Web Browsers as Operating Systems: Supporting Robust and Secure Web Programs

Charles Reis

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2009

Program Authorized to Offer Degree: Computer Science and Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Charles Reis

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Co-Chairs of the Supervisory Committee:

_____

Steven D. Gribble

_____

Henry M. Levy

Reading Committee:

_____

Steven D. Gribble

_____

Henry M. Levy

_____

Dan Grossman

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Proquest Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, 1-800-521-0600, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

Web Browsers as Operating Systems:
Supporting Robust and Secure Web Programs

Charles Reis

Co-Chairs of the Supervisory Committee:
Associate Professor Steven D. Gribble
Computer Science and Engineering

Professor Henry M. Levy
Computer Science and Engineering

The World Wide Web has changed significantly since its introduction, facing a shift in its workload from passive web pages to active programs. Current web browsers were not designed for this demanding workload, and web content formats were not designed to express programs. As a result, the platform faces numerous robustness and security problems, ranging from interference between programs to script injection attacks to browser exploits.

This dissertation presents a set of contributions that adapt lessons from operating systems to make the web a more suitable platform for deploying and running programs. These efforts are based upon four architectural principles for supporting programs. First, we must recognize web programs and precisely identify the boundaries between them, while preserving compatibility with existing content. Second, we must improve browser architectures to effectively isolate web programs from each other at runtime. Third, publishers must have the ability to authorize the code that runs within the programs they deploy. Fourth, users must be able to enforce policies on the programs they run within their browser.

In this work, I incorporate these architectural principles into web browsers and web content, and I use experiments to quantify the improvements to robustness and performance while preserving backward compatibility. Additionally, some of these efforts have been incorporated into the Google Chrome web browser, demonstrating their practicality.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I am extremely fortunate to have spent my graduate school career with such a strong and supportive group of advisors, colleagues, and friends, without whom I could not have reached this point.

I must first thank my advisors, Steve Gribble and Hank Levy, who taught me to always pursue another level of depth without getting lost in the details. From them, I learned the importance of telling a good story, both in writing and in presenting, as well as asking the right questions. I have immensely enjoyed working with them.

I am particularly grateful to Brian Bershad, whose relentless feedback helped me embark on this work in earnest, and whose support allowed me to have impact within the very impressive Google Chrome team. I also thank Linus Upson, Darin Fisher, and Nicolas Sylvain for their immense support during my time at Google, and I look forward to working more with all of them.

Dan Grossman has also been a valuable member of my committee and a fun source of advice through many years of PL seminars.

I have been fortunate to work with a wonderful set of additional co-authors on the work that appears in this dissertation, including Helen Wang, John Dunagan, Opher Dubrovsky, Saher Esmeir, Yoshi Kohno, and Nick Weaver. I have also greatly enjoyed working with Adam Barth and Collin Jackson on related papers and projects, as well as with David Wetherall, John Zahorjan, Ratul Mahajan, and Maya Rodrig on my earlier wireless networking research.

All of my friends, office mates, house mates, and team mates have made my time at UW CSE a fantastic experience. In particular, Steve Balensiefer, Ben Lerner, Stef Schoenmackers, Kevin Wampler, and Brian Ferris have helped me enjoy both work and life outside of it. Adrien Treuille was always willing to throw at a moment's notice, and Office 502 understood

the value of putting on a good TGIF. I am also grateful to have shared a lab with Roxana Geambasu, Alex Moshchuk, Tanya Bragin, Jon Hsieh, Dave Richardson, Nick Murphy, and Mark Zbikowski, and I wish them the best in their work. Also, without Lindsay Michimoto's advice and hard work behind the scenes, we would all surely be lost.

Most of all, I thank Kate Everitt for her endless support and affection, through both the highest and lowest points of grad school. My family's pride and support through these years have also been invaluable.

Finally, I would be remiss not to acknowledge the many sources of funding that allowed me to pursue this work. For that, I am grateful to the National Science Foundation, the Torode and Wissner-Slivka families, Google, Microsoft Research, Cisco, and Intel.

DEDICATION

To my grandfather, John Skinner, who holds my respect, encouraged me intellectually from a young age, and supported my education.

Chapter 1

INTRODUCTION

In this dissertation, I discuss ways to improve the architecture of the World Wide Web to reflect a significant shift that has occurred in its workload. Web browsers are now commonly used to run networked applications written in script code, not simply to navigate among linked documents. Web browsers and the content types they support were not designed for this active workload, leading to demonstrable robustness and security failures when running programs within browsers. This work aims to address such failures by revising the underlying browser architectures and definitions of web content, adapting lessons from modern operating systems and desktop applications.

1.1 Motivation

The nature of web content has shifted significantly over the past several years. Web pages were once predominantly static documents, containing at most small snippets of active code. Today, the web has become a viable platform for deploying applications, and it has the potential to become the predominant way to do so in the future. It is now common for web pages to contain substantial amounts of script code meant to run within the client's web browser, and role of this code has changed from manipulating document content to providing client-side libraries for interactive user interfaces. This changes web pages from *passive documents* into *full-featured programs*, ranging from email clients to calendar applications to word processors.

This workload shift changes the role of the web browser. Rather than acting solely as a document renderer, the browser must now perform many of the duties of an operating system: executing the code of multiple independent programs safely and with high performance. Like an operating system, the browser must provide runtime environments for

programs that demand user interaction, network support, and local storage. The browser must isolate these programs to prevent unwanted interactions, and it must identify principals on which to enforce security policies. In these ways, the browser faces many of the same challenges posed to early operating systems, which had to discover appropriate principals, mechanisms, and policies to support diverse programs.

However, web browsers have not traditionally been designed with an operating system perspective. They have slowly evolved into their current role as features have been added to web content, such as support for asynchronous network communication using `XmlHttpRequests`. As a result, the underlying architectures of current browsers have not benefitted from the design of modern operating systems. Instead, these architectures assume each page can be rendered and displayed with little regard to its client-side workload, and they provide subtle and inconsistent security policies that expose unexpected interactions between principals.

Similarly, web content formats like HTML and JavaScript were designed for document layout and not for application development and deployment. These formats thus provide little guidance to the browser about properties relevant to running programs, from issues as basic as defining program boundaries to the ability to restrict what code is allowed to run in a program. This leaves browsers ill-equipped to isolate or enforce appropriate policies on the programs that users interact with today.

In both browsers and web content, we thus find a mismatch between the architecture used to define and execute code from the web and the workload this architecture currently faces. As a result, the web faces substantial challenges for supporting this new set of active programs. These challenges have led to concrete problems that can be observed today:

- Interference between programs within the browser can disrupt users' experiences, due to a lack of clear boundaries and insufficient isolation mechanisms. For example, browser crashes terminate many independent programs and unresponsiveness in one program may prevent users from interacting with other programs [99, 102].

- A user's credentials with one web site may be put at risk by the actions of another web site, because the sites that act as principals on the web are not effectively isolated.

This puts the browser in danger of acting as a confused deputy, resulting in cross-site request forgery (CSRF) attacks [57, 132].

- Attackers have numerous opportunities to inject malicious code into web pages, because code and data are often difficult to distinguish in web content. This has caused cross-site scripting (XSS) attacks to become one of the most widely reported classes of vulnerabilities [28].

- Browsers themselves are significantly complex codebases that routinely face untrusted input, making vulnerability exploits a pressing concern [101, 21].

These problems are not an exhaustive set of the concerns with today's web, which include additional security issues such as phishing attacks. However, they do represent a class of issues that are direct symptoms of the mismatch between the current web architecture and the interactive programs it is trying to support.

To address these problems and improve the web as a platform for deploying networked applications, we draw comparisons to the world of operating systems and desktop applications. In this world, programs are clearly defined and isolated from each other. There are mechanisms that provide failure isolation and concurrent program execution, mechanisms to verify the integrity of program contents, and opportunities to interpose on program behavior to enforce novel policies.

Conversely, there are some notable differences between the web and desktop worlds that are both desirable and important to consider when revising the web architecture. For example, web content is much less trusted than installed desktop applications and has correspondingly fewer privileges. These restricted privileges allow users to freely explore web content with fewer trust decisions than in the desktop application world. They also lower the barrier to entry for publishers, who must first establish reputations in the desktop world. However, this lack of trust also increases the importance of identifying principals and placing limits on their behavior.

Similarly, composition among independent programs and principals is more common on the web than among desktop applications, despite the mutually untrusted nature of

principals on the web. This creates interesting compatibility challenges for strongly isolating principals from each other, while still supporting popular mashups and uses of available content. It is thus important to recognize the boundaries between principals while providing opportunities for them to collaborate.

These differences between the web and the desktop worlds are desirable properties, despite the challenges they pose. As a result, the ideal web browser and web content format will not merely be a recasting of today's operating systems and applications into the web. The web architecture must instead adapt lessons from these worlds, while preserving the ease with which web content can be published, composed, and explored.

1.2 Thesis and Contributions

My thesis is that web browsers and web content can effectively support the current trend toward web-based programs by adapting principles and mechanisms from operating systems. By applying lessons learned in the operating system space to the architecture of web browsers and the ways web-based programs are defined, we can alleviate the class of problems discussed above.

To support this thesis, I outline a set of four architectural principles to make the web a safer platform for deploying programs [104], and I describe my specific research contributions associated with each principle. These principles are detailed below and are derived from properties of modern operating systems. They directly address the challenges we face in supporting programs in both web browsers and web content formats.

1. *Identify Program Boundaries.* Desktop programs generally have clearly defined boundaries, allowing the operating system to run program instances independently in separate processes. This represents one of the core functions of modern operating systems: allowing independent programs to run concurrently, free from interference from other programs. Surprisingly, there is no clear program abstraction for web browsers and web content. This leads to architectures with significant robustness and security problems as web-based programs become increasingly complex. I argue that precise *web program* abstractions are required to safely support programs within web browsers,

and that we must make some compromises from ideal definitions to maintain compatibility with existing web content. I have contributed a set of such abstractions that identify *site instances* as an isolatable unit within the browser [102].

2. *Isolate Programs From Each Other.* Given adequate program abstractions, operating systems can effectively isolate program instances using processes. The process mechanism provides each program instance a separate address space, failure isolation, concurrency, resource management, and independent access control. Current browser architectures provide inadequate isolation between web program instances, resulting in an unsafe environment for program code. I argue that operating system processes are an appropriate mechanism for use within the browser's architecture for preventing interference between web program instances, without precluding any data sharing necessary between instances. I have contributed multiple implementations of such an architecture, using the site instance abstraction as the basis of isolation [99, 102].

3. *Authorize Program Code.* Desktop program publishers often sign program installers or include checksums to verify the integrity of their own code. It is thus uncommon for other principals to inject code into such programs before they are installed. We face a different scenario on the web, in which it is hard to restrict what active code appears in a web program. Unexpected code may be injected as the program is generated on the web server, as in an XSS attack, or while it is in transit to the client's browser. Such in-flight content changes occur surprisingly often in practice, as I demonstrate in a measurement study [103]. I argue that integrity mechanisms are an important tool for web publishers to permit reasoning about the behavior of their programs. I have contributed such mechanisms to detect in-flight modifications to web content, analogous to checksums on program installers [103], and mechanisms to provide whitelists of authorized script code for web programs, analogous to program manifests.

4. *Enforce Policies on Program Behavior.* For both desktop applications and web programs, it can be desirable to enforce policies on program behavior that were not fore-

seen by the program's authors or the developers of the runtime environment. These policies may limit the access rights of a program or modify its output or behavior in particular ways. Many interposition techniques exist to enforce such policies on desktop applications, ranging from system-call interposition to binary rewriting to source-code rewriting. I argue that similar interposition is even more important for web programs, because such programs carry less trust and may each be used in a wider array of scenarios. Complete interposition is difficult but can be achieved in a flexible way by rewriting web content. I have contributed a content interposition layer based on code rewriting in the BrowserShield system, which acts as a reference monitor for web programs [101].

Together, these four principles allow users and publishers to reason about what a web program can and cannot do on the user's machine. Fully supporting them requires changes to both browsers and content, but the changes can be deployed in backward compatible ways with acceptable overhead. By adopting these changes, web browsers and web content can provide a safer platform for deploying web programs.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 provides background about the architecture of today's browsers and web content and the shift toward active programs within the browser. It also provides an outline of the failings of this architecture, and how they arise due to the change in workload. Chapter 3 then surveys related work on robust and secure systems in general, as it relates to improving the robustness of web browsers and web content.

The next several chapters discuss my contributions toward improving the robustness and security of the web, based on the architectural principles presented above. Chapter 4 introduces abstractions for web program instances that are compatible with existing web content, and it discusses the tradeoffs necessary to achieve this compatibility. Chapter 5 shows how to modify the browser's architecture to isolate these web program instances from each other using operating system processes, and it evaluates the resulting robustness and

performance benefits. Chapter 6 shows how publishers can detect unwanted modifications made to their web content before it reaches the client's browser. It discusses a measurement study that shows surprisingly many and diverse in-flight changes to actual web content, with many negative consequences for both publishers and users. Chapter 7 introduces a script whitelist mechanism for explicitly authorizing all script code in a page, to defend against other types of script injection attacks. Finally, Chapter 8 presents the BrowserShield content rewriting system, which interposes on web content to enforce flexible policies, such as preventing exploits of known browser vulnerabilities.

Chapter 9 presents a vision for future web browsers and content, as well as further opportunities for supporting each of the aforementioned architectural principles. Chapter 10 concludes the dissertation.

Chapter 2

BACKGROUND

In this chapter, I introduce concepts and terminology relevant to current web content and web browsers, to provide context for the remainder of the dissertation. I discuss how the original underlying philosophies of the web led to its current document-based architecture, and I show how a shift from a passive workload to an active one is taking place. This shift is not yet reflected in the architectures of most web browsers, and I discuss how this mismatch directly leads to a set of concrete robustness and security problems.

2.1 Web Content Background

Web content is based on the markup of interlinked documents. These documents can contain active code alongside embedded objects, but the design of web content formats reflects a web of pages, not of programs. This is evident in both the terminology and philosophies surrounding web content, despite the current shift toward more active client-side code.

Terminology We can observe the document-centric nature of the web from the concepts and terminology used to define content. Web content fundamentally consists of *pages* written in Hypertext Markup Language (HTML). These pages contain text formatted with markup *tags*, which define styles, document structure, and the links between pages. Tags are also used to embed objects into pages, including child pages known as *frames*, *images* shown within the page, and *Cascading Style Sheet (CSS)* files that affect the way the page is rendered. In this sense, most tags are passive, instructing the browser how to lay out web content visually or navigate between pages.

More active types of objects can be embedded into HTML pages as well, providing interactive content or media. These objects are generally interpreted by browser plug-ins, including interactive environments like Flash, Silverlight, or Java, and media players such

as Quicktime or Windows Media. We will refer to these objects as *plug-in media*, noting that the browser generally relies on third parties to provide the runtime environments for this media.

In general, though, most active code in web content now exists as *script code* embedded within HTML pages. This code is predominantly written in JavaScript that is directly interpreted by the web browser, although VBScript is also supported in Internet Explorer. Script code can be embedded either as self-contained *script tags* in HTML pages, attributes on other tags that define *event handlers*, or in separate *script library* files referenced by script tags. Each of these techniques contributes *script fragments* that interact with the HTML page's contents, as described in Section 2.2. As a result, the contents of web pages can change at runtime, driven by input events or network communication.

Philosophy The basic abstractions described above reflect essentially passive documents, not active programs. Active code was introduced primarily as a way to augment and manipulate pages in minor ways (e.g., form-input validation), and support for script code was added to browsers well after the design of HTML. This late addition allowed script fragments to be included in a wide variety of contexts, with little attention given to distinguishing code from data. Like support for code itself, notions of principals and access control were added as an afterthought to the initial web design. In these ways, programs are not first class citizens on the web.

Web content has also followed a "best effort" philosophy that attempts to minimize the number of errors that users encounter. In other words, web browsers typically attempt to render, display, and execute content even if it contains errors or illegal syntax. This behavior may range from ignoring syntax errors to algorithmically guessing (i.e., "sniffing") document content types if they are not properly identified by the server [20]. This philosophy creates a low barrier to entry for web developers, who do not need to be skilled to publish basic web pages, and it reflects an emphasis of convenience above rigid structure. However, it also introduces significant challenges for security, since it is difficult for web publishers or security professionals to anticipate how various browsers will interpret malformed input.

Another key philosophy of the web is its "openness." Web pages are not only allowed to link to any other page on the web, but also embed images, scripts, and other objects across site boundaries. This makes it easy for publishers to compose content into flexible and useful services, often referred to as "mashups," while also making it more difficult for browsers to distinguish between principals and their intents.

Shift Toward Programs In recent years, there has been a dramatic shift in the amount and complexity of active code included in popular web pages, and in the purposes of such code. Rather than performing simple form validation or providing basic menus, script code is now being used to construct interactive user interfaces for client-side applications. These applications are still delivered to the browser as web pages and use web technologies to communicate with servers, but they represent a new workload for browsers that is very different than the original workload of passive content.

Our measurements show substantial recent increases in the number of pages using JavaScript and the amount of JavaScript per page [99]. The percentage of popular pages with at least 10 KB of JavaScript code was 35% in 2003, and rose to 71% in 2006, with an accelerating rate of change. During the same time period, the average amount of JavaScript code per page increased four-fold from 15.3 KB to 64.9 KB.

This increase in client-side code is being used to deploy programs with the functionality of many desktop applications. It is now common to find web sites that offer email clients, calendars, word processors, spreadsheets, interactive maps, and other applications. These programs implement much of their interactive logic in JavaScript, often with the support of frameworks such as Dojo, Prototype, or the Google Web Toolkit. They also exchange data with servers within the context of a single page, using the `XmlHttpRequest` object in JavaScript. This approach changes the basic page paradigm of the web, allowing users to stay on a single page that changes substantially over time, such as an email client.

As a result of this shift, web-based programs are becoming an increasingly heavy workload for clients, with additional CPU and memory requirements. This has important implications for browser architecture and program robustness and security, as we discuss in Section 2.3.

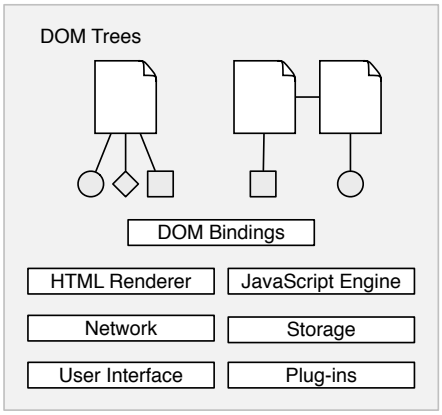

Figure 2.1: All web content and browser components share fate in a single operating system process in monolithic browser architectures, in contrast to the multiprocess architecture we present in Figure 5.1.

2.2 Browser Architecture Background

Most current web browsers were designed for a passive workload of rendering and navigating among pages. The browser's main responsibilities thus include fetching, parsing, and rendering pages, with support for navigation via URLs, bookmarks, links, and going back and forward in a given window's history. Browsers generally parse and interpret script fragments in their entirety as they are encountered, under the assumption that they will have little impact on the overall browsing experience.

In this section, we present terminology for the various browser components that support these tasks, and we discuss how the browser enforces security on code that has been grafted onto web pages. We also show how current monolithic browser architectures, as depicted in Figure 2.1, are well-suited for a workload of passive pages but not one of active programs.

Terminology Most modern web browsers have a variety of components that are necessary to render and interact with current web content. While the exact set of components varies from browser to browser, we introduce a general list below, covering topics that are relevant

to discussions throughout this dissertation. These components necessarily interact with each other within the browser, but the browser's architecture can influence how failures in one component affect others.

Potentially one of the most complex parts of any browser [21], the *content renderer* logic is responsible for parsing and visually laying out HTML pages into a graphical form. This logic is influenced by CSS and images included with the HTML, and it is used to build a *Document Object Model (DOM)* tree to represent the page. DOM trees provide a run-time representation of a page that can be manipulated by script code, which causes the browser to rerender the affected regions of the page.

JavaScript code is then parsed and interpreted by the browser's *script engine*. The engine itself is generally only responsible for running JavaScript code, with little knowledge of browser integration or DOM trees. However, JavaScript code in web pages is primarily useful because of how it can interact with the DOM trees of the pages. The browser's *DOM bindings* connect code running in the script engine with the browser's internal representations of pages, as well as runtime libraries for basic tasks. In this sense, the DOM bindings act much like the system call interface for programs in the browser. Much of the browser's security model is also implemented in the DOM bindings, which act as a reference monitor to restrict which objects in the browser a piece of script code may access.

The browser also provides *storage components* for numerous tasks, although web pages have no direct access to the client's filesystem. As part of the Hypertext Transfer Protocol (HTTP), web pages and other objects can set *cookies* on the client as a limited form of key-value storage. Cookies are generally sent on future HTTP requests to the same web server, to allow pages to be customized based on a user's history or past authentication. Browsers also store many types of objects to improve performance and usability, requiring access to the client's filesystem. These objects include cached web content, browser preferences, stored passwords, and downloaded files, among other types. While many stored objects can only be accessed by the web site they are associated with (e.g., cookies, stored passwords), others are universal (e.g., cached objects).

Fundamentally, web browsers are designed to fetch web content from the network using HTTP. The browser's *network component* must fetch pages and embedded objects, as well as support asynchronous requests for data via the `XmlHttpRequest` object in JavaScript.

The browser maintains its own *user interface*, often referred to as *chrome*, that is separate from the graphical layout of web pages. This UI includes a series of windows and tabs that can display web pages (along with nested pages inside frames), as well as browser-specific menus, toolbars, and status indicators such as address bars. The browser's UI is generally considered more trusted than the UI of web pages, and it is important for displaying security-sensitive information such as the URL of the current page and encryption indicators such as padlock icons for HTTPS (i.e., HTTP over a secure transport layer).

Beyond the core functionality provided in the browser itself, most browsers provide some means for extensibility. First, most web browsers support additional content types via *plug-ins* as described in Section 2.1, including Flash, Silverlight, Java applets, Quicktime, and others. Plug-ins are generally installed as arbitrary client programs that receive content from the browser and render it. The rendered plug-in media is then displayed in the browser, often within a rectangular portion of a web page. In most cases, plug-ins interact with the browser using the NPAPI interface designed for Netscape Navigator [93]. Basic scripting support is also often present, allowing plug-in media to interact with the DOM trees of web pages.

Second, some browsers additionally support *extensions* to their own logic and interfaces, to allow customization of the browser. For example, Firefox supports a wide range of extensions that can modify both web content and the browser's user interface. Because they have greater privileges, extensions require greater trust from the user than web content. Firefox extensions are often written in JavaScript and an XML User Interface Language (XUL), although they can contain arbitrary binary code as well. Extension architectures tend to be browser-specific, when supported at all.

All of these components must cooperate to render web pages and provide a runtime environment for active web content. As we will see, the way that these components are architected can impact how robust this runtime environment is to failures and interference.

Browser Security Model At a high level, web browsers enforce a security model that aims to prevent pages from different web sites from accessing each other. This model is important both for establishing trust and protecting private information. Users can generally trust that the content on a web page is influenced only by the page's publisher and parties that the publisher trusts, not by other pages that happen to be open in the browser. Also, as more web content becomes authenticated and specialized for each user, it is important to prevent untrusted pages from accessing the contents of sensitive pages, such as bank account information.

This security model is embodied in the Same Origin Policy [106], which affects script code, cookies, XmlHttpRequests, and other resources. We discuss the details and subtleties of the Same Origin Policy in Chapter 4.

Monolithic Browser Architectures Until very recently, most web browsers used monolithic architectures that placed all browser components in a single operating system (OS) process, like the one shown in Figure 2.1. Many browsers run entirely within one process (e.g., Firefox, Safari, Opera), at least for a given user profile. Other browsers are capable of running multiple instances in separate processes but are still essentially monolithic, since each process still contains all browser components (e.g., Internet Explorer 7 and earlier, Konqueror).

Monolithic architectures are reasonable for a workload of passive web pages, since expensive computation is usually only expected when a user navigates from one page to another. However, monolithic architectures face challenges as the workload shifts toward mutually untrusted programs that compete for resources as they execute, because they do not provide sufficient isolation between programs. We have shown each of the above browsers to be vulnerable to robustness problems due to their architecture [99]. Chapter 5 discusses how more modular browser architectures can better support the workload of today's web.

2.3 Symptoms of Weak Program Support

As we move toward a workload of active and untrusted programs on the web, both web content and web browsers are stressed in new ways. We find that many of the current

robustness and security problems on the web are symptoms of the mismatch between an architecture focused on documents and a workload of programs.

Interference Interference occurs between independent programs in the browser due to poor isolation between programs in the browser's architecture. This interference manifests itself as crashes that affect multiple programs or pages in the browser, unresponsiveness that can prevent users from interacting with one program while another is computing, and contention for resources across programs [99]. Stronger isolation between browser components and instances of programs would alleviate these symptoms.

Authentication Abuse As web content becomes more specialized for each user, the use of authentication credentials becomes more important to protect sensitive information. Cookies and other credentials are frequently used to ensure that only authorized users can access content or enact server-side state changes. However, browsers are susceptible to the "confused deputy" problem [57], where an adversary can fool the browser into abusing the user's credentials with another site. In the context of the web, these are called cross-site request forgery (CSRF) attacks.

As an 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. Note that this attack does not require placing content on the victim web site; only visiting an adversary-controlled page is required.

Such attacks can succeed because browsers do not distinguish between separate authenticated program instances. Instead, they 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 discussed for several years [132], and many recent concrete vulnerabilities have surfaced, including ones affecting Netflix, Google Mail,

and several open source web applications [40, 127, 72]. Stronger support for identifying the boundaries between program instances could help with this problem, as we discuss in Chapter 4 and Chapter 9.

Script Injection Script injection attacks (also known as cross-site scripting, or XSS) represent a significant threat on the web today, as one of the most reported vulnerabilities of any type [28]. These attacks occur when adversaries are able to place their own script code in another publisher's web program. This code then runs when users load the web program in their browser, allowing adversaries to take control over the program. Often these attacks are used to steal users' cookies or private data, modify the victim web site, or launch distributed denial of service attacks against particular servers [53].

Script injection vulnerabilities are widespread for many reasons. For many years, web publishers were not aware of the threat, so legacy web content often contains many XSS bugs just as legacy desktop programs contained many buffer overflows. Even with knowledge of the problem, however, web publishers face challenges to prevent script injection. Web programs often contain user-provided content such as blogs and discussion boards, which presents many opportunities for adversaries to inject scripts. In web content, code and data can be interspersed in many ways, and browsers attempt to interpret even malformed input. This makes content sanitization a very difficult task, as demonstrated by the success of the Samy and Yamanner worms on MySpace and Yahoo Mail [2, 18].

We thoroughly discuss classes and examples of script injection attacks in Chapter 7, where we show that giving publishers better support for authorizing their program code can eliminate these attacks.

Browser Exploits Finally, browser exploits pose a significant threat to web users, because they typically give adversaries full control of a user's account on the client machine. These exploits attack vulnerabilities and flaws in the browser itself, which tend to be common due to the complexity of the browser's components. Unfortunately, browser complexity is only increasing as web content continues to evolve and gain more interactive features. Additionally, monolithic browser architectures do little to contain exploits when they do occur,

since the exploited components are typically given much higher privileges than they need. To exacerbate the problem, patches to remove vulnerabilities are often not applied immediately, leaving a dangerous time window for adversaries to reverse engineer patches and attack vulnerable users.

More modular browser architectures can mitigate the damage exploits cause [21], in addition to providing the robustness and performance benefits we describe in Chapter 5. Additionally, interposition layers for web content can help enforce policies that protect users during the dangerous time period until browser patches are applied, as we discuss in Chapter 8.

2.4 Summary

Both web content and web browsers were designed for a passive workload. As we move toward a workload of complex, active, and untrusted programs, we encounter many problems as symptoms of an insufficient architecture, ranging from program interference to browser exploits. This dissertation looks at ways to address these problems, starting with a discussion of related work in the next chapter.

Chapter 3

RELATED WORK

This chapter presents a wide range of work related to my thesis and the contributions presented in this dissertation. I first discuss foundational systems research that provides guiding principles for robustly and securely supporting programs. This work shows that isolation and interposition are important concepts that are missing from the current architecture of the web.

Second, I describe more recent efforts to address concrete problems on the web. These efforts often aim at the symptoms without improving support for web programs, leaving the underlying problems unsolved. Adapting principles from operating systems, as proposed in this thesis, can more thoroughly address these issues.

Finally, I present a set of non-goals to place this dissertation in context with other web research. While poor support for programs is an important problem on today's web, it is orthogonal to many security concerns such as phishing.

3.1 Foundational Systems Work

I first discuss foundational work in robustness and security for operating systems and language runtime environments. Research in these areas provides a strong starting point for improving support for programs on the web, through many approaches to isolate programs and interpose on their behavior.

3.1.1 Isolation and Confinement

On any multiprocessing system, isolating independent programs is important to prevent unwanted interference between them. For traditional operating systems, the *process* abstraction has long offered a mechanism for isolating programs in their own virtual environments. Modern OS processes feature separate address spaces and pre-emptive multitasking, along

with tools for resource management and accounting. These attractive properties have led to research that incorporates process-like abstractions into other runtime environments, such as Java [19, 69]. My work finds that browsers are hampered by both a lack of process-like isolation for web programs, as well as a proper program abstraction to place in separate processes. I contribute such a program abstraction, along with browser architectures that directly use OS processes to prevent unwanted interactions between web programs.

Process-based isolation can come at a cost, however, due to the high overhead for interprocess communication (IPC). As a result, many researchers have investigated lightweight fault domains to allow intraprocess isolation [126, 35, 34, 117]. For example, Software Fault Isolation (SFI) [126] provides memory safety by confining untrusted code in sandboxes, while Asbestos [34] offers lightweight isolated contexts within an OS process, using a new event process abstraction. Such approaches often trade slight overhead in the common case for substantially faster cross-domain communication. It is possible that such techniques could be useful to provide program isolation within browsers with less memory or IPC overhead than OS processes, particularly on more resource constrained devices.

In the case of untrusted programs, confining program accesses and behavior becomes even more important. Lampson discusses general concerns for confining untrusted code, including addressing explicit and covert channels for communication [78]. 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 [39]), others are continually discovered (e.g., visited link history [67]). While discovering all such channels is not a goal of this work, providing better isolation between web programs and their associated browser components may help to reveal such interactions.

Closer to the domain of the web, Borenstein [25] and Ousterhout et al. [95] aim to support untrusted code in email messages by confining their behavior with Safe-Tcl. Safe-Tcl restricts the privileges of untrusted programs by running them in a separate interpreter without the full rights of the user. While scripts in email have not become popular, JavaScript presents a similar challenge for running untrusted programs on the web. The JavaScript engine generally provides scripts with little access to the client, but it often runs with the

full rights of the user. New browser architectures can better confine untrusted script code using sandboxes [21].

3.1.2 Interposition and Filtering

Providing security or extensible policies for a system requires the ability to monitor its behavior or filter its inputs. Interposition is thus a fundamental tool in system design: using a reference monitor to allow or deny a program access to particular resources, based on a policy. OS protection mechanisms are perhaps the most basic example, interposing with reference monitors on resources such as the filesystem or network. Saltzer and Schroeder outline basic principles of protection [107], including design principles and mechanisms such as access control lists and capabilities. Unsurprisingly, many of their concepts translate directly to browsers and web content. I argue that interposing between web content and the browser's core components is important for supporting flexible security policies that govern web program behavior.

The need for extensible security policies in systems is generally 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 [45]. 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 [42], and he proposes a delegation architecture to avoid these pitfalls [43]. Nooks [117] takes a different approach, using wrapper code to interpose on calls between the kernel and its extensions. These various approaches may be adapted within web browsers to support extensible security policies for web programs. While this dissertation instead proposes a language rewriting approach in Chapter 8, future interposition layers within the browser itself may be worth pursuing, as discussed in Chapter 9.

At a higher level than the OS, Wallach et al. explore extensible security architectures for Java [128]. 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).

Interposing at the level of program code can also be attractive, as it requires no changes to the underlying platform [36, 38]. For example, Erlingsson and Schneider use code rewriting to support inline reference monitors [36]. These monitors enforce extensible security policies within the code of an untrusted application. Chapter 8 shows how BrowserShield takes the same approach, injecting an interposition layer into web program code to enforce policies at runtime.

Filtering inputs is another important technique for improving program security, as it can be applied without modifying the program's code or behavior. For example, firewalls and proxies are often used to filter network input before it can reach vulnerable systems. More specifically, Shield [130] uses a filtering technique that interposes on network packets bound for an application. Shield filters this traffic for exploits of known vulnerabilities, using vulnerability-specific state machines. Shield's network-based approach shares deployment advantages with code rewriting, as neither requires changes to the underlying platform. This filtering approach is attractive for preventing browser exploits, but it must be able to handle active code in web programs that can generate exploits after arriving in the browser. BrowserShield handles this by rewriting the code to filter exploits at runtime, as discussed in Chapter 8.

3.1.3 Summary

Research in operating systems provides a wealth of approaches for supporting robust and secure program execution. These approaches illustrate how programs can be isolated from each other, confined from the surrounding system, and governed with extensible policies by interposition layers. Each of these techniques can be used to better support programs within the browser.

3.2 Web Robustness and Security

In this section, I cover more direct efforts to address the concrete symptoms of the web's architectural problems. Many research proposals have considered how to better isolate web

content, prevent abuse of authentication credentials, defeat script injection, and address browser exploits. Rather than address the symptoms directly, my work aims to solve the underlying problems by better supporting programs on the web.

3.2.1 Interference Between Web Programs

Existing browsers restrict code access between web programs with the Same Origin Policy, but they face many robustness challenges without additional isolation in their architectures. Researchers have aimed to rearchitect the browser to reduce such interference, but they often do so at the expense of compatibility with existing content.

Specifically, several research proposals have decomposed traditionally monolithic browsers into modular architectures to improve robustness and security. However, most have done so at the cost of compatibility by not identifying existing program boundaries. This makes them difficult to deploy on today's web, where pages might break without warning. For example, the OP browser isolates each web page instance using a set of processes for various browser components [52]. The authors do not discuss communication or DOM interactions between pages of the same web program instance, which poses a challenge for this architecture.

