TOWARDS SECURE WEB BROWSING

BY

SHUO TANG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Assistant Professor Samuel T. King, Chair, Director of Research
Professor Carl A. Gunter
Professor José Meseguer
Professor Henry M. Levy, University of Washington
Dr. Pablo Montesinos Ortego, Qualcomm Inc.

ABSTRACT

The Web is now the dominant platform for delivering interactive applications to hundreds of millions of users. Correspondingly, web browsers have become the *de facto* operating system for hosting these web-based applications (web apps). Unfortunately, web apps, browsers, and operating systems have all become popular targets for web-based attacks, intensifying the need for secure web browsing systems.

Current research efforts to retrofit today's web browsers help to improve security, but fail to address the fundamental design flaws of current browsing systems. To overcome those issues, in this dissertation, we rethink the way we build secure browsing systems, hoping to define the principles that should be followed.

To achieve this goal, we strive to learn through building experimental systems for secure web browsing. Specifically, we design and implement a new operating system and a new web browser. We also investigate other generic approaches to help secure these systems even further, including formal methods and heuristics.

The first system we build is called the Illinois Browser Operating System (IBOS). IBOS is an operating system co-designed with a new browser that reduces the trusted computing base for web browsing. We demonstrate that by exposing browser-level abstractions directly at the lowest software layer – the OS kernel – we are able to remove almost all traditional OS components and services from our trusted computing base. We show that this architecture is flexible enough to enable new browser security policies, can still support traditional applications and adds little overhead to the overall browsing experience.

We also propose the OP2 secure browser architecture that can be used on top of commodity operating systems. We combine operating system design principles with formal methods to design this secure web browser by

drawing on the expertise of both communities. Our design philosophy is to partition the browser into smaller subsystems and make all communications between subsystems simple and explicit. At the core of our design is a small browser kernel that manages the browser subsystems and interposes on all communications between them to enforce our new browser security features.

Through the experiences of building these systems, we are able to summarize the principles of building secure browsing systems: 1) make security decisions at the lowest layer of software and make it simple; 2) enforce strong isolation between distinct browser-level components; 3) employ simple and explicit communication between components; 4) provide the right set of operating system abstractions; 5) maintain compatibility with current browser standards; 6) expose enough browser states and events to enable new browser security policies.

Overall, we demonstrate in this dissertation that, by following these principles, our new browsing systems are not vulnerable to many forms of web-based attacks. We believe that the work presented in the dissertation makes one step towards secure web browsing.

To mom, dad, and Miaomiao

ACKNOWLEDGMENTS

During my journey towards a Ph.D., I had the fortune to have a group of amazing advisors, colleagues, and friends. Without their help and support, I could not reach this point. I would like to thank all of them.

First, I would like to thank my advisor, Sam King. He was the best advisor I could imagine. From Sam, I learned how to find a question, how to analyze the problem, and how to synthesize a good solution. More importantly, I also learned how to present a research work, both in write and presentation. Sam was a good model to me both on and off research, and would be the example for me to follow in the next stage of my life.

I would like to thank my committee, Carl, Jose, Hank, and Pablo for their valuable insight and feedback on my dissertation. Even during the dissertation writing, I learned a lot more from them of telling a better story.

I would like to thank Chris, Haohui, and Nathan for their help of my research projects included in this dissertation. I would also like to thank all the other member of our research group – Anthony, Anh, Hui, Matt, and Murph, and other students in our department – Michael, and Ralf. I always enjoyed discussing with them.

I would like to thank Doreen and Onur from Samsung, and David from Facebook for providing great internship experiences during my Ph.D. study.

I would like to thank Katie, who helped me a great deal of my English towards preparing my speaking test and OSDI presentation, which would also be an asset for my whole life.

Finally, I would particularly thank my cousin, Ge, who initially encouraged me to purse an advanced degree in the states, and continued to give me advice and support in the past years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Improving security is one of the greatest challenges and most important tasks across all domains in today's computing. This dissertation focuses on the security of web browsing. Specifically, we design and implement a new operating system and a new web browser to illustrate the general principles of building secure browsing systems.

1.1 Motivation

The Web has become a focal point in our daily lives. Tremendous amount of sensitive information is transmitted via the Web, signifying the demand of secure web browsing.

However, web-based applications (web apps), browsers, and operating systems have become popular targets for web-based attacks. Vulnerabilities in web apps are widespread and increasing. For example, cross-site scripting (XSS), which is effectively a form of script injection into a web app, recently overtook the ubiquitous buffer overflow as the most common security vulnerability [96]. Vulnerabilities in web browsers are less common than in web apps, but still occur often. For example, in 2009 Internet Explorer, Chrome, Safari, and Firefox had 349 new security vulnerabilities [94], and attackers exploit browsers commonly [67, 77, 78, 94, 105]. Vulnerabilities in libraries, system services, and operating systems are less common than vulnerabilities in browsers, but are still problematic for modern systems. For example, libc, Gtk, Linux, and X had 114 new security vulnerabilities in 2009 [1]. And evidence has already shown that they can be used by web-based exploitation to subvert the operating system kernel [12].

The cost of web-based attacks is extensive to both organizations and individuals. For organizations, it leads to loss of customer confidence, trust,

and reputation, with the consequent harm to brand equity, revenue, and profitability. It could also come with the cost related to repairing the damage done and securing the compromised web apps. For individuals, it could result in credit card fraud, identity theft, and subsequent financial loss. In extreme cases, one could even face legal liability if the attacker uses web-based exploitation, remotely controlling the victim's computer to attack other computers.

