–8, Oct. 1998.

[23] A. Herzberg and A. Gbara. Trustbar: Protecting (even naive) web users from spoofing and phishing attacks. In *Cryptology ePrint Archive: Report 2004/155*, 2004.

[24] J. Howell, C. Jackson, H. J. Wang, and X. Fan. Mashupos: Operating system abstractions for client mashups. In *HotOS XI*, Apr. 2007.

[25] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai. Web application security assessment by fault injection and behavior monitoring. In *WWW*, May 2003.

[26] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *WWW*, May 2004.

[27] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguichi. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In *AINA*, 2004.

[28] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *WWW*, May 2006.

[29] C. Jackson and H. J. Wang. Subspace: Secure cross-domain communication for web mashups. In *WWW*, May 2007.

[30] Java Community Process. Jsr 121: Application isolation api. `http://jcp.org/en/jsr/detail?id=121`, June 2006.

[31] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW*, May 2007.

[32] M. Johns and J. Winter. Requestrodeo: Client side protection against session riding. In *OWASP Europe Conference*, May 2006.

[33] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *SecureComm*, Aug. 2006.

[34] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A client-side solution for mitigating cross-site scripting attacks. In *ACM Symposium on Applied Computing (SAC)*, 2006.

[35] D. Kristol and L. Montulli. Http state management mechanism. RFC 2109, Feb. 1997.

[36] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[37] G. Maone. Noscript - whitelist javascript blocking for a safer firefox experience. `http://www.noscript.net`, 2006.

[38] G. Maone. Personal communication, Nov. 2006.

[39] Microsoft. Microsoft silverlight. `http://www.microsoft.com/silverlight/`, Apr. 2007.

[40] A. Moshchuk, T. Bragin, D. Deville, S. D. Gribble, and H. M. Levy. Spyproxy: On-the-fly protection from malicious web content. In *Usenix Security (to appear)*, Aug. 2007.

[41] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware on the web. In *NDSS*, Feb. 2006.

[42] J. K. Ousterhout, J. Y. Levy, and B. B. Welch. The safe-tcl security model. Manuscript, 1996.

[43] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. In *OSDI*, Nov. 2006.

[44] J. Ruderman. The same origin policy. `http://www.mozilla.org/projects/security/components/same-origin.html`, 2001.

[45] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Communications of the ACM*, 17(7), 1974.

[46] T. Schreiber. Session riding: A widespread vulnerability in today's web applications. `http://www.securenet.de/papers/Session_Riding.pdf`, Dec. 2004.

[47] D. Scott and R. Sharp. Abstracting application-level web security. In *WWW*, May 2002.

[48] C. Smoak. Seattle bus monster. `http://www.busmonster.com/`, 2005.

[49] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *SOSP*, 2003.

[50] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, 2007.

[51] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.

[52] J. Walker. Csrf attacks or how to avoid exposing your gmail contacts. `http://getahead.org/blog/joe/2007/01/01/csrf_attacks_or_how_to_avoid_exposing_your_gmail_contacts.html`, Jan. 2007.

[53] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for java. In *SOSP*, Oct. 1997.

[54] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *SIGCOMM*, Aug. 2004.

[55] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *NDSS*, 2006.

[56] P. Watkins. Cross-site request forgeries. `http://www.tux.org/~peterw/csrf.txt`, 2001.

[57] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security*, Aug. 2006.

[58] Z. Ye and S. Smith. Trusted paths for browsers. In *Proceedings of the 11th Usenix Security Symposium*, 2002.