Tahoma isolates web applications on the client, both from each other and from the client's operating system [30]. It uses separate virtual machines and browser instances for each web application, managing them with a "browser operating system" (BOS) outside the browser. The BOS confines web applications based on their self-provided manifest files. Unlike Tahoma, I argue for rearchitecting the browser itself, rather than running existing browsers within isolated containers. By trusting some components of the browser, I can leverage lighter-weight isolation mechanisms than Tahoma, such as OS processes. To protect the underlying OS from exploits of browser flaws, I propose vulnerability filtering techniques similar to Shield [130].

Compared with Tahoma, I also place greater emphasis on backward compatibility. Tahoma requires manifest files that specify confinement policies for each web site, which are not currently available. Tahoma also places a greater burden on the user to manage

and understand the manifest for each web application before approving it. In contrast, my work isolates web programs 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, as discussed in Chapter 4.

SubOS tries to improve browser security with multiple processes, but the authors do not discuss the granularity of the process model nor the interactions between processes [64]. Chapter 4 and Chapter 5 cover these details in greater depth to show how web programs can be identified and isolated.

Internet Explorer 8 introduces a multiprocess architecture that can offer some of the same benefits as those discussed in Chapter 5, while preserving compatibility with existing content [142, 143]. IE8 separates browser and renderer components, and it runs renderers with limited privileges. However, IE8 does not distinguish or isolate *web program instances* from each other. Instead, it ensures that pages at different trust levels are isolated, such as intranet and internet pages. It also places unrelated groups of pages into processes without regard for site boundaries. We provide a separate contribution, identifying sites as web program boundaries and isolating site instances within the browser.

Finally, 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 [14]. Their model describes when objects should be isolated or shared between contexts. While thorough isolation and security policies for browsers are important, I propose them at a deeper level than the scripting engine, since interference can occur in the form of crashes or CPU content as well. OS processes can thus provide more effective isolation, given appropriate program abstractions as presented in Chapter 4.

3.2.2 Authentication Abuse

While the underlying cause for authentication abuse and CSRF attacks is rooted in the browser's lack of support for identifying program instances, these attacks are often considered server-side problems in practice. As a result, many proposed solutions for CSRF require careful development practices on web servers [108]. A typical defense requires fresh

tokens in a page 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 [72].

Client-side defenses for CSRF attacks are attractive, however, as they are closer to the source of the problem and do not require changes to all web servers. For example, Johns and Winter propose RequestRodeo [71], 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 may confuse users on legitimate navigations across sites: the first page view will not be authenticated, but subsequent page views will be. While Chapter 4 and Chapter 9 only discuss potential defenses for CSRF attacks, they do so in a way that recognizes the differences between program instances within the browser.

3.2.3 Script Injection

There are a wide variety of approaches to address script injection and XSS attacks, ranging from eliminating bugs on the server to limiting damage on the client. Most such approaches are fail-open, though, in that they may not eliminate all opportunities for injection attacks. These approaches also do not leverage the publisher's knowledge about which code is authorized to run. We consider a range of these approaches below.

One crude solution to script injection is to simply disable JavaScript in the browser. Most browsers have JavaScript blocking built in as a preference, but this approach becomes less attractive as more and more pages become JavaScript-based programs. The NoScript extension for Firefox partly addresses this concern by selectively allowing JavaScript on a per-site basis [79]. This URL whitelist approach does not protect against malicious scripts that are injected into pages on a URL whitelist, though.

Several researchers have proposed server-side solutions to prevent attackers from injecting JavaScript. These include the use of a gateway on the server side for applying a comprehensive security policy [109], fault injection tools [62], and static analysis or runtime protection for server-side Web components [137, 63], among many others. While

these techniques may be useful to improve the security of server-side components, they are not effective for all types of JavaScript injection attacks. For example, Klein notes that server-oriented solutions often do not address input validation flaws in script code on the client [76]. The above proposals also do not address JavaScript injection via third-party scripts or bookmarklets, indicating an incomplete solution.

A number of recent papers have instead attempted to mitigate JavaScript injection attacks on the client side. Vogt et al. propose browser modifications to support taint analysis, preventing the transfer of sensitive information to a third-party [125]. Such a solution only protects against privacy leaks, and not integrity or availability attacks. Protecting page integrity would require distinguishing scripts that should and should not write to the page, which the browser is unable to do without additional context, such as the whitelists we discuss in Chapter 7. It also is difficult to taint enough data (such as sensitive content found within the page itself) while avoiding false positives. Other client-side approaches include a client proxy to detect "reflected" XSS attacks [65] (i.e., those that only require visiting a malicious crafted URL to succeed) and a client firewall that blocks suspicious connections in the browser [75]. These approaches also fall victim to false positives and false negatives, as they must infer the intent of the Web developer through heuristics. Finally, Content Restrictions [81] have been proposed to prevent scripts from taking certain actions on a page. Content Restrictions are a fail-open solution that allow any actions not specifically restricted, and the difficulty of specifying which actions should be restricted in practice remains unclear.

In contrast, we view script injection as a problem with web program specifications. To address this, publishers should provide enough context to browsers to distinguish authorized scripts from unauthorized ones, and browsers should interpose on script code to run only authorized code. We present such a technique with script whitelists in Chapter 7.

Concurrent work with our script whitelists implementation proposed a similar mechanism, known as Browser Enforced Embedded Policies (BEEP) [70]. BEEP provides a whitelist mechanism analogous to the strawman proposal in Chapter 7, along with a more general framework for enforcing publisher-specified policies. The authors do not consider some of the challenges posed by generated scripts, third-party scripts, dynamic scripts, or

varying sets of scripts in as much detail as our discussion, but we share their overall goal of better capturing publishers' intent by providing additional information to the browser.

3.2.4 Browser Exploits

Browser exploits pose a significant threat to users, because they often provide attackers with the full privileges of the user. Researchers have shown that this threat is widespread by observing a large number of web pages that perform "drive-by downloads," installing malware by exploiting browser vulnerabilities [89, 131, 98].

To defend against these attacks, some researchers propose measurement and testing techniques that identify malicious content before it reaches the browser. Strider HoneyMonkey uses data about offending sites to pursue legal action and blacklists [131], while SpyProxy tests content in a VM for safety before sending it to the client [88].

On the client-side, privilege separation can help limit the damage that browser exploits can cause, much like the safe interpreters proposed in Safe-Tcl [95]. Tahoma sandboxes the entire browser in a VM to contain any damage from malware [30]. Privilege separation can be made possible at a finer granularity using the multiprocess browser architecture discussed in Chapter 5, as demonstrated by the Google Chrome sandbox [21].

I argue that a filtering approach similar to Shield [130] is also valuable for preventing exploits of known browser vulnerabilities in the first place. Chapter 8 discusses how this can be accomplished for active web content using BrowserShield's interposition strategy. While BrowserShield cannot block exploits of vulnerabilities that have not been discovered by browser developers (i.e., zero-day exploits), it can defend known vulnerabilities in browsers until patches are applied.

3.2.5 Summary

Many of the concrete robustness and security problems that plague today's web stem from its poor support for programs. Rather than solely addressing the symptoms, this dissertation aims to adapt systems principles for isolation and interposition to web browsers and web

content, while preserving the desirable properties of the web and compatibility with existing content.

3.3 Related Non-Goals

There remain many web security concerns that do not arise from the mismatch between the architecture of the web and the shift toward active content. Addressing these concerns is not a primary goal of my thesis, but I provide a brief outline of these concerns to place my 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 stealing private information like credit card numbers or passwords. Researchers have investigated phishing practices [33] and proposed mechanisms to help users detect attacks [32, 58]. 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 [32]. Like most phishing defenses (including those offered by Tahoma [30] and Ye and Smith [138]), 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, they are outside the scope of this dissertation. Phishing represents a breakdown in the user's understanding, as opposed to a lack of architectural support for web programs.

Server-side Security Many researchers have considered ways to improve the security of web applications on the server. Some of the many proposals include security gateways for policy enforcement [109], fault injection tools for testing [62], and static or dynamic analyses to eliminate particular threats [63, 137]. For example, Xie and Aiken suggest the use of static analysis of PHP programs to prevent SQL injection attacks [137].

Approaches to improve server-side security are complementary to this dissertation. Clearly, security flaws in web servers present a risk in the form of compromised web programs. However, my thesis focuses on client-side architectural improvements, as well as additions to web

content that can be leveraged by browsers to better support programs once they are delivered to the client.

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 [115]. 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 [4]). Malicious data providers could include arbitrary script code in such files, taking full control of the mashup site.

Recent proposals have sought to improve data exchange between partially trusted principals in the browser. JSONRequest [31] offers a cross-origin RPC primitive that sanitizes JSON data before using it. Subspace [68] 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. MashupOS [61] proposes more principled abstractions in the browser to isolate content while supporting controlled communication and various trust relationships.

It is clear that clean composition models are important for preserving the desirable properties of the web while improving its security. This particular challenge is not covered in this dissertation, as existing proposals offer a reasonable path forward once sufficient isolation is provided to principals on the web.

3.4 Summary

Experience from operating systems can guide us towards principles for supporting programs, such as effective isolation and interposition. As the workload of the web shifts towards programs that run within browsers, it becomes important to adapt these principles to the web. By providing better support for programs in browsers and web content, we can improve web program robustness and security in a fundamental way.

Chapter 4

DEFINING WEB PROGRAMS

We begin discussing the contributions of the dissertation with improved support for existing programs within the web browser. This chapter describes how to identify program boundaries from existing content, and the next chapter uses the resulting abstractions to isolate programs in the browser's architecture. The work in these two chapters appeared at the EuroSys conference in 2009 [102].

4.1 Motivation

The shift toward complex and resource-demanding programs on the web creates challenges for current web browser architectures, which are still designed primarily for rendering basic pages. By not providing sufficient isolation between competing programs within the browser, these architectures admit many types of interference that affect the fault tolerance, memory management, and performance of web-based programs.

These reliability problems are familiar from early PC operating systems. OSes like MS-DOS and MacOS only supported a single address space, allowing programs to interfere with each other. Modern OSes isolate programs from each other with processes to prevent these problems.

Surprisingly, web browsers do not yet have a program abstraction that can be easily isolated. Neither pages nor origins are appropriate isolation boundaries, because some groups of pages, even those from different origins, can interact with each other within the browser. To prevent interference problems in the browser, we face three key challenges: *(1)* finding a way for browsers to identify program boundaries, *(2)* addressing the complications that arise when trying to preserve compatibility with existing web content, and *(3)* rearchitecting the browser to isolate separate programs.

We address the first two of these challenges in this chapter by introducing new abstractions. We show that web content can in fact be divided into separate web programs, and we show that separate instances of these programs can exist within the browser. In particular, we consider the relationships between web objects and the browser components that interact with them, and we define web program instances based on the access control rules and communication channels between pages in the browser. Our goal is to use these abstractions to improve the browser's robustness and performance by isolating web program instances. We find they are also useful for reasoning about the browser's execution and trust models, though we leave secure isolation of these instances via the browser's architecture as a goal for future work.

We show that these divisions between web program instances can be made without losing compatibility with existing content or requiring guidance from the user, although doing so requires compromises. We define a web program as pages from a given *site* (i.e., a collection of origins sharing the same domain name and protocol), and a web program instance as a *site instance* (i.e., pages from a given site that share a communication channel in the browser). Compatibility with existing web content limits how strongly site instances can be isolated, but we find that isolating them can still effectively address many interference problems.

The rest of this chapter is organized as follows. We present ideal program abstractions and show how they can be applied to real web content and browser components in Section 4.2. We discuss complications that arise when incorporating these abstractions into real browsers in Section 4.3, and we show what compromises are necessary to maintain compatibility in Section 4.4.

4.2 Finding Programs in Browsers

We currently lack a precise program abstraction for browsers, leaving most browsers with monolithic architectures in which all content and browser components are combined in one address space and process (see Figure 2.1). Unfortunately, it is challenging to find an appropriate way to define program boundaries in today's browsers. Consider a strawman approach that treats each web page as a separate program, isolating pages from each other.

Web Program	A set of related web pages and their sub-resources that provide a common service.
Web Program Instance	Copies of pages from a web program that are tightly coupled within the browser.
Site	A concrete way to define a web program based on access control between pages.
Browsing Instance	A set of connected windows and frames in the browser that share DOM-based communication channels.
Site Instance	A concrete way to define a web program instance, using the set of same-site pages within a browsing instance.

Table 4.1: Ideal abstractions (bold) and concrete definitions (italic) to identify program instances within browsers.

This approach breaks many real web programs that have multiple communicating pages. For example, mapping sites often allow a parent page to script a separate map page displayed in a frame. Similarly, calendar pop-up windows are frequently used to fill in dates in web forms. Isolating every page would break these interactions.

Origins are another inadequate strawman. Two copies of the same page are often independent and can be isolated from each other, while two pages from partially differing origins can sometimes access each other. Thus, isolating pages based solely on origin would be too coarse in some situations and too fine grained in others.

In this section, we show that it is possible for a browser to identify program boundaries in a way that lets it isolate program instances from each other, while preserving backward compatibility. We provide ideal abstractions to capture the intuition of these boundaries, and we provide concrete definitions that show how programs can be defined with today's content, as summarized in Table 4.1. We then discuss the complications that arise due to the browser's execution and trust models, and what compromises are required to maintain compatibility with existing content.

4.2.1 Ideal Abstractions

We first define two idealized abstractions that represent isolated groups of web objects in the browser.

A *web program* is a set of conceptually related pages and their subresources, as organized by a web publisher. For example, the Gmail web program consists of a parent page, script libraries and images, pages in embedded frames, and optional pop-out windows for chatting and composing messages. The browser combines all of these web objects to form a single coherent program, and it keeps them distinct from the objects in other web programs.

Web programs are easy to understand intuitively but difficult to define precisely. Some research proposals have called for explicit mechanisms that enumerate the components of each web program [104, 30], but in this work we seek to infer web program boundaries from existing content without help from web publishers or users.

Browsers make it possible to visit multiple copies of a web program at the same time. For example, a user may open Gmail in two separate browser windows. In general, these two copies do not communicate with each other in the browser, making them mostly independent. We introduce a second abstraction to capture this: a *web program instance* is a set of pages from a web program that are connected in the browser such that they can manipulate each other's contents. Pages from separate web program instances cannot modify each other's contents directly and can be safely isolated.

4.2.2 Concrete Definitions

The browser can only isolate instances of web programs if it is able to recognize the boundaries between them. To achieve this, we show how to concretely define web programs using *sites*, and how to define web program instances using *site instances*.

Sites The browser already distinguishes between unrelated pages with its access control rules, using the Same Origin Policy [106]. This policy permits some pages to communicate with each other within the browser, so any practical program boundaries must take it into account. To reflect this, we define web programs based on which pages may be able to

access each other. This may imperfectly group together pages from the same origin that are logically unrelated pages to each other, but such relationships are not always evident to the browser. Instead, this approach approximates program boundaries using existing information, and it preserves backward compatibility.

Taking this approach, pages permitted to interact should be grouped together into web programs, while pages that are not should be isolated from each other. We focus on *pages* and not other types of web objects because the Same Origin Policy bases its access control decisions on the origin (i.e., protocol, full host name, and port) of each page. Subresources in a page, such as script libraries, are governed based on the origin of their enclosing page, not their own origin. Note that pages loaded in frames are considered separate from their parent pages, both by the Same Origin Policy and by our definitions.

When pages are allowed to interact, they do so by accessing each other's Document Object Model (DOM) trees. DOM trees provide a representation of each page's contents that can be manipulated by script code. Within the browser, particular components are responsible for supporting this behavior and enforcing access control. The HTML rendering component generates DOM trees from pages. Script code runs within the JavaScript engine and can only interact with these trees via the DOM bindings component, which acts as a reference monitor. The bindings allow a page's code to manipulate and call functions in other pages from the same origin. If the origins do not match, the code is denied access to the page's DOM tree and can only use a small API for managing the page's window or frames. In most cases, this means that pages from different origins can be isolated from each other.

However, origins do not provide perfect program boundaries because a page can change its origin at runtime. That is, a page's code can modify its `document.domain` property, which the DOM bindings use for access control checks. Fortunately, this property can only be modified within a limited range: from subdomains to more general domains (e.g., from `a.b.c.com` to `c.com`) and only up to the "registry-controlled domain name." A registry-controlled domain name is the most general part of the host name before the public suffix (e.g., `.com` or `.co.uk`) [91]. Note that any port specified in a URL (e.g., 8080 in

`http://c.com:8080`) becomes irrelevant for access control checks when a page changes its origin, so pages served from different ports may also access each other.

Since a page's origin can change, we instead define a web program based on the range of origins to which its pages may legally belong. We denote this as a *site*: the combination of a protocol and registry-controlled domain name. Sites provide a concrete realization of the web program abstraction. Pages from the same site may need the ability to access and modify each other within the browser, while pages from different sites can be safely isolated from each other.

While effective and backward compatible, using sites for isolation does represent a somewhat coarse granularity, since this may group together logically independent web programs hosted on the same domain. We discuss the implications of this further in Section 4.4.

Browsing Instances Defining web program instances requires knowing which pages share communication channels inside the browser. A page can access the contents of other pages only if it has references to them, and the browser's DOM bindings only provide such references in certain cases. Thus, we must consider how groups of communicating pages are formed.

DOM-based communication channels actually arise between the containers of pages: windows[1] and frames. We say two such containers are *connected* if the DOM bindings expose references to each other. This occurs if a page opens a second window, where the first page is returned a reference to the second, and the second can access the first via `window.opener`. This also occurs if a child frame is embedded in a parent page, where they become accessible via `window.frames` and `window.parent`. Connections match the lifetime of the container and not the page: if a window or frame is navigated to a different page, it will still have a reference to its parent or opener window.

These persistent connections imply that we can divide the browser's page containers into connected subsets. We define a set of connected windows and frames as a *browsing instance*, which matches the notion of a "unit of related browsing contexts" in the HTML 5 specification [59]. New browsing instances are created each time the user opens a fresh

[1]We consider windows and tabs equivalent for this purpose.

browser window, and they grow each time an existing window creates a new connected window or frame, such as a chat window in Gmail. Because browsing instances are specific to containers and not pages, they may contain pages from multiple web programs (i.e., sites).

In current browsers, named windows are another way to obtain a reference to a page's container. A page can attempt to open a new window with a given name, and if the browser finds an existing window with the same name, it can return a reference to it instead of creating a new window. In theory, this allows any two windows in the browser to become connected, making it difficult to isolate browsing instances. However, this feature is not covered in the HTML 4 specification and its behavior is left to the user agent's discretion in HTML 5. Specifically, browsers are permitted to determine a reasonable scope for window names [59]. To allow web program instances to be isolated, we recommend shifting from a global namespace for windows to a separate namespace for each browsing instance. This change will have little impact on most browsing behavior, and it more intuitively separates unrelated windows in the browser.

Pages do share some other communication channels in the browser, but not for manipulating DOM trees. For example, pages from the same origin may access a common set of cookies using the browser's storage component. Such channels can continue to work even if browsing instances are isolated from each other.

Site Instances We can further subdivide a browsing instance into groups of pages from the same web program, for a concrete realization of a web program instance. Specifically, we define a *site instance* as a set of connected, same-site pages within a browsing instance. Since all pages within a browsing instance are connected, there can be only one site instance per site within a given browsing instance.

We show an example of how web content in the browser can be divided into browsing instances and site instances in Figure 4.1. Each site instance contains pages from a single site but can have subresources from any site. All pages in a browsing instance can reference each other's containers, but only those from the same site share DOM access, as shown by the solid and dashed gray lines between pages.

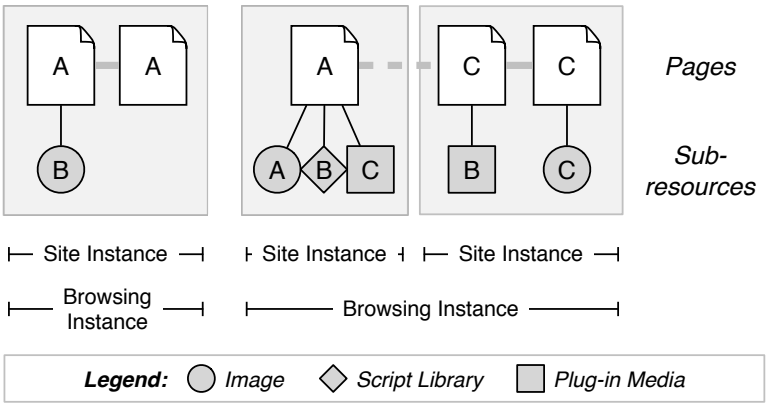

Figure 4.1: We divide web content into independent site instances, based on the sites of pages (e.g., A, B, C) and the connections between page containers (horizontal gray lines).

Note that the same site can have multiple site instances in separate browsing instances (e.g., site A). In this case, pages from different site instances have no references to each other. This means that they have no means to communicate, even though the Same Origin Policy would allow them to.

Similarly, a single browsing instance may contain site instances from different sites (e.g., sites A and C). In this case, pages in different site instances do have references to each other, but the Same Origin Policy does not allow them to access each other's DOM trees.

In both cases, pages in separate site instances are independent of each other. This means that site instances can be isolated from each other in the browser's architecture without disrupting existing web content.

It is also worth noting that the browsing instance and site instance boundaries are orthogonal to the groupings of windows and tabs in the browser. It is unfortunately not possible to visually identify a browsing instance by looking at a set of browser windows, because a single window may contain tabs from different browsing instances. This is because a single browsing instance may have connected tabs in separate windows. This is especially true in browsers that allow tabs to be dragged to different windows, such as Google Chrome.

As a result, the browser's user interface does not reflect the boundaries between web program instances.

Summary With these concrete definitions, we have identified precise boundaries between groups of web objects in the browser. We use sites as a proxy for web programs, based on whether documents can access each other. We use site instances as a proxy for web program instances, based on whether those documents' containers (e.g., windows) can access each other. Each of these definitions can act as an architectural boundary in the browser, with various strengths that we discuss in Section 5.4.

4.3 Practical Complications

Some aspects of the browser's runtime environment present constraints and complications that affect how site instances can be implemented and isolated from each other. For example, isolating site instances for security is desirable, but this is difficult to achieve without changing the behavior of web content. In this section, we show how the execution model for web programs suggests using a single thread and address space for all code within a site instance, and how the browser's trust model prevents site instances from being perfectly isolated.

4.3.1 Execution Model

Web-program execution primarily involves two tasks: page rendering and script execution. These tasks are interleaved as an HTML page is parsed, and both tasks can result in DOM-tree mutations. The browser presents a single-threaded view of these events to avoid concurrent modifications to the DOM. This is partly necessary because JavaScript is a single-threaded language with no concurrency primitives, and it suggests that a given page's rendering and script execution should be implemented on a single thread of execution.

Moreover, communicating pages within a site instance have full access to each other's DOM tree values and script functions. Because there are effectively no boundaries between these pages, it is important that no race conditions occur during their code's execution. We find that such race conditions are unfortunately possible in the Internet Explorer and Opera

browsers, which both execute different windows in different threads, even if their pages can access each other.

To prevent concurrent DOM modifications and provide a simple memory model for shared DOM trees, we recommend that browser architectures place all web objects for a site instance, as well as their supporting browser components, into a single address space with a single thread of execution.

4.3.2 Trust Model

The trust model of the browser reveals the extent to which site instances can be isolated, because it defines how web objects can interact. We now consider how credentials can specialize a web program for a user and how the browser tries to prevent the resulting confidential information from flowing into other web programs. We find that site instance boundaries alone are not sufficient for enforcing the browser's trust model, so we cannot yet rely on them for security.

Credentials Many web programs, such as mail or banking sites, are only useful when specialized for individual users. Browsers provide several credential mechanisms (e.g., cookies, client certificates) that allow sites to identify users on each request. As a result, these web programs may contain confidential information that should not leak to other web programs. We refer to the combination of a web program and the user's credentials as a *web principal*.

In most cases, sites give credentials to the browser after the user has authenticated (e.g., using a password). The main challenge is that browsers attach credentials to each request based solely on the destination of the request and not which web program instance is making the request. Thus, sites are usually unable to distinguish between a user's site instances on the server side. As a result, site instances from the same site can only be partly isolated in the browser; they have independent DOM trees but share credentials. This property can lead to user confusion (e.g., in cases where the user attempts to purchase separate items from the same site in different browser windows), as well as cross-site request forgery attacks against other site instances (e.g., when a malicious page makes a cross-site request that carries the user's credentials).

Information Flow Given that both web programs and other resources on the user's computer may contain confidential information, the browser's trust model must ensure that this information does not leak into the wrong hands. In terms of information flow, the browser must prevent any confidential information from flowing into an unauthorized site instance, because it places no restrictions on what information can flow out.

Specifically, a site instance can send information to any site, because the browser permits it to embed subresources from any site. A site instance can simply encode information in the HTTP requests for subresources, either as URL parameters, POST data, or in the URL path itself.

As a result, today's browsers must already take steps to prevent information from other web principals or from the *user principal* (i.e., the resources outside the browser to which the user has access) from flowing into a web principal.

To protect the user principal, the browser abstracts away the user's resources and denies access to local files and devices. These restrictions reflect the fact that web programs are not granted the same level of trust as installed applications and should not have the same privileges.

The browser faces a harder challenge to protect web principals. This is because site instances are free to embed subresources from any site. These subresources carry credentials based on their own origin, not the origin of their enclosing page (unlike the access control rules described in Section 4.2.2). Such subresources may therefore contain information to which the site instance should not have access. For example, if a user logs into a bank in one site instance and visits an untrusted page in another, the untrusted page can request authenticated objects from the bank's site. This is an instance of the confused deputy problem [57], and it can lead to cross-site request forgery attacks for sites that do not defend against it [132].

In practice, today's browsers try to keep subresources *opaque* to their enclosing page, to prevent such information from flowing into the site instance. For example, script libraries can be executed but not directly read by a page's code, and a page's code cannot access the contents of binary objects like images. Pages also cannot transmit subresources back to their own site. In this way, code running within a site instance cannot directly read

information from another site that is embedded within the page, even though it is displayed seamlessly to the user.

The consequence of this approach is that the browser must rely on subtle logic in its components, such as the script engine and DOM bindings, to maintain its trust model within each site instance. Providing strong walls between site instances is not sufficient to enforce this trust model, because each site instance must be restricted from accessing the cross-site subresources within its own pages.

4.4 Compromises for Compatibility

The practical constraints discussed above demonstrate the challenges for isolating web program instances in the browser in a compatible way. We conclude Chapter 4 by discussing the main compromises needed to maintain backward compatibility, which is a critical goal for web browsers that seek substantial adoption by real users. These compromises permit compatibility but hold us back from perfectly isolating web program instances and their confidential information.

Coarse Granularity Our use of sites to define web programs represents one compromise. A site like `google.com` may host many logically separate applications (e.g., search, mail, etc.), perhaps even on different subdomains. However, the architecture can identify these only as part of the `google.com` site. Any two pages from the same site instance could legally access each other, so we cannot isolate them a priori. Fortunately, this coarse granularity of web programs is offset by the fact that separate instances can be isolated. Thus, mail and map programs from the same site need not share fate, if the user opens them in separate browsing instances.

Imperfect Isolation A second compromise is that site instances cannot be fully isolated, for two reasons. First, a small subset of the DOM API is not subject to the Same Origin Policy, so site instances within a browsing instance are not entirely independent. This API is shown in Table 4.2 and mainly allows a page to access properties of connected windows and frames, without reaching into the contents of their pages. Fortunately, it can

Writable Properties	location, location.href
Readable Properties	closed, frames, history, length, opener, self, top, window
Callable Methods	blur, close, focus, postMessage, toString, history.back, history,forward, history.go, location.reload, location.replace, location.assign

Table 4.2: Properties and methods of the `window` object that can be accessed by cross-site pages in the same browsing instance.

be implemented without interacting with the DOM trees of different site instances, allowing them to be placed in different address spaces. Second, multiple site instances from the same site share credentials and other storage, such as cached objects. As a result, the browser's storage component can at best be partitioned among sites, not site instances.

Subresource Credentials Ideally, the information contained in web principals could be strongly isolated, but the browser's policies for subresources and credentials require another compromise. For example, the architecture could prevent one web principal from using another web principal's credentials when requesting subresources. This approach would reduce the importance of keeping subresources within a site instance opaque, since they would not contain another web principal's sensitive information. Unfortunately, since credentials are currently included based only on the destination of a request, this may break some mashup sites that depend on embedding authenticated subresources from other sites. Thus, we let the browser rely on components like the DOM bindings to keep these subresources opaque.

Overall, we find that compatibility does tie our hands, but we can still provide architectural improvements to browsers by making reasonable compromises. We choose a coarse

granularity for web programs, we require shared credentials and a small API between site instances, and we limit our ambitions for securely isolating web principals.

4.5 Summary

Although web browsers do not currently have a program abstraction, we have shown that it is possible to identify independent web program instances in a backward compatible way, without additional information from web publishers. We can achieve this with the site instance abstraction, making reasonable compromises to preserve compatibility with existing content. This abstraction can be used to robustly isolate web program instances from each other, although providing secure boundaries between web program instances in the browser's architecture remains a topic for future work.

Chapter 5

ISOLATING WEB PROGRAMS IN BROWSERS

In the previous chapter, we showed how the browser can be divided into independent instances of web programs, without sacrificing compatibility with existing web sites. In this chapter, we isolate these web program instances with a new browser architecture that can prevent many types of unwanted interference between programs.

5.1 Motivation

Most current browsers employ the monolithic architecture shown in Figure 2.1, combining all web program instances and browser components into a single OS process. This leads to interference between web program instances in the form of fault tolerance, accountability, memory management, and performance.

To better prevent interference between web program instances, we present a browser architecture that uses OS processes as an isolation mechanism. The architecture dedicates one process to each program instance and the browser components required to support it, while the remaining browser components run safely in a separate process. These web program processes leverage support from the underlying OS to reduce the impact of failures, isolate memory management, and improve performance. As a result, the browser becomes a more robust platform for running active code from the web. Web program processes can also be sandboxed to help enforce some aspects of the browser's trust model, as we discuss in a related technical report that is outside the contributions of this dissertation [21].

Using OS processes as an isolation mechanism does represent a tradeoff. Giving each web program instance its own process eliminates some sharing opportunities between instances of the browser's components and it introduces overhead for IPC and context switching. As we discuss in Section 3.1.1, there may be alternatives for lower-cost isolation of program instances. However, we have found OS processes to be an appropriate mechanism with

benefits that outweigh the drawbacks, including ease of use, good fault isolation, and built-in OS tools that are sufficient for resource management.

Google has implemented the browser architecture described above in the open source Chromium web browser, with some caveats. The Google Chrome browser is based on the Chromium source code; we will refer to both browsers as Chromium. While at Google, I helped add support for site-instance isolation to Chromium's multiprocess architecture, allowing each site instance to run in a separate process. While the current implementation does not always provide strict isolation of pages from different sites, we argue that it achieves most of the potential benefits and that strict isolation is feasible.

We evaluate the improvements this multiprocess architecture provides for Chromium's robustness and performance. We find that it provides a more robust and responsive platform for web programs than monolithic browsers, with acceptable overhead and compatibility with existing web content.

In Section 5.2, we present a multiprocess architecture that can isolate these abstractions. We describe Chromium's implementation of the architecture in Section 5.3 and how it can address concrete robustness problems in Section 5.4. Finally, we evaluate the benefits and costs of the architecture in Section 5.5 and conclude in Section 5.6.

5.2 A Multiprocess Browser Architecture

We first show how to divide the browser's components among different OS processes to isolate web program instances from each other. We choose OS processes as an isolation mechanism to leverage their independent address spaces and OS-supported scheduling and parallelism. We define three types of processes to isolate the browser's components and explain why certain components belong in each type. Our architecture is shown in Figure 5.1.

Rendering Engine We create a rendering engine process for each instance of a web program. This process contains the components that parse, render, and execute web programs. The HTML rendering logic belongs here, as it has no dependencies between instances and contains complex code that handles untrusted input. The DOM trees it produces similarly belong in this process; this keeps the pages of a site instance together but those of differ-

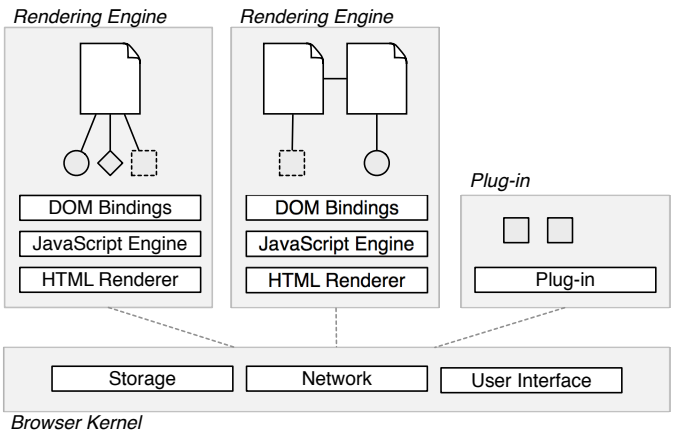

Figure 5.1: A multiprocess architecture can isolate web program instances and their supporting browser components, from each other and from the rest of the browser. Plug-ins can also be isolated from the rest of the browser. Gray boxes indicate processes.

ent instances isolated. The JavaScript engine and the DOM bindings connecting it to the DOM trees are also placed here. These components share a single thread for rendering and script execution within a web program instance, but distinct instances can run on separate threads in separate processes. As we discuss in a separate report on Chromium's security architecture [21], these processes can also be sandboxed to remove unnecessary privileges like filesystem access.

Browser Kernel We place most of the remaining browser components in a single process known as the browser kernel. All storage functionality (e.g., cookies, cache, history) is shared among instances of a web program and requires filesystem privileges, so it is placed in this process. The network stack is also privileged and can be shared among instances. Finally, the logic for managing the browser's user interface can be placed in this process, as it is independent of the execution of web program instances.