On one hand, the web-based attacks are prevalent because the Web is popular and attacking it is profitable [95]. Web browsers provide a simple and convenient interface to access the content and services in the Web, bringing in an unprecedented number of computer users. Today, both traditional and emerging businesses rely on the Web to deliver content and sell product to their customers. However, web apps have been developed and deployed with minimal attention given to security risks, resulting in a surprising number of web sites that are vulnerable to hackers [94]. The ease of accessing sensitive information through attacking the Web certainly stimulates the exploitation.

On the other hand, traditional browsing systems fail to provide sufficient protection, which also contributes to the trend of exploitation. In this dissertation, we refer *browsing system* to the whole client-side software stack that enables web browsing, including the web browser, runtime libraries, OS services, and kernel. Traditional browsing systems could not meet of requirements of secure web browsing because:

- The design and architecture of traditional web browsers are fudementally flawed. Traditional web browser design still roots in the original model of browser usage where users viewed several different static pages and the browser itself was the application. However, recent web browsers have evolved into a platform for hosting web apps, where each distinct page (or set of pages) represents a logically different application. The single-application model provides little isolation or security between these distinct applications.

- Traditional web browsers have wide attack surfaces. Browsers run untrusted web apps from all over the Internet, host external applications for rendering non-HTML content (i.e., plugins), and interact with other applications on the system. These interactions are often implicit, and can be carried out using various communication channels. As a result,

it is infeasible to monitor them, resulting possible avenues for attack.

- Commodity operating systems fail to regulate the behavior of web browsers. Commodity operating systems provide a set of general-purpose abstractions, often overly permissive, to browsers. Without restrictions, a compromised browser component can be further exploited to affect other parts of the browsing system, including the operating system itself. For example, a compromised HTML engine could download a malware and use the `exec()` Linux system call to execute it.

Based on the above observation, we argue that to address these problems fundamentally and achieve the desired security of web browsing, a resign of web browser and operating system is required.

1.2 Thesis and contributions

My thesis is:

General operating system design principles and techniques can be used to help building browsing systems that improve the security of web browsing.

To support this thesis, we embrace microkernel [50], Exokernel [34], safety kernel design principles, heuristic, and formal methods to build new browsing systems from scratch to illustrate how to enable secure web browsing. Through the experiences of building these systems, we are able to summarize a set of design and architecture principles that should be followed when building a secure browsing platform that is required to protect the web apps, the browsers, and the underlying operating systems.

1. *Make security decisions at the lowest layer of software and make it simple.* Security vulnerabilities are commonplace and costly across all layers in the software stack. It is necessary to provide protection for all the layers. However, without securing the lowest layer, it is impossible to provide guarantee for other layers built on top of it. Ideally, a secure system should have a simple lowest layer responsible for all the security decisions. By doing this, we could have a small and potentially verifiable TCB for the whole system, thus improving the overall security assurance.

2. *Enforce strong isolation between distinct browser-level components.* Providing isolation between browser-level components reduces the likelihood of unanticipated and unaudited interactions and allows us to make stronger claims about general security and the specific policies we implement. Specifically, the browser should be decomposed into several subsystems, each of which performs dedicated functions (e.g., HTML rendering, cookie management) and runs in its own protection domain. At the same time, web apps that come from different sources should also be isolated as they represent different principals in the Web.

3. *Employ simple and explicit communication between components.* Clean separation between functionality and security with explicit interfaces between components reduces the number of paths that can be taken to carry out an action. This makes reasoning about correctness, both manually and automatically, much simpler. We argue that a microkernel-like architecture should be adapted to enforce explicit communications between different components. Meanwhile, a microkernel-like architecture also enables using formal methods as a practical methodology for validating browser system design and implementation.

4. *Provide the right set of operating system abstractions.* A fudemental problem of using commodity operating systems as the building bases for secure browsing systems is that they often expose a set of overly permissive interfaces for browser-level components. Typically, one could use rule-based OS sandboxing mechanisms to restrict these components. However, sandboxing systems can be complex (the Ubuntu 10.04 SELinux reference policy uses over 104K lines of policy code) and difficult to implement correctly [38, 97]. Given the opportunity of redesigning the whole browsing system, we argue that one should follow the Exokernel principles to expose the set of OS abstractions that are just enough for web browsing to avoid further use of OS sandboxing.

5. *Maintain compatibility with current browser standards.* Even though we are building a new browsing system, our primary goal is to improve the enforcement of current browser policies without changing current web apps. Current web apps already provide a rich set of features and were designed according to existing security policies. Without proper

backwards compatibility support, it is impractical for users to accept the new system. Meanwhile, traditional desktop applications provide an alternative set of functionalities. We argue that a browsing system should provide adequate support for them and enforce controlled sharing between web apps and traditional apps.

6. *Expose enough browser states and events to enable new browser security policies.* The Web is fast evolving. Browsing system should be flexible to adapt to the evolution. Consequently, a desired architecture should expose enough browser states and events to the lowest layer to enable novel browser security policies. Nevertheless, web developers have been slow to use these new browser security features [90]. Fortunately, with those states and events, one could use simple heuristics to add protections to web apps automatically at the client side. We validate this claim through multiple examples shown in this dissertation.

Following these six principles, we design and implement the Illinois Browser Operating System (IBOS) – an operating system and web browser co-designed to reduce drastically the trusted computing base for web browsers and to simplify browsing systems. To achieve this improvement, we build IBOS with browser abstractions as first-class OS abstractions and removed traditional shared system components and services from its TCB.

In addition, we demonstrate a secure browser architecture – the OP2 web browser that can be used when a specialized operating system is not desired. OP2 adopts a microkernel-like design to enforce strong isolation and explict communications between browser components to improve browsing security. We also show a generic browser-based mechanism for adding protection to legacy web apps and how to use formal methods to help design of browsing systems.

Overall, by design, our new systems are not vulnerable to many forms of web-based attacks, making one step towards secure web browsing.