Plug-ins We can introduce a third process type for running browser plug-ins like Adobe Flash, since plug-ins are effectively black boxes to the rest of the browser. Plug-in compo-

nents could be loaded into each rendering engine process that needs them, but the Google Chrome team points out that this causes two problems [50]. First, many plug-ins require privileges that are stripped from sandboxed rendering engine processes, such as filesystem access. Second, running plug-ins in the rendering engine process would let a third party plug-in cause crashes in web program instances. Placing plug-ins in a separate process avoids these problems, preserving compatibility with existing plug-ins but losing potential security benefits of sandboxing them. Also, creating one plug-in process for each type of active plug-in avoids overhead from running multiple instances of a given plug-in component. This represents another compromise for compatibility, and an opportunity for improvement for future browser plug-ins, as we discuss in Chapter 9.

Using this architecture, instances of web programs can be safely run in separate processes from the rest of the browser without losing compatibility with existing content.

5.3 Implementation

In an earlier technical report, I describe the implementation of a similar multiprocess architecture based on the Konqueror web browser [99]. The Google Chrome team has implemented the architecture described here in the Chromium web browser. My work on Chromium's implementation helps to isolate site instances using separate rendering engine processes. Chromium also supports several other compatibility-preserving models for assigning web content to processes, based on the definitions from Section 4.2.2. These models are summarized in Table 5.1. We discuss them below to reveal their implementation requirements and properties, and we then present caveats in Chromium's current implementation. Users can choose between the models with command line arguments, as documented online [51].

- **Monolithic:** Chromium is designed to use separate processes for the browser kernel, rendering engines, and plug-ins, but it is capable of loading each of these components in a single process. This model acts as a baseline for comparisons, allowing users to evaluate differences between browser architectures without comparing implementations of different browsers.

Model	*Creates Process*
Monolithic	At browser startup
Process-Per-Browsing-Instance	For each user-created window
Process-Per-Site-Instance	For each user-created window, or when visiting a new site in an existing window
Process-Per-Site	When visiting a new site

Table 5.1: Summary of compatible process models for browser architectures, showing when each model creates new OS processes.

- **Process-Per-Browsing-Instance:** Chromium's simplest multiprocess model creates a separate rendering engine process for each browsing instance. Implementing this model requires mapping each group of connected browser tabs to a rendering engine process. The browser kernel must display the process's output and forward user interactions to it, communicating via IPC. Frames can always be handled in the same process as their parent tab, since they necessarily share a browsing instance.

  A browsing instance, and thus a rendering engine process, is created when the user opens a new tab, and it grows when a page within the browsing instance creates a new connected tab. Chromium maintains the orthogonality between browsing instances and the visual groupings of tabs, allowing connected tabs to be dragged to different windows.

  While this model is simple and provides isolation between pages in unconnected windows, it makes no effort to isolate content from different sites. For example, if the user navigates one tab of a browsing instance to a different web site than the other tabs, two unrelated web program instances will end up sharing the rendering engine

process. This process model thus provides some robustness but does not isolate web program instances.

- **Process-Per-Site-Instance:** The above process model can be refined to create a separate renderer process for each site instance, providing meaningful fate sharing and isolation for each web program instance. This model provides the best isolation benefits and is used in Chromium by default.

  To implement this, Chromium supports switching a tab from one rendering process to another if the user navigates it to a different site. Ideally, Chromium would strictly isolate site instances by also rendering frame pages in processes determined by their site. This is not yet supported, as we discuss in the caveats below.

- **Process-Per-Site:** As a final but less robust model, Chromium can also consolidate all site instances from a site and simply isolate web programs, rather than web program instances. This model requires extra implementation effort to ensure that all pages from the same site are rendered by the same rendering engine process, and it provides less robustness by grouping more pages together. However, it still isolates sites from each other, and it may be useful in low resource environments where the overhead of additional processes is problematic, because it creates fewer rendering engine processes than process-per-site-instance.

Caveats Chromium's current implementation does not yet support strict site isolation in the latter two process models. There are several scenarios in which a single rendering engine process may host pages from multiple sites, although this could be changed in future versions.

This is partly because Chromium does not yet support the API permitted between windows showing different sites (from Table 4.2), if those windows are rendered by different processes. We argue that supporting this API in Chromium is feasible, but until the support is implemented, Chromium should avoid breaking sites that may depend on the API. It does so by limiting the cases in which a tab is switched to a new rendering engine process

during navigations, so that the API calls can complete within the same process. Specifically, it does not swap a tab's process for navigations initiated within a page, such as clicking links, submitting forms, or script redirects. Chromium only swaps processes on navigations via the browser kernel, such as using the location bar or bookmarks. In these cases, the user has expressed a more explicit intent to move away from the current page, so severing script connections between it and other pages in the same browsing instance is acceptable.

Similarly, Chromium does not yet render frames in a separate process from their parents. This would be problematic if secure isolation of site instances were a goal, but it is sufficient for achieving robustness goals.

Chromium also places a limit on the number of renderer processes it will create (usually 20), to avoid imposing too much overhead on the user's machine. This limit may be raised in future versions of the browser if RAM is plentiful. When the browser does reach this limit, existing rendering engine processes are re-used for new site instances.

Future versions could resolve these caveats to provide strict isolation of pages from different sites. Nonetheless, the current implementation can still provide substantial robustness benefits in the common case.

5.4 Robustness Benefits

A multiprocess browser architecture can address a number of robustness problems that afflict current browsers, including fault tolerance, accountability, memory management, and performance issues. We discuss how the architecture is relevant for these problems, and how Chromium further leverages it for certain security benefits.

Fault Tolerance Faults in browser components or plug-ins are unfortunately common in these complex and evolving codebases, and they typically lead to a crash in the process where they occur. The impact of such a crash depends on both the browser architecture and which component is responsible.

In monolithic browsers, a crash in any component or plug-in will lead to the loss of the entire browser. To mitigate this, some browsers have added session restore features to reload lost pages on startup. However, users may still lose valuable data, such as purchase receipts

or in-memory JavaScript state. Also, web programs that cause deterministic crashes may prevent such browsers from restoring any part of a session, since the browser will crash on each restore attempt. We have encountered this situation in practice.

In a multiprocess architecture, many crashes can be confined to have much lower impact. Crashes in a rendering engine component, such as HTML renderers or JavaScript engines, will cause the loss of only one rendering engine process, leaving the rest of the browser and other web program instances usable. Depending on the process model in use, this process may include pages from a single site instance, site, or browsing instance. Similarly, plug-in components can be isolated to prevent plug-in crashes from taking the rest of the browser with them.

Note that crashes in browser kernel components, such as storage or the user interface, will still lead to the loss of the entire browser. However, in a related technical report [21], we find that over the past year, the rendering engine components of popular browsers contained more complexity and had twice as many vulnerabilities as the browser kernel components. Thus, tolerating failures in the rendering engine will likely provide much of the potential value.

Accountability As web pages evolve into programs, their demands for CPU, memory, and network resources grow. Thus, resource accounting within the browser becomes important for locating misbehaving programs that cause poor performance. In a monolithic browser, the user can typically only see resource usage for the entire browser, leaving him to guess which web program instances might be responsible. In multiprocess architectures, resource usage for each process is available from the OS, allowing users to accurately diagnose which web program instances are to blame for unreasonable resource usage.

Memory Management Web program instances can also interfere with each other in monolithic browsers in terms of memory management. Browsers tend to be much longer-lived than web program instances, so a monolithic browser must use a single address space to execute many independent programs over its lifetime. Heavy workloads and memory leaks in these web program instances or in their supporting components can result in a large and

fragmented heap. This can degrade performance and require large amounts of memory, and it can lead to large challenges for browser developers that seek to reduce memory requirements [96]. In contrast, placing each web program instance and the components supporting it in a new process provides it a fresh address space, which can be disposed when the process exits. This simplifies memory reclamation and isolates the memory demands of web program instances.

Performance Performance is another concern as the code in web programs becomes more resource demanding. In monolithic browsers, a web program instance can interfere with the performance of both unrelated web programs and the browser itself. This occurs when separate instances are forced to compete for CPU time on a single thread, and also when the browser's UI thread can be blocked by web program actions, such as synchronous `XmlHttpRequest`s. This is most evident in terms of responsiveness, where a large or misbehaving program instance can cause user-perceived delays in other web programs or the browser's UI. By isolating instances, a multiprocess architecture can delegate many performance and scheduling issues to the OS. This allows web program instances to run in parallel, improving responsiveness and taking advantage of multiple cores when available.

Security The browser's process architecture can also be used to help enforce security restrictions, although this is not the main focus of this chapter. Monolithic architectures rely entirely on the logic in browser components, such as the DOM bindings, to enforce the trust model we discuss in Section 4.3.2. However, bugs may allow malicious web programs to bypass this logic, letting attackers install malware, steal files, or access other web principals.

In a recent report, we have shown how Chromium leverages its multiprocess architecture to help enforce isolation between the user principal and web principals [21]. Chromium uses sandboxes that restrict rendering engine processes from accessing the filesystem and other resources, which can help protect the user principal if a rendering engine is compromised.

It is also desirable to isolate web principals architecturally, to help prevent exploited rendering engines from stealing or modifying information in other web principals. As we discuss in Section 4.4, compatibility poses challenges for this because site instances can

embed subresources from any site with the user's credentials. The architecture provides little benefit if confidential subresources are loaded by a malicious page in a compromised rendering engine. Thus, we leave a study of secure isolation of web principals to future work.

5.5 Evaluation

In this section, we evaluate the benefits and costs of moving from a monolithic to a multiprocess architecture in the Chromium browser. We first demonstrate how this change improves robustness in terms of fault tolerance, accountability, and memory management. We then ask how the architecture impacts Chromium's performance, finding many advantages despite a small penalty for process creation. Finally, we quantify the memory overhead for the architecture and discuss how Chromium satisfies backward compatibility.

Note that Chromium makes it possible to compare browser architectures directly by selecting between the monolithic mode and the process-per-site-instance mode using command line flags. We take this approach rather than comparing different web browsers to avoid capturing irrelevant implementation differences in our results.

Our experiments are conducted using Chromium 0.3.154 on a dual core 2.8 GHz Pentium D computer running Windows XP SP3. We use multicore chips because they are becoming increasingly prevalent and can exploit the parallelism between independent web programs. All of the network requests in our tests were played back from disk from a previously recorded browsing session to avoid network variance in the results.

5.5.1 Is multiprocess more robust?

We first use real examples to demonstrate that the multiprocess architecture can improve Chromium's robustness.

Fault Tolerance We verified that crashes in certain browser components can be isolated in multiprocess architectures. Chromium supports a built-in "`about:crash`" URL that simulates a fault in a rendering engine component. In monolithic mode, this causes the loss of the entire browser. In Chromium's multiprocess architecture, only a single rendering

engine process is lost. Any pages rendered by the process are simply replaced with an error message while the rest of the browser continues to function. Similarly, terminating a plug-in process in Chromium causes all plug-in instances (e.g., all Flash movies) to disappear, but the pages embedding them continue to operate.

Accountability Chromium's multiprocess architecture allows it to provide a meaningful Task Manager displaying the CPU, memory, and network usage for each browser process. This helps users diagnose problematic site instances. In one real world scenario, we found that browsing two photo galleries caused the browser's memory use to grow considerably. In monolithic mode, Chromium's Task Manager could only report 140 MB of memory use for the entire browser, but in process-per-site-instance mode we could see that one gallery accounted for 104 MB and the other for 22 MB. This gave us enough information to close the problematic page.

Memory Management Many real web programs present heavy memory demands today, and leaks can occur in both web pages and browser components. For example, the image gallery mentioned above presents a heavy workload, while a recent bug in Chromium caused a rapid memory leak when users interacted in certain ways with news.gnome.org [48]. In both cases, Chromium's multiprocess architecture quickly reclaims the memory used by the offending site instance when its windows are closed, simply by discarding the process's address space. In contrast, the monolithic architecture continues to hold the allocated memory after offending windows are closed, only slowly releasing it over time as the browser's process attempts to free unused resources at a finer granularity.

5.5.2 Is multiprocess faster?

A multiprocess architecture can impact the performance of the browser because it allows site instances to run in parallel. We measure three aspects of the architecture's impact on Chromium's performance: the responsiveness of site instances, the speedups when multiple instances compute concurrently, and the latency introduced by process creation.

Workload	Monolithic	Multiprocess
Alone	9 (14)	4 (6)
With Top 5 Pages	1408 (2536)	6 (7)
With Gmail	3307 (5590)	6 (7)

Table 5.2: Average (and worst case) delay in milliseconds observed when interacting with a loaded page while other pages are loading or running.

Responsiveness To test the browser's responsiveness, we look at the user-perceived latency for interacting with a page while other activity is occurring in the browser. Specifically, we inserted code into the browser kernel to measure the time interval between right-clicking on a page and the corresponding display of the context menu. This processing is handled by the same thread as rendering and script execution, so it can encounter contention as other pages load or compute. We perform a rapid automated series of 5 right clicks (500 ms apart) on a loaded blank page while other workloads are in progress. These workloads include loading the 5 most popular web pages according to Alexa [9], and loading an authenticated Gmail session. We report both the average and worst case latencies in Table 5.2, comparing Chromium's monolithic and multiprocess architectures.

The monolithic architecture can encounter significant delays while other pages are loading or computing. Such "thread capture" may occur if the rendering and script computations performed by other pages prevent the active page from responding to events. The multiprocess architecture completely masks these delays, keeping the focused page responsive despite background activity.

Speedup In some cases, multiple web program instances have enough pending work that considerable speedups are possible by parallelizing it, especially on today's multicore computers. Speedups can occur when several pages are loading at once, or when pages in the background perform script operations while the user interacts with another page.

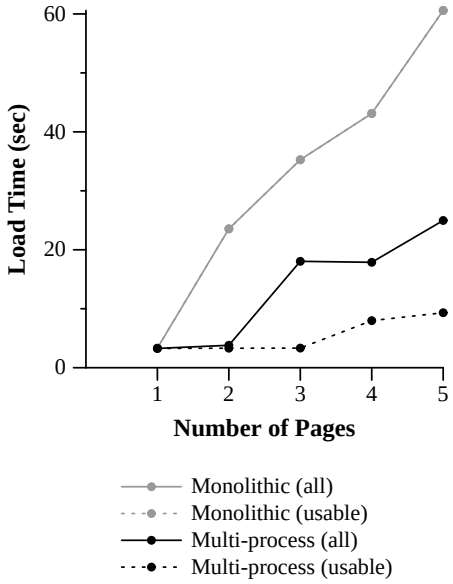

Figure 5.2: Solid lines show the total load time for restoring a session of realistic pages. Dotted lines show the time until at least one of the restored pages is usable. Both lines for the monolithic architecture overlap.

We consider the case of session restore, in which the browser starts up and immediately loads multiple pages from a previous session. Our test loads sessions containing between 1 and 5 Google Maps pages in different tabs, and we report the time taken for all pages to finish loading, based on measurements recorded within the browser. We compare results for Chromium's monolithic and multiprocess architectures.

We also report the time until at least one of the pages becomes usable (i.e., responds to input events). In the monolithic case, this occurs when the last page finishes loading, because each loading page interferes with the others. For multiprocess, we found we could interact with the first page as soon as it finished.

Our results are shown in Figure 5.2. On a dual core computer, the multiprocess architecture can cut the time to fully load a session to less than half, and it allows users to begin interacting with pages substantially sooner.

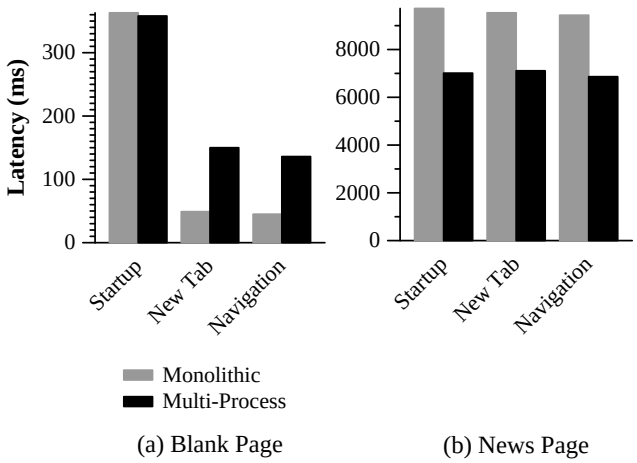

Figure 5.3: Latency for browser startup, tab creation, and a cross-site navigation, for both (a) a blank page and (b) a popular news page. Each task creates a process in the multiprocess architecture.

Latency Although a multiprocess browser architecture brings substantial benefits for responsiveness and speedups, it can also impose a latency cost for process creation. We use code in the browser to measure this latency for Chromium's monolithic and multiprocess architectures in three common scenarios, each of which creates a process: starting the browser with a page, opening a page in a new tab, and navigating to a different site in a given tab. We present results for both a blank page and a popular news site, to put the additional latency in context.

Figure 5.3 shows the results. Startup times can actually improve for both pages in the multiprocess architecture, because the browser kernel and rendering engine can initialize in parallel. For the blank page, new tab creation and cross-site navigations incur about a 100 ms penalty in the multiprocess browser, due to process creation. As seen in Figure 5.3 (b), this is more than offset by speedups offered by the multiprocess architecture. Such speedups are possible because the browser kernel must also perform computations to render a page, such as cache and network request management, which can run in parallel with the rendering engine.

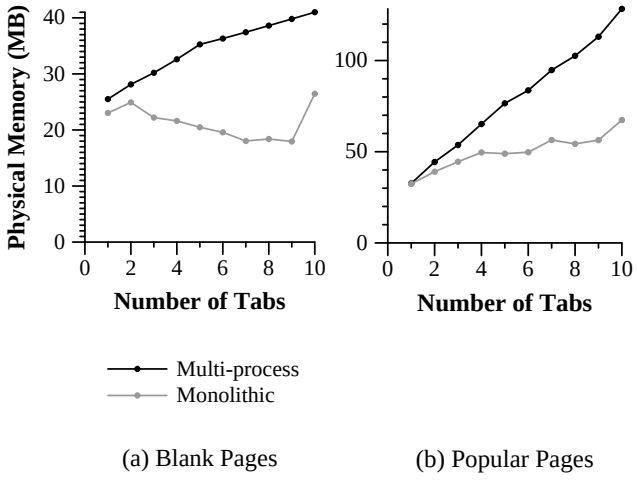

Figure 5.4: Memory overhead for loading additional blank or realistic pages in new tabs.

5.5.3 What is the overhead for multiprocess?

Moving to a multiprocess browser architecture does incur a cost in terms of memory overhead. Each rendering engine process has its own copies of a set of browser components, causing the footprint of a new site instance to be larger than simply the footprint of the pages it contains.

Measuring this overhead poses a challenge because multiple browser processes may share large amounts of memory [49]. We attempt to avoid double-counting this memory in our measurements. Specifically, the physical memory we report includes the private bytes and shareable (but not actually shared) bytes allocated by all browser processes, plus an approximation of the amount of shared memory. This approximation is the sum of each browser process's shared bytes divided by the number of processes.

Using this metric, we report the physical memory sizes of the monolithic and multiprocess architectures after loading a number of pages in separate tabs. We consider the footprints of both blank pages and a set of the 10 most popular pages, as reported by Alexa.

Our results are shown in Figure 5.4. The blank page tests for the monolithic architecture show some variability, due to unrelated memory reclamations after browser startup.

We confirmed this hypothesis by introducing page activity before running the tests, which eliminated the variability. We do not include this initial activity in our actual tests because it is biased against monolithic mode, which cannot reclaim the memory as easily as multiprocess mode.

As expected, the multiprocess architecture requires more memory per site instance than the monolithic architecture. For blank pages, the average site instance footprint rises from 0.38 MB to 1.7 MB between architectures. For popular pages, it rises from 3.9 MB to 10.6 MB. The greater disparity in this case is likely due to caches and heaps that must be instantiated in each rendering engine as pages execute.

These footprints may vary widely in practice, depending on web program complexity. Many typical machines, though, have upwards of 1 GB of physical memory to dedicate to tabs, supporting a large number of average pages in either architecture: 96 in multiprocess mode compared to 263 in monolithic mode. Also note that Chromium currently creates at most 20 rendering engine processes and then reuses existing processes, as discussed in Section 5.3.

5.5.4 Does multiprocess remain compatible?

The multiprocess architecture is designed to be compatible with existing web sites, as discussed in Chapter 4. Chromium's implementation also uses the WebKit rendering engine that is shared by Safari and other browsers, to avoid introducing a new engine for web publishers to test against. We are aware of no compatibility bugs for which the architecture, not the implementation, is responsible.

Nonetheless, both the architecture and Chromium's implementation exhibit some minor differences from monolithic browsers that may be observed in uncommon circumstances. Because of the shift from a global namespace for window names to a per-browsing-instance namespace, it is possible to have multiple windows with the same name. For example, the Pandora music site allows users to open a small player window. If a player window is already open, attempting to open another player from a second browsing instance will simply refresh the current player in monolithic browsers, but it will open a second player

window in multiprocess browsers. This is arguably a logical choice, as the user may consider the two browsing instances independent.

Chromium's implementation also does not yet support cross-process calls for the small JavaScript API that is permitted between page containers from different origins. As discussed in Section 5.3, Chromium attempts to keep such pages in the same process when they might try to communicate with this API. In practice, this is unlikely to affect many users, since most interwindow communication is likely to occur between pages from the same site.

5.6 Summary

The reliability problems in today's browsers are symptoms of an inadequate browser architecture, designed for a different workload than browsers currently face. We have shown that a multiprocess architecture can address these reliability problems effectively by isolating instances of web programs and the browser components that support them. To do so, we have identified program abstractions within the browser while preserving compatibility with existing web content. These abstractions are useful not just for preventing interference between independent groups of web objects, but also for reasoning about the browser and its trust model. We hope that they can push forward discussions of future browser architectures and types of web content.

Our evaluation shows that Google's Chromium browser effectively implements such a multiprocess architecture. It can be downloaded as Google Chrome from `http://www.google.com/chrome/`, and its source code is available at `http://dev.chromium.org/`.

Chapter 6

DETECTING WEB PROGRAM ALTERATIONS

Even with strong boundaries between web programs on the client, it remains important to understand what code is running within a given web program. This chapter and the next discuss efforts to address this problem by detecting changes to program code and by explicitly authorizing code. The work in this chapter appeared at the NSDI conference in 2008 [103].

6.1 Motivation

Most web pages and web programs are sent from servers to clients using HTTP. It is well-known that Internet Service Providers (ISPs) or other parties between the server and the client *could* modify this content in flight; however, the common assumption is that, barring a few types of client proxies, no such modifications take place. In this chapter, we show that this assumption is false. Not only do a large number and variety of in-flight modifications occur to web pages, but they often result in significant problems for users or publishers or both.

We present the results of a measurement study to better understand what in-flight changes are made to web pages in practice, and the implications these changes have for end users and web publishers. The web server used in our measurement study recorded any changes made to the HTML code of its web pages for visitors from over 50,000 unique IP addresses.

Changes to our page were seen by 1.3% of the client IP addresses in our sample, drawn from a population of technically oriented users. We observed many types of changes caused by agents with diverse incentives. For example, ISPs seek revenue by injecting ads, end users seek to filter annoyances like ads and popups, and malware authors seek to spread worms by injecting exploits.

Many of these changes are undesirable for publishers or users. At a minimum, the injection or removal of ads by ISPs or proxies can impact the revenue stream of a web publisher, annoy the end user, or potentially expose the end user to privacy violations. Worse, we find that several types of modifications introduce bugs or even vulnerabilities into many or all of the web pages a user visits—pages that might otherwise be safe and bug-free. Such problems are likely to become more severe as the code in web programs becomes more complex. We demonstrate the threats these modifications already pose by building successful exploits of the vulnerabilities.

These discoveries reveal a diverse ecosystem of agents that modify web pages. Because many of these modifications have negative consequences, publishers may have incentives to detect or even prevent them from occurring. Detection can help publishers notify users that a page or program might not appear as intended, take action against those who make unwanted changes, debug problems due to modified pages, and potentially deter some types of changes. Preventing modifications may sometimes be important, but there may also be types of page changes worth allowing. For example, some enterprise proxies modify web pages to increase client security, such as Blue Coat WebFilter [24] and our own BrowserShield system [100].

HTTPS offers a strong but rigid and costly solution for these issues. HTTPS encrypts web traffic to prevent in-flight modifications, though proxies that act as HTTPS endpoints may still alter pages without any indication to the server. Encryption can prevent even beneficial page changes, as well as web caching, compression, and other useful services that rely on the open nature of HTTP.

As a result, we propose that concerned web publishers adopt *web tripwires* on their pages to help understand and react to any changes made in flight. Web tripwires are client-side JavaScript code that can detect most modifications to unencrypted web pages. Web tripwires are not secure and cannot detect all changes, but they can be made robust in practice. We present several designs for web tripwires and show that they can be deployed at a lower cost than HTTPS, do not require changes to web browsers, and support various policy decisions for reacting to page modifications. They provide web servers with practical integrity checks against a variety of undesirable or dangerous modifications.

The rest of this chapter is organized as follows. Section 6.2 describes our measurement study of in-flight page changes and discusses the implications of our findings. Section 6.3 compares several web tripwire implementation strategies that allow publishers to detect changes to their own pages. We evaluate the costs of web tripwires and their robustness to adversaries in Section 6.4. Section 6.5 illustrates how our web tripwire toolkit is easy to deploy and can support a variety of policies. Finally, we conclude in Section 6.6.

6.2 Measuring In-Flight Modifications

Despite the lack of integrity guarantees in HTTP, most web publishers and end users expect web pages and web programs to arrive at the client as the publisher intended. Using measurements of a large client population, we find that this is not the case. ISPs, enterprises, end users, and malware authors all have incentives to modify pages, and we find evidence that each of these parties does so in practice. These changes often have undesirable consequences for publishers or users, including injected advertisements, broken pages, and exploitable vulnerabilities. These results demonstrate the precariousness of today's web, and that it can be dangerous to ignore the absence of integrity protection for web content.

To understand the scope of the problem, we designed a measurement study to test whether web pages arrive at the client unchanged. We developed a web page that could detect changes to its HTML source code made by an agent between the server and the browser, and we attracted a diverse set of clients to the page to test many paths through the network. Our study seeks to answer two key questions:

- What kinds of page modifications occur in practice, and how frequently?

- Do the changes have unforeseen consequences?

We found that clients at over 1% of 50,000 IP addresses saw some change to the page, many with negative consequences. In the rest of this section, we discuss our measurement technique and the diverse ecosystem of page modifications that we observed.

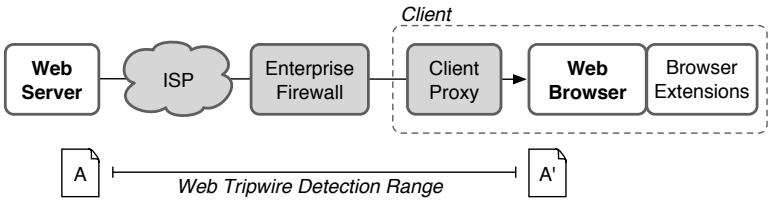

Figure 6.1: Web tripwires can detect any modifications to the HTML source code of a page made between the server and the browser.

6.2.1 Measurement Infrastructure

Our measurement study identifies changes made to our web page between the web server and the client's browser, using code delivered by the server to the browser. This technique allows us to gather results from a large number of clients in diverse locations, although it may not detect agents that do not modify every page.

Technology. Our measurement tool consists of a web page with JavaScript code that detects page modifications. We refer to this code as a *web tripwire* because it can be unobtrusively placed on a web page and triggered if it detects a change. As shown in Figure 6.1, our web tripwire detects changes to HTML source code made anywhere between the server and browser, including those caused by ISPs, enterprise firewalls, and client-side proxies. We did not design the web tripwire to detect changes made by browser extensions, because extensions are effectively part of the browser, and we believe they are likely installed with the knowledge and consent of the user. In practice, browser extensions do not trigger the tripwire because they operate on the browser's internal representation of the page and not the HTML source code itself.

Our web tripwire is implemented as JavaScript code that runs when the page is loaded in the client's browser. It reports any detected changes to the server and displays a message to the user, as seen in Figure 6.2. Our implementation can display the difference between the actual and expected contents of the page, and it can collect additional feedback from the user about her environment. Further implementation details can be found in Section 6.3.

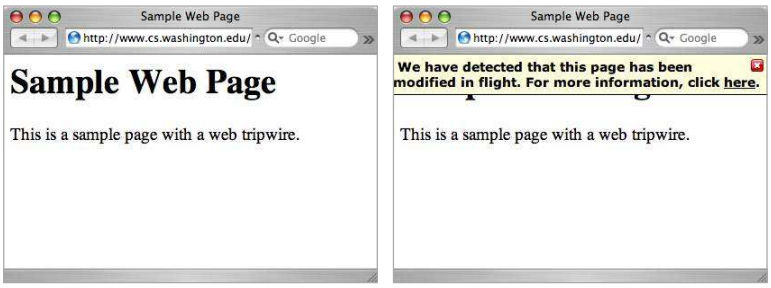

Figure 6.2: If a web tripwire detects a change, it displays a message to the user, as in the screenshot on the right.

We note two caveats for this technique. First, it may have false negatives. Modifying agents may choose to only alter certain pages, excluding those with our web tripwires. We do not expect any false positives, though, so our results are a lower bound for the actual number of page modifications.[1] Second, our technique is not cryptographically secure. An adversarial agent could remove or tamper with our scripts to evade detection. For this study, we find it unlikely that such tampering would be widespread, and we discuss how to address adversarial agents in Section 6.4.2.

Realism. We sought to create a realistic setting for our measurement page, to increase the likelihood that agents might modify it. We included HTML tags from web authoring software, randomly generated text, and keywords with links.

We were also guided by initial reports of ISPs that injected advertisements into their clients' web traffic, using services from NebuAd [12]. These reports suggested that only pages from `.com` top-level domains (TLDs) were affected. To test this, our measurement page hosts several frames with identical web tripwires, each served from a different TLD. These frames are served from `vancouver.cs.washington.edu`, `uwsecurity.com`, `uwprivacy.org`, `uwcse.ca`, `uwsystems.net`, and 128.208.6.241.

[1] In principle, a false positive could occur if an adversary forges a web tripwire alarm. Since this was a short-term measurement study, we do not expect that we encountered any adversaries or false positives.

We introduced additional frames during the experiment, to determine if any agents were attempting to "whitelist" the domains we had selected to evade detection. After our measurement page started receiving large numbers of visitors, we added frames at `www.happyblimp.com` and `www2.happyblimp.com`.

In the end, we found that most changes were made indiscriminately, although some NebuAd injections were `.com`-specific and other NebuAd injections targeted particular TLDs with an unknown pattern.

Exposure. To get a representative view of in-flight page modifications, we sought visitors from as many vantage points as possible. Similar studies such as the ANA Spoofer Project [23] attracted thousands of participants by posting to the Slashdot news web site, so we also pursued this approach.

Although our first submission to Slashdot was not successful, we were able to circulate a story among other sites via Dave Farber's "Interesting People" mailing list. This led another reader to successfully post the story to Slashdot.

Similarly, we attracted traffic from Digg, a user-driven news web site. We encouraged readers of our page to aid our experiment by voting for our story on Digg, promoting it within the site's collaborative filter. Within a day, our story reached the front page of Digg.

6.2.2 Results Overview

On July 24, 2007, our measurement tool went live at `http://vancouver.cs.washington.edu`, and it appeared on the front pages of Slashdot and Digg (among other technology news sites) the following day. The tool remains online, but our analysis covers data collected for the first 20 days, which encompasses the vast majority of the traffic we received.

We collected test results from clients at 50,171 unique IP addresses. 9,507 of these clients were referred from Slashdot, 21,333 were referred from Digg, and another 705 were referred from both Slashdot and Digg. These high numbers of referrals indicate that these sites were essential to our experiment's success.

The modifications we observed are summarized in Table 6.1. At a high level, clients at 657 IP addresses reported modifications to at least one of the frames on the page. About

70% of the modifications were caused by client-side proxies such as popup blockers, but 46 IP addresses did report changes that appeared to be intentionally caused by their ISP. We also discovered that the proxies used at 125 addresses left our page vulnerable to cross-site scripting attacks, while 3 addresses were affected by client-based malware.

6.2.3 Modification Diversity

We found a surprisingly diverse set of changes made to our measurement page. Importantly, these changes were often misaligned with the goals of the publisher or the end user. Publishers wish to deliver their content to users, possibly with a revenue stream from advertisements. Users wish to receive the content safely, with few annoyances. However, the parties in Figure 6.1, including ISPs, enterprises, users, and also malware authors, have incentives to modify web pages in transit. We found that these parties do modify pages in practice, often adversely impacting the user or publisher. We offer a high level survey of these changes and incentives below.

ISPs. ISPs have at least two incentives to modify web traffic: to generate revenue from advertising and to reduce traffic using compression. Injected advertisements have negative impact for many users, who view them as annoyances.

In our results, we discovered several distinct ISPs that appeared to insert ad-related scripts into our measurement page. Several companies offer to partner with ISPs by providing them appliances that inject such ads. For example, we saw 5 IP addresses that received injected code from NebuAd's servers [6]. Traceroutes suggested that these occurred on ISPs including Red Moon, Mesa Networks, and XO, as well as an IP address belonging to NebuAd itself. Other frequently observed ad injections were caused by MetroFi, a company that provides free wireless networks in which all web pages are augmented with ads. We also observed single IP addresses affected by other similar companies, including LokBox, Front Porch, PerfTech, Edge Technologies, and `knects.net`.

Notably, these companies often claim to inject ads based on behavioral analysis, so that they are targeted to the pages a user has visited. Such ads may leak private information about a user's browsing history to web servers the user visits. For example, a server could

Category	**IPs**	ISP	Enterprise	User	Attacker	**Examples**
Popup Blocker	277			✓		Zone Alarm (210), **CA Personal Firewall (17)**, Sunbelt Popup Killer (12)
Ad Blocker	188			✓		**Ad Muncher (99)**, Privoxy (58), **Proxomitron (25)**
Problem in Transit	118	✓				Blank Page (107), Incomplete Page (7)
Compression	30	✓				**bmi.js (23)**, Newlines removed (6), **Distillation (1)**
Security or Privacy	17		✓	✓		Blue Coat (15), **The Cloak (1)**, AnchorFree (1)
Ad Injector	16	✓				MetroFi (6), FairEagle (5), LokBox (1), Front Porch (1), PerfTech (1), Edge Technologies (1), knects.net (1)
Meta Tag Changes	12		✓	✓		Removed meta tags (8), Reformatted meta tags (4)
Malware	3				✓	W32.Arpiframe (2), Adware.LinkMaker (1)
Miscellaneous	3			✓		New background color (1), **Mark of the Web (1)**

Table 6.1: Categories of observed page modifications, the number of client IP addresses affected by each, the likely parties responsible, and examples. Each example is followed by the number of IP addresses that reported it; examples listed in bold introduced defects or vulnerabilities into our page.

use a web tripwire to determine which specific ad has been injected for a given user. The choice of ad may reveal what types of pages the user has recently visited.

We also observed some ISPs that alter web pages to reduce network traffic. In particular, several cellular network providers removed extra whitespace or injected scripts related to image distillation [41]. Such modifications are useful on bandwidth-constrained networks, though they may also unintentionally cause page defects, as we describe in Section 6.2.4.

Enterprises. Enterprises have incentives to modify the pages requested by their clients as well, such as traffic reduction and client protection. Specifically, we observed proxy caches that remove certain `meta` tags from our measurement page, allowing it to be cached against our wishes. Such changes can go against a publisher's desires or present stale data to a user. Our results also included several changes made by Blue Coat WebFilter [24], an enterprise proxy that detects malicious web traffic.

End Users. Users have several incentives for modifying the pages they receive, although these changes may not be in the best interests of the publishers. We found evidence that users block annoyances such as popups and ads, which may influence a publisher's revenue stream. Users also occasionally modify pages for security, privacy, or performance.

The vast majority of page modifications overall are caused by user-installed software such as popup blockers and ad blockers. The most common modifications come from popup blocking software. Interestingly, this includes not only dedicated software like Sunbelt Popup Killer, but also many personal firewalls that modify web traffic to block popups. In both types of software, popups are blocked by JavaScript code injected into every page. This code interposes on calls to the browser's `window.open` function, much like Naccio's use of program rewriting for system-call interposition [38].

Ad blocking proxies also proved to be quite popular. We did not expect to see this category in our results, because our measurement page contained no ads. That is, ad blocking proxies that solely removed ads from pages would have gone unnoticed. However, we detected numerous ad blocking proxies due to the JavaScript code they injected into our page. These proxies included Ad Muncher, Privoxy, Proxomitron, and many others.

Beyond these annoyance blocking proxies, we found user-initiated changes to increase security, privacy, and performance. AnchorFree Hotspot Shield claims to protect clients on wireless networks, and Internet Explorer adds a "Mark of the Web" comment to saved pages to prevent certain attacks [87]. Users also employed web-based anonymization services such as The Cloak [8], as well as proxies that allowed pages to be cached by removing certain `meta` tags.

Malware Authors. Surprisingly, our measurement tool was also able to detect certain kinds of malware and adware. Malware authors have clear incentives for modifying web pages, either as a technique for spreading exploit code or to receive revenue from injected advertisements. These changes are clearly adversarial to users.

In one instance, a client that was infected by Adware.LinkMaker [118] visited our measurement page. The software made extensive changes to the page, converting several words on the page into doubly underlined links. If the user hovered his mouse cursor over the links, an ad frame was displayed.

Two other clients saw injected content that appears consistent with the W32.Arpiframe worm [119]. In these cases, the clients themselves may not have been infected, as the Arpiframe worm attempts to spread through local networks using ARP cache poisoning [135]. When an infected client poisons the ARP cache of another client, it can then act as a man-in-the-middle on HTTP sessions. Recent reports suggest that web *servers* may also be targeted by this or similar worms, as in the recent case of a Chinese security web site [27].

6.2.4 Unanticipated Problems

In the cases discussed above, page modifications are made based on the incentives of some party. However, we discovered that many of these modifications actually had severe unintentional consequences for the user, either as broken page functionality or exploitable vulnerabilities. The threats posed by careless page modifications thus extend far beyond annoyances such as ad injections.

Page Defects

We observed two classes of bugs that were unintentionally introduced into web pages as a result of modifications. First, some injected scripts caused a JavaScript stack overflow in Internet Explorer when they were combined with the scripts in our web tripwire. For example, the `xpopup.js` popup blocking script in CA Personal Firewall interfered with our calls to `document.write`. Similar problems occurred with a compression script called `bmi.js` injected by several ISPs. These bugs occasionally prevented our web tripwire from reporting results, but users provided enough feedback to uncover the issue. In general, such defects may occur when combining multiple scripts in the same namespace without the ability to test them sufficiently.

Second, we discovered that the CA Personal Firewall modifications interfered with the ability to post comments and blog entries on many web sites. Specifically, code injected by the firewall appeared in users' comments, often to the chagrin of the users. We observed 28 instances of "`_popupControl()`" appearing on MySpace blogs and comments, and well over 20 sites running the Web Wiz Forums software [133] that had the same code in their comments. We reproduced the problem on Web Wiz Forums' demo site, learning that CA Personal Firewall injected the popup blocking code into the frame in which the user entered his comments. We observed similar interference in the case of image distillation scripts that contained the keyword "`nguncompressed`."

It is worth noting that such bugs are likely to become more prevalent as the client-side logic in web programs becomes more complex. Many of the in-flight modifications appeared to use simple text substitutions to alter pages, and such changes may impact complex programs in unanticipated ways.

Vulnerabilities

More importantly, we discovered several types of page changes that left the modified pages vulnerable to cross-site scripting (XSS) attacks. The impact of these vulnerabilities should not be understated: the modifications made *most or all* of the pages a user visited ex-

ploitable. Such exploits could expose private information or otherwise hijack any page a user requests.

Ad Blocking Vulnerabilities. We observed exploitable vulnerabilities in three ad-blocking products: two free downloadable filter sets for Proxomitron (released under the names Sidki [113] and Grypen [55]), plus the commercial Ad Muncher product [10]. At the time of our study, each of these products injected the URL of each web page into the body of the page itself, as part of a comment. For example, Ad Muncher injected the following JavaScript comment onto Google's home page:

```
// Original URL:   http://www.google.com
```

These products did not escape any of the characters in the URL, so adversaries were able to inject script code into the page by convincing users to visit a URL similar to the following:

```
http://google.com/?</script><script>alert(1);
```

Servers often ignore unknown URL parameters (following the '?'), so the page was delivered as usual. However, when Ad Muncher or Proxomitron copied this URL into the page, the "</script>" tag terminated the original comment, and the script code in the remainder of the URL was executed as part of the page. To exploit these vulnerabilities, an adversary must convince a user to follow a link of his own construction, possibly via email or by redirecting the user from another page.

It is worth noting that our measurement tool helped us discover these vulnerabilities. Specifically, we were able to search for page changes that placed the page's URL in the body of the page. We flagged such cases for further security analysis.

We developed two exploit pages to demonstrate the threat posed by this attack. Our exploit pages first detect whether a vulnerable proxy is in use, by looking for characteristic modifications in their own source code (e.g., an "Ad Muncher" comment).

In one exploit, our page redirects to a major bank's home page.[2] The bank's page has a login form but is served over HTTP, not HTTPS. (The account name and password are intended to be sent over HTTPS when the user submits the form.) Our exploit injects script code into the bank's page, causing the login form to instead send the user's account name and password to an adversary's server.

In a second exploit, we demonstrate that these vulnerabilities are disconcerting even on pages for which users do not normally expect an HTTPS connection. Here, our exploit page redirects to Google's home page and injects code into the search form. If the user submits a query, further exploit code manipulates the search results, injecting exploit code into all outgoing links. This allows the exploit to retain control of all subsequent pages in the browser window, until the user either enters a new URL by hand or visits an unexploited bookmark.

In the case of Ad Muncher (prior to v4.71), any HTTP web site that was not mentioned on the program's exclusion list is affected. This list prevents Ad Muncher from injecting code into a collection of JavaScript-heavy web pages, including most web mail sites. However, Ad Muncher did inject vulnerable code into the login pages for many banks, such as Washington Mutual, Chase, US Bank, and Wachovia, as well as the login pages for many social networking sites. For most social networking sites, it is common to use HTTPS only for sending the login credentials, and then revert to HTTP for pages within the site. Thus, if a user is already logged into such a site, an adversary can manipulate the user's account by injecting code into a page on the site, without any interaction from the user. This type of attack can even be conducted in a hidden frame, to conceal it from the user.

In both Proxomitron filter sets (prior to September 8, 2007), all HTTP traffic is affected in the default configuration. Users are thus vulnerable to all of the above attack scenarios, as well as attacks on many web mail sites that revert to HTTP after logging in (e.g., Gmail, Yahoo Mail). Additionally, Proxomitron can be configured to also modify HTTPS traffic, intentionally acting as a "man in the middle." If the user enables this feature, all

[2]We actually ran the exploit against an accurate local replica of the bank's home page, to avoid sending exploit code to the bank's server.

SSL encrypted pages are vulnerable to script injection and thus leaks of critically private information.

We reported these vulnerabilities to the developers of Ad Muncher and the Proxomitron filter sets, who have released fixes for the vulnerabilities.

Internet Explorer Vulnerability. We identified a similar but less severe vulnerability in Internet Explorer. IE injects a "Mark of the Web" into pages that it saves to disk, consisting of an HTML comment with the page's URL [87]. This comment is vulnerable to similar attacks as Ad Muncher and Proxomitron, but the injected scripts only run if the page is loaded from disk. In this context, the injected scripts have no access to cookies or the originating server, only the content on the page itself. This vulnerability was originally reported to Microsoft by David Vaartjes in 2006, but no fix is yet available [123].

The Cloak Vulnerabilities. Finally, we found that the "The Cloak" anonymization web site [8] contains two types of XSS vulnerabilities. The Cloak provides anonymity to its users by retrieving all pages on their behalf, concealing their identities from web servers. The Cloak processes and rewrites many HTML tags on each page to ensure no identifying information is leaked. It also provides users with options to rewrite or delete all JavaScript code on a page, to prevent the code from exposing their IP address.

We discovered that The Cloak replaced some tags with a comment explaining why the tag was removed. For example, our page contained a `meta` tag with the name "generatorversion." The Cloak replaced this tag with the following HTML comment:

```
<!-- the-cloak note - deleting possibly dangerous
  META tag - unknown NAME 'generatorversion' -->
```

We found that a malicious page could inject script code into the page by including a carefully crafted `meta` tag, such as the following:

```
<meta name="foo--><script>alert(1);</script>">
```

This script code runs and bypasses The Cloak's policies for rewriting or deleting JavaScript code. We reported this vulnerability to The Cloak, and it has been resolved as of October 8, 2007.

Additionally, The Cloak faces a more fundamental problem because it bypasses the browser's "same origin policy," which prevents documents from different origins from accessing each other [106]. To a client's browser, all pages appear to come from `the-cloak.com`, rather than their actual origins. Thus, the browser allows all pages to access each other's contents. We verified that a malicious page could load sensitive web pages (even HTTPS encrypted pages) from other origins into a frame and then access their contents. This problem is already known to security professionals [60], though The Cloak has no plans to address it. Rather, users are encouraged to configure The Cloak to delete JavaScript code to be safe from attack.

OS Analogy. These vulnerabilities demonstrate the power wielded by software that rewrites web pages, and the dangers of any flaws in its use. By considering web browsers as operating systems for web programs, we can better understand the severity of the problem. Most XSS vulnerabilities affect a single web site, just as a security vulnerability in a program might only affect that program's operation. However, vulnerabilities in page-rewriting software can pose a threat for *most or all* pages visited, just as a root exploit may affect all programs in an operating system. Page-rewriting software must therefore be carefully scrutinized for security flaws before it can be trusted.

6.3 Practical Detection with Web Tripwires

Our measurement study reveals that in-flight page modifications can have many negative consequences for both publishers and users. As a result, publishers have an incentive to seek integrity mechanisms for their content. There are numerous scenarios where detecting modifications to one's own web page may be useful:

- Search engines could warn users of injected scripts that might alter search results.
- Banks could disable login forms if their front pages were modified.

- Web mail sites could debug errors caused by injected scripts.

- Social networking sites could inform users if they detect vulnerable proxies, which might put users' accounts at risk.

- Sites with advertising could object to companies that add or replace ads.

Publishers may also wish to *prevent* some types of page changes, to prevent harm to their visitors or themselves.

HTTPS provides one rigid solution: preventing page modifications using encryption. However, the use of HTTPS excludes many beneficial services, such as caching by web proxies, image distillation by ISPs with low bandwidth networks, and security checks by enterprise proxies. HTTPS also imposes a high cost on the server, in terms of financial expense for signed certificates, CPU overhead on the server, and additional latency for key exchange.

In cases where HTTPS is overly costly, we propose that publishers deploy web tripwires like those used in our measurement study. Web tripwires can effectively detect most HTML modifications, at low cost and in today's web browsers. Additionally, they offer more flexibility than HTTPS for reacting to detected changes.

6.3.1 Goals

Here, we establish a set of goals a publisher may have for using a web tripwire as a page integrity mechanism. Note that some types of tripwires may be worthwhile even if they do not achieve all of the goals.

First, a web tripwire should detect any changes to the HTML of a web page after it leaves the server and before it arrives at the client's browser. We exclude changes from browser extensions, as we consider these part of the user agent functionality of the browser. We also currently exclude changes to images and embedded objects, although these could be addressed in future work.

Goal	**Count Scripts**	**Check DOM**	**XHR then Overwrite**	**XHR then Redirect**	**XHR on Self**	**HTTPS**
Detects all HTML changes	✗	✓	✓	✓	✓	✓
Prevents most changes*	✗	✗	✓	✗	✗	✓
Displays difference	✗	✗	✓	✓	✓	✗
Preserves semantics	✓	✓	✗	✓	✓	✓
Renders incrementally	✓	✓	✗	✗	✓	✓
Supports back button	✓	✓	✗	✗	✓	✓

Table 6.2: Comparison of how well each tripwire implementation achieves the stated goals. (*Neither "XHR then Overwrite" nor HTTPS can prevent all changes. The former allows full page substitutions; the latter allows changes by proxies that act as the encryption endpoint, at the user's discretion.)

Second, publishers may wish for a web tripwire to prevent certain changes to the page. This goal is difficult to accomplish without cryptographic support, however, and it may not be a prerequisite for all publishers.

Third, a web tripwire should be able to pinpoint the modification for both the user and publisher, to help them understand its cause.

Fourth, a web tripwire should not interfere with the functionality or performance of the page that includes it. For example, it should preserve the page's semantics, support incremental rendering of the page, and avoid interfering with the browser's back button.

6.3.2 Designs & Implementations

Several implementation strategies are possible for building web tripwires. Unfortunately, limitations in popular browsers make tripwires more difficult to build than one might expect. Here, we describe and contrast five strategies for building JavaScript-based web tripwires.[3] We also compare against the integrity properties of HTTPS as an alternative mechanism. The tradeoffs between these strategies are summarized in Table 6.2.

Each of our implementations takes the same basic approach. The web server delivers three elements to the browser: the *requested page*, a *tripwire script*, and a *known-good representation* of the requested page. The known-good representation may take one of several forms; we use either a checksum of the page or a full copy of the page's HTML, stored in an encoded string to deter others from altering it. A checksum may require less space, but it cannot easily pinpoint the location of any detected change. When all three of the above elements arrive in the user's browser, the tripwire script compares the requested page with the known-good representation, detecting any in-flight changes.

We note that for all tripwire implementations, the web server must know the intended contents of the page to check. This requirement may sound trivial, but many web pages are simply the output of server-based programs, and their contents may not be known in advance. For these *dynamic* web pages, the server may need to cache the contents of the page (or enough information to reconstruct the content) in order to produce a tripwire with the known-good representation. Alternatively, servers with dynamic pages could use a web tripwire to test a separate static page in the background. This technique may miss carefully targeted page changes, but it would likely detect most of the agents we observed.

We have implemented each of the strategies described below and tested them in several modern browsers, including Firefox, Internet Explorer, Safari, Opera, and Konqueror. In many cases, browser compatibility limited the design choices we could pursue.

Count Scripts Our simplest web tripwire merely counts the number of script tags on a page. Our measurement results indicate that such a tripwire would have detected 90% of the

[3] We focus on JavaScript rather than Flash or other content types to ensure broad compatibility.

modifications, though it would miss any changes that do not affect script tags (*e.g.*, those made by the W32.Arpiframe worm). Here, the known-good representation of the page is simply the expected number of script tags on the page. The tripwire script compares against the number of script tags reported by the Document Object Model (DOM) to determine if new tags were inserted.

If a change is detected, however, it is nontrivial to determine which of the scripts do not belong or prevent them from running. This approach does miss many types of modifications, but it is simple and does not interfere with the page.

Check DOM For a more comprehensive integrity check, we built a web tripwire that compares the full page contents to a known-good representation. Unfortunately, JavaScript code has no means to directly access the actual HTML string that the browser received. Scripts only have access to the browser's internal DOM tree, through variables such as `document.documentElement.innerHTML`. This internal representation varies between browsers and often even between versions of the same browser. Thus, the server must pre-render the page in all possible browsers and versions in order to provide a known-good representation of the page for any client. This technique is thus generally impractical.

Additionally, the server cannot always accurately identify a client's user agent, so it cannot know which representation to send. Instead, it must send all known page representations to each client. We send a list of checksums to minimize space overhead. The tripwire script verifies that the actual page's checksum appears in the array. Because checksums are used, however, this strategy cannot pinpoint the location of a change.

XHR then Overwrite Rather than checking the browser's internal representation of the page, our third strategy fetches the user's requested page from the server as data. We achieve this using an `XmlHttpRequest` (XHR), which allows scripts to fetch the contents of XML or other text-based documents, as long as the documents are hosted by the same server as the current page. This is an attractive technique for web tripwires for several reasons. First, the tripwire script receives a full copy of the requested page as a string, allowing it to perform comparisons. Second, the request itself is indistinguishable from a

typical web page request, so modifying agents will modify it as usual. Third, the response is unlikely to be modified by browser extensions, because extensions expect the response to contain XML data that should not be altered. As a result, the tripwire script can get an accurate view of any in-flight modifications to the page.

In our first XHR-based web tripwire, the server first sends the browser a small *boot page* that contains the tripwire script and a known-good representation of the requested page (as an encoded string). The tripwire script then fetches the requested page with an XHR. It compares the response with the known-good representation to detect any changes, and it then overwrites the contents of the boot page, using the browser's `document.write` function.

This strategy has the advantage that it could *prevent* many types of changes by always overwriting the boot page with the known-good representation, merely using the XHR as a test. However, adversaries could easily replace the boot page's contents, so this should not be viewed as a secure mechanism.

Unfortunately, the overwriting strategy has several drawbacks. First, it prevents the page from rendering incrementally, because the full page must be received and checked before it is rendered. Second, the use of `document.write` interferes with the back button in Firefox, though not in all browsers. Third, we discovered other significant bugs in the `document.write` function in major browsers, including Internet Explorer and Safari. This function has two modes of operation: it can append content to a page if it is called as the page is being rendered, or it can replace the entire contents of the page if called after the page's `onload` event fires. Many web sites successfully use the former mode, but our tripwire must use the latter mode because the call is made asynchronously. We discovered bugs in `document.write`'s latter mode that can cause subsequent XHRs and cookie accesses to fail in Safari, and that can cause Internet Explorer to hang if the resulting page requests an empty script file. As a result, this overwriting approach may only be useful in very limited scenarios.

However, our measurement tool in Section 6.2 was small and simple enough that these limitations were not a concern. In fact, we used this strategy in our study.

XHR then Redirect We made a small variation to the above implementation to avoid the drawbacks of using `document.write`. As above, the tripwire script retrieves the originally requested page with an XHR and checks it. Rather than overwriting the page, the script redirects the browser to the requested page. Because we mark the page as cacheable, the browser simply renders the copy that was cached by the XHR, rather than requesting a new copy from the server. However, this approach still prevents incremental rendering, and it loses the ability to prevent any changes to the page, because it cannot redirect to the known-good representation. It also consistently breaks the back button in all browsers.

XHR on Self Our final implementation achieves all of our stated goals except change prevention. In this XHR-based approach, the server first delivers the requested page, rather than a small boot page. This allows the page to render incrementally. The requested page instructs the browser to fetch an external tripwire script, which contains an encoded string with the known-good representation of the page. The tripwire script then fetches another copy of the requested page with an XHR, to perform the integrity check. Because the page is marked as cacheable (at least for a short time), the browser returns it from its cache instead of contacting the server again.[4]

This strategy cannot easily prevent changes, especially injected scripts that might run before the tripwire script. However, it can detect most changes to the requested page's HTML and display the difference to the user. It also preserves the page's semantics, the ability to incrementally render the page, and the use of the back button. In this sense, we view this as the best of the implementations we present. We evaluate its performance and robustness to adversarial changes in Section 6.4.

HTTPS Finally, we compare the integrity properties of HTTPS with those of the above web tripwire implementations. Notably, the goals of these mechanisms differ slightly. HTTPS is intended to provide confidentiality and integrity checks for the *client*, but it offers no indication to the server if these goals are not met (e.g., if a proxy acts as the en-

[4]If the page were not cached, the browser would request it a second time from the server. In some cases, the second request may see a different modification than the first request.

cryption end point). Web tripwires are intended to provide integrity checks for the *server*, optionally notifying the client as well. Thus, HTTPS and web tripwires can be seen to complementary in some ways.

As an integrity mechanism, HTTPS provides stronger security guarantees than web tripwires. It uses encryption to detect all changes to web content, including images and binary data. It prevents changes by simply rejecting any page that has been altered in transit. It also preserves the page's semantics and ability to incrementally render.

However, HTTPS supports fewer policy decisions than web tripwires, such as allowing certain beneficial modifications. It also incurs higher costs for the publisher, as we discuss in Section 6.4.

6.4 Evaluation

To evaluate the strengths and weaknesses of web tripwires for publishers who might deploy them, we ask three questions:

1. Are web tripwires affordable, relative to HTTP pages without tripwires?

2. How do the costs of web tripwires compare to the costs of HTTPS?

3. How robust are web tripwires against adversaries?

We answer these questions by quantifying the performance of pages with and without web tripwires and HTTPS, and by discussing how publishers can react to adversarial page modifications.

6.4.1 Web Tripwire Overhead

To compare the costs for using web tripwires or HTTPS as page integrity mechanisms, we measured the client-perceived latency and server throughput for four types of pages. As a baseline, we used a local replica of a major bank's home page, served over HTTP. This is a realistic example of a page that might deploy a tripwire, complete with numerous embedded images, scripts, and stylesheets. We created two copies of this page with web tripwires, one

of which was rigged to report a modification. In both cases, we used the "XHR on Self" tripwire design, which offers the best strategy for detecting and not preventing changes. We served a fourth copy of the page over HTTPS, without a web tripwire.

All of our experiments were performed on Emulab [136], using PCs with 3 GHz Xeon processors. We used an Apache 2 server on Fedora Core 6, without any hardware acceleration for SSL connections.

Latency. For each page, we measured client-perceived latency using small scripts embedded in the page. We measured the start latency (i.e., the time until the first script runs) to show the responsiveness of the page, and we measured the end latency (i.e., the time until the page's onload event fires) to show how long the page takes to render fully. We also measured the number of bytes transferred to the client, using Wireshark [29]. Our tests were conducted with a Windows XP client running Firefox, using a simulated broadband link with 2 Mbps bandwidth and 50 ms one-way link latency. Each reported value is the average of 5 trials, and the maximum relative error was 3.25%.

Figure 6.3 shows that the pages with web tripwires did not increase the start latency over the original page (i.e., all were around 240 ms). In comparison, the extra round trip times for establishing an SSL connection contributed to a much later start for the HTTPS page, at 840 ms.

The time spent rendering for the web tripwires was longer than for the HTTP and HTTPS pages, because the tripwires required additional script computation in the browser. The web tripwire that reported a modification took the longest, because it computed the difference between the actual and expected page contents. Despite this, end-to-end latencies of the tripwire pages were still lower than for the HTTPS page.

Table 6.3 shows that transmitting the web tripwire increased the size of the transferred page by 17.3%, relative to the original page. This increase includes a full encoded copy of the page's HTML, but it is a small percentage of the other objects embedded in the page.

Future web tripwire implementations could be extended to check all data transferred, rather than just the page's HTML. The increase in bytes transferred is then proportional

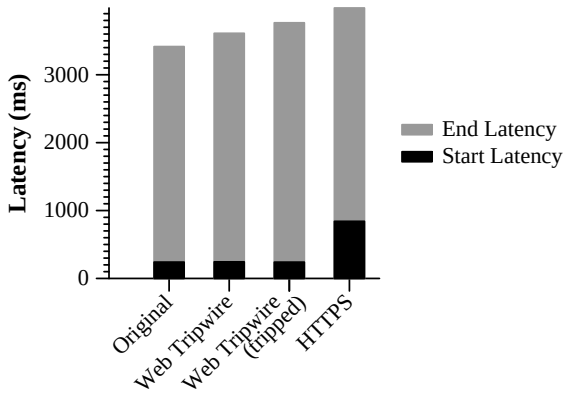

Figure 6.3: Impact of web tripwires and HTTPS on client perceived latency.

Technique	**Data Transferred**
Original	226.6 KB
Web Tripwire	265.8 KB
Web Tripwire (tripped)	266.0 KB
HTTPS	230.6 KB

Table 6.3: Number of kilobytes transferred from server to client for each type of page.

to the number of bytes being checked, plus the size of the tripwire code. If necessary, this overhead could be reduced by transmitting checksums or digests instead of full copies.

Throughput. We measured server throughput using two Fedora Core 6 clients running httperf, on a 1 Gbps network with negligible latency. For each page, we increased the offered load on the server until the number of sustained sessions peaked. We found that the server was CPU bound in all cases. Each session simulated one visit to the bank's home page, including 32 separate requests.

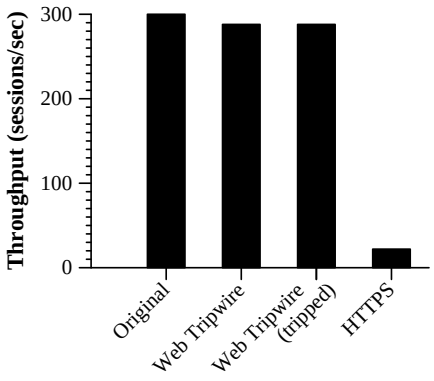

Figure 6.4: Impact of web tripwires and HTTPS on server throughput.

Figure 6.4 shows our results. The web tripwire caused only a 4% degradation of throughput compared to the original page. In comparison, the throughput dropped by over an order of magnitude when using HTTPS, due to the heavy CPU load for the SSL handshake.

For well-provisioned servers, HTTPS throughput may be improved by using a hardware accelerator. However, such hardware introduces new costs for publishers.

6.4.2 Handling Adversaries

In some cases, agents that modify web pages may wish for their behavior to remain undetected. For example, adversarial agents in the network may wish to inject ads, scripts, or even malicious code without being detected by the user or the publisher. Similarly, end users may wish to conceal the use of some proxies, such as ad-blockers, from the publisher.

In general, web tripwires cannot detect all changes to a page. For example, web tripwires cannot detect *full-page substitutions*, in which an adversary replaces the requested content with content of his choice. Similarly, BrowserShield's complete interposition on script behavior can render it transparent to web tripwires, as we discuss in Chapter 8. Thus, we cannot address adversaries who are determined to undetectably modify content at all costs.

Instead, we consider a threat model in which adversaries wish to preserve the functionality of a page while introducing changes to it, without the overhead of complete interposition.

This model assumes that adversaries can observe, delay, and modify packets arbitrarily. However, it reflects the fact that end users often have some expectation of a page's intended contents.

Under such a threat model, we hypothesize that publishers can make web tripwires effective against most adversaries. Barring complete interposition techniques like BrowserShield, adversaries must both identify *and disable* any web tripwire on a page. Publishers can make both tasks difficult in practice using code obfuscation, using approaches popular in JavaScript malware for evading signature-based detection (e.g., code mutators [120], dynamic JavaScript obfuscation [141], and frequent code repacking [54]). Several additional techniques can challenge an adversary's ability to identify or disable tripwires on-the-fly: creating many variants of web tripwire code, employing web tripwires that report an encoded value to the server even if no change is observed, and randomly varying the encoding of the known-good representation. Also, integrating web tripwire code with other JavaScript functionality on a page can disguise tripwires even if adversaries monitor the behavior of a page or attempt to interpret its code.

Ultimately, it is an open question whether an arms race will occur between publishers and agents that modify pages, and who would win such a race. We feel that the techniques above may help make web tripwires an effective integrity mechanism in practice, by making it more difficult for adversaries to disable them. However, using HTTPS (alternatively or in addition to web tripwires) may be appropriate in cases where page integrity is critical.

6.4.3 Summary

Overall, web tripwires offer an affordable solution for checking page integrity, in terms of latency and throughput, and they can be much less costly than HTTPS. Finally, though they cannot detect all changes, web tripwires can be robust against many types of agents that wish to avoid detection.

6.5 Configurable Toolkit and Service

Based on our findings, we developed an open source toolkit to help publishers easily integrate web tripwires into their own pages. When using tripwires, publishers face several policy

decisions for how to react to detected modifications. These include: (1) whether to notify the end user, (2) whether to notify the server, (3) whether the cause can be accurately identified, and (4) whether an action should be taken. Our toolkit is configurable to support these decisions.

The web tripwire in our toolkit uses the same "XHR on Self" technique that we evaluated in Section 6.4. We offer two implementations with different deployment scenarios: one to be hosted entirely on the publisher's server, and a second to be hosted by a centralized server for the use of many publishers.

The first implementation consists of two Perl CGI scripts to be hosted by the publisher. The first script produces a JavaScript tripwire with the known-good representation of a given web page, either offline (for infrequently updated pages) or on demand. The second script is invoked to log any detected changes and provide additional information about them to the user. Publishers can add a single line of JavaScript to a page to embed the web tripwire in it.

Our second implementation acts as a web tripwire service that we can host from our own web server. To use the service, web publishers include one line of JavaScript on their page that tells the client to fetch the tripwire script from our server. This request is made in the background, without affecting the page's rendering. Our server generates a known-good representation of the page by fetching a separate copy directly from the publisher's server, and it then sends the tripwire script to the client. Any detected changes are reported to our server, to be later passed on to the publisher. Such a web tripwire service could easily be added to existing web site management tools, such as Google Analytics [47].

In both cases, the web tripwire scripts can be configured for various policies as described below.

Notifying the User. If the web tripwire detects a change, the user can be notified by a message on the page. Our toolkit can display a yellow bar at the top of the page indicating that the page has changed, along with a link to view more information about the change. Such a message could be beneficial to the user, helping her to complain to her ISP about injected ads, remove adware from her machine, or upgrade vulnerable proxy software.

However, such a message could also become annoying to users of proxy software, who may encounter frequent messages on many different web sites.

Notifying the Server. The web tripwire can report its test results to the server for further analysis. These results may be stored in log files for later analysis. For example, they may aid in debugging problems that visitors encounter, as proposed in AjaxScope [73]. Some users could construe such logging as an invasion of their privacy (e.g., if publishers objected to the use of ad blocking proxies). We view such logging as analogous to collecting other information about the client's environment, such as IP address and user agent, and use of such data is typically described under a publisher's privacy policy.

Identifying the Cause. Accurately identifying the cause of a change can be quite difficult in practice. It is clearly a desirable goal, to help guide both the user and publisher toward an appropriate course of action. In our own study, for example, we received feedback from disgruntled users who incorrectly assumed that a modification from their Zone Alarm firewall was caused by their ISP.

Unfortunately, the modifications made by any particular agent may be highly variable, which makes signature generation difficult. The signatures may either have high false negative rates, allowing undesirable modifications to disguise themselves as desirable modifications, or high false positive rates, pestering users with notifications even when they are simply using a popup blocker.

Our toolkit allows publishers to define patterns to match known modifications, so that the web tripwire can provide suggestions to the user about possible causes or decide when and when not to display messages. We recommend to err on the side of caution, showing multiple possible causes if necessary. As a starting point, we have built a small set of patterns based on some of the modifications we observed.

Taking Action. Even if the web tripwire can properly identify the cause of a modification, the appropriate action to take may depend highly on the situation. For example, users may choose to complain to ISPs that inject ads, while publishers may disable logins or other

functionality if dangerous scripts are detected. To support this, our toolkit allows publishers to specify a callback function to invoke if a modification is detected.

6.6 Summary

Using measurements of a large client population, we have shown that a nontrivial number of modifications occur to web pages on their journey from servers to browsers. These changes often have negative consequences for publishers and users: agents may inject or remove ads, spread exploits, or introduce bugs into working pages. Worse, page-rewriting software may introduce vulnerabilities into otherwise safe web sites, showing that such software must be carefully scrutinized to ensure the benefits outweigh the risks. Overall, page modifications can present a significant threat to publishers and users when pages are transferred over HTTP.

To counter this threat, we have presented "web tripwires" that can detect most modifications to web pages. Web tripwires work in current browsers and are more flexible and less costly than switching to HTTPS for all traffic. While they do not protect against all threats to page integrity, they can be effective for discovering even adversarial page changes. Our publisher-hosted and service-hosted implementations are easy to add to web pages, and they are available at the URL below:

`http://www.cs.washington.edu/research/security/webtripwires.html`

Chapter 7

AUTHORIZING WEB PROGRAM CODE

In-flight changes are not the only way that unwanted code is injected into web programs. In this chapter, we address how publishers can effectively prevent script injection attacks that allow adversaries to hijack web programs. We focus on a fail-closed approach, where publishers explicitly authorize which code is allowed to run in their web programs.

7.1 Motivation

Web programs are now prevalent, relying on active content such as JavaScript to provide users with a rich application experience. JavaScript lets publishers take advantage of advanced browser functions and perform client-side processing, ranging from event handling to manipulation of page content to asynchronous exchange of data with web servers. Using it, publishers can build surprisingly sophisticated and responsive web applications [44]. However, the presence of JavaScript code in a web program can also expose users to security risks.

This chapter examines JavaScript injection attacks [26] that let adversaries exert control over web programs within users' browsers. These attacks occur when an adversary manipulates a user's browser into executing JavaScript code of the attacker's choosing. In one common attack, an adversary submits carefully crafted content to a web server (e.g., by adding a comment to a blog). When a user visits the affected web page, the server unknowingly delivers the attacker's JavaScript code embedded in the intended page content. Many high profile sites, including Yahoo Mail and MySpace, have fallen victim to these attacks [18, 2]. The consequences of an attack can be severe, ranging from the leak of confidential user data to distributed denial-of-service attacks on third parties.