1.3 Dissertation organization

In this dissertation, we discuss designing and implementing client-side systems to advance the state of the art in secure web browsing.

In Chapter 2, we provide background material around the Web. As this dissertation focuses on the principles of building browsing systems, we present common architectures of operating systems and web browsers. We also discuss previous work on web app security, browser security, operating system security, and applying formal method to browsing systems.

In Chapter 3, we present an operating system and a browser co-designed using principles from both microkernel and Exokernel to reduce drastically the TCB for web browsers and to simplify browsing systems.

In Chapter 4, we show how to use microkernel-like design to build a secure web browser, and how to make it practical in terms of compatibility and performance.

In Chapter 5, we discuss how to only use information available at the client side to improve the security of web apps. And we use four case studies to show that our approaches are simple yet effective.

In Chapter 6, we discuss how to use formal method to validate of design of secure web browser. We also examine the possibility of verifying IBOS TCB.

In Chapter 7, we present the security analysis and performance evaluation of our systems. We show that our design is able to achieve improved protection without sacrificing performance.

We discuss potential future directions in Chapter 8, and conclude in Chapter 9.

CHAPTER 2

BACKGROUND AND RELATED WORK

Security has been a goal of computer systems designers for a long time. As the Web become increasingly popular, software systems for web browsing have become the dominant sources of computer vulnerabilities. Our work therefore focuses on improving the security of web browsing.

In this chapter, we present background material on building secure browsing systems. First, we provide the definitions of web-based applications and trused computing base. We then discuss web-based attacks and how common operating system architectures and browser architectures handle them. Finally, we describe previous approaches to improving general web security.

2.1 Definitions

In this section, we describe the definitions and background information about web-based applications and trusted computing base.

2.1.1 Web-based application

When the Web was first invented, it was a collection of static web pages. Now, the Web is the dominant platform for delivering interactive applications. By definition, a web app is an application that is accessed via the Web using primarily HTTP (Hypertext Transfer Protocol) connections and hosted primarily in web browsers.

The World Wide Web's markup language has always been HTML (HyperText Markup Language). Although HTML was primarily designed as a language for semantically describing scientific documents, its general design and adaptation over the years have enabled it to become the dominating language for describing the structure of web pages.

HTML itself in most cases is not adequate to specify all of features of today's web apps. For complement, extensibility mechanisms (e.g., plugins) have been developed. At the same time, HTML specifications are also being updated to include new features, such as video support, to address the issues of web development raised in the past few years [101].

In general, plugins are external applications that browsers use to render non-HTML content. One common example of a plugin is the Flash player that enables browsers to play Flash content. One can treat plugins as traditional desktop applications or libraries, except that they are launched by the browser and the system gives them access to browser states and events through a standard plugin programming interface, called the NPAPI [2].

CSS, Document Object Model (DOM), and JavaScript have also been developed as the complements to HTML to provide enough power for developing interactive and user-friendly web apps.

CSS, which stands for Cascading Style Sheets, is a style sheet language used to express styles that define how to present web pages, including colors, layout, and fonts. CSS provides a powerful, yet flexible, mechanism, helping web apps to match the look and feel of desktop applications.

The DOM is a representation – a model – of the document and its content. The DOM is not just an API; operations on the in-memory HTML document are also defined. HTML elements in the DOM implement and expose a series of interfaces to scripts to make the documents themselves programmable. DOM also provides the interface of `XMLHttpRequest` to allow scripts to programmatically connect back to their originating server via HTTP.

JavaScript is the predominant client-side script language for web apps. JavaScript utilizes the interfaces exposed by the DOM to dynamically change the web pages, resembling interactive traditional desktop applications. For example, together with XML and the `XMLHttpRequest` capability, Asynchronous JavaScript and XML (AJAX) is used to allow web apps to retrieve data from the server asynchronously in the background without interfering with the display and behavior of the existing page, thus enabling interactive applications.

The Web, as a multi-users platform, requires a mechanism to authenticate and distinguish different users. The most commonly used method is cookie. A cookie, or an HTTP cookie, is a piece of text stored on a user's computer by

his or her browser, typically consisting of one or more name-value pairs that the server and client pass back and forth. Initially developed as a method for implementing reliable virtual shopping carts, cookies were later pervasively used as the *de facto* way of authenticating users to web sites and storing the login information so that a web user does not have to keep entering their username and password each time he or she visits a same web site. Cookies can also be used to store identifiers so that web servers can track what the users have done during the visit.

2.1.2 Trusted computing base

In computer security terminology, the *trusted computing base* is the set of all hardware, software and procedural components that enforce the security policy. Consequently, a compromised TCB would jeopardize the security properties of the entire system. While parts of a computer system outside the TCB, regulated by security policies, could only incur limited damage within the privileges granted to them.

Design and implemention of the system's TCB is critical to overall system security. Modern operating systems strive reduce the size of their TCB so that manual review or formal verification of the TBC could be feasible.

Systems without all of their TCBs as part of their design do not provide security of their own. They are secure only when the external means provide required security. For example, a cryptography algorithm that uses private key to encrypt would require the key to be securely kept in the memory. If a successful memory-based attack is able to steal the key, the encryption could no longer provide the security assurance even if the algorithm itself is flawless.

Similarly, mechanisms developed to protect web apps in the browsers would be only as secure as the browsers (or the underlying operating systems) themselves. Consequently, to improve the security of web browsing, we have to carefully design and implement the TCB of our browsing systems.