Existing defenses against JavaScript injection attacks are ad hoc and incomplete. Most rely on web publishers to follow best practices for validating input from users, in an attempt

to prevent scripts from being unwittingly injected. Worse, today's defenses are *fail-open:* a developer error or oversight can leave a web site and its users vulnerable to attack.

We propose a *fail-closed* solution that prevents the execution of any script not explicitly intended by the publisher. Our solution uses *script whitelists* that capture the publisher's intent: web pages are delivered along with a whitelist that defines exactly which scripts the browser will permit to execute while rendering the page. If a script is delivered to the browser that is not part of the whitelist, the browser blocks its execution. Our solution requires modifications to both web pages and web browsers to prevent attacks. However, it is backward compatible, allowing existing pages and browsers to continue working.

The contributions of this chapter are as follows:

- we describe the five classes of JavaScript injection attacks and their potential consequences, and we discuss the shortcomings of existing defenses;

- we propose a strawman script whitelisting technique, and we show how common practices in modern web sites (such as JavaScript code obfuscation) are incompatible with this strawman;

- we describe effective whitelisting techniques, and we propose a syntax, architecture, and its implementation in the context of the Firefox web browser and a PHP-based server-side whitelist generator;

- we evaluate the effectiveness, backward compatibility, and performance of our technique.

Our evaluation shows that script whitelists are effective at blocking known attacks, are feasible to incorporate into complex web sites, and perform well.

The rest of this chapter is organized as follows. Section 7.2 lays a foundation for understanding JavaScript injection attacks, covering several classes of exploits and their consequences, as well as previous solutions. We discuss script whitelists in Section 7.3, starting with a simple architecture and then extending it to handle the challenges posed by real

web sites. In Section 7.4 we present our client and server based implementations of script whitelists, and we evaluate their success at preventing JavaScript injection attacks in Section 7.5. We conclude in Section 7.6.

7.2 JavaScript Injection

To defend against JavaScript injection attacks, we must first understand them. This section describes five classes of injection attacks that allow adversaries to place malicious code on a victim's web site, and explores their potential consequences. These range from leaking private information, to falsifying information on a page, to distributed denial-of-service attacks. We categorize the previously proposed defenses against JavaScript injection attacks, describing why they are unlikely to be effective in general.

7.2.1 Exploit Classes

JavaScript injection attacks occur when an adversary coerces visitors of a victim web site into running the adversary's JavaScript code as if it were part of the victim site. These attacks are often referred to as "cross-site scripting" (XSS) attacks, as they let adversaries run code across web site boundaries. Script injection can be accomplished in many ways, such as hiding script code in the posts on a discussion board or crafting a URL that causes script code to appear on the destination page. Below, we describe five known classes of injection attacks, with one notable example of each.

Stored XSS. In these attacks, adversaries exploit server-side, application-level vulnerabilities to add scripts to a page in a persistent way. This usually occurs on sites that accept user-contributed content, such as comments on blog sites, posts on bulletin boards, or user profiles on social networking sites. Such sites must use input validation to ensure that any user-contributed content does not contain script code. However, as we describe in Section 7.2.3, input validation can be difficult to implement. Thus, adversaries often sneak script code past an input filter, allowing it to be stored and displayed for all future site visitors.

As a simple example, version 2.0.18 of the phpBB bulletin board software [7] was vulnerable to a stored XSS attack. If the site administrator let visitors post comments with a set of simple HTML text formatting tags (including bold, italic, underline, and preformatted tags), then an adversary could hide script code inside one of those tags. For example, an adversary could use the following code, which uses a quoted "`>`" character to confuse the input filter [17]:

```
<B C=">" onmouseover="alert('attack code')" X="<B ">text</B>
```

Reflected XSS. Adversaries can exploit similar server-side vulnerabilities using a carefully crafted URL instead of user-contributed content. Many sites display the value of URL parameters, such as search engine queries, on the page itself. If the site does not properly validate these parameters, adversaries can build a URL that contains script code, causing the code to run if a user follows the URL. These malicious URLs can then be distributed via phishing emails or other social-engineering attacks.

For example, the *New York Times* was vulnerable to the following reflected XSS attack [1], in which script code in the URL was included in the resulting page:

```
http://www.nytimes.com/auth/login?URI=">><script>alert('nytimes')</script>
```

DOM-based XSS. These attacks resemble reflected XSS, except the URL parameters are parsed by client-side JavaScript code instead of server-side code. DOM-based XSS exploits take the same form as reflected XSS exploits. However, the attack code is never present in the HTML of the page itself; it is instead added to the page via existing JavaScript code. The vulnerable code below shows one example [76], in which a script adds content from the URL to the page:

```
<script>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</script>
```

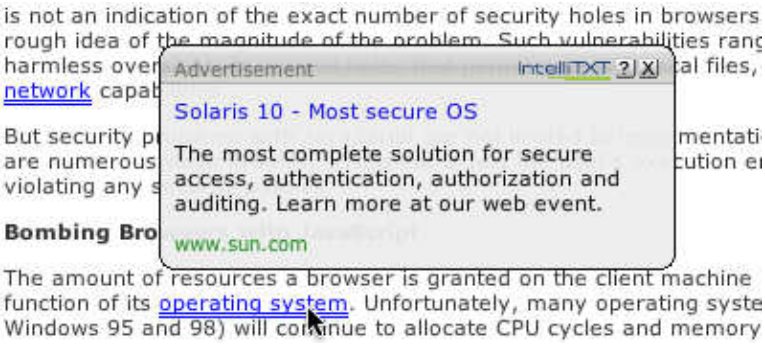

Figure 7.1: Keywords in the content of an article are automatically transformed into advertising links by a third-party script.

Note that this approach can be used to exploit vulnerabilities in HTML files stored on the local computer. In this case, the attack code runs with greater privileges, giving the attacker the ability to access the local filesystem.

Third-Party Scripts. Beyond the commonly recognized XSS attacks described above, adversaries can use other approaches to inject malicious script code into the content of a victim web server. For example, many web sites directly embed JavaScript files from third-party servers, such as advertisers, in their pages. In fact, 48 of the top 100 domains reported by Alexa in 2006 [9] embed scripts from third-parties on their front page. These third-party script files have the same capabilities as any other script on the page: they can modify the page or generate new JavaScript code. As a nonmalicious example, IntelliTXT advertising uses a third-party script to transform certain keywords in an article into advertising links [5], as shown in Figure 7.1.

Because these scripts are delivered directly from the third-party server to the client, the host web site cannot verify whether they contain the expected content. Hence, the security of the host web site becomes dependent on the security of the third-party server. That is, if the third-party server is compromised, it may deliver malicious JavaScript to visitors without the host web site's knowledge.

Bookmarklets. Most modern browsers support "bookmarklets," which are bookmarks that contain JavaScript code instead of simple URLs. Bookmarklets are becoming popular among web developers and advanced users because they aid in debugging and modifying page structure (e.g., blocking ads). However, they can also be used maliciously, since the bookmarklet code is run as if it is part of the currently displayed web page. For example, the following malicious bookmarklet will secretly leak the cookie for the page the user is viewing, before redirecting the user to the expected page:

```
javascript:(function() {i=new Image(1,1);
i.src="http://evil.com/"+document.cookie;
location.href="http://expected.com"})()
```

Bookmarklets, unlike XSS attacks, do not rely on a flaw in the victim page to be effective. Instead, an adversary could use social engineering to convince a user to bookmark a malicious link, similar to many phishing attacks. While we are not yet aware of such attacks in practice, they may become popular if other XSS flaws are corrected.

7.2.2 Potential Consequences

Each attack strategy described above grants an adversary significant control over a user's web browser. Currently, web browsers isolate script code in different web programs using the Same Origin Policy [106], creating in effect a sandbox for each web site. The policy blocks script code on one web site from viewing content from or opening socket-like connections to other web sites in the browser. The goal is to prevent adversaries from spying on a user's browsing behavior or modifying the contents of other pages. With JavaScript injection attacks, however, adversaries can run their own code in the sandbox of a victim web program. This code can then leak information back to the adversary, modify the victim web program, or otherwise control the user's browser.

A user's *privacy* can be violated if an adversary leaks information from the user's browser to the adversary's web server. The Same Origin Policy prevents injected script code from reading data from other servers over the network with XHRs, but an adversary can use other channels to leak information from the page's sandbox. For example, a script can

embed an image from the adversary's server into the page, with private data encoded in the URL. Such data may include cookies from the victim web site (letting the adversary log in with the user's credentials) or any information displayed on the page or entered by the user, such as usernames, passwords, or credit card numbers. Additionally, the adversary could use asynchronous network requests such as XHRs to retrieve additional information from the victim web site using the client's credentials. For example, a script injected into a web email program could request and then leak the contents of the user's address book.

An adversary can violate the *integrity* of the victim web program by modifying any content in the program's pages. This could involve subtle changes that falsify a site's content. In one actual incident, an adversary injected humorous fake news articles onto the web sites of CBS News and BBC [97]. Alternatively, adversaries could make the site appear faulty or launch attacks against browser vulnerabilities. For example, the web site for Super Bowl XLI became victim to an XSS attack that installed a keylogger and backdoor on visitor's PCs [134]. Adversaries could also inject a form into the site to obtain additional sensitive information from the user, such as a credit card or PIN number.

Using injected scripts, an adversary could also deny *availability* to a server of his choice. This can be accomplished by employing the web browsers of all his visitors to launch a distributed denial-of-service attack, as suggested by Lam et al [77]. For example, a stored XSS attack on a popular web site could include code to barrage a victim server with requests, overwhelming the victim server with traffic. Grossman notes that such an attack could be compounded if an advertiser's third-party script were compromised, as all sites that embed the third-party script would then display the attack code on their pages [53].

Combined, these consequences of successful attacks represent a critical security threat for web users. Prominent examples of these attacks have been seen in practice. In an example from 2005, the Samy worm allowed a single MySpace user to add himself as a "hero" to the profile of anyone who viewed an infected page [2, 53]. The worm used JavaScript code injected into a profile to simulate the actions of adding the author as a hero, and it copied itself to the victim profile. Within 24 hours, over one million profiles had been infected, causing the MySpace server to become unavailable. In another example from 2006, the Yammaner worm targeted Yahoo's web-based email client [18]. The worm forwarded itself

to all contacts in a victim's address book, and it collected email addresses on a central server to be sold to spammers. It propagated by injecting JavaScript code that was not properly filtered from email messages. These examples show that high profile web sites with extensive JavaScript blocking mechanisms have nonetheless been vulnerable to JavaScript injection attacks.

7.2.3 Current Defenses

Defenses for JavaScript injection attacks have been proposed in the literature and implemented in practice. However, these defenses are unlikely to provide effective protection against injection attacks. Each approach faces difficulties in identifying unintended script code on a page or preventing scripts from causing harm. We now discuss three categories of proposed solutions and why they remain unsatisfactory.

Script Blocking. The most straightforward way to prevent damage from JavaScript injection attacks is simply to disable JavaScript in the browser. All browsers offer an option to turn off JavaScript, and this approach is sure to defeat any JavaScript injection attack. However, it is increasingly unacceptable because many popular web sites rely on JavaScript to provide important features.

In response, some browser extensions let users selectively allow certain pages to use JavaScript. For example, the NoScript extension for Firefox lets users to specify a whitelist of trusted sites allowed to run JavaScript code [79]. Unfortunately, this site-based approach is ineffective for JavaScript injection attacks, because malicious script code can still be placed on web sites on the whitelist.

Preventing Injection. The current accepted practice for defeating JavaScript injection attacks is to fix any web application vulnerability that could allow code injection. This approach is noble in its goal but ultimately unattainable. Like finding bugs, it is difficult to know when all security vulnerabilities have been found. Additionally, it creates a fail-open scenario, where a single missed defect can render a site vulnerable to injection attacks.

Proper input validation is the most commonly used technique for preventing injection attacks. In this technique, web publishers should inspect all input supplied by the user to ensure that it does not contain potentially executable script code. In some cases, this entails replacing characters such as "<" with HTML entities like "<". In other cases, user input may be allowed to contain some HTML tags but not others.

In general, input validation is notoriously difficult. There are numerous, often undocumented ways to add JavaScript code to a page. Indeed, one site documents over 90 techniques to specify JavaScript code on a page in an effort to help test input filters [56]. Some sites may be able to restrict all user input to letters and numbers, but any site that lets users specify some formatting tags (e.g., in MySpace profiles or bulletin board posts) faces a difficult validation problem.

Moreover, most web browsers tolerate malformed input. These browsers do a "best effort" job to render malformed HTML, which may allow script code to bypass an input filter but still be interpreted as code by the browser. The Samy worm [2] bypassed JavaScript filters on MySpace because it inserted a newline character into the word "javascript" in a URL. The filter did not recognize the attack as code, but Internet Explorer ignored the newline character and executed the worm code.

For these reasons, relying on input validation leaves many opportunities for adversaries to inject code in previously unanticipated ways. Thus, we seek a more comprehensive solution to the problem.

Damage Control. Several approaches attempt to limit the damage that injected code can inflict within the browser rather than trying to prevent the injection itself. These include client-side taint analysis [125, 140], connection blocking [65, 75], and preventing certain actions in scripts [81]. However, distinguishing malicious from legitimate behavior can prove difficult. Thus, these approaches invariably incur false positives and false negatives. For example, taint analysis prevents cross-site network connections after a script on a page reads certain data (e.g., cookies), but it is unclear how much data on a page should be considered tainted to prevent leaks while still ensuring that valid scripts are not hampered. Preventing cross-site image requests after a script makes any DOM access will break legitimate sites,

but allowing such image requests after a script reads information on the page could allow sensitive data to leak. This general technique is also fail-open, as adversaries can still cause damage if the approach fails to contain enough dangerous behavior.

7.3 Script Whitelists

A viable solution to JavaScript injection attacks must precisely identify which scripts are intended to be on a page and which are not. The original publishers of a web site are in the best position to list the set of intended scripts accurately. We therefore propose a server-side mechanism called *script whitelists* to allow them to do so. Enhanced browsers enforce this whitelist of approved scripts and refuse to run any other scripts injected by an adversary. In this way, whitelists offer a fail-closed solution: an incomplete whitelist may block legitimate code from running, but it will never allow unintended code to run.

In this section, we first describe the capabilities that we assume attackers have. Next, we propose a strawman script whitelisting solution, in which a list of digests of intended scripts is included at the top of each web page. While simple and effective, this solution is unfortunately impractical given how modern web servers generate and employ JavaScript. After discussing common practices that confound the strawman, we describe extensions that allow whitelists to be incorporated even into complex web sites.

7.3.1 Threat Model

Current security models for web browsers underestimate the power of today's adversaries. They assume that web page content is controlled by the server that delivers the page. However, JavaScript injection attacks violate this assumption. We propose a stronger threat model that better reflects the capabilities of adversaries and the realities of injection attacks; in particular, we make the following assumptions:

- *Some prefix of each web page can be considered tamper-proof.* Through injection attacks, adversaries can potentially influence the content of web pages. However, we assume that web servers are able to place a block of content at the top of a web page that cannot be influenced or modified by attackers. For example, this "tamper-proof"

block might span until the first point that dynamically generated or user-influenced content is included on a page.

- *The adversary can insert strings into the page at any position after this prefix.* Once past the tamper-proof prefix, we assume that the remainder of the page is vulnerable to injection attacks. We assume that the adversary can insert any string, including text, HTML, or script code, at any place after this prefix. Note that this inserted content may hide the remainder of the page from the user (e.g., in a comment).

- *The adversary has control over URL parameters and bookmarklets.* Phishing and other social engineering attacks have been successful in practice [33]. We thus assume that an adversary can convince users to follow or bookmark a link of their construction.

- *The adversary can gain control over third-party script content.* In general, web sites that embed scripts from third-parties may have little control over these scripts or the security of the third-party servers. We thus assume that such servers can be compromised to distribute malicious script code.

- *The adversary can observe the page before injecting code.* This assumption is important for the design of an effective solution: if a security mechanism assumes the adversary does not know some secret information on the page, the mechanism can be compromised. For example, some pages use asynchronous network requests to fetch new HTML snippets to add to the page (e.g., AJAX email clients). An adversary may have the ability to sniff a client's web traffic to observe the page. He can then inject code with knowledge of the secret information (e.g., by sending the client an email that is subsequently added to the page).

These assumptions are sufficient to encompass attacks observed in practice. Our goal is to use to them to analyze the strength our proposed solutions. Note that we do not aim to prevent injection of HTML or other nonscript content. Our reasoning is that such injection attacks are strictly less powerful than JavaScript injection attacks. For example, they cannot leak sensitive information to an adversary's server.

We also do not aim to prevent in-flight page changes, as discussed in Chapter 6. Such script injections may be unwanted by users or publishers, but we observed few such changes to be malicious in practice. This is likely because network attacks that can rewrite HTTP traffic represent a more difficult goal for adversaries to achieve than those described in this threat model.

7.3.2 A Strawman Script Whitelist Solution

We now propose a strawman solution that uses simple script whitelists to block JavaScript injection attacks. Our strawman mechanism can effectively prevent all injected script code from running, but it can be deployed only on a web page with predictable, *static script* content. As we show in Section 7.3.3, this is unrealistic for many web sites.

In our strawman approach, at the time a page is authored, the web publisher places a whitelist in an HTML comment at the top of the page, within the page's tamper-proof prefix. This whitelist specifies precisely which script code is allowed to run on the page. As shown in Figure 7.2, it contains a secure digest (e.g., a SHA-1 hash) of each *script fragment* that appears on the page. Script fragments are pieces of JavaScript code that are interpreted independently by the browser, including script tags, event handlers, JavaScript links, and external scripts, as well as any code generated at runtime in the browser. We define fragments at the same granularity as used by the browser's JavaScript engine, as we discuss in Section 7.4.

Enhanced web browsers parse this whitelist from the top of the page. Because it is assumed to reside in the tamper-proof prefix of the page, an adversary cannot modify the whitelist. Before the browser runs a script fragment, it first ensures that a corresponding digest appears in the whitelist. If not, the browser simply ignores the script fragment.

This simple solution is attractive because it captures the intent of the web publisher. It ensures that the only script fragments that can run on a given web page are those approved in the whitelist. Thus, adversaries cannot coerce users into running arbitrary script code, even if such code is injected into the page. This whitelist mechanism is also backward compatible with existing web sites and existing browsers. An enhanced browser will run

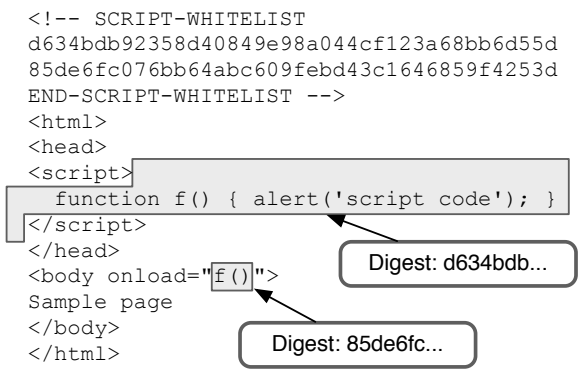

Figure 7.2: A simple web page with a script whitelist. The whitelist contains a SHA-1 digest for each of the two script fragments on the page, as indicated.

any script on a page that lacks a whitelist, allowing unmodified sites to continue to work. Similarly, an unmodified browser will simply ignore any whitelist it encounters, as it resides in an HTML comment.

7.3.3 Limitations of the Strawman Approach

Our experience has shown that it is not feasible for many web sites to know all script code in advance of user requests. The pages delivered from these web sites are often the output of complex programs, and the script code that appears on a page may change from one page view to another. Moreover, the content of a given page may be streamed over the network as it is generated, so the tamper-proof prefix of the page may already be sent before the full set of scripts is known. Thus, the strawman solution for script whitelists cannot be easily deployed on such sites. This section describes several of the practical challenges to using the strawman approach.

Varying Sets of Scripts. Many web pages are not static HTML files, but instead are generated by server-side programs. These programs may choose to include different sets of script fragments on a page depending on the state of the program. For example, phpBB

includes a particular script fragment on each page only if a user is logged in. More complex sites may choose from among a large set of candidate script fragments.

In such cases, it may be difficult to form a practical script whitelist. One might take a conservative approach and include digests for all potential script fragments, but this could form an impractically large list on some pages. It is also undesirable to delay sending a page until the full set of scripts is known, as this would impact the page's responsiveness.

Dynamic Scripts. Script fragments with dynamic content pose another challenge. Many web sites include script fragments with portions that change on every request (e.g., Google's front page). These portions can include timestamps, session IDs, advertising variables, or even user input. In these cases, script content cannot be known before a request arrives from a user. In fact, in many cases script content may not be known until *after* the top of the page has been transmitted to the user. Thus, a static whitelist cannot suffice for pages with dynamic scripts.

Generated Scripts. Generated script fragments present yet another difficulty for defining whitelists. In this case, as an existing script fragment on a page executes, it generates additional JavaScript code into a string variable. It then runs this code via the JavaScript `eval` function or `Function` constructor, or it inserts the code into the page as a new script tag. This generated script may never be seen by the server or web publisher. Unfortunately, generated script fragments are common on many popular sites. For example, many sites appear to use JavaScript obfuscators to obscure proprietary code. These obfuscators often use generated code to further mask a script's behavior.

In certain cases, the web publisher or server-side software may be able to predict the code that will be generated on the client-side and supply a digest for it in the whitelist. However, generated code will not always be predictable on the server-side. Thus, it may not be possible to include digests for all generated script fragments in the whitelist.

Third-Party Scripts. A similar challenge is presented by third-party scripts, which are delivered directly from a third-party server to the client. In many cases, the host server

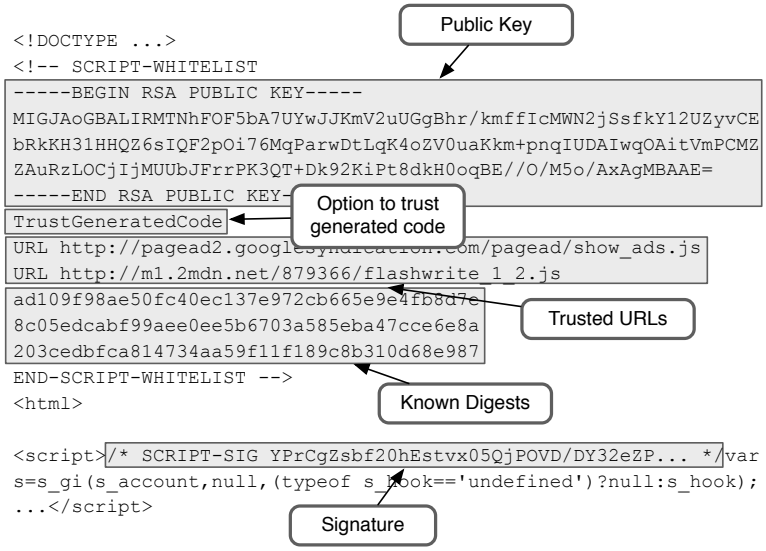

Figure 7.3: The full syntax for whitelists, including our extensions to the strawman. Each part of the whitelist is optional and included only if necessary. The public key is required if any script fragments have signatures rather than digests.

may never know the precise contents of the script code sent to its visitors, leaving it unable to generate digests for the code. Like generated script fragments, it may be possible for the host server to know some third-party scripts in advance, but this is not always the case.

7.3.4 An Effective Script Whitelist Strategy

Because the digest-based strawman solution cannot be effectively integrated into many real web sites, we propose a set of extensions to handle the challenges posed by these sites. These extensions comprise our complete and practical whitelist architecture. Due to the use of arguably unsafe practices on many web sites, some of our extensions represent a tradeoff between security and compatibility. That is, use of the extensions may weaken the guarantees offered by script whitelists to support a broader set of web sites. We thus provide recommendations for how to use whitelists most effectively, allowing publishers to rely on weaker solutions only when necessary.

Script Type	Digests	Signatures	Trust Generated	Trusted URLs
Static	✓	✓	✗	✗
Varying Sets	✗	✓	✗	✗
Dynamic	✗	✓	✗	✗
Generated (predictable)	✓	✓	✓	✗
Generated (unpredictable)	✗	✗	✓	✗
Third-Party (predictable)	✓	✓	✗	✓
Third-Party (unpredictable)	✗	✗	✗	✓

Table 7.1: Effective whitelist mechanisms for each type of script on real pages. Checkmarks indicate when a mechanism will work, and crosses indicate when it will not work. The most protective option for each script type is shaded, showing that each mechanism is necessary in some situations.

The syntax for whitelists is shown in Figure 7.3. The whitelist is placed near the top of the page, following the `DOCTYPE` declaration but preceding any HTML tags, text, or other comments. As no variable content comes first, the whitelist resides in our assumed tamper-proof prefix of the page. The whitelist optionally contains declarations for each of our extensions, followed by a list of known digests for any script fragments that are known in advance. Our extensions are described below and their effectiveness is summarized in Table 7.1. As the table shows, each mechanism is necessary in some cases.

Digests

For static script content, we can enforce digests as described in the strawman solution from Section 7.3.2.

Signatures

To support pages with either varying sets of scripts or dynamic scripts, we must allow some script fragments to be logically added to the whitelist after the whitelist itself has been sent over the network. To achieve this safely, such script fragments must be prefixed with both their own digest and some proof that they were inserted by the server and not an adversary.

We use digital signatures for this purpose. The server can include a public key in the whitelist and can prefix dynamic script fragments with signatures of their contents. The signature is placed in a JavaScript comment before the code, as shown in Figure 7.3. Enhanced browsers can use the public key to verify the signature. If the script fragment is signed properly, the browser executes it.

The use of a signature lets the server prove that the script fragment is intended to be on the page. The signature is not intended to act as proof of the server's identity. Thus, no public key infrastructure (PKI) is required, and the server can generate key pairs at its own discretion. Because public key cryptography is expensive, we note that this technique is required only when the contents of a script fragment cannot be known before the whitelist is transmitted to the user. In other cases, publishers can use static digests.

Signatures maintain the security guarantees of the strawman solution, as long as web publishers ensure that signed script fragments contain no malicious code. Dynamic scripts with well-defined variable content, such as session IDs or timestamps, pose little threat. Unfortunately, in some cases dynamic scripts may be influenced by user input. In these cases, we are forced to fall back on input validation for protection. This is far from ideal, but we have at least narrowed the scope of the validation task. Rather than guarding against dangerous user input anywhere on a page, publishers must guard against it only within dynamic script fragments.

Finally, we note that noncryptographic solutions may be acceptable if an adversary were unable to observe the page before injecting scripts. For example, a random token that appears in the whitelist could be repeated in each dynamic script fragment before the digest is computed. Script Keys [80] offers a similar approach. However, we note in Section 7.3.1 that this assumption may be flawed in some situations. That is, an adversary could observe

the random token and later use it to successfully inject scripts. For this reason, we rely on signatures as stronger proof that only the server (who posses the corresponding private key) can approve scripts not listed in the whitelist.

Trusting Generated Scripts

Generated script fragments might not be predictable by the server. We strongly recommend that publishers include only predictable generated code on their pages so that corresponding digests can be provided in the whitelist. For example, it may be possible to create JavaScript obfuscators that either avoid the use of generated code or produce code that can be predicted. However, in some cases unpredictable generated code may be required. For such pages, we provide the ability to specify a `TrustGeneratedCode` option in the whitelist to instruct browsers to bypass the whitelist check for generated script fragments.

Note that only script fragments that are already approved can attempt to run generated code; thus, this option does not provide a direct opportunity for injection attacks. However, the generated code itself may be vulnerable to a DOM-based XSS attack, so this option does weaken the security of the whitelist. Thus, if this option is used, publishers must use caution to prevent any vulnerabilities in the generated code.

Trusting Script URLs

Like generated script fragments, the contents of third-party scripts may or may not be known to the server. Again, we strongly recommend that publishers embed only third-party scripts for which they can provide digests. This requires third-parties to provide all script code to a web site's publishers before their code can be included on the page. It also requires that third-party scripts remain static, because it is impractical to generate digests or signatures for third-party code on each page request. However, this requirement offers an attractive guarantee for both the host web site and its visitors: the host web site has access to (and can thus audit) all script code delivered to its visitors.

In practice, however, there may be scenarios where a site must use dynamic or unpredictable third-party scripts. Thus, we allow publishers to list a set of trusted script URLs

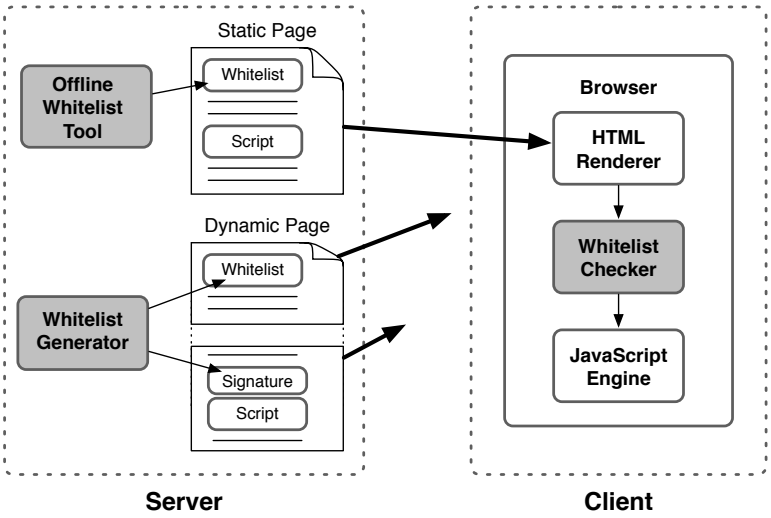

Figure 7.4: Script whitelist architecture. Whitelists can be added to static pages offline and to dynamic pages with an online tool. Signed script fragments can be added to the page as it is streamed to the client. Browsers then check each script fragment as it is sent from the page renderer to the JavaScript engine.

in the whitelist, as shown in Figure 7.3. Enhanced browsers will bypass the whitelist check for any script retrieved from one of these URLs. No guarantees can be made about the script code delivered via these URLs, so we recommend that this option be used sparingly.

7.4 Architecture and Implementation

We now describe the architecture and implementation of our script whitelist prototype. The high-level architecture of our approach is shown in Figure 7.4. For pages with static script content, web publishers use an offline tool to create whitelists and store them in the page. For pages that are generated by server-side software, we provide a whitelist generation library that computes whitelists and signatures on the fly.

When the client's web browser fetches a page, it detects and parses the whitelist (if present). For the purposes of whitelists, a page is defined to be a single HTML document.

As with existing browser security policies, pop up windows and frames are considered to be separate pages.

In an unmodified browser, script fragments are passed to the JavaScript engine by the HTML renderer. Our architecture requires browser modifications to interpose a checker between these components. The checker validates scripts against the whitelist, permitting them to pass to the JavaScript engine only if they are approved. JavaScript can be encountered in a variety of manners. Script tags (e.g., `<script>f()</script>`) encapsulate code that is executed as the page is rendered. Event handlers (e.g., `onload="f()"`) contain code that is invoked when a specified event fires. JavaScript hyperlinks (e.g., `<a href="javascript:f()">`) contain code that is invoked when the link is clicked. External scripts (e.g., `<script src="ext.js">`) contain code that must be retrieved before being invoked. In all cases, the code must be passed to our checker before it flows to the JavaScript engine.

7.4.1 Implementation

We now describe the implementation of our client-side and server-side whitelist components. On the client side, we enhanced the Firefox web browser to detect and enforce whitelists if they are present. On the server side, we built a set of tools to help web publishers incorporate whitelists into their web pages. As a demonstration, we then used these tools to add whitelist generation capabilities into phpBB, a popular and relatively complex web application.

Client Side

We modified the Firefox browser to detect and enforce script whitelists. An ideal implementation would parse and validate the whitelist in the browser's page rendering logic and it would verify scripts in the JavaScript compiler. For ease of implementation, our prototype performs these functions in an external Perl process, as shown in Figure 7.5.

To parse the whitelist, we modified Firefox to search for comments that begin with `SCRIPT-WHITELIST`. When the browser finds a whitelist comment, it opens a socket to the

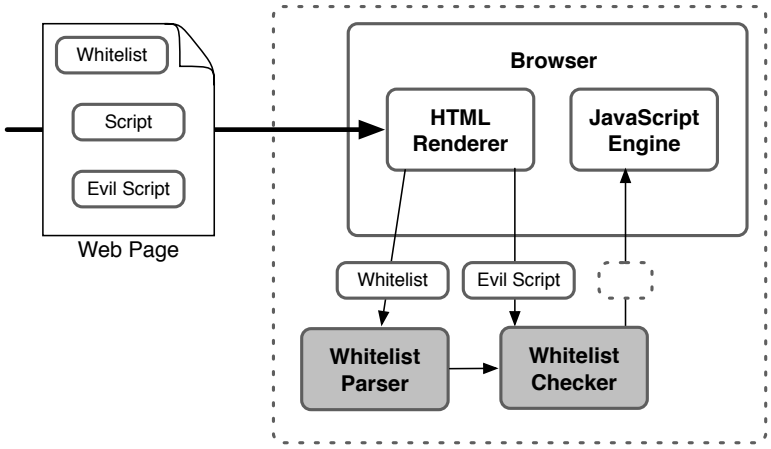

Figure 7.5: Architecture of enhanced browser prototype. The parsing logic delivers each page's whitelist to an external checking mechanism. The browser then consults the checking mechanism before giving each script to its JavaScript engine.

```
if (script is prefixed by a signature) {
    if (whitelist has a public key) {
        if (signature is valid) { accept; }
    }
} else if (script is generated code) {
    if (TrustGeneratedCode) { accept; }
} else if (script is from third party) {
    if (url is trusted in whitelist) { accept; }
} else if (script's digest is in whitelist) {
  accept;
} else reject;
```

Figure 7.6: Pseudocode for checking a script fragment in the browser.

Perl process to store the whitelist for the page. When the browser finds JavaScript code on the page, it passes the code to the Perl process, which then consults the stored whitelist to approve or reject the fragment according to the logic in Figure 7.6.