2.2 Web-based attacks

Unfortunately, current browser security policies and browsing systems are not always effective at guarding the Web. Several attacks operate without violating those policies. Besides attacking scenarios that target only to the Web, attackers can also use the Web as the medium to carry out traditional attacks such as using the method of so-called "drive-by download". The Web also provides a perfect avenue for social engineering attacks. One example is phishing, in which the attacker creates a fake website that has almost identical look and feel to the legitimate one, in order to fool users to give in sensitive information such as usernames, passwords and credit card details. In this section, we discuss the same-origin policy (SOP) and present details about XSS and drive-by download. We leave other policies and attacks to be discussed inline in later chapters when necessary.

2.2.1 The same-origin policy

The primary security policy that all modern browsers implement is the SOP, in which most security decisions are predicated on the origin of the web apps. An origin of a web app is define as the `<protocol, domain name, port>` tuple of the uniform resource locator (URL) it originates from. Loosely speaking, the SOP acts as a non-interference policy for the Web and provides isolation for web pages and states that come from different origins. If the browser runs one web app from `victim.com` and another from `attack.com`, the browser isolates these two web apps from each other. Unfortunately, Chrome, IE8, Safari, and Firefox all enforce the SOP using a number of checks scattered throughout the millions of lines of browser code and current browsers have had trouble implementing the SOP correctly [25].

In a browser, a frame is a container that encapsulates a HTML document and any material included in that HTML document. Web pages are frames, and web developers can embed additional frames within web pages – these frames are called `IFRAME`s. Developers can include `IFRAME`s from the same origin as the hosting frame, or from a different origin. Each frame is labeled with the origin of the main HTML document used to populate the frame, meaning that a cross-origin `IFRAME` has a different label than the hosting web page.

In general HTML documents include references to network objects that the browser will download and display to form the web page. These network objects can be images, JavaScript, and CSS. Browsers can download these objects from any domain and the browser labels them with the origin of the hosting frame. For example, if a page from `uiuc.edu` includes a script from `foo.com`, that script runs with full `uiuc.edu` permissions and can access any of the states in that web page. Browsers can also download HTML documents and other objects using XMLHttpRequests (used for Ajax), but the SOP dictates that these objects must come from the same origin as the hosting frame.

2.2.2 XSS

XSS is effectively a form of code injection attack, where an attacker injects malicious scripts into the victim web app and operates using the victim's origin and credentials. Some potential damage an XSS could cause includes stealing the victim's credentials, gaining elevated access privileges to sensitive page content, and carrying out actions on behalf of the victim. In fact, XSS is the most prevalent vulnerability on modern computer systems, accounting for more vulnerabilities than all other vulnerabilities combined [96]. There are three types of XSS attacks:

Persistent XSS Attackers inject malicious code into a Web app in the server side and are able to affect all the users that use the web app. Typical affected websites are public Internet forums or Wiki-like sites. An attacker could submit content of JavaScript (which should be plain text or legitimate HTML code) to web servers and every user that visits the site would be affected by the malicious JavaScript code.

Reflective XSS User supplied data can be used to generate a temporary page sent back to the same user. If the user supplied data is not properly sanitized and it contains, for example, malicious JavaScript code, the user could subjected to reflective XSS attack. In a simple attack scenario, the attacker could foul the victim to click a URL with a malicious payload that can be embedded in the web page delivered to the victim.

DOM-based XSS This type of XSS can be similar to reflective XSS and may have been overlooked. Instead of generating a malicious page by the server side logic (e.g., a PHP), attackers can leverage client side logic to effectively deliver attack code. We will discuss more details of this type of XSS in Chapter 5.

2.2.3 Drive-by download

Drive-by download means unintended download occurs during visiting a website. In usual case, it is used to download malware for nefarious purpose. It could happen when a user authorizes the download but without understanding the scene behind or the consequences. For example researchers from Google found that attackers fool users into downloading and executing malicious content from adult web sites by making them think they are installing a new video codec in an attempt to view "free" videos [78]. It could also mean download of spyware, a computer virus or any kind of malware that happens without a person's knowledge. Current web technology enables high-degree dynamic content. One can easily use JavaScript to initiate download automatically without consent from user.

Drive-by download can also happen in legitimate websites. Buying advertisements, for example, can sometimes allow attackers to have their malicious code included in pages that display the advertisements [77]. Also, by exploiting security vulnerabilities in web apps, attackers can often automatically modify these sites to host their malicious code. The downloaded content does not need to be executable binaries either. For example, in a recent iOS vulnerability, the attacker can simply serve a specifically crafted PDF file to exploit victims' operating systems [12].

2.3 Operating system architectures

As the browser transforms into a platform of hosting web apps, it becomes the *de facto* operating system. Before diving into different design decisions of the browsers themselves, we first look at common operating system architectures to discuss how they could accommodate web browsers and what we can learn from them.

Generally speaking, operating system (or the kernel) manages computer hardware resources, and provides common services for execution of various application software. These management logics are often called operating system services. Depending on the organization of these services, we have different types of kernel architectures.

2.3.1 Monolithic kernel

A monolithic kernel includes all (or at least, most) of its services in a single privileged address space – the kernel space. Consequently, all the services run as supervisor mode and can modify any other parts in the operating system. Some examples are Linux, BSD, and early Windows versions (95, 98 and ME).

One advantage of this architecture is performance superiority. Since all the services reside in the same address space, there is no context switching overhead in the kernel for managing different types of resources.

This architecture, on the other side, is not quit flexible as all the management logics are in a single binary. To alleviate the problem, modern monolithic kernel enables "loadable module", such as Linux.

Fundamentally, this architecture of operating system is no different from that of a user-level application (such as traditional web browsers). Since everything runs in the same address space, if any component is compromised, the security of the whole system is jeopardized.

2.3.2 Microkernel

A Microkernel tries to run most services, like network stacks, file systems, etc., as servers in user space. The kernel only provides basic services that are needed to implement the whole operating system – memory allocation, scheduling, and inter-process communication (IPC).

This architecture often result a minimal amount of software in the kernel address space. When the correctness of this small kernel is assured, this architecture has the advantage of providing highly guaranteed reliability and security. Naturally, every OS services runs in its own user-level address space. When one crashes, it is hard for it to affect other parts in the system.

In the context of security the minimality principle of microkernels is a direct consequence of the principle of "least privilege". A smaller kernel as the TCB also leads to an easier and feasible effort to formal verification [57]. While services could still be compromised, the kernel is able to ensure that they could only incur damages confined within the privileges provided.

A drawback is the amount of messaging and context switching involved, which makes microkernels conceptually slower than monolithic kernels. However, when security is highly desired and usage scenario would not incur frequent context switching (e.g., computation intensive rather than I/O intensive), this architecture is preferred.

Operating systems designed to reduce the trusted computing base for applications are not new and typically choose the microkernel-like architecture. For example, several recent OSes propose using information flow to allow applications to specify information flow policies that are enforced by a thin kernel [32,59,112]; KeyKOS [21], EROS [88], and seL4 [57] provide capability support using a small kernel; and Microkernels [40, 50, 51] push typical OS components into user space.

In IBOS, we apply these principles to a new application – the web browser – and include support for user interface components and window manager operations. Also, these previous approaches support general purpose security mechanisms, like information flow and capabilities, and shared resources and device drivers are part of the TCB. The IBOS security policy is specific to web browsers, and although this is less general, we can track this policy to hardware abstractions and can remove drivers and other shared components from IBOS TCB.

At the same time, for our standalone secure browser design, we also choose a microkernel-like architecture. Using a secure architecture like in OP2, we are able to enforce the clear separation between functionality and security, employing a small "browser kernel" to provide overall security assurance of the web browser.

2.3.3 Hybrid kernel

In practice, there are not many commercialized operating systems that use pure-microkernel architecture. Instead, to balance performance, reliability,

and security, system designers mostly choose a hybrid kernel architecture.

A hybrid kernel is, as its name indicates, a hybrid between a monolithic kernel and a microkernel. Unlike a microkernel where everything takes place in user-level servers and drivers, or a monolithic kernel where all are included in the kernel space, the designers of a hybrid kernel may decide to keep several components inside kernel and some outside.

In fact, most of the commercialized operating systems we use today are hybrid architectures, including Windows NT series (NT, XP, Vista, 7, Server 2003, Server 2008), and MacOS (also iOS).

2.3.4 Exokernel

Exokernel is not really a whole different architecture not included in above discussions, but rather an orthogonal design principle.

Exokernels [34, 56] attempt to separate security and performance from abstraction. The kernel almost does nothing but securely multiplex the hardware. The goal is to avoid forcing any particular abstraction upon applications, instead allowing them to use or implement whatever abstractions that are best suited to their task without having to layer them on top of other abstractions which may impose limits or unnecessary overhead. In the original Exokernel design, this is done by moving abstractions into untrusted user-space libraries called "library operating systems" (libOSes), which are linked to applications and call the operating system on their behalf.

Both Exokernels and L4 [50] could be regarded as the efforts of rethinking low-layer software abstractions. In both projects, they advocate exposing abstractions that are close to the underlying hardware to enable applications to customize for improved performance. In IBOS we build on these previous works – in fact we use the L4Ka::Pistachio L4 [9] MMU abstractions and message passing implementation directly. However, the key difference between our work and L4 and Exokernel is that we expose high-level application abstractions at our lowest layer of software, not low-level hardware abstractions. Our focus is on making web browsers more secure and the system software we use to accomplish this improved security. At the same time, we manage to build a Unix-like layer to provide support for traditional applications.

2.3.5 Operating system security

Many efforts have been put into operating system security research. Previous studies in device driver security, and secure window managers, while not directly applicable to the systems we build, could still be learned.

Device driver security Device driver security has focused on three main topics. First, several projects focus on restricting driver access to I/O ports and device access to main memory via DMA. For example, RVM uses a software-only approach to restrict DMA access of devices [107], SVA prevents the OS from accessing driver registers via memory mapped I/O through memory safety checks [29], and Mungi [61] relies on using a hardware IOMMU to limit which memory regions are accessible from devices. Second, system designers isolate drivers from the rest of the system. This isolation can be achieved by running drivers in user-mode, which has been a staple of Microkernel systems [40,51,62], using software to protect the OS from kernel drivers [35,114], or by using page table protections within the OS [92,93]. The driver security architecture in IBOS differs from these approaches because our system provides fine-grained protection for individual requests within a shared driver in addition to isolating the driver from the rest of the system.

Secure window managers A number of recent projects have looked at reducing the TCB for window managers. For example DoPE [36] and Nitpicker [37] move widget rendering from the server to the client, leaving the server to only manage shared buffers. CMW [108], EWS [89], and TrustGraph [74] also use clients for rendering, but are able to apply capabilities and mandatory access control policies to application user-interface elements. In IBOS, we deprecate the general window notion of modern computer systems in favor of the simpler browser chrome and tab motif, allowing us to track our security policies down to the underlying graphics hardware on our system.

2.4 Browser architectures

In the early stage of the history of the Web, Netscape was the most popular web browser – yet a traditional desktop application. With the success of

Netscape showing the importance of the Web, Microsoft created Internet Explorer to compete and eventually take over the market of web browsing.

At the beginning of this century, due to lack of competition, Internet Explorer stagnated in version 6 for 5 years until 2006, resulting in slow innovation in browser technology at the time. As web pages gradually transform into web apps, there are greater demands of building new and more secure browsing systems.