In our Firefox implementation, we modified the `js_NewTokenStream` function in the browser's JavaScript engine. This function translates a string of JavaScript code into a token stream for compilation. All script code passes through this function before being executed, so it acts as a natural "choke point" for checking a script fragment.

To determine if a script fragment is generated, we pass an extra parameter to the `js_NewTokenStream` function. This parameter is set to true if the script fragment arrived via a call to `eval` or the `Function` constructor. Scripts can also generate code by inserting new script tags into the DOM. Our prototype does not yet identify these tags, so digests for them must be provided in the whitelist. For external scripts, we can examine the URL of the script that is passed to `js_NewTokenStream` and check it against the optional list of trusted URLs in the whitelist.

Server Side

The tools needed to generate whitelists for a web page depend on how the page is generated and the types of scripts it includes. For pages with a static set of scripts, it is sufficient to use a simple offline tool to compute digests. For web applications that decide which scripts to include as the page is generated, or that include dynamically generated scripts, signatures must be generated on the fly and the application must be updated to print these signatures along with the scripts. To deal with third-party and generated scripts, the application must be updated to include appropriate whitelist declarations. We implemented a whitelist generation library to help with these tasks. We now describe our implementations of these tools and libraries, and how we used them to incorporate whitelists into a real web application.

Whitelist Generation Tools. For sites containing pages with predictable, static scripts, we built a Greasemonkey [3] tool to help create whitelists. The publisher uses the tool to load and render the page in her browser. The tool traverses the DOM, locates all script fragments on the page, computes their digests, and assembles a valid whitelist. It also displays each script fragment it used to build the whitelist, so that the publisher can identify any unintended or missing scripts. We built a second web tool to add and remove

digests manually, in the event that publishers want to edit a whitelist. For example, this is necessary to include digests for generated code, which our Greasemonkey tool does not currently detect.

Whitelist PHP Library. With more complex sites that contain dynamically generated or selected scripts, the server must generate the whitelist and signatures for script fragments on the fly. To support this, we built a small library in PHP, a popular web scripting language. Web publishers can use the following library API to incorporate whitelists and signatures into their pages:

- `WhitelistGenerator(publicKey, privateKey)` — builds a new generator object with an optional public and private key pair.
- `void setTrustGeneratedCode(bool)` — sets whether the whitelist will trust generated code. Defaults to false.
- `void addTrustedScriptURL(url)` — adds a third-party URL that will be trusted if encountered on the page.
- `void addKnownDigest(digest)` — adds a digest for a script for inclusion in the whitelist.
- `string printWhitelist()` — returns a string with the whitelist that should be placed at the top of the page, including the public key, trust options, and any known digests.
- `string signScript(script)` — returns the given script code, prefixed by a JavaScript comment containing the corresponding signature, for insertion in the page.

7.4.2 A case study: incorporating whitelists into phpBB

Using our PHP library, we incorporated whitelists into the phpBB bulletin board web application [7]. phpBB has a large number of available script fragments, and it has highly

dynamic pages whose contents vary based on the state of the application (e.g., whether the user is logged in). This makes it difficult to assemble a static digest for a given page, without extensive knowledge of the application. Instead, signatures must be generated for script fragments as they are added to a page.

phpBB places most of its HTML and script code in template files that are separate from the application logic. On each page, it uses a custom preprocessing stage to convert the templates into traditional PHP code. This generated code then fills in variable values and prints the final HTML to send to the client. Many of the script fragments contain variables that are not assigned until after this preprocessing stage, so signatures cannot be placed directly in the template files. Instead, we introduced a new token in the template files to identify JavaScript code to be signed. We then modified the preprocessing logic to identify these tokens and provide signatures for the resulting JavaScript code, as the final page is printed. With this in place, all scripts are signed as they are added to the page.

Overall, we found that our tools and PHP library were useful and sufficient for incorporating script whitelists into phpBB. Moreover, because of how phpBB dynamically generates JavaScript fragments, we found it necessary to use the signature whitelist extension; the simple digests provided by the strawman approach would not suffice.

7.5 Evaluation

In this section, we evaluate the benefits and costs of script whitelists. More specifically, we ask and answer three questions:

- Do script whitelists successfully block script injection attacks?
- Are script whitelists compatible with real sites?
- What is the performance impact of script whitelists?

7.5.1 Do script whitelists successfully block script injection attacks?

To evaluate the effectiveness of script whitelists, we tested them using both synthetic and real-world injection attacks. The synthetic attacks demonstrate that whitelists prevent all of

the exploit classes described in Section 7.2.1. The real-world tests, taken from attacks used against phpBB and MySpace, demonstrate the effectiveness of whitelists against threats found, "in the wild."

Synthetic Attacks

We built a series of test pages containing injection attacks from each of the five known exploit classes. We first visited each page using an unmodified browser, i.e., one without script whitelist support. Next, we added whitelists to each page and visited them using our whitelist-enhanced Firefox browser. We verified that attacks succeed against the unmodified browser but failed against our whitelist-enhanced browser.

Stored XSS. These attacks succeed by sneaking JavaScript past an input filter on a vulnerable site. Thus, we built pages to test the various ways to include JavaScript code on a page. We included 8 straightforward tests: script tags, external scripts, event handlers, JavaScript links, `eval`, `Function` constructors, and scripts inserted into the head and body of the page. We also included 23 JavaScript injection attacks from ha.ckers.org that are designed to sneak past input validation filters [56]. We found that our Firefox prototype ran the script code on all test pages in the absence of whitelists. When a whitelist that excluded the test code was added, our browser rejected the test code in every case but one.

In this case, a Firefox-specific feature allowed an attacker to run JavaScript via an XML file. We were surprised to find that Firefox used the XML file and not the test page as the origin of the JavaScript, yet it still allowed the JavaScript code to access the test page. After investigating, we found that this violation of the Same Origin Policy has been reported as a bug in Firefox [111].

We also tested the `TrustGeneratedCode` option on relevant test pages and found that it did allow the browser to run code passed to `eval` and the `Function` constructor.

Reflected XSS. These attacks are similar to stored XSS attacks, in that they attempt to evade input validation tests. Thus, our results above serve for this class of attacks as well. To further test reflected XSS attacks, we built a sample PHP application that reads

a username from the URL on the server and echoes it on the resulting page. Adding script code in place of the username resulted in a successful reflected XSS attack. Including a whitelist on the page prevents the attack.

DOM-based XSS. These attacks are exploited in the same way as reflected XSS attacks. We built a test HTML page that read a URL parameter from JavaScript code and added it to the page. We found it was vulnerable to injection attacks but a whitelist prevents the injected code from running.

Third-party scripts. Script files on a compromised server can be replaced with malicious code. We built a demonstration of such an attack, replacing the code in an external script file with attack code. Adding a whitelist with the digest for the expected script defeated the attack, but adding a whitelist with a trusted URL for the script does not.

Bookmarklets. The code within bookmarklets is passed to the JavaScript engine in the same way as any other code on a page. Thus, adding a whitelist to a page prevents all bookmarklets from running. This may be an overly conservative solution, as many bookmarklets may offer desirable features. Browsers could allow users to subscribe to a service providing whitelists for trusted bookmarklets. Alternatively, browsers could disable bookmarklets by default and allow advanced users to enable them.

Real-world Attacks

We examined known injection vulnerabilities within real applications to see whether whitelists would have been effective. We based our modifications to phpBB in Section 7.4.1 on version 2.0.18, which is vulnerable to a stored XSS attack [17]. By incorporating whitelists into phpBB, we found that the attack code was no longer run by our prototype browser, despite being present in the page.

We also analyzed the Samy worm that affected MySpace in 2005 [2]. This worm bypassed MySpace's input filters by inserting a newline into the word "`javascript`" in a JavaScript URL. Internet Explorer ignored the newline character and interpreted the content as a

	Yahoo	MSN	Google	MySpace	Ebay	Live.com	Microsoft	YouTube	Amazon	Blogger	**Top 25**
Varying Sets	✓	✗	✗	✓	✗	✗	✓	✗	✗	✗	40%
Dynamic	✓	✓	✓	✓	✗	✗	✓	✗	✓	✗	48%
Generated	✓	✓	✗	✓	✓	✗	✓	✗	✓	✓	68%
Third-Party	✗	✗	✗	✓	✗	✗	✗	✓	✗	✓	44%

Table 7.2: Types of scripts in use by the 10 most popular web sites, including varying sets of scripts, scripts with dynamic contents, generated scripts, and third-party scripts. The last column shows the percentage of the top 25 sites that were observed to use each type on their front page.

script. We were unable to test this precise attack in our prototype browser because Firefox was not vulnerable to the attack. However, our manual inspection confirms that implementing our whitelist architecture in Internet Explorer would have prevented this attack.

7.5.2 Are script whitelists compatible with real sites?

To determine whether real web sites can support script whitelists, we studied the front pages of the top 25 domains reported by Alexa in October 2006 [9]. We fetched each front page twice over a 30 minute interval to see whether script code changed between requests. By examining those changes, we saw an indication of what kinds of scripting techniques are used by each site. Our findings are summarized in Table 7.2.

We detect varying sets of scripts by observing when entire script fragments change between two page requests. We declare a script fragment to be dynamic if it has portions that change between requests. We identify generated code through the presence of `eval` or `Function` constructor calls. Finally, we note any site that included a third-party script from a different organization (e.g., ebay.com and ebaystatic.com are considered the same organization). Each of these techniques is conservative (e.g., a script could change based

on whether a user is logged in, not just after a second page view), and therefore we may be underreporting the prevalence of each practice.

As Table 7.2 shows, a substantial portion of popular web pages use each of these practices. Because almost half of these sites use varying sets of scripts and dynamic scripts, they would need to use signatures to approve script fragments as the page is generated (or random tokens if one assumes a weaker threat model in which the attacker cannot observe the page contents). Many sites appear to use generated code to obfuscate their scripts, and it is unclear whether this code could be predicted on the server. Thus, the `TrustGeneratedCode` option may be necessary in some cases. Finally, 44% of the sites include third-party scripts. We recommend providing digests for these files when possible, but specifying trusted URLs in the whitelist may sometimes be necessary.

7.5.3 What is the performance impact of script whitelists?

Client browsers must enforce whitelists by checking digests and signatures, and servers must either include a static whitelist or compute one on each page request. To quantify the performance impact of these actions, we measured client latency and server throughput for a set of four test pages. The pages are based on the front page of MSN, which presents a realistic (and challenging) script workload. They contain 20 separate script fragments, plus two small scripts that we added for measurement purposes.

All four pages are PHP programs that print the scripts to the page from an array of strings; they differ only in their use of whitelists. The first page ("original") uses no whitelist and has a size of 128 KB. The second page ("static digests", 129 KB) includes a precomputed whitelist with digests for each script. The third page ("dynamic digests", 129 KB) computes the digest for every script fragment on every page request and includes the digests in the whitelist. The fourth page ("dynamic signatures", 132 KB) computes a signature for each script fragment on every page request.

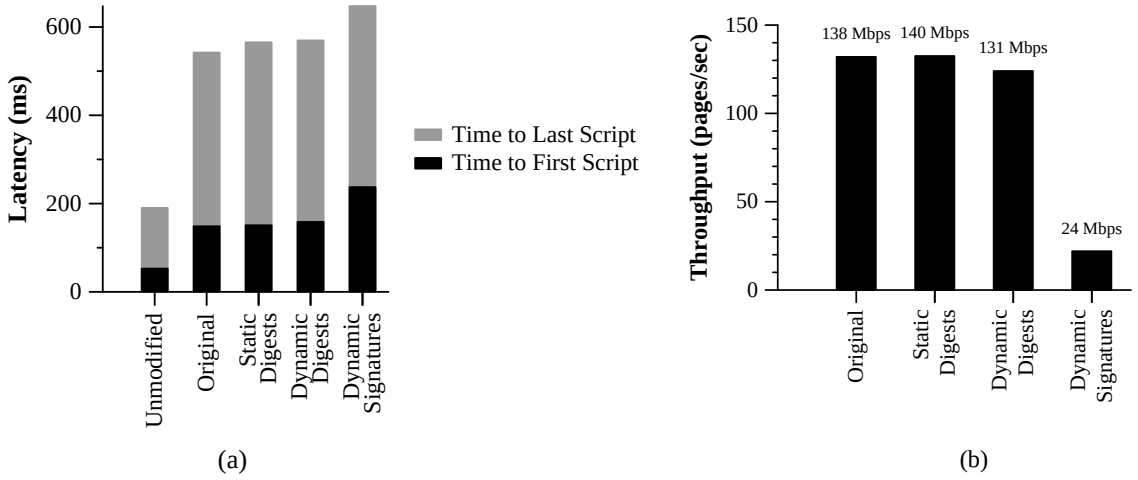

Figure 7.7: Performance impact of whitelists. (a) Impact on client perceived latency. (b) Impact on server throughput. In both graphs, the four pages use no whitelist, a static whitelist, a whitelist of digests generated on the fly, and a whitelist of signatures generated on the fly, respectively. For client latency, we also show the performance of an unmodified browser on the original page.

Latency

Figure 7.7(a) shows the latency measurements of our prototype browser on our test pages. For comparison, the leftmost bar ("unmodified") shows the performance of an unmodified Firefox browser on the "original" page. The bars show a breakdown of the time until the first script runs (the start of rendering) and the time until the final script on the page runs (the completion of rendering).

The first two bars in Figure 7.7(a) shows that our use of an external Perl process has added 400 ms of overhead. Note that this is *not* the cost of checking whitelists, but rather the cost of using an expedient implementation for the prototype. This cost could be eliminated by merging our checking logic directly into the Firefox codebase.

From the other bars in Figure 7.7(a), we see that the incremental cost for performing digest-based script whitelist verification is negligible: less than a millisecond per script

fragment compared to the "original" page. The cost of signature verification is larger (a few tens of milliseconds total), but would still be imperceptible to the end user.

Throughput

To measure the impact of whitelists on server throughput, we served the same set of four pages using Apache 2.0.53 and PHP 4.3.11. We used httperf on two clients to measure throughput; all machines involved were connected by a gigabit LAN, and had dual core, dual processor 3.4 GHz Intel Xeon processors with 4 GB of memory.

Figure 7.7(b) shows our results. Because of its use of PHP, the server was CPU bound in all four tests. To the server, the sole difference between the "original" and "static digests" pages is needing to send extra bytes for the whitelist. Thus, the difference in throughput was tiny, with both tests achieving around 139 Mbps. The "dynamic digests" test computed 22 digests on each request, causing throughput to drop slightly to 131 Mbps. Finally, the "dynamic signatures" test imposed significant overhead, as the server computes 22 public key signatures on each request. This caused throughput to drop to 23.8 Mbps (approximately 22 requests per second, or 484 signatures per second). Having 22 separate signed fragments on a page is perhaps excessive. To reduce the number of signatures needed, we encourage publishers to supply precomputed digests, to compute dynamic digests before sending the page header, or to aggregate many script fragments into a smaller number of scripts when possible.

7.5.4 Evaluation Summary

Whitelists effectively block JavaScript injection attacks and can be incorporated into real web sites. Though the use of dynamically generated signatures can affect server throughput, we found that computing digests incurs negligible cost on the server. As well, we found that whitelist verification has negligible impact on client-perceived latency.

7.6 Summary

JavaScript injection attacks present a critical security threat for the web, and current solutions do not provide effective protection. In this chapter, we have proposed script whitelists as a mechanism to allow publishers to authorize which code is allowed to run in their web programs, and we have shown that this approach supports the complexities of modern web sites. We find whitelists provide full protection against JavaScript injection attacks when all scripts can be known in advance, and they offer adequate protection if publishers are careful with code that cannot be known in advance. As well, our evaluation shows that digest-based whitelists introduce acceptably small performance overhead, though signature-based whitelists have a more significant impact on server throughput. While our technique requires changes to both web clients and existing web pages, it provides fail-closed protection and also allows existing pages and browsers to continue to work.

Chapter 8

ENFORCING POLICIES ON WEB PROGRAMS

To support the final architectural principal in this dissertation, we seek to allow users or administrators to impose policies on web program behavior. Such policies may not be foreseen by either browser developers or program authors but can still provide value or additional security, such as preventing exploits of known browser vulnerabilities. This chapter discusses how to enforce policies on web programs using an interposition layer based on code rewriting. This work appeared at OSDI in 2006 [100] and in ACM Transactions on the Web in 2007 [101].

8.1 Motivation

Web browsers have become a significant target for attacks. Several factors contribute to this, including their widespread use, their tendency to interpret and execute untrusted content, and their complex codebases. The last factor in particular leads to many vulnerabilities that attackers can aim to exploit. During 2005, 8 out of 29 critical Microsoft security bulletins (corresponding to 19 vulnerabilities) were due to flaws in Internet Explorer (IE) or its extensions (e.g., ActiveX controls) [84]. During the same time period, there were also 7 critical security bulletins (corresponding to 16 vulnerabilities) for Firefox [90].

Software patches are currently the primary way to defend against browser exploits, ensuring that vulnerabilities are removed from the browser as they are discovered. However, studies have shown that the deployment of software patches is often delayed after the patches become available. For example, services such as Windows Update download patches automatically, but they typically delay enactment if the patch requires a reboot or application restart. This delay helps both home and corporate users save work and schedule downtime. In corporate settings, patches are also typically tested prior to deployment, to avoid the potentially high costs for recovering from a faulty patch [22].

As a result, there is a dangerous time window between patch release and patch application. Attackers often use this time window to reverse-engineer patches to understand the vulnerability and then launch attacks. One study showed that a large majority of existing attacks target known vulnerabilities [16].

For vulnerabilities that are exploitable through application level protocols (e.g., HTTP, RPC), Shield [130] can address the patch deployment problem using network traffic filtering. Shield filters malicious traffic at a firewall above the transport layer, based on vulnerability signatures. Each signature uses a state machine to characterize all possible message sequences that may lead to attacks, along with the message formats that can trigger an exploit of the application (e.g., an overly long field of a message that triggers a buffer overrun). Importantly, Shield is able to cleanse the network data without modifying the code of the vulnerable application, making signature deployment (and rollback if needed) easier than it is for patches. Vulnerability signature deployment can thus be automatic rather than user-driven, using the same deployment model as antivirus signatures.

These desirable features motivated us to explore a Shield-like approach for removing exploits from web pages. At first glance, Shield can filter passive HTML files by treating HTML as a protocol on top of HTTP. The challenge lies in active code within a web page, which can change the page's contents at runtime. Thus, attackers could easily evade Shield filters by generating and injecting exploits via script code, possibly using obfuscation to hide their intent. Statically determining whether a script will exploit a vulnerability is undecidable.

To cleanse active web content, our approach is to instead rewrite HTML pages and their embedded scripts into safe equivalents, before they are rendered by the browser. The safe equivalent pages contain logic for applying run-time checks to the web content. To this end, we have designed *BrowserShield*, a system that instruments HTML and JavaScript code and that admits policies for changing web program behavior. One such policy is a vulnerability signature, which can prevent exploits of a known browser vulnerability, even when created by dynamically generated script code. Figure 8.1 shows an overview of the system, which transforms HTML and JavaScript using a set of policies as input. We focus on JavaScript

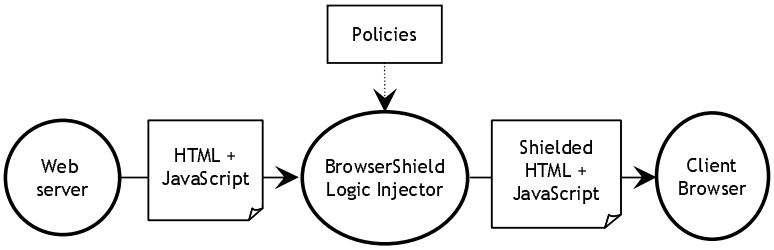

Figure 8.1: BrowserShield transforms HTML and JavaScript into safe equivalents, using a set of policies as input.

because it is the predominant form of script code on the web; a full-fledged system would require rewriting or disabling other types of active code as well.

Interposing on program behavior via code rewriting has been used in other contexts, but several properties of JavaScript make this a more difficult challenge for the browser space. For example, code rewriting has been used to isolate faults of software extensions [126], and Java-bytecode rewriting has been used to enable security policies [36, 114]. In contrast, JavaScript is a prototype-based language, and the combination of this with JavaScript's scoping rules, implicit garbage collection, and pervasive reflection requires a number of techniques not needed by previous work on code rewriting.

We have designed BrowserShield to adhere to well established principles for protection systems: complete interposition of the underlying resource (i.e., the HTML document tree), tamper-proofness and transparency [13, 36, 107]. In addition, BrowserShield is a general framework that supports applications other than vulnerability-driven filtering. For example, we have authored policies that add UI invariants to prevent certain phishing attempts.

Because BrowserShield protects web browsers by transforming their inputs and not the browsers themselves, its logic injector can be deployed at client or edge firewalls, browser extensions, or web publishers that republish third-party content such as ads.

Deployments outside the server or browser can be considered in-flight page changes, as we discuss in Chapter 6. However, much like Blue Coat WebFilter [24], BrowserShield may be viewed as a positive change to increase client security, and it aims to interpose on

program behavior transparently without introducing annoyances, bugs, or vulnerabilities. In theory, BrowserShield's complete interposition can render it transparent to web tripwires, as we discuss in Section 8.4.3.

In this work, we have implemented a prototype of the BrowserShield system that injects the rewriting logic into a web page at an enterprise firewall, acting as an in-flight page change. The rewriting logic is then executed by the browser at rendering time. Our prototype can transparently render many familiar websites that contain JavaScript (e.g., www.google.com, www.cs.washington.edu, www.mit.edu). We also successfully translated and ran a large intranet portal application (Microsoft SharePoint) that uses 549 KB of JavaScript libraries.

We chose the firewall deployment scenario because it offers the greatest manageability benefit, as BrowserShield updates can be centralized at the firewall. This protects all client machines in the organization without any BrowserShield-related installation at either clients or web servers. The main disadvantage of this deployment scenario is that firewalls have no visibility into end-to-end encrypted traffic. Nevertheless, commercial products [110] already exist that force traffic crossing the organization boundary to use the firewall (instead of a client within the organization) as the encryption endpoint, trading client privacy for aggregate organization security.

In scenarios where end-to-end encryption is critical, the browser extension and web publisher deployment scenarios can transparently handle encrypted traffic. While a client-side deployment of BrowserShield would require policy updates for every client, this approach still offers advantages over relying on software patches. In particular, BrowserShield policies do not require application restarts, and they are easy to roll back if they prove to be faulty. Thus, the policies can be automatically distributed much like antivirus signatures.

We evaluate the effectiveness of the BrowserShield design and the performance of our implementation. Our analysis of recent IE vulnerabilities shows that BrowserShield significantly advances the state-of-the-art; existing firewall and antivirus techniques alone can provide patch-equivalent protection for only 1 of the 8 IE patches from 2005, but combining these two with BrowserShield is sufficient to cover all 8. We evaluated BrowserShield's performance on real-world pages containing over 125 KB of JavaScript. Our evaluation shows

```
var ms04_040_policy = function (tag) {
  var len = 255; // not the actual limit

  // Look for long attribute values
  if ((contains("name", tag.attrs) &&
       tag.attrs["name"].length > len) &&
      (contains("src", tag.attrs) &&
       tag.attrs["src"].length > len)) {
    // Remove all attributes to be safe
    tag.attrs = [];
    // Return false to indicate exploit
    return false;
  }
  // Return true to indicate safe tag
  return true;
}
```

Figure 8.2: JavaScript code snippet to identify exploits of the MS04-040 vulnerability.

a 22% relative increase in firewall CPU utilization, and client rendering latencies that are comparable to the original page latencies for most pages.

The rest of this chapter is organized as follows. In Section 8.2, we describe a typical browser vulnerability that we would like to filter. We discuss the design of BrowserShield in Section 8.3, and we present BrowserShield's JavaScript rewriting approach in detail in Section 8.4. We discuss how to author and register policy functions in Section 8.5, listing additional potential applications in Section 8.6. We describe our implementation in Section 8.7, and we evaluate BrowserShield's effectiveness and performance in Section 8.8. We then conclude in Section 8.9.

8.2 Browser Exploits

As an example to motivate vulnerability-driven filtering, we consider *MS04-040*: the HTML Elements Vulnerability [83] of IE from December, 2004. In this case, IE had a vulnerable buffer that was overrun if both the `name` and the `src` attributes were too long in an `iframe`, `frame`, or `embed` HTML element.

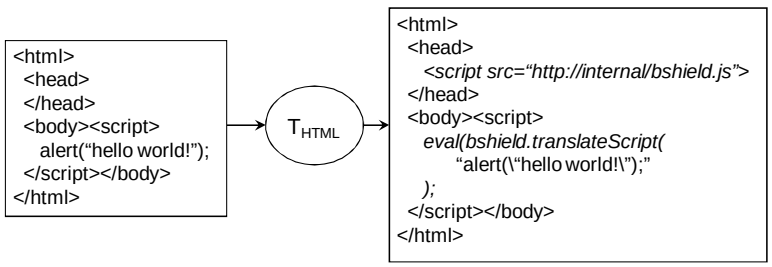

Figure 8.3: T_{HTML} Translation

Figure 8.2 shows a corresponding snippet of JavaScript code that can be used to identify and to remove exploits of this vulnerability. As input, the function takes an object representing an HTML tag, including an associative array of its attributes. When invoked on an `<iframe>`, `<frame>` or `<embed>` tag, the function determines whether the relevant attributes exceed the size of the vulnerable buffer.

The goal of BrowserShield is to take such vulnerability-specific filtering functions as policies and apply them to all occurrences of vulnerable content whether they are in static HTML pages or dynamically generated by scripts. The framework could react in many ways to detected exploits; our current system simply removes the exploit from the HTML before rendering. Vulnerability-driven filtering, used as a patch alternative or intermediary, should prevent all exploits of the vulnerability (i.e., zero false negatives), and should not disrupt any exploit-free pages (i.e., zero false positives). We design BrowserShield to meet these requirements.

8.3 BrowserShield Overview

The BrowserShield system consists of a JavaScript library that translates web pages into safe equivalents and a logic injector (such as a firewall) that modifies web pages to use this library.

BrowserShield uses two separate translations along with policies that are enforced at run-time. The first translation, T_{HTML}, translates the HTML. It tokenizes an HTML page, modifies the page according to its policies (such as the one depicted in Figure 8.2)

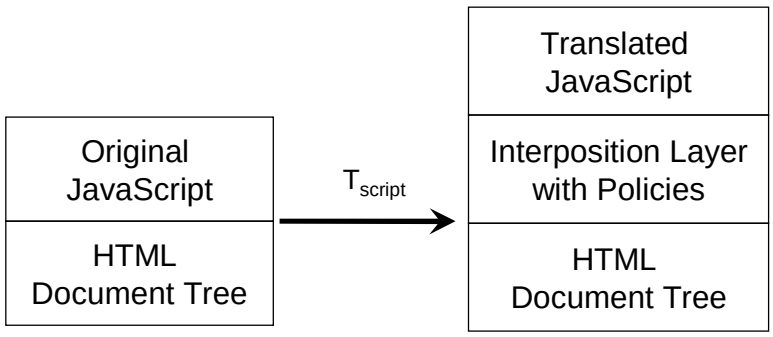

Figure 8.4: T_{script} Translation

and wraps the script elements so that the work of the second translation, T_{script}, will be deferred to page rendering at the browser. T_{HTML} is depicted in Figure 8.3 using `bshield.translateScript(...)` to invoke T_{script}. T_{script}, as depicted in Figure 8.4, parses and rewrites JavaScript to access the HTML document tree through an interposition layer. This layer regulates all accesses and manipulations of the underlying document tree, recursively applies T_{HTML} to any dynamically generated HTML, and recursively applies T_{script} to any dynamically generated script code. Additionally, the interposition layer enforces policies, such as filtering exploits of known vulnerabilities.

Since users can choose to disable scripting in their web browsers, we must ensure BrowserShield protects such users even if our JavaScript library is not executed. We handle such clients by applying T_{HTML} at the logic injector, independent of the user's browser. Any modifications due to T_{script} are still in place, but disabling scripts has made them irrelevant, along with the original script code.

Browser extensions, such as ActiveX controls, can also manipulate the document tree. The security model for such extensions is that they have the same privileges as the browser, and thus we focus on interposing between script and the extensions, not between the extensions and the document tree. This allows BrowserShield to prevent malicious scripts from exploiting known vulnerabilities in trusted browser extensions.

We have designed BrowserShield to adhere to the following established principles for protection systems [13, 36, 107]:

- *Complete interposition*: All script access to the HTML document tree must be mediated by the BrowserShield framework.

- *Tamper-proof*: Web pages must not be able to modify or tamper with the BrowserShield framework in unintended ways.

- *Transparency*: Apart from timing considerations and reasonable increases in resource usage, web pages should not be able to detect any changes in behavior due to the BrowserShield framework. The sole exception is for policy enforcement (e.g., the behavior of a page containing an exploit is visibly modified).

- *Flexibility*: We desire the BrowserShield framework to have a good separation between mechanism and policy, to make the system flexible for many applications.

8.4 Language-Based Interposition

We now give a detailed discussion of the BrowserShield script library. We describe how rewriting JavaScript differs from other languages, and which mechanisms are necessary to construct a complete interposition layer for it.

While much previous work uses code rewriting for interposition [36, 37, 38, 126], our approach is heavily influenced by the fact that our code lives in the same name space as the code it is managing, and also by several subtleties of JavaScript. First, JavaScript is a prototype-based language [121], not a class-based language like Java. In prototype-based languages, objects are created using other objects as prototypes. They can then be modified to have a different set of member variables and methods. JavaScript also has no static typing, so different data types can be assigned to the same variable, even for references to functions and object methods. Second, scoping issues must be dealt with carefully, as assigning a method to a new object causes any use of the `this` keyword in the method to bind to the new object. Thus, any interposition mechanisms must ensure that `this` is always evaluated in the intended context. Third, JavaScript uses a garbage collector that is not exposed to the language, creating challenges for freeing interposition state when the

Construct	**Original Code**	**Rewritten Code**
Function Calls	`foo(x);`	`bshield.invokeFunc(foo, x);`
Method Calls	`document.write(s);`	`bshield.invokeMeth(` `  document, "write", s);`
Object Properties	`obj.x = obj.y;`	`bshield.propWrite(obj, "x",` `  bshield.propRead(obj, "y") );`
Object Creation	`var obj = new Foo(x);`	`var obj = bshield.createObj(` `  "Foo", [x]);`
`with` Construct	`with (obj) { x = 3; }` `// x refers to obj.x`	`(bshield.undefined(obj.x) ? x = 3 :` `  bshield.propWrite(obj, "x", 3));`
Variable Names	`bshield = x;`	`bshield_ = x;`

`in` Construct	`for (i in obj) {...}`	`for (i in obj) {`
		`  if (i=="bshieldProp") continue;`
		`  ...`
		`}`

Table 8.1: Sample Code for BrowserShield Rewrite Rules.

associated objects are collected. Fourth, the language has pervasive reflection features that let a script explore its own code and object properties.

As a result of these JavaScript subtleties, BrowserShield must use a series of interposition mechanisms: *method wrappers*, *new invocation syntax*, and *name-resolution management*. We justify and describe these mechanisms in the following subsections, organized by our goals for the framework.

8.4.1 Complete Interposition

To provide complete interposition, BrowserShield must mediate all possible accesses and manipulations allowed by the DOM over the HTML document trees (including script elements). In this subsection, we detail how we achieve this using script rewriting to interpose on function calls, object-method calls, object-property accesses, object creation, and control constructs. These interposition points are sufficient to mediate script access to the DOM. We summarize our rewriting rules in Table 8.1.

Function and Object-Method Calls JavaScript supports the definition of global functions and methods on objects. Both functions and methods can be passed by reference and aliased, so that multiple names could refer to the same executable code. The sole difference between functions and methods is that the `this` keyword will be bound to the enclosing object for methods. All other references are bound at the time the function or method is created.

There are two techniques to rewrite function or method calls for interposition: callee rewriting or caller rewriting. Below, we evaluate each and find that both techniques are necessary.

In *callee* rewriting, the original function or method definition is first saved under a different name, and then the original function or method is redefined to allow interception before calling the saved original. We call the redefined function the *wrapper*. For example, a `foo` method on an object could be renamed to `fooOriginal`, and a new `foo` method could check the arguments to the method before invoking `fooOriginal`. The benefit of callee rewriting is that the rewritten code is localized — only the functions or methods of interest

are modified, and not their invocations throughout the code. However, callee rewriting does not work in cases where functions or methods cannot be redefined.

In *caller* rewriting, the invocation is rewritten to call an interposition function without changing the original function's definition. The interposition function looks up the appropriate interposition logic based on the identity of the target function or method. For example, `foo(3)` could be rewritten as `invokeFoo(3)`, which first checks the arguments before invoking `foo`. Although caller rewriting causes more pervasive code changes, it can interpose on those functions or methods that cannot be overwritten.

In BrowserShield, we have to use a hybrid of both approaches to accommodate the previously mentioned JavaScript subtleties.

Specifically, JavaScript contains some native functions that cannot be redefined (e.g., `alert` in Internet Explorer), which necessitates caller rewriting. The first row of Table 8.1 shows how BrowserShield indirectly invokes a function with its list of parameter values by passing it to `bshield.invokeFunc(func, paramList)`, where `bshield` is a global object that we introduce to contain BrowserShield library code.

However, using caller rewriting alone for interposing on method calls requires maintaining references to state otherwise eligible for garbage collection. That is, caller rewriting requires maintaining a map from functions and methods of interest to their associated interposition logic. Storing this map in a global table would require maintaining a reference to methods of interest on every object ever created. This is because each object may correspond to a distinct prototype requiring distinct interposition logic. These global table references would prevent reclamation of objects otherwise eligible for garbage collection, possibly causing pages that render normally without BrowserShield to require unbounded memory. To avoid this, BrowserShield uses callee rewriting to maintain the necessary interposition logic on a wrapper for each method, allowing unused state to be reclaimed.