2.4.1 Monolithic browser architecture

New breed of web browsers were introduced, including Firefox, Opera, and newer versions of Internet Explorer, to facilitate web browsing. However, these browsers still use a monolithic architecture, where all browser services such as user interface(UI), cookie management, and network services, together with plugins, browser addons (e.g., Firefox extensions), and web apps from different sources are running in the same address space with the same privileges. The single-application mode provides little isolation or security between these distinct applications hosted within the same browser, or between different applications aggregated on the same web page. A compromise occurring on any part of the browser, including plugins, results in a total compromise of all web apps running within the browser and the browser itself.

2.4.2 Secure browser architecture

A number of recent academic and industry projects have proposed new browser architectures resembling operating systems including SubOS [52,53], safe web programs [79], OP [46], Chrome [18, 80], Gazelle [104], and ServiceOS [69]. Without exception, these secure web browsers choose to use microkernel-like architectures to enforce strong isolation and explicit communication between different components. Although the browser portion of IBOS and OP2 do resemble some of these works, they all run on top of commodity OSes and include complex libraries and window managers in their TCB, something that IBOS avoids by focusing on the OS architecture of our system.

All these systems require the use of host OS sandboxing to restrict

browsers. The idea of sandboxing browsers was first introduced by Goldberg et al. [39]. We also use this type of sandboxing in our OP2 browser as the starting point for our security, and we focus on more fine-grained interactions within the browser itself. Plus, by breaking our browser into different components, we can apply different sandboxing rules to each subsystem, giving us even more control over our browser's interactions with the underlying system. Moreover, the browser abstractions that IBOS exposes for its browser components avoid the complex sandboxing process, and simplify the browsing system.

As the representative commercial secure browser, Google Chrome uses operating system processes to separate instances of page rendering in different ways. Chrome provides four different process creation models that can provide isolation between different browser entities [18]. Unlike OP2, Chrome does not support protections between frames in a web browser and does not enforce security policy for plugin content. Chrome also relies on the parsing and rendering engine to correctly control network requests, placing security decisions in the same process as the rendering engine. We separate the security enforcement from rendering and allow security decisions to be made by the browser kernel, allowing plugins to be subject to the policies enforced by the browser kernel.

2.4.3 Browser-oriented OSes

Researchers and engineers have also tried to build specialized systems that are used primarily for web browsing.

In the Tahoma browser [28], the authors propose using virtual machine monitors (VMMs) to enable web apps to specify code that runs on the client. Tahoma uses server-side manifests to specify the security policy for the downloaded code and the VMM enforces this security policy. Tahoma does expose a few browser abstractions from their VMM to help manage UI elements and network connections, but operates mostly on hardware-level abstractions. Because Tahoma operates on hardware-level abstractions, Tahoma is unable to provide full backwards-compatible web semantics from the VMM and more fine-grained protection for browsers, such as isolating `iframes` embedded in a web application. Also, many modern VMMs use a full-blown commodity

OS in a privileged virtual machine or host OS for driver support, leaving tens of millions of lines of code in the TCB potentially.

The webOS from Palm [75] and the upcoming ChromeOS from Google [43] run a web browser on top of a Linux kernel. ChromeOS includes kernel hardening using trusted boot, mandatory access controls, and sandboxing mechanisms for reducing the attack surface of their system. However, ChromeOS and IBOS have fundamentally different design philosophies. ChromeOS starts with a large and complex system and tries to remove and restrict the unused and unneeded portions of the system. In contrast, IBOS starts with a clean slate and only adds to our system functionality needed for our browser. Although our approach does require implementing from scratch low-level software and fitting device drivers to a new driver model, the end result has 2 to 3 orders of magnitude fewer lines of code in the TCB, while still retaining nearly all of the same functionality.

2.5 General web security

In addition to the effort of designing more secure browser architectures we discussed, there are many more approaches for improving the general web security.

One way to mitigate the problem is to educate the user. Some examples are using the latest patched software, installing protection programs, staying off illegitimate websites, and being aware of suspicious content. However, educating users is hard. Under certain circumstances, such as encountering with technical or financial difficulty, it is infeasible for a user to upgrade the software or purchase protection programs. More alarmingly, even if the user only visits legitimate websites and keeps awareness, he or she could be subjected to stealth web-based attack. It has been reported that legitimate websites could serve malicious advertisements, resulting in malware downloaded and executed in background [77].

The other is to fix the software stack for web browsing. Mitigation techniques can involve the server, the server and the client, or just the client to provide protection to users. The first, and often most accepted, solution to web app vulnerabilities is simple: fix the bug or write better programs in the server. However, recent research has argued that purely server-side

techniques are flawed due to differences in browser implementations [70], ultimately limiting the effectiveness. Hybrid server-client solutions use browser modifications to allow web developers to express security constraints to the browser directly. Some recent examples of this type of defensive architecture include introducing new HTML tags for fine-grain sandboxing of scripts [103]. Two downsides of hybrid solutions are that servers and clients must both be modified, introducing a high barrier to adoption, and hybrid solutions provide little support for legacy systems. Client-side prevention is positioned so that clients can defend themselves against servers even if the servers are malicious or unpatched. Fundamentally, code executes within browsers, making the browser a natural location to detect and remove malicious code, or contain the damage. But having browsers changed alone might raise potential compatibility issue with unchanged web apps. This constricts the development of client-side mitigation techniques and has caused some designers to deploy conservative designs for maintaining compatibility to "avoid breaking the Web" [81].

Both server-side and client-side improvement are necessary and they are complementary to each other and would also benefit the systems we build. In the following paragraphs, we discuss a series of recent work in details, including XSS defenses, clickjacking defenses, cookie protection, taint tracking, and formal methods for web security.