It might seem tempting to maintain this interposition logic as a property on the object, rather than using wrappers. Unfortunately, aliases to the interposed method can be created, and these aliases provide no reference to the object containing the interposition logic. For example, after "`f = document.write`", any interposition logic associated with `document.write` is not associated with `f`; finding the logic would require a global scan of

JavaScript objects. This justifies using wrappers for methods that require interposition logic. The wrappers are installed by replacing the original method with the wrapper and saving the original method as a property on the wrapper (which is itself an object). Because we interpose on object-property accesses, object creation, and method invocations, we can install wrappers when an object is first created or used.

Thus far we have justified caller rewriting for functions and callee rewriting for methods. Because JavaScript allows functions to be aliased as methods on objects (e.g., "`obj.m = eval`"), we also must perform caller rewriting for method calls. The rewritten method invocations can then check for potential aliased functions.

JavaScript scoping introduces additional complexity in method interposition. The original method cannot simply be called from the method wrapper. This is because the original method is now a property of the wrapper, so the `this` keyword binds to the wrapper instead of the intended object. To avoid this problem, we use a *swapping* technique: The wrapper temporarily restores the original method during the wrapper execution, and then reinstalls the wrapper for the object method before the wrapper returns.

The first step in swapping is to restore the original method. One challenge here is that the method name may not be the same as when the method wrapper was installed, so we may not know where to place the original method. This is because methods can be reassigned to different names. For example, `document.write` may be aliased to `otherObject.foo`. The `document.write` wrapper needs to be informed of the method name being called. We solve this problem again with caller rewriting. In the rewritten method invocation syntax `invokeMeth(obj, methName, paramList)`, we pass the name of the method to the method wrapper, which performs the swap. Continuing the example, the rewritten method call is `invokeMeth(otherObject, "foo", ...)`, allowing `document.write` to swap to `otherObject.foo`.

The swapping process requires an additional check to handle recursive methods. As described above, a recursive call to a method would bypass the method's wrapper, which is swapped out after the first call. Instead, the `bshield.invokeMeth` method first checks to see if the method being called has a wrapper that has already been swapped out. If so, `invokeMeth` invokes the wrapper again, ignoring any swapping logic until the original

recursive call completes. Because JavaScript is single threaded, we have not needed to handle concurrency during this process.

Object Properties The HTML document tree can be accessed and modified through object-property reads and writes in JavaScript. For example, the HTML in a page can be modified by assigning values to `document.body.innerHTML`, or an individual script tag can be changed by modifying its `text` property. To interpose on such actions, BrowserShield replaces any attempts to read or write object properties with calls to `bshield`'s `propRead(obj, propName)` and `propWrite(obj, propName, val)` methods, as shown in Table 8.1. We apply this rewriting rule to *all* object-property reads and writes, because it is not possible to know statically whether an arbitrary read or write will require interposition logic. Instead, we use the object's identity at run-time to check whether an assignment will create new HTML or script code. If so, `propWrite` applies T_{HTML} or T_{script} to the new code as needed. As part of the identity check, we can determine if an object is part of the HTML document tree by calling certain JavaScript library functions. Note that we ensure BrowserShield uses the authentic JavaScript library functions (and not malicious replacements) by creating private aliases of the functions before any script code begins to run.

Interposing on property accesses is necessary for two additional reasons. First, it is required for installing wrappers when an object is first accessed. Using this technique, we avoid installing wrappers on DOM objects unless the particular method requiring interposition is about to be accessed. Second, while wrappers are swapped out during method execution, `propRead` must ensure that any attempts to access the original method are redirected to the swapped-out wrapper.

Object Creation For some objects, BrowserShield must ensure that method wrappers are initialized on each object as it is created. For example, it is straightforward to check the type of DOM objects at any point during their lifetime, making it easy to then decide on an appropriate interposition policy. In contrast, determining the appropriate interposition policy for the output of the `ActiveXObject` constructor is best done at creation time, as otherwise important context has been lost. To achieve this, we rewrite the instantiation

of new objects to use `bshield`'s `createObj(name, paramList)` method. The `createObj` method is also responsible for interposing on the JavaScript `Function` constructor, which can create new executable functions from its parameters as follows:

```
f = new Function("x", "return x+1;");
```

In this case, `createObj` applies T_{script} to the arguments before instantiating the function.

Control Constructs For control constructs (e.g., `if-then` blocks, loops, etc.), the bodies of the constructs are translated by T_{script}. The bodies of traditional function constructors (e.g., `function foo() {...}`) are translated by T_{script} as well.

JavaScript's `with` construct presents a special case, as it has the ability to modify scope. As shown in Table 8.1, free variables within a `with` block are assumed to refer to properties on the designated object, unless such properties are undefined. This construct is effectively "syntactic sugar" for JavaScript, and we handle this case with a syntactic transformation.

8.4.2 Tamper-Proof

Preventing scripts from tampering with BrowserShield is challenging because BrowserShield logic lives in the same name space as the code it is managing. To address this, we use *name-resolution management* to ensure that all BrowserShield logic is inaccessible. This entails renaming certain variables and modifying the output of reflection in some cases.

Variable Names In the common case, variable names in a script can remain unchanged. However, we make the `bshield` name inaccessible to scripts to prevent tampering with the global BrowserShield data structure.

To do this, we rename any variable references to `bshield` by appending an underscore to the end of the name. We also append an underscore to any name that matches the `bshield(_*)` regular expression (i.e., that begins with `bshield` and is optionally followed by any number of underscores). Note that JavaScript places no limit on variable name length, so appending characters will not cause a conflict.

Reflection Reflection in JavaScript allows script code to explore the properties of objects as well as its own code, using two pervasive language features: the syntax for accessing object properties (such as `myScript.text` or `myScript[i]`), and the JavaScript `in` construct.

In the first case, BrowserShield must hide some object properties, because it maintains per-object interposition state (as described in Section 8.4.3) on some objects. Such state is stored on a `bshieldProp` property of the object, which we hide using property-access interposition. Specifically, if a call to `propRead` or `propWrite` attempts to access a property name beginning with `bshieldProp`, we simply append an underscore to the name, thus returning the property value that the original script would have seen. Since array indices can also be used to access object properties, we must return the appropriate value for the given index.

In the second case, the `in` construct allows iteration through all of an object's properties by name. For example, `for (i in obj)` sets up a loop that iterates through each property name in `obj`. The `bshieldProp` property of an object must be hidden during the iteration if it is present. Thus, BrowserShield inserts a check as the first line of the iteration loop, jumping to the next item if the property name is `bshieldProp`. This is accomplished using the rewrite rule shown in Table 8.1.

8.4.3 Transparency

The BrowserShield framework must also ensure its presence is transparent to the original script's semantics. The techniques for preventing tampering described in Section 8.4.2 contribute to this goal by making BrowserShield inaccessible. Transparency additionally requires that we present to scripts the context they would have in the absence of BrowserShield.

Shadow Copies Scripts can access both their own script code and HTML, which BrowserShield modifies for interposition. To preserve the intended semantics of such scripts, BrowserShield retains a "shadow copy" of all original code before rewriting it. The shadow copy is stored on a `bshield` property of the object. Interposition on property reads and writes allows the shadow copy to be exposed to scripts for access and modification.

Shadowing translated HTML requires additional care. During T_{HTML} transformation, a policy may rewrite static HTML elements. We must similarly create shadow copies for such translated HTML elements, but we cannot directly create a JavaScript object in HTML to store the shadow copy. Thus, we persist the shadow copy to a `bshield` HTML tag attribute during T_{HTML}, which is later instantiated by the BrowserShield library. For example, a policy function that rewrites link URLs may modify the `href` attribute of `<a>` tags during the T_{HTML} transformation. Then, the persisted shadow copy looks like this:

```
<a href="http://translatedLink"
   bshield="{href:'http://originalLink'}">
```

When BrowserShield looks for the `bshield` property of the DOM object corresponding to this tag, it interprets this string into an actual `bshield` property with a shadow copy for the `href` attribute.

Because scripts can only interact with shadow copies of their code and not modified copies, our transformations are not required to be idempotent. That is, we will never apply T_{HTML} or T_{script} to code that has already been transformed.

Preserving Context The JavaScript `eval` function evaluates a string as script code in the current scope, and any occurrence of the `this` keyword in the string is bound to the current enclosing object. Thus, if `eval` were to be called from within `bshield.invokeFunc`, the `this` keyword might evaluate differently than in the original context.

For this reason, the rewriting rule for functions is actually more complex than shown in Table 8.1. Instead, the rewritten code first checks if the function being invoked is `eval`. If so, the parameter is translated using T_{script} and then evaluated in the correct context; otherwise, `invokeFunc` is called as described before. Thus, the code is rewritten as follows:

```
bshield.isEval(bshield.func = foo) ?
  eval(bshield.translate(x)) :
  bshield.invokeFunc(bshield.func, x);
```

Note that the function expression `foo` is assigned to a temporary state variable on the `bshield` object, so that the expression is not evaluated a second time in the call to `invokeFunc`.

This check is a special case that is only needed for `eval`, because `eval` is the only native function in JavaScript that accesses `this`. Other native functions, such as `alert` or `parseInt`, do not access `this`, and can be evaluated within `invokeFunc`.

External Requests Scripts in web programs may make network requests for data using the `XmlHttpRequest` (XHR) facility. While our current BrowserShield prototype does not interpose on XHRs, it is generally important to ensure that any HTML or JavaScript files requested via XHR are not rewritten by BrowserShield before being delivered to the web program. This could be achieved by flagging XHRs in a way that instructs the BrowserShield rewriter to deliver the results unmodified. BrowserShield would also have to ensure such requests do not go directly to the browser's cache, since the browser's cache may contain rewritten copies of the files. This could be achieved by rewriting the URL in the XHR to avoid a cache hit. Using these techniques, web programs would receive the data they expect, unaltered by BrowserShield.

Note that adding this support would render BrowserShield transparent to the web tripwires discussed in Chapter 6. This is consistent with our experience that web tripwires cannot detect all page changes.

8.4.4 Flexibility

The final goal of BrowserShield is to support flexible policy enforcement. This can be achieved by separating mechanism from policy: Our mechanism consists of the rewrite rules for translating HTML and script code, and our policy consists of the run-time checks invoked by the rewritten code. Some run-time checks are critical for complete interposition, such as applying T_{script} to any string passed to `eval` or the `Function` constructor, or applying T_{HTML} to any string passed to `document.write` or assigned to `document.body.innerHTML`. These checks are always applied, regardless of what policy is in place.

Because the interposition logic itself is policy-driven, our system can be made incrementally complete. For example, if an undocumented API is discovered that can manipulate the document tree, we simply add a new policy to interpose on this API.

The remaining run-time checks are used for enforcing flexible policies, such as the MS04-040 vulnerability filter in Figure 8.2. Such policy functions are separate from the remainder of the BrowserShield code, and they can be updated and customized based on the intended application. We provide a more detailed discussion of policy functions in the next section.

8.5 Policies

BrowserShield policies are functions that can be registered at any interposition point in the framework. These functions are given the chance to inspect and modify script state during execution, allowing them to enforce invariants on script behavior in a comprehensive way. In this section, we discuss how policy functions are written, how to register them with the BrowserShield framework, and how they can be distributed.

8.5.1 Writing Policies

Policy functions are implemented in JavaScript. They are the only code outside the BrowserShield framework that runs without being translated, and thus they have access to the BrowserShield framework itself. Because these policy functions are given control over the page and are responsible for protecting the browser from exploits, they are part of the trusted computing base. Thus, we have not established mechanisms to prevent interference between the policies and the framework itself.

Each policy function takes in arguments relevant to the interposition point at which it is registered, such as the arguments to a JavaScript function or a representation of an HTML tag. The policy function inspects these arguments and determines whether they should be modified before the intended action is performed. It has access to the entire state of the page and the BrowserShield framework itself, in case additional context is required. We also allow policy writers to introduce new global state and functions as part of the global `bshield` object, or local state and methods for all objects or for specific objects.

For vulnerability-specific policies, there are many reasonable reaction strategies. Our current prototype is configured to strip the offending code from the page, to prevent an exploit from occurring. Optionally, BrowserShield could instead redirect to a page with an error message, or it could notify an administrator of an attempted exploit if desired.

```
bshield.addHTMLTagPolicy("iframe", ms04_040_policy);
bshield.addHTMLTagPolicy("frame", ms04_040_policy);
bshield.addHTMLTagPolicy("embed", ms04_040_policy);
```

Figure 8.5: JavaScript code to register policy function for the MS04-040 vulnerability.

8.5.2 Policy Registration

The BrowserShield framework exposes several hooks for registering policy functions at interposition points. These points include when HTML tags are encountered, when functions and methods are called, and when object properties are read and written. By registering policy functions at these hooks, policy authors can comprehensively restrict or modify the behavior of any HTML or script code in a web document.

We describe the API for registering policy functions below. The registration functions are defined on the `bshield` object and are invoked as part of BrowserShield's initialization logic. In general, the registration functions take in a representation of the interposition point, such as the name of an HTML tag, along with a reference to the policy function itself. As BrowserShield encounters each interposition point at runtime, it invokes the policy functions registered at that point.

- `addHTMLTagPolicy(tagName, policy)` This hook allows a policy function to be registered with a given HTML tag name. Whenever BrowserShield encounters a tag with the given name on a page, it passes a JavaScript object representing the tag and its attributes to the policy function. This occurs before the tag is rendered by the browser, allowing the policy to change the tag if necessary. In this case, the policy function returns a boolean value indicating whether the tag was left unmodified.

  For example, the code in Figure 8.5 would register the *MS04-040* policy function in Figure 8.2.

  The tags are presented as part of a token stream of tags and text, without a full parse tree. It is possible for policy functions to reconstruct a subset of the parse tree to gain

additional context if necessary, although we have not yet encountered a need for this in the policies we have authored.

- `addJSConstructorPolicy(name, policy)` This hook causes BrowserShield to invoke the given policy function each time the constructor of the given name is invoked. BrowserShield passes the list of constructor arguments to the policy function, allowing it to alter them if necessary. The policy function is expected to return the newly constructed object. This allows the policy function to wrap methods on the object if necessary (e.g., to alter the behavior of method calls).

- `addJSFunctionPolicy(function, policy)` This hook causes BrowserShield to invoke the given policy function each time a particular function is invoked. This registration function takes in a pointer to the function of interest rather than its name, to avoid the need for an additional mechanism to track function aliasing, and because functions cannot simply be wrapped on first access. Each time the function is called, BrowserShield passes the arguments to the policy function. The policy function is expected to call the function and return the result, altering it if necessary.

- `addJSMethodPolicy(obj, methodName, policy)` This hook causes BrowserShield to invoke the given policy function each time a method with the given name on the given object is invoked. Because the method can be wrapped on first access, subsequent aliasing of the method is not a concern. Again, the policy function takes in the arguments for the method and returns the result of calling the method, altering them if necessary.

- `addJSPropWritePolicy(obj, policy)` This hook causes BrowserShield to invoke the given policy function on any write to a property of the given object. The policy function takes in the name of the property and the value to write. It can then apply the change, alter it, or block it as necessary. The policy function returns no value in this case.

- `addJSDOMPropWritePolicy(nodeName, policy)` This hook is similar to the object property write hook above, except that it applies to all DOM objects of a particular type. Rather than taking in a pointer to the object of interest, it takes in the name of a DOM node. This acts as an analog of a class name, allowing policies to be applied to any instance of a given type of DOM node.

Additional registration functions are possible, e.g. `addJSPropReadPolicy`, but we have not yet encountered a need for them in practice.

Finally, we note that multiple policy functions can be registered for the same function, HTML tag, or other hook. The BrowserShield framework simply calls the policy functions in the order that they were registered.

8.5.3 Policy Distribution

Policy functions are downloaded separately from the remainder of the BrowserShield framework, and they can be updated and customized based on the intended application. In the firewall deployment scenario, policy functions are downloaded to the firewall, and clients fetch them in a file separate from the BrowserShield framework. This allows the BrowserShield framework to be cached on the client, with a longer cache expiration time than the policy functions.

We envision that a centralized service provider or vendor could provide BrowserShield policy files. Alternatively, a firewall administrator could subscribe to policy files from a number of vendors. These vendors would be responsible for keeping vulnerability policies up to date and notifying subscribers when new updates are available. This scenario is analogous to distributing virus signatures for antivirus tools.

8.6 Applications

Our motivating application for BrowserShield is to prevent exploits of known browser vulnerabilities. The BrowserShield framework is general, however, and it can be useful for a wide variety of applications, related to browser security or not. In this section, we explore

how the framework can be used to translate links, aid in script debugging, sandbox scripts, and enable certain antiphishing measures.

Comprehensive Link Translation One application of BrowserShield outside security involves comprehensive link translation. URL rewriting is offered by some web proxies (e.g., EZproxy [122]) and web servers (e.g., Apache [15]), but in these cases rewriting can only be applied to links that appear in the HTML page as it is transferred over the network. Any links that are generated by script code on the page will not be rewritten. In contrast, BrowserShield policies can intercept any link on a page that a user attempts to visit, whether it appears in the original source of the page or not.

Comprehensive link translation can be useful in many contexts. For example, corporate intranet sites, such as Microsoft's SharePoint product [86], often contain links that are only accessible within the intranet. Providing remote authenticated access to employees may be desirable, but in such cases, links must be translated to use a proxy server. Using BrowserShield, all links and redirects in a page can be comprehensively identified and translated, whether they exist in the original HTML or are generated by scripts. This would allow sites such as SharePoint to easily export a view for remote authenticated access.

Similarly, a link translation policy may be useful for search engines that provide cached results, such as Google and MSN. Currently, users visit the cached copies of a search result when the original server is unavailable, but any embedded links and images continue to point to the unavailable server. A BrowserShield policy could translate HTML and script-generated links in cached pages to redirect into the search engine's cache, allowing users to browse entirely within the cache. If deployed at a search engine and combined with the vulnerability policies discussed in this chapter, this search engine cache could provide a safer search and browsing experience as well.

Script Debugging As another application outside the realm of security, BrowserShield policies can be used to provide extensible JavaScript debugging or profiling functionality. Many browsers have integrated JavaScript debuggers, but these debuggers have interfaces that vary across browsers and cannot always be extended with new features. Debugging

using BrowserShield policies could offer a browser-agnostic interface and the opportunity to register custom hooks or logging functionality within a page's code. Perhaps even more importantly, BrowserShield allows JavaScript developers to debug JavaScript code running on the client without requiring any trust from the client. BrowserShield debugging policies could easily include call traces, conditional breakpoints, and profiling information.

Much like Live Monitoring [74], BrowserShield policies could be used to instrument and analyze web applications. This would allow debugging, profiling, and maintaining web applications as they are used in practice.

Script Sandboxing BrowserShield policies can be useful for security related tasks beyond browser-exploit prevention. For example, many web sites today host user-contributed content, such as wikis, blogs, and social networking sites. These sites currently attempt to prevent any scripts from appearing in user-contributed content, because such scripts could leak private information to an attacker. Identifying and removing scripts can be challenging, however, often leaving these sites vulnerable to cross-site scripting attacks [26].

To address this, BrowserShield policies could be created to sandbox the user-contributed content on such sites. Such policies might prevent any scripts from running in certain areas of a page, as proposed by Jim et al. [70]. Alternatively, the authors of these community-driven sites might want to allow certain kinds of user-contributed dynamic content, such as simple scripts or function calls approved by the site authors. BrowserShield policies could be used to enforce script whitelists like those in Chapter 7, or to enforce a corresponding sandbox or otherwise restrict the actions that scripts on a page may take. Content Restrictions [81] is an example of a proposed set of restrictions on script actions. These approaches could help publishers authorize program behavior and prevent unauthorized actions.

Antiphishing Aids Finally, BrowserShield policies can be used to enforce invariants on the user interface of the browser, which can be useful to prevent deceptive behavior employed by some phishing sites. For example, some phishing sites use JavaScript to change the text in the status bar to hide the true destination of links, and some open new windows without location bars. BrowserShield policies can be used to prevent such behavior, ensuring that

the status bar and location bar always display authentic information. This can aid users in determining whether they are actually visiting the site they expect.

Overall, BrowserShield's comprehensive interposition makes it useful for security policies such as exploit prevention, but its general design supports a wide range of additional applications as well.

8.7 Implementation

We have implemented a prototype of BrowserShield as a service deployed at a firewall and proxy cache. Our prototype consists of a standard plugin to Microsoft's Internet Security and Acceleration (ISA) Server 2004 [82], along with a JavaScript library that is sent to the client with transformed web documents. The ISA plugin plays the role of the BrowserShield logic injector.

We implemented our ISA plugin in C++ with 2,679 lines of code. Our JavaScript library has 3,493 lines (including comments). Most of the ISA plugin code is devoted to parsing HTML, while about half of the JavaScript library is devoted to parsing HTML or JavaScript. This is a significantly smaller amount of code than in a modern web browser because it can leave out the majority of the browser's complexity (e.g., rendering, layout, etc). This implies that our trusted computing base is small compared to the code base we are protecting.

The ISA plugin is responsible for applying the T_{HTML} transformation to static HTML. The ISA plugin first inserts a reference to the BrowserShield JavaScript library into the web document. Because this library is distributed in a separate file, clients automatically cache it, reducing network traffic on later requests. T_{HTML} then rewrites all script elements such that they will be transformed using T_{script} at the client before they are executed. Figure 8.3 depicts this transformation; note that it does not require translating the JavaScript at the firewall.

In our implementation, the firewall component applies T_{HTML} using a streaming model, such that the ISA Server can begin sending transformed data to the client before the entire page is received. This streaming model also means that we do not expect the filter to be vulnerable to state-holding DoS attacks by malicious web pages.

One complexity is that BrowserShield's HTML parsing and JavaScript parsing must be consistent with that of the underlying browser. Any inconsistency will cause false positives and false negatives in BrowserShield run-time checks. For our prototype, we have sought to match IE's behavior through testing and refinement. If future versions of browsers exposed this logic to other programs, it would make this problem trivial.

When the browser starts to run the script in the page, the library applies T_{script} to each piece of script code, translating it to call into the BrowserShield interposition layer. This may sometimes require decoding scripts, a procedure that is implemented in publicly available libraries [124] and which does not require cryptanalysis, though we have not yet incorporated it in our implementation.

A final issue in T_{script} is translating scripts that originate in source files linked to from a source tag. T_{HTML} rewrites such source URLs so that they are fetched through a proxy. The proxy wraps the scripts in the same way that script code embedded directly in the page is wrapped. For example, a script source URL of `http://foo.com/script.js` would be translated to `http://rewritingProxy/ translateJS.pl?url=http://foo.com/ script.js`. T_{script} is then applied at the client after the script source file is downloaded.

8.8 Evaluation

We evaluate BrowserShield's effectiveness and performance by considering several questions. How much coverage can BrowserShield provide for actual browser vulnerabilities? How difficult is it to author vulnerability filters? How much overhead does BrowserShield introduce at firewalls and at end hosts? We find that BrowserShield provides substantial protection against browser exploits while introducing reasonable overhead in most cases.

8.8.1 Vulnerability Coverage

We first evaluated BrowserShield's ability to prevent exploits of all critical IE vulnerabilities for which Microsoft released patches in 2005 [84]. Of the 29 critical patches that year, 8 are for IE, corresponding to 19 IE vulnerabilities. These vulnerabilities fall into three classes: IE's handling of (i) HTML, script, or ActiveX components, (ii) HTTP, and (iii) images or other files. Table 8.2 shows how many vulnerabilities there were in each area, and whether

BrowserShield or another technology could provide protection equivalent to applying the software patch.

In the first class, BrowserShield can successfully handle all 12 of the vulnerabilities because its design focuses on HTML, script, and ActiveX controls. These include vulnerabilities where the underlying programmer error is at a higher layer of abstraction than a buffer overrun (e.g., a cross-domain scripting vulnerability).

In the second class, handling HTTP accounted for 3 of the 19 vulnerabilities. Perhaps surprisingly, 2 out of 3 of these vulnerabilities required BrowserShield in addition to an existing HTTP filter, such as Snort [116] or Shield [130]. This is because malformed URLs could trigger the HTTP-layer vulnerabilities regardless of whether the URL came over the network or was generated internally by the browser. BrowserShield is able to prevent the HTML/script layer from triggering the generation of these bad HTTP requests.

In the third class, processing images or other files accounted for the remaining 4 vulnerabilities. Patch-equivalent protection for these vulnerabilities is already available using existing antivirus solutions [112].

vulnerability		**protected by**		
type	**#**	**BrowserShield**	**HTTP filter**	**antivirus**
HTML, script, ActiveX	12	12	0	0
HTTP	3	2*	3*	0
images and other files	4	0	0	4

Table 8.2: BrowserShield Vulnerability Coverage. *Two of the HTTP vulnerabilities required both BrowserShield and an HTTP filter to provide patch-equivalent protection.

Because management and deployment costs are often incurred on a per-patch basis, we also analyze the vulnerabilities in Table 8.2 in terms of the corresponding patches. For the 8 IE patches released in 2005, combining BrowserShield with standard antivirus and HTTP

filtering would have provided patch-equivalent protection in every case, greatly reducing the costs associated with multiple patch deployments. In the absence of BrowserShield, antivirus and HTTP filtering would have provided patch-equivalent protection for only 1 of the IE patches.

8.8.2 Authoring Vulnerability Filters

To evaluate the complexity of vulnerability filtering, we discuss how to author filters for three vulnerabilities from three different classes: HTML Elements Vulnerability (MS04-040), COM Object Memory Corruption (MS05-037), and Mismatched DOM Object Memory Corruption (MS05-054).

We filtered for the MS04-040 vulnerability using the function shown in Figure 8.2, registered as shown in Figure 8.5.

COM object vulnerabilities typically result from IE instantiating COM objects that have memory errors in their constructors. IE patches for such vulnerabilities blacklist particular COM objects (identified by their `clsid`). Implementing an equivalent blacklist in BrowserShield requires adding checks for an HTML tag (the `OBJECT` tag) and sometimes a JavaScript function (the `ActiveXObject` constructor, which can be used to instantiate a subset of the COM objects accessible through the `OBJECT` tag). In the case of MS05-037, it does not appear to be possible to instantiate the vulnerable COM object using the `ActiveXObject` constructor. The `OBJECT` tag filter we use is conceptually similar to the function shown in Figure 8.2.

The MS05-054 vulnerability results when the `window` object, which is not a function, is called as a function in the outermost scope. Our interposition layer itself prevents `window` from being called as a function in the outermost scope since all function calls are mediated by BrowserShield with `invokeFunc`. Hence there is no need for a filter. Nevertheless, if this vulnerability had not depended on such a scoping constraint, we could simply have added a filter to prevent calling the object as a function.

To test the correctness of our vulnerability filters, we installed an unpatched image of Windows XP Pro within a virtual machine, and created web pages for each of the vulner-

abilities that caused IE to crash. Applying BrowserShield with the filters caused IE not to crash upon viewing the malicious web pages. We tested the fidelity of our filters using the same set of URLs that we used in our evaluation of BrowserShield's overhead (details are in Section 8.8.3). Under side-by-side visual comparisons, we found that the filters had not changed the behavior of any of the web pages, as desired.

8.8.3 Firewall Performance

We evaluated BrowserShield's performance by scripting multiple IE clients to download web pages (and all their embedded objects) through an ISA server running the BrowserShield firewall plugin. The ISA firewall ran on a Compaq Evo PC containing a 1.7GHz Pentium 4 microprocessor and 1 GB RAM. Because the experiments were conducted within a corporate intranet, our ISA server connected to another HTTP proxy, not directly to web sites over the internet. We disabled caching at our ISA proxy, and we fixed our IE client cache to contain only the BrowserShield JavaScript library, consistent with the scenario of a firewall translating all web sites to contain a reference to this library.

We ran 10 IE processes concurrently using 10 pages that IE could render quickly (so as to increase the load on the firewall), and repeatedly initiated each page visit every 5 seconds. We used manual observation to determine when the load on the ISA server had reached a steady state.

We chose these 10 pages out of a set of 70 URLs that are the basis for our client performance macrobenchmarks. This set is based on a sample of 250 of the top 1 million URLs clicked on after being returned as MSN Search results in Spring 2005, weighted by click-through count. Specifically, the 70 URLs are those that BrowserShield can currently render correctly; the remaining URLs in the sample encountered problems due to incomplete aspects of our implementation, such as JavaScript parsing bugs. We believe that a more complete parser implementation could handle the remaining pages.

We measured CPU and memory usage at the firewall, as shown in Table 8.3. CPU usage increased by about 22%, resulting in a potential degradation of throughput by 18.1%; all

resource	unmodified	browsershield
cpu utilization	15.0%	18.3%
virtual memory	317 MB	319 MB
working set	45.5 MB	46.6 MB
private bytes	26.3 MB	27.3 MB

Table 8.3: BrowserShield Firewall overheads. "Virtual memory" measures the total virtual memory allocated to the process; "working set" measures memory pages that are referenced regularly; "private bytes" measures memory pages that are not sharable.

aspects of memory usage we measured increased by negligible amounts. We also found that network usage increased only slightly (as we discuss further in Section 8.8.4).

8.8.4 Client Performance

We evaluated the client component of our BrowserShield implementation through microbenchmarks on the JavaScript interposition layer and macrobenchmarks on network load, client memory usage, and the latency of page rendering.

Microbenchmarks

We designed microbenchmarks to measure the overhead of individual JavaScript operations after translation. Table 8.4 lists our microbenchmarks and their respective BrowserShield slowdown. Our results are averages over 10 trials, where each trial evaluated its microbenchmark repeatedly, and lasted over half a second. For the first 11 microbenchmarks, the standard deviation over the 10 trials was less than 2%; in the last case it was less than 8%. The slowdown ratio was computed using the average time required per microbenchmark evaluation with and without the interposition framework.

Microbenchmarks 1-4 measure operations for which we expect no changes during rewriting, and hence no slowdown. The only slowdown we measure is in the case of the `if` statement. Further examination showed that the BrowserShield translation inserted a semi-colon

	operation	**slowdown**
1	i++	1.00
2	a = b + c	1.00
3	if	1.07
4	string concat ('+')	1.00
5	string concat ('concat')	61.9
6	string split ('split')	21.9
7	no-op function call	44.8
8	x.a = b (property write)	342
9	eval of minimal syntactic structure	47.3
10	eval of moderate syntactic structure, minimal computation	136
11	eval of moderate syntactic structure, significant computation	1.34
12	image swap	1.07

Table 8.4: BrowserShield Microbenchmarks. Slowdown is the ratio of the execution time of BrowserShield translated code and that of the original code.

(e.g., `var a = 1 (linebreak)` changed to `var a = 1; (linebreak)`). This results in a 7% slowdown.

Microbenchmarks 5-8 measure operations we expect to incur a slowdown comparable to an interpreter's slowdown. As detailed in Section 8.3, BrowserShield translation introduces additional logic around method calls, function calls, and property writes, leading to a slowdown in the range of 20x-400x. This slowdown is in line with good interpreters [105], but worse than what is achieved by rewriting systems targeting other languages, e.g., Java bytecode [36]. BrowserShield is paying a price for the JavaScript subtleties that previous rewriting systems did not have to deal with.

We were curious about the difference in slowdown between the two string methods; an additional experiment showed that the difference can be attributed to the JavaScript built-in `concat` method requiring about 3 times as much CPU as the built-in `split` method. Also, it is not surprising that property writes have a greater slowdown than function or

method calls because property writes need to both guard the BrowserShield namespace and interpose on writes to DOM elements (such as the text property of scripts).

Microbenchmarks 9-11 explore the overhead of translating JavaScript code of various complexity. The "eval of minimal syntactic structure" microbenchmark measures the cost of translating and then evaluating a simple assignment. The cause of the large slowdown is the additional work done by `eval` in the BrowserShield framework: parsing, constructing an AST, modifying the AST, and outputting the new AST as a JavaScript program. The two subsequent "eval of moderate syntactic structure" microbenchmarks measure the cost of translating and evaluating a simple `for(;;)` loop. This simply demonstrates that as the cost of the computation inside the simple loop increases, the cost of translating the code can decrease to a small fraction of the overall computational cost.

The last microbenchmark measures the overhead of performing a simple manipulation of the DOM – swapping two 35 KB images. This microbenchmark is designed to measure the relative importance of overheads in the JavaScript engine when the JavaScript is manipulating the layout of the HTML page. The JavaScript code to swap these two images requires two property writes (i.e., `img.src = 'newLink'`), and we described above how BrowserShield translation adds significant overhead to property writes. Nonetheless, the overall slowdown is less than 8%. In particular, the raw time to swap the image only increases from 26.7 milliseconds to 28.5 milliseconds. This suggests that even the large overhead that BrowserShield translation adds to some language constructs may still be quite small in the context of a complete web page.

In summary, BrowserShield incurs a significant overhead on the language constructs where it must add interpreter-like logic, but these overheads can be quite small within the context of the larger DOM manipulations in embedded scripts.

Macrobenchmarks

We designed macrobenchmarks to measure the overall client experience when the BrowserShield framework is in place. In particular, the macrobenchmarks include all the dynamic parsing and translation that occurs before the page is rendered, while the microbenchmarks

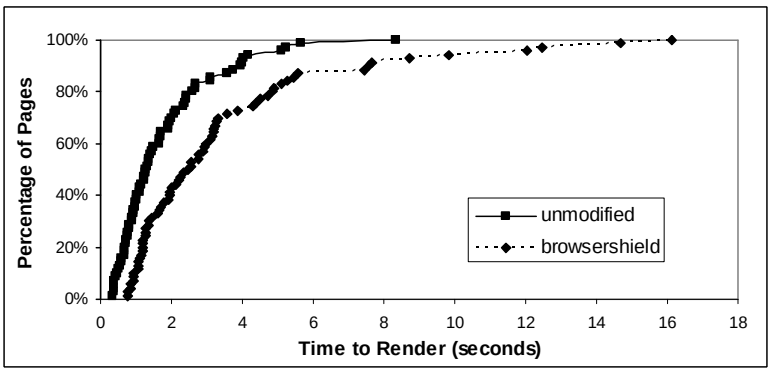

Figure 8.6: Latency CDF with and without BrowserShield.

primarily evaluated the performance of the translated code accomplishing a task relative to the untranslated code accomplishing that same task. To this end, we scripted an instance of IE to download each of the 70 web pages in our workload 10 times. For the same reasons given in our evaluation of the BrowserShield ISA component, we maintained that the only object in the IE cache was the BrowserShield JavaScript library. These caching policies represent a worst-case for client latency. This measurement includes the overhead of the three filters that we discussed in Section 8.8.1. We then repeated these measurements without the BrowserShield framework and translation.