XSS defenses XSS defenses are closely related to our `HttpOnly` cookie defense in ZAN because one common use of XSS is to steal authentication cookies via injected JavaScript, which is something our defense is designed to prevent. XSS Auditor [41] and IE8 [81] use heuristics to detect script-like entities embedded in URLs to prevent reflected XSS attacks. A number of recent projects enable the programmer to use annotations to specify portions of the HTML document where the browser prevents scripts from running [42, 55,98,102,103]. Similarly, two recent project propose automated client/server hybrid systems [48,70] that automatically mark portions of the HTML where scripts are not allowed to run. Finally, the Firefox NoScript extension [64] white-lists trusted script source locations. The mechanism we propose in ZAN differs from these approaches because it focuses on identifying and isolating authentication cookies rather than determining what JavaScript code should be allowed to run.

Clickjacking defenses Frame-based attacks were first reported in 2008 when Hansen and Grossman introduced the term "clickjacking" [49]. In a clickjacking attack, the attacker chooses a clickable region on the target website that the user is currently authenticated on (e.g., a "like" button in a Facebook page). To perform the attack, a malicious website will load a page from the victim website inside an `IFRAME`, using Cascading Style Sheets (CSS) to make it transparent. At the same time, this transparent clickable element is placed on top of some visible, fake, but interesting clickable gadget (e.g., click to win a free iPad). As a result, the user would "like" an attacker chosen page in Facebook instead of unrealistically winning a free iPad when he or she clicks it. Clickjacking defenses are related to our `X-Frame-Options` defense in Zan because clickjacking is enabled by attackers including framed pages and occluding the framed content to fool users.

ClearClick, which is part of NoScript [64], tries to prevent clickjacking by notifying the user anytime they interact with an framed element that has been occluded. This mechanism essentially infers the user's intent by reasoning about visual elements and any occlusion that the page might induce on embedded elements. ClickIDS [14] uses ClearClick as part of an automated testing tool that synthesizes clicks on pages and runs them with ClearClick and without ClearClick enabled. By comparing the results of the two pages they can infer a possible clickjacking attack by detecting differences between the two. Our complementary `X-Frame-Options` defense differs from these techniques by instead inferring programmer intentions (i.e., frame busting code) and preventing the page from being framed rather than inferring user intentions.

Cookie protection One recent project that aims to protect cookies is the Doppelganger project [86]. Doppelganger provides more flexible cookie policies for users by recording and replaying web sessions to detect if modifying a cookie would affect a web site. This information enables Doppelganger to make decisions about deleting cookies that would otherwise be stored by the browser.

Recent work by Vogt *et al.* [100], proposes using dynamic taint tracking to prevent cookies from being sent to a remote site via JavaScript. In our work we strive to identify and isolate login cookies, whereas they assume that cookies are tainted and track the effects of these cookies as JavaScript

code accesses them. One could imagine combining these two complementary techniques so that their taint tracking system only taints cookies that ZAN identifies as `HttpOnly` cookies.

Taint tracking for web app security *Taint tracking* or data flow tracking has been used extensively to improve application security. Existing techniques such as Perl supports a taint-mode [76] to prevent unsafe use of untrusted input. Sekar [85] provides an effective and language-independent taint tracking approach to prevent injection attacks. Vogt et al. use client-side taint tracking to identify requests that leak sensitive information across domain boundaries [100]. Their technique works inside of the browser and provides security alerts when sensitive information is sent to a third-party domain. Resin [111] is another tool that tracks data flow to propagate policies to improve application security. However, Resin only requires that untrusted data be sanitized to prevent XSS vulnerabilities and assumes the correctness of the sanitization.

Secure browser extensibility Most internet users do not expect the performance of web apps to be the same as desktop applications, which are driven by code created from high-quality compilers and designed to run natively. Browser extensibility mechanisms, such as plugins, provide a way to use native code modules as part of a web app, but also result in new avenues of vulnerabilities.

One software project that strives for security yet still offers native performance is Xax [30]. Xax separates native instruction execution from native OS access, leveraging legacy code to deliver desktop applications on the Web. In contrast to Xax, which relies on the memory management unit for memory isolation and a kernel system-call patch to prevent OS access, Google's Native Client takes a different approach [110]. Using an OS-portable sandbox, Native Client relies on x86 segmentation hardware to enforce memory isolation and on a binary validator to isolate the OS interface, preventing direct access to the OS and resources such as the file system and the network.

Xax and Native Client are but two of the software technologies designed to close the performance gap by using legacy software and strengthen the security of web browsers. While our focus in this dissertation is the secure architectures of browsing systems, these techniques can be supplement to our

design when safe extensibility is desired.

Formal methods for web security Researchers have been using model checking and symbolic execution to find subtle bugs in operating systems such as Coreutils [23], file systems [109], network stack implementations [22]. In seL4 [57], the authors use formal verification to guarantee that a implementation of L4 microkernel is free of programming errors.

Formal methods have also been applied to web browsers. In a recent work by Chen *et al.* [24], the authors examined cases that allow the address bar in the browser to mismatch the page content for Internet Explorer. They use model checking to search for violations of invariants specified for GUI elements in Internet Explorer under normal operation. Bohannon et al. proposed a formal specification of the core functionality of a web browser [20]. However, they did not really verify any security property or provide security enforcement in the work.

2.6 Summary

Current research efforts to retrofit today's web browsers help to improve security, but fail to address the fundamental design flaws of current browsing systems. Mechanisms that run within current web browsers to provide better web app security are only as secure as the browser they run within, which currently is not very secure. Moreover, web browsers still run on top of commodity operating systems and use general abstractions designed for a wide range of traditional applications, forcing browsers to rely on a huge TCB. Without redesign of the whole browsing system, it is infeasible to achieve the desired security.