We set a 30-second upper limit on the time to render the web page, including launching secondary (popup) windows and displaying embedded objects, but not waiting for secondary windows to render. We visually verified that the programmatic signal that rendering had completed indeed corresponded to the user's perception that the page had rendered. IE hit the 30-second timeout several times in these trials, and it hit the timeouts both when the BrowserShield framework and translation were present and when the framework and translation were absent. We did not discern any pattern in these timeouts, and because our experiments include factors outside our control, such as the wide-area network and the servers originating the content, we do not expect page download times to be constant over our trials. We reran the trials that experienced the timeouts.

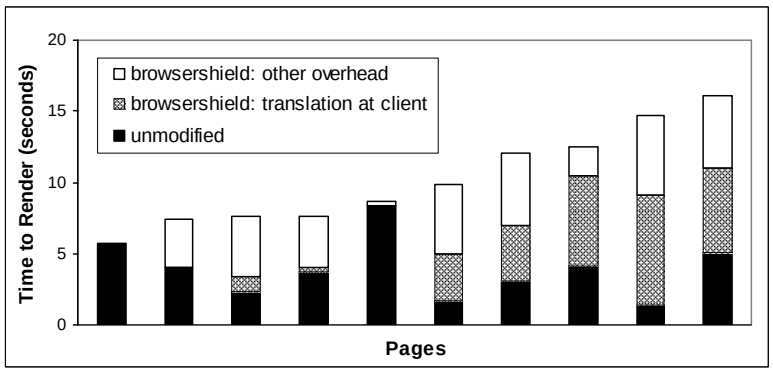

Figure 8.7: Breakdown of latency for slowest 10 pages under BrowserShield.

Figure 8.6 shows the CDF of page rendering with and without BrowserShield. On average BrowserShield added 1.7 seconds to page rendering time. By way of contrast, the standard deviation in rendering time without BrowserShield was 1.0 seconds.

In Figure 8.7, we further break down the latency for the 10 pages that took the most time to render under BrowserShield. They experienced an average increase in latency of 6.3 seconds, requiring 3.9 seconds on average without BrowserShield and 10.2 seconds on average with BrowserShield. Of this 6.3 seconds of increased latency, we found that 2.8 seconds (45%) could be attributed to the overhead of dynamically translating JavaScript and HTML within IE. We attribute the remaining overhead to effects such as the overhead of evaluating the translated code, and the time to modify the HTML at the firewall.

We broke down the latency of dynamic translation for both HTML and JavaScript into 2 parts each: time to parse the JavaScript/HTML into an AST and convert the modified AST back to a string, and the time to modify the AST. We found that the time to parse the JavaScript to and from a string was always more than 70% of the overall latency of dynamic translation, and it averaged 80% of the overall latency. Figure 8.8 shows the JavaScript parsing time versus the number of kilobytes. Fitting a least-squares line to this data yields an average parse rate of 4.1 KB of JavaScript per second, but there was significant variation; the slowest parse rate we observed was 1.3 KB/second. These results

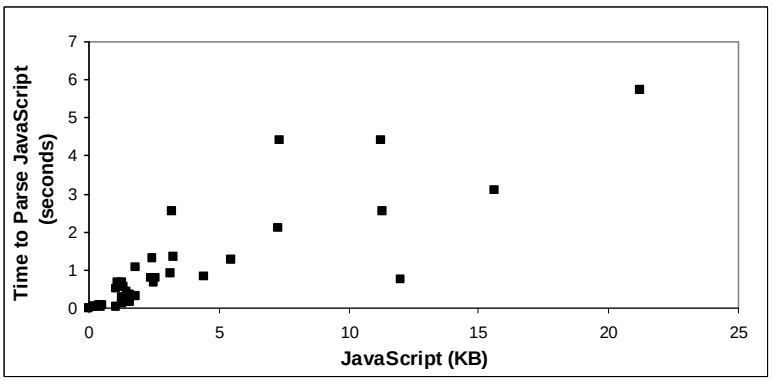

Figure 8.8: Latency of JavaScript parsing.

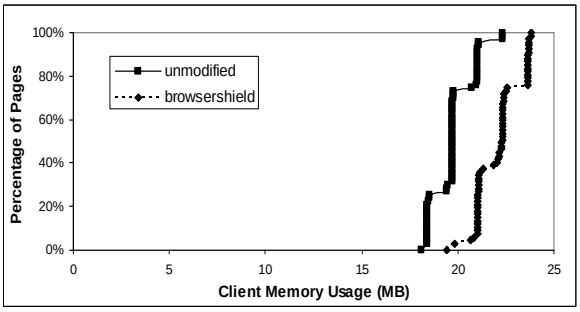

Figure 8.9: Memory Usage at Client.

suggest that more efficient parsing logic in the BrowserShield library and faster JavaScript engines in newer browsers may significantly reduce the client-side performance overhead.

Figure 8.9 shows the memory usage of page rendering with and without BrowserShield. We found that private bytes (memory pages that are not sharable) was the client memory metric that increased the most when rendering the transformed page. Private memory usage increased on average by 11.8%, from 19.8 MB to 22.1 MB. This increase was quite consistent; no page caused memory usage to increase by more than 3 MB.

We also measured the increased network load over a single run through the pages both with and without BrowserShield. We measured an average increase of 9 KB, less than the standard deviation in the network load over any individual trial due to background traffic

during our measurements. We expect BrowserShield rewriting to increase the network load only slightly, because the firewall just adds script wrappers, while the translation itself happens at the client.

Overall, we have shown that BrowserShield is both effective for preventing browser exploits and efficient in most cases in practice. BrowserShield's vulnerability filters can cover most modern browser vulnerabilities, and they can be authored with only modest effort. BrowserShield imposes a reasonable overhead on firewalls and is often quite efficient on the client. While it can occasionally introduce moderate overheads in client performance, current trends in browsers suggest that faster script engines may largely reduce this concern.

8.9 Summary

Web browser vulnerabilities have become a popular vector for attacks, partly due to the dangerous time window between patch release and application. Filtering exploits of these vulnerabilities is difficult by the active nature of web programs. In this chapter, we have presented BrowserShield as a way to filter exploits of vulnerabilities even from active web content.

BrowserShield provides a general framework that rewrites HTML pages and embedded scripts to enforce policies on run-time behavior. These policies may include vulnerability filters, but may also span other uses that were not foreseen by browser developers or program authors. BrowserShield can enforce these policies using complete interposition on the HTML document tree, in a transparent and tamper-proof manner.

Because BrowserShield transforms content rather than browsers, it supports deployment at clients, firewalls, or web publishers. Our evaluation shows that adding this approach to existing firewall and antivirus techniques increases the fraction of IE patches from 2005 that can be protected at the network level from 12.5% to 100%, and that this protection can be done with only moderate overhead.

Chapter 9

FUTURE WORK

This dissertation has presented several ways to improve web browsers and web content to better support the programs being deployed on today's web. These changes represent only the beginning of a promising opportunity to shape how future applications are developed and deployed. This chapter lays out a vision of where the web architecture could head, and it shows how the four principles detailed in this dissertation can continue to guide useful improvements to robustness and security as the web continues to evolve.

9.1 Future Web Browsers and Content

Far from the early usage model of the web, we are headed toward a web of applications that can free users from many of the ties to their particular devices. These applications will be personalized, universally accessible across devices, composable along clean interfaces, and protected by secure boundaries. The direction of this trend is evident from a number of emerging mechanisms and research proposals in both the browser and content spaces: support for powerful features and new communication interfaces in HTML5 [59], Gears [46], and MashupOS [61]; richer content environments in Flash, AIR [11], and Silverlight [85]; stronger isolation mechanisms in Chromium, OP [52], Gazelle [129], Native Client [139], and others; stronger user identification with OpenID [94], and many other proposals.

As a necessary part of this transition, we will see more of a convergence between operating systems and web browsers. Programs running within the browser or using browser technologies will have fewer UI distinctions from desktop applications, as foreshadowed by projects like Mozilla Prism [92]. Many web programs will also adopt richer features that take better advantage of client-side resources. New support for local storage, device access, and longer-lived applications will bring challenges for ensuring appropriate trust boundaries are maintained.

Operating systems themselves may also change to support these trends, providing more tailored support for browsers and web programs. In particular, lightweight but stronger isolation mechanisms than current processes may be desirable to support vast numbers of untrusted programs and content from the web. Each principal from the web may merit its own form of user account, for example, with OS-enforced access control on resource access. Inexpensive and securable communication channels will also be important.

Beyond the convergence that is likely to occur on typical desktops and laptops, the architecture for web programs must adapt to better target a diverse set of platforms and user agents. Phones, netbooks, televisions, and other devices are all gaining viable web access. However, each has very different capabilities than traditional browsers, ranging from form factor to input techniques to performance. Similarly, the popularity of Firefox extensions have shown a significant demand for customizable user agents, which can better serve the needs of particular users. Such variations in the underlying platform must not compromise the robustness or security of web programs.

These trends point to a number of important challenges for web browsers and web content that must be addressed. Trust models for applications will necessarily change as web programs gain richer features. How can these features be added without violating users' expectations and mental models? How dangerous will it become to visit a web page, and how manageable will it be to grant particular web programs greater privileges than others? How will diverse user agents affect the robustness and security of the platform, or impact the expectations of web program authors? It seems clear that evolution of the web standards, such as work on HTML5, will be critical to help all user agents address these questions, and that this evolution must be informed by research on long-term views of the web architecture and the problems it will face.

In the rest of this chapter, I discuss how the architectural principles laid out in this dissertation can help to guide us toward answers to many of these questions.

9.2 Web Program Definitions

Chapter 4 introduces the site instance abstraction as a backward compatible way to recognize the boundaries between web programs. This abstraction allows current browsers to

improve their support for web programs, but it has several limitations that suggest the need for improved definitions in the future. For example, site instances have a coarse boundary based on registry controlled domain names, which may encompass several logical web applications. They also do not reflect applications that may have portions hosted across domains, such as mashups.

In the long run, the Same Origin Policy is an imperfect approach to identify programs and govern their behavior. Future research should consider alternative policies that improve upon its limitations and inconsistencies, either from a clean slate approach or as incremental changes that web publishers can optionally adopt.

One example of such a direction could include using secrets, in the form of public key pairs, as proof of program authorship [104]. This eliminates dependence on the domain name system, which has its own security concerns such as DNS rebinding attacks [66]. It also provides greater flexibility for publishers to define program boundaries independently of hosting decisions.

Future research should also consider how web publishers can provide greater information to browsers about how their programs should be treated. Content Restrictions and BEEP provide two recent examples of work in this direction [81, 70], allowing the browser to enforce stricter policies when publishers deem it appropriate. As another example, publishers could instruct browsers to treat certain types of credentials as specific to a site instance rather than a site. Browsers could then provide more comprehensive defenses against CSRF attacks and authentication abuse, since malicious pages in different browsing instances would be unable to make requests with the user's credentials.

By providing clearer boundaries and additional metadata, web publishers will have more effective means to create independent programs at appropriate granularities, with behavior that both publishers and users can reason about.

9.3 Web Program Isolation

The process and security architecture of Chromium is designed to make it difficult for attackers to access the user's local resources, even if a rendering engine is exploited [21]. The architecture does not currently enforce secure isolation between web program instances,

however. Thus, if a rendering engine is exploited, attackers may breach the Same Origin Policy to access the user's other web accounts.

Ideally, the architecture of the browser would provide isolation of web principals even if a rendering engine were compromised. The OP and Gazelle browsers aim for this goal [52, 129], but they have compatibility constraints that make them difficult to deploy in practice. Future work may find secure and compatible approaches for isolating content from different web sites, resulting in a stronger notion of web principal.

Browser plug-ins also present a challenge here, if they are not well architected to support separate instances for separate principals. For example, Chromium currently uses one instance of Flash for the entire browser. The choice of content formats should not impact the isolation properties of the browser architecture, so it is worth pursuing plug-in architectures that can support stronger isolation between web program instances.

9.4 Authorizing Program Content

Chapter 6 presents the concerns that arise in practice when transferring web content over HTTP without integrity protection. While web tripwires offer one means for detecting unwanted changes to HTML, there are many situations in which preserving web content integrity is important, while confidentiality may not be. As more complex programs are delivered to the browser, for example, in-flight changes become more likely to introduce bugs and vulnerabilities. As a result, some web publishers may desire inexpensive integrity mechanisms in protocols like HTTP, without defeating network caches or introducing extra round-trip times like SSL or TLS.

Similarly, Chapter 7 argues for publishers to have the ability to authorize script code within a web program. Whitelisting approaches should provide a more complete solution, however, since any type of injected content poses a risk, particularly active content. Plug-in media is one prominent example, since it often has a scripting interface to the DOM. In many cases, stricter guarantees than whitelists may be desirable as well, if calling an authorized function in an unauthorized context presents a risk. In this sense, web publishers may seek the ability to enforce stronger isolation between code and data in programs that accept user

input. This enforcement could be provided on an opt-in basis, perhaps based on information provided in program manifests.

9.5 Enforcing Policies

The BrowserShield system introduced in Chapter 8 offers a flexible policy enforcement mechanism, but it faces many challenges by running outside the browser and in the same namespace as the content it modifies. If particular browsers interpret input differently than the BrowserShield rewriting logic expects, or if new DOM APIs are introduced in some browsers, then the interposition may be incomplete.

Ultimately, the browser itself may be a more appropriate location to enforce policies on web content behavior. This can ensure more complete coverage of browser APIs while eliminating any guesswork as to how the browser will interpret its input. Policies can then be provided to the browser itself, rather than to BrowserShield proxies.

This approach may also be useful for uniformly enforcing policies on not just HTML and JavaScript, but plug-in media and browser extensions as well. If plug-in instances can run on top of a browser-provided API rather than an OS-provided API, the browser can more effectively govern their behavior and mitigate the damage of plug-in exploits.

9.6 Summary

While this dissertation has provided a framework for enhancing the robustness and security of web programs, it is important to continue improving browsers and content towards this goal. Each of the architectural principles presented here can be further pursued to achieve a safer web platform in the future.

Chapter 10

CONCLUSION

We are in the midst of a significant change in the workload on the World Wide Web, which calls for us to revisit how the underlying architecture of the web is designed. The new web of personalized applications puts complex and active code within clients' web browsers, though browsers and content formats were intended for passive documents. This workload places browsers in the role of operating systems, and publishers in the role of application developers.

My thesis is that we can improve the robustness and security of programs on the web by adapting principles and mechanisms from operating systems. This dissertation has presented several contributions to better support web programs, based on four architectural principles:

- We can *identify program boundaries* by using site instances as backward compatible program abstractions.

- We can *isolate programs from each other* by redesigning the web browser architecture. Google Chrome now isolates site instances from each other in separate OS processes.

- We can *authorize program code* within a web program by detecting unwanted in-flight changes with web tripwires, and by introducing whitelists of script fragments that prevent arbitrary script injection.

- We can *enforce policies on program behavior* by rewriting web content to insert an interposition layer, with uses such as protecting users against exploits of known vulnerabilities.

Each of these contributions improve the robustness and security of web programs. Many opportunities remain, however. Future work in this space can continue to make the web a

safe and viable platform for deploying powerful, personalized applications to a wide array of devices.

BIBLIOGRAPHY

[1] Point Blank Security - Cross Site Scripting Blacklist 1. http://www.pointblanksecurity.com/xss/, 2002.

[2] Technical explanation of The MySpace Worm. http://namb.la/popular/tech.html, 2005.

[3] Greasemonkey. http://greasemonkey.mozdev.org/, 2006.

[4] Introducing JSON. http://www.json.org/, 2006.

[5] Vibrant Media - The In-Text Advertising Leader. http://www.vibrantmedia.com/, 2006.

[6] NebuAd / Service Providers. http://www.nebuad.com/providers/providers.php, August 2007.

[7] phpBB.com - Creating Communities. http://www.phpbb.com/, 2007.

[8] The Cloak: Free Anonymous Web Surfing. http://www.the-cloak.com, October 2007.

[9] Alexa Top 500 Global Sites. http://www.alexa.com/topsites, 2009.

[10] Ad Muncher. The Ultimate Popup and Advertising Blocker. http://www.admuncher.com/, 2007.

[11] Adobe. Rich Internet Applications - Adobe AIR. http://www.adobe.com/products/air/, 2008.

[12] Ben Anderson. fair eagle taking over the world? http://benanderson.net/blog/weblog.php?id=D20070622, June 2007.

[13] J. P. Anderson. Computer Security Technology Planning Study Volume II. ESD-TR-73-51, Vol. II, Electronic Systems Division, Air Force Systems Command, Hanscom Field, Bedford, MA, October 1972.

[14] Vinod Anupam and Alain Mayer. Security of Web Browser Scripting Languages: Vulnerabilities, Attacks, and Remedies. In *USENIX Security Symposium*, January 1998.

[15] Apache Foundation. The Apache HTTP Server Project. http://httpd.apache.org, 2007.

[16] W. A. Arbaugh, W. L. Fithen, and J. McHugh. Windows of Vulnerability: a Case Study Analysis. *IEEE Computer*, December 2000.

[17] Maksymilian Arciemowicz. phpBB 2.0.18 XSS and Full Path Disclosure. http://archives.neohapsis.com/archives/fulldisclosure/2005-12/0829.html, December 2005.

[18] Charles Babcock. Yahoo Mail Worm May Be First Of Many As Ajax Proliferates. http://www.informationweek.com/security/showArticle.jhtml?articleID=189400799, June 2006.

[19] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.

[20] Adam Barth, Juan Caballero, and Dawn Song. Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves. In *IEEE Symposium on Security and Privacy*, 2009.

[21] Adam Barth, Collin Jackson, Charles Reis, and Google Chrome Team. The Security Architecture of the Chromium Browser. Technical report, Stanford University, 2008. http://crypto.stanford.edu/websec/chromium/chromium-security-architecture.pdf.

[22] Steve Beattie, Seth Arnold, Crispin Cowan, Perry Wagle, and Chris Wright. Timing the Application of Security Patches for Optimal Uptime. In *Large Installation System Administration Conference (LISA)*, 2002.

[23] Robert Beverly. ANA Spoofer Project. http://spoofer.csail.mit.edu/, November 2006.

[24] Blue Coat. Blue Coat WebFilter. http://www.bluecoat.com/products/webfilter, October 2007.

[25] Nathaniel 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.

[26] CERT. CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests. http://www.cert.org/advisories/CA-2000-02.html, February 2000.

[27] Chinese Internet Security Response Team. ARP attack to CISRT.org. `http://www.cisrt.org/enblog/read.php?172`, October 2007.

[28] Steven M. Christey. Vulnerability Type Distribution in CVE. `http://www.attrition.org/pipermail/vim/2006-September/001032.html`, September 2006.

[29] Gerald Combs. Wireshark: The World's Most Popular Network Protocol Analyzer. `http://www.wireshark.org/`, October 2007.

[30] Richard S. Cox, Jacob Gorm Hansen, Steven D. Gribble, and Henry M. Levy. A Safety-Oriented Platform for Web Applications. In *IEEE Symposium on Security and Privacy*, 2006.

[31] Douglas Crockford. JSONRequest. `http://www.json.org/JSONRequest.html`, 2006.

[32] Rachna Dhamija and J. D. Tygar. The Battle Against Phishing: Dynamic Security Skins. In *Symposium on Usable Privacy and Security (SOUPS 2005)*, July 2005.

[33] Rachna Dhamija, J. D. Tygar, and Marti Hearst. Why Phishing Works. In *ACM Conference on Human Factors in Computing Systems (CHI)*, April 2006.

[34] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazieres, Frans Kaashoek, and Robert Morris. Labels and Event Processes in the Asbestos Operating System. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[35] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[36] Úlfar Erlingsson and Fred B. Schneider. IRM Enforcement of Java Stack Inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

[37] Úlfar Erlingsson and Fred B. Schneider. SASI Enforcement of Security Policies: A Retrospective. In *WNSP: New Security Paradigms Workshop*. ACM Press, 2000.

[38] David Evans and Andrew Twyman. Flexible Policy-Directed Code Safety. In *IEEE Symposium on Security and Privacy*, pages 32–45, 1999.

[39] Edward W. Felten and Michael A. Schneider. Timing Attacks on Web Privacy. In *ACM Conference on Computer and Communications Security (CCS)*, pages 25–32, 2000.

[40] Dave Ferguson. Netflix.com XSRF vuln. http://www.webappsec.org/lists/websecurity/archive/2006-10/msg00063.html, October 2006.

[41] Armando Fox and Eric A. Brewer. Reducing WWW Latency and Bandwidth Requirements by Real-Time Distillation. In *International Conference on World Wide Web (WWW)*, May 1996.

[42] Tal Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition based Security Tools. In *Network and Distributed System Security Symposium (NDSS)*, 2003.

[43] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Network and Distributed System Security Symposium (NDSS)*, 2004.

[44] Jesse James Garrett. Ajax: A New Approach to Web Applications. http://www.adaptivepath.com/publications/essays/archives/000385.php, February 2005.

[45] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer. A Secure Environment for Untrusted Helper Applications. In *USENIX Security Symposium*, July 1996.

[46] Google. Gears. http://gears.google.com, 2007.

[47] Google. Google Analytics. http://www.google.com/analytics/, 2008.

[48] Google. Issue 3666 - chromium. http://code.google.com/p/chromium/issues/detail?id=3666, October 2008.

[49] Google. Memory Usage Backgrounder (Chromium Developer Documentation). http://dev.chromium.org/memory-usage-backgrounder, 2008.

[50] Google. Plugin Architecture (Chromium Developer Documentation). http://dev.chromium.org/developers/design-documents/plugin-architecture, 2008.

[51] Google. Process Models (Chromium Developer Documentation). http://dev.chromium.org/developers/design-documents/process-models, 2008.

[52] Chris Grier, Shuo Tang, and Samuel T. King. Secure Web Browsing with the OP Web Browser. In *IEEE Symposium on Security and Privacy*, 2008.

[53] Jeremiah Grossman. Cross-Site Scripting Worms and Viruses. http://www.whitehatsec.com/downloads/WHXSSThreats.pdf, April 2006.

[54] Dmitry Gryaznov. Keeping up with Nuwar. http://www.avertlabs.com/research/blog/index.php/2007/08/15/keeping-up-with-nuwar/, August 2007.

[55] Grypen. CastleCops: About Grypen's Filter Set. http://www.castlecops.com/t124920-About_Grypens_Filter_Set.html, June 2005.

[56] Robert Hansen. XSS (Cross Site Scripting) Cheat Sheet. http://ha.ckers.org/xss.html, 2007.

[57] Norm Hardy. The Confused Deputy (or why capabilities might have been invented). *Operating Systems Review*, 22(4):36–8, October 1988.

[58] Amir Herzberg and Ahmad Gbara. TrustBar: Protecting (even Naive) Web Users from Spoofing and Phishing Attacks. In *Cryptology ePrint Archive: Report 2004/155*, 2004.

[59] Ian Hickson and David Hyatt. HTML 5. http://www.w3.org/html/wg/html5/, October 2008.

[60] Billy Hoffman. The SPI Laboratory: Jikto in the Wild. http://devsecurity.com/blogs/spilabs/archive/2007/04/02/Jikto-in-the-wild.aspx, April 2007.

[61] Jon Howell, Collin Jackson, Helen J. Wang, and Xiaofeng Fan. MashupOS: Operating System Abstractions for Client Mashups. In *Workshop on Hot Topics in Operating Systems (HotOS)*, April 2007.

[62] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *International Conference on World Wide Web (WWW)*, May 2003.

[63] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D. T. Lee, and Sy-Yen Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *International Conference on World Wide Web (WWW)*, May 2004.

[64] Sotiris Ioannidis and Steven M. Bellovin. Building a Secure Web Browser. In *FREENIX Track of the 2001 USENIX Annual Technical Conference*, June 2001.

[65] Omar Ismail, Masashi Etoh, Youki Kadobayashi, and Suguru Yamaguichi. A Proposal and Implementation of Automatic Detection/Collection System for Cross-Site Scripting Vulnerability. In *International Conference on Advanced Information Networking and Application (AINA)*, 2004.

[66] Collin Jackson, Andrew Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting Browsers from DNS Rebinding Attacks. In *ACM Conference on Computer and Communications Security (CCS)*, October 2007.

[67] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. Protecting Browser State from Web Privacy Attacks. In *International Conference on World Wide Web (WWW)*, May 2006.

[68] Collin Jackson and Helen J. Wang. Subspace: Secure Cross-Domain Communication for Web Mashups. In *International Conference on World Wide Web (WWW)*, May 2007.

[69] Java Community Process. JSR 121: Application Isolation API. `http://jcp.org/en/jsr/detail?id=121`, June 2006.

[70] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *International Conference on World Wide Web (WWW)*, May 2007.

[71] Martin Johns and Justus Winter. RequestRodeo: Client Side Protection against Session Riding. In *OWASP Europe Conference*, May 2006.

[72] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing Cross Site Request Forgery Attacks. In *ICST Conference on Security and Privacy in Communication Networks (SecureComm)*, August 2006.

[73] Emre Kiciman and Benjamin Livshits. AjaxScope: A Platform for Remotely Monitoring the Client-Side Behavior of Web 2.0 Applications. In *ACM Symposium on Operating Systems Principles (SOSP)*, November 2007.

[74] Emre Kiciman and Helen J. Wang. Live Monitoring: Using Adaptive Instrumentation and Analysis to Debug and Maintain Web Applications. In *Workshop on Hot Topics in Operating Systems (HotOS)*, April 2007.

[75] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks. In *ACM Symposium on Applied Computing (SAC)*, 2006.

[76] Amit Klein. DOM Based Cross Site Scripting. `http://www.webappsec.org/projects/articles/071105.html`, July 2005.

[77] V.T. Lam, S. Antonatos, P. Akritidis, and K. G. Anagnostakis. Puppetnets: Misuing Web Browsers as a Distributed Attack Infrastructure. In *ACM Conference on Computer and Communications Security (CCS)*, 2006.

[78] Butler W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, 1973.

[79] Giorgio Maone. NoScript - Whitelist JavaScript blocking for a safer Firefox experience. http://www.noscript.net, 2006.

[80] Gervase Markham. Script Keys. http://www.gerv.net/security/script-keys/, April 2005.

[81] Gervase Markham. Content Restrictions. http://www.gerv.net/security/content-restrictions/, January 2006.

[82] Microsoft. Internet Security and Acceleration Server. http://www.microsoft.com/isaserver/default.mspx, 2004.

[83] Microsoft. Microsoft Security Bulletin MS04-040. http://www.microsoft.com/technet/security/Bulletin/MS04-040.mspx, December 2004.

[84] Microsoft. Microsoft Security Bulletin Summaries and Webcasts. http://www.microsoft.com/technet/security/bulletin/summary.mspx, 2005.

[85] Microsoft. Microsoft Silverlight. http://www.microsoft.com/silverlight/, April 2007.

[86] Microsoft. SharePoint. http://www.microsoft.com/sharepoint, 2007.

[87] Microsoft Developer Network. Mark of the Web. http://msdn2.microsoft.com/en-us/library/ms537628.aspx, October 2007.

[88] Alexander Moshchuk, Tanya Bragin, Damien Deville, Steven D. Gribble, and Henry M. Levy. SpyProxy: On-the-fly Protection from Malicious Web Content. In *USENIX Security Symposium*, August 2007.

[89] Alexander Moshchuk, Tanya Bragin, Steven D. Gribble, and Henry M. Levy. A Crawler-based Study of Spyware on the Web. In *Network and Distributed System Security Symposium (NDSS)*, February 2006.

[90] Mozilla. Mozilla Foundation Security Advisories. http://www.mozilla.org/security/announce, 2005.

[91] Mozilla. Public Suffix List. http://publicsuffix.org/, 2007.

[92] Mozilla. Prism. https://developer.mozilla.org/en/Prism, 2008.

[93] Mozilla. Plugins. https://developer.mozilla.org/en/Plugins, March 2009.

[94] OpenID Foundation. OpenID. http://openid.net/, 2008.

[95] John K. Ousterhout, Jacob Y. Levy, and Brent B. Welch. The Safe-Tcl Security Model. Manuscript, 1996.

[96] Stuart Parmenter. Firefox 3 Memory Usage. `http://blog.pavlov.net/2008/03/11/firefox-3-memory-usage/`, March 2008.

[97] Positive Technologies. George Bush appoints a 9 year old to be the chairperson of the Information Security Deportment. `http://www.securitylab.ru/news/extra/272756.php`, 2006.

[98] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. All Your iFRAMEs Point to Us. In *USENIX Security Symposium*, July 2008.

[99] Charles Reis, Brian Bershad, Steven D. Gribble, and Henry M. Levy. Using Processes to Improve the Reliability of Browser-based Applications. Technical Report UW-CSE-2007-12-01, University of Washington, December 2007.

[100] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006.

[101] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. *ACM Transactions on the Web*, 1(3), 2007.

[102] Charles Reis and Steven D. Gribble. Isolating Web Programs in Modern Browser Architectures. In *ACM European Conference on Computer Systems (EuroSys)*, April 2009.

[103] Charles Reis, Steven D. Gribble, Tadayoshi Kohno, and Nicholas C. Weaver. Detecting In-Flight Page Changes with Web Tripwires. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2008.

[104] Charles Reis, Steven D. Gribble, and Henry M. Levy. Architectural Principals for Safe Web Programs. In *Workshop on Hot Topics in Networks (HotNets)*, November 2007.

[105] Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. The Structure and Performance of Interpreters. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[106] Jesse Ruderman. The Same Origin Policy. `http://www.mozilla.org/projects/security/components/same-origin.html`, 2001.

[107] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Communications of the ACM*, 17(7), 1974.

[108] Thomas Schreiber. Session Riding: A Widespread Vulnerability in Today's Web Applications. `http://www.securenet.de/papers/Session_Riding.pdf`, December 2004.

[109] David Scott and Richard Sharp. Abstracting Application-Level Web Security. In *International Conference on World Wide Web (WWW)*, May 2002.

[110] Secure Computing. Webwasher SSL Scanner. `http://www.securecomputing.com/pdf/WW-SSLscan-PO.pdf`, 2006.

[111] SecuriTeam. Gecko Based Browsers -moz-binding XSS. `http://www.securiteam.com/securitynews/5LP051FHPE.html`, February 2006.

[112] Larry Seltzer. Eweek: Anti-Virus Protection for WMF Flaw Still Inconsistent. `http://www.eweek.com/article2/0,1895,1907102,00.asp`, December 2005.

[113] sidki. Proxomitron. `http://www.geocities.com/sidki3003/prox.html`, September 2007.

[114] Emin Gün Sirer, Robert Grimm, Arthur J. Gregory, and Brian N. Bershad. Design and Implementation of a Distributed Virtual Machine for Networked Computers. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1999.

[115] Chris Smoak. Seattle Bus Monster. `http://www.busmonster.com/`, 2005.

[116] Snort. The Open Source Network Intrusion Detection System. `http://www.snort.org/`, 2005.

[117] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[118] Symantec.com. Adware.LinkMaker. `http://symantec.com/security_response/writeup.jsp?docid=2005-030218-4635-99`, February 2007.

[119] Symantec.com. W32.Arpiframe. `http://symantec.com/security_response/writeup.jsp?docid=2007-061222-0609-99`, June 2007.

[120] Peter Ször and Peter Ferrie. Hunting For Metamorphic. In *Virus Bulletin Conference*, September 2001.

[121] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *OOPSLA*, 1987.

[122] Useful Utilities. EZproxy by Useful Utilities. `http://www.usefulutilities.com`, 2007.

[123] David Vaartjes. XSS via IE MOTW feature. `http://securityvulns.com/Rdocument866.html`, August 2007.

[124] Virtual Conspiracy. Windows Script Decoder. `http://www.virtualconspiracy.com`, 2005.

[125] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Network and Distributed System Security Symposium (NDSS)*, 2007.

[126] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient Software-Based Fault Isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1993.

[127] Joe 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`, January 2007.

[128] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible Security Architectures for Java. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 1997.

[129] Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. Technical Report MSR-TR-2009-16, Microsoft Research, 2009. `http://research.microsoft.com/pubs/79655/gazelle.pdf`.

[130] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *ACM SIGCOMM*, August 2004.

[131] Yi-Min Wang, Doug Beck, Xuxian Jiang, Roussi Roussev, Chad Verbowski, Shuo Chen, and Sam King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*, 2006.

[132] Peter Watkins. Cross-Site Request Forgeries. `http://www.tux.org/~peterw/csrf.txt`, 2001.

[133] Web Wiz Guide. Web Wiz Forums - Free Bulletin Board System, Forum Software. http://www.webwizguide.com/webwizforums/, October 2007.

[134] Websense. Security Labs Alert: Super Bowl XLI / Dolphin Stadium. http://securitylabs.websense.com/content/Alerts/1346.aspx, February 2007.

[135] Sean Whalen. An Introduction to Arp Spoofing. http://www.rootsecure.net/content/downloads/pdf/arp_spoofing_intro.pdf, April 2001.

[136] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[137] Yichen Xie and Alex Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *USENIX Security Symposium*, August 2006.

[138] Zishuang Ye and Sean Smith. Trusted Paths for Browsers. In *USENIX Security Symposium*, 2002.

[139] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narul, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy*, 2009.

[140] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript Instrumentation for Browser Security. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2007.

[141] Bojan Zdrnja. Raising the bar: dynamic JavaScript obfuscation. http://isc.sans.org/diary.html?storyid=3219, August 2007.

[142] Andy Zeigler. IE8 and Loosely-Coupled IE. http://blogs.msdn.com/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx, March 2008.

[143] Andy Zeigler. IE8 and Reliability. http://blogs.msdn.com/ie/archive/2008/07/28/ie8-and-reliability.aspx, July 2008.

VITA

Charles Reis was born and raised in St. Louis, Missouri. He studied Computer Science at Rice University, where he received a Bachelor of Arts degree in 2002 and a Master of Science degree in 2003. His research experiences at Rice led him to pursue a doctorate at the University of Washington, with interests in systems, security, and languages. After finding his footing with web browser research and joining the Google Chrome team, he earned his Ph.D. degree in 2009.