CHAPTER 3

THE ILLINOIS BROWSER OPERATING SYSTEM

Web browsing becomes the primary task in today's computing systems. However, modern web browsers still run on top of commodity operating systems, use general-purpose OS abstractions, and inherite the cruft needed to implement and access these general OS abstractions. In this chapter, we examine the possibility of building a operating system specifically for secure web browsing and present the design and implemention of our prototype system.

3.1 Introduction

Current research efforts into more secure web browsers help improve the security of browsers, but remain susceptible to attacks on lower layers of the computer stack. The OP web browser [46], Gazelle [104], Chrome [18], and ChromeOS [43] propose new browser architectures for separating the functionality of the browser from security mechanisms and policies. However, these more secure web browsers are all built on top of commodity operating systems and include complex user-mode libraries and shared system services within their trusted computing base (TCB). Even kernel designs with strong isolation between OS components (e.g., microkernels [40, 50, 51] and information-flow kernels [32, 59, 112]) still have OS services that are shared by all applications, which attackers can compromise and still cause damage. Here are a few ways that an attacker can still cause damage to more secure web browsers built on top of traditional OSes:

- A compromised Ethernet driver can send sensitive HTTP data (e.g., passwords or login cookies) to any remote host or change the HTTP response data before routing it to the network stack.
- A compromised storage module can modify or steal any browser related

persistent data.

- A compromised network stack can tamper with any network connection or send sensitive HTTP data to an attacker.
- A compromised window manager can draw any content on top of a web page to deploy visual attacks, such as phishing.

In this chapter we describe IBOS, an operating system and a browser co-designed to reduce drastically the TCB for web browsers and to simplify browser-based systems. Our key insight is that our lowest-layer software can expose browser-level abstractions, rather than general-purpose OS abstractions, to provide vastly improved security properties for the browser *without* affecting the TCB for traditional applications. Some examples of browser abstractions are cookies for persistent storage, hypertext transfer protocol (HTTP) connections for network I/O, and tabs for displaying web pages. To support traditional applications, we build UNIX-like abstractions on top of our browser abstractions.

IBOS improves on past approaches by removing typically shared OS components and system services from our browser's TCB, including device drivers, network protocol implementations, the storage stack, and window management software. All of these components run above a trusted *reference monitor* [11], which enforces our security policies. These components operate on browser-level abstractions, allowing us to map browser security policies down to the lowest-level hardware directly and to remove drivers and system services from our TCB.

This architecture is a stark contrast to current systems where *all* applications layer application-specific abstractions on top of general-purpose OS abstractions, inheriting the cruft needed to implement and access these general OS abstractions. By exposing application-specific abstractions at the OS layer, we can cut through complex software layers for one particular application without affecting traditional applications adversely, which still run on top of general OS abstractions and still inherit cruft. We choose to illustrate this principle using a web browser because browsers are used widely and have been prone to security failures recently. Our goal is to build a system where a user can visit a trusted web site safely, even one or more of the components on the system have been compromised.

Our contributions are:

- IBOS is the first system to improve browser and OS security by making browser-level abstractions first-class OS abstractions, providing a clean separation between browser functionality and browser security.
- We show that having low-layer software expose browser abstractions enables us to remove almost all traditional OS components from our TCB, including device drivers and shared OS services, allowing IBOS to withstand a wide range of attacks.
- We demonstrate that IBOS can still support traditional applications that interact with the browser and shared OS services without compromising the security of our system.

3.2 The IBOS architecture

This dissertation presents the design and implementation of the IBOS operating system and browser that reduce the TCB for browsing drastically. Our primary goals are to enforce today's browser security policies with a small TCB, without restricting functionality, and without slowing down performance. To withstand attacks, IBOS must ensure any compromised component (1) cannot tamper with data it should not have access to, (2) cannot leak sensitive information to third parties, and (3) cannot access components operating on behalf of different web sites.

In this section we discuss the design principles that guide our design and the overall system architecture. In Section 3.3 we discuss the security policies and mechanisms we use.

3.2.1 Design principles

We embrace microkernel [50], Exokernel [34], and safety kernel design principles in our overall architecture. By combining these principles with our insight about exposing browser abstractions at the lowest software layer we hope to converge on a more trustworthy browser design. Five key principles guide our design:

1. *Make security decisions at the lowest layer of software.* By pushing our security decisions to the lowest layers we hope to avoid including the millions of lines of library and OS code in our TCB.

2. *Use controlled sharing between web apps and traditional apps.* Sharing data between web apps and traditional apps is a fundamental functionality of today's practical systems and should be supported. However, this sharing should be facilitated through a narrow interface to prevent misuse.

3. *Maintain compatibility with current browser security policies.* Our primary goal is to improve the enforcement of current browser policies without changing current web-based applications.

4. *Expose enough browser states and events to enable new browser security policies.* In addition to enforcing current browser policies, we would like our architecture to adapt easily to future browser policies.

5. *Avoid rule-based OS sandboxing for browser components.* Fundamentally, rule-based OS sandboxing is about restricting unused or overly permissive interfaces exposed by today's operating systems. However, sandboxing systems can be complex (the Ubuntu 10.04 SELinux reference policy uses over 104K lines of policy code) and difficult to implement correctly [38, 97]. If our architecture requires OS sandboxing for browser components then we should rethink the architecture.

3.2.2 Overall architecture

Figure 3.1 shows the overall IBOS architecture. The IBOS architecture uses a basic microkernel approach with a thin kernel for managing hardware and facilitating message passing between processes. The system includes user-mode device drivers for interacting directly with hardware devices, such as network interface cards (NIC), and browser API managers for accessing the drivers and implementing browser abstractions. The key browser abstractions that the browser API managers implement are HTTP requests, cookies and local storage for storing persistent data, and tabs for displaying user-interface (UI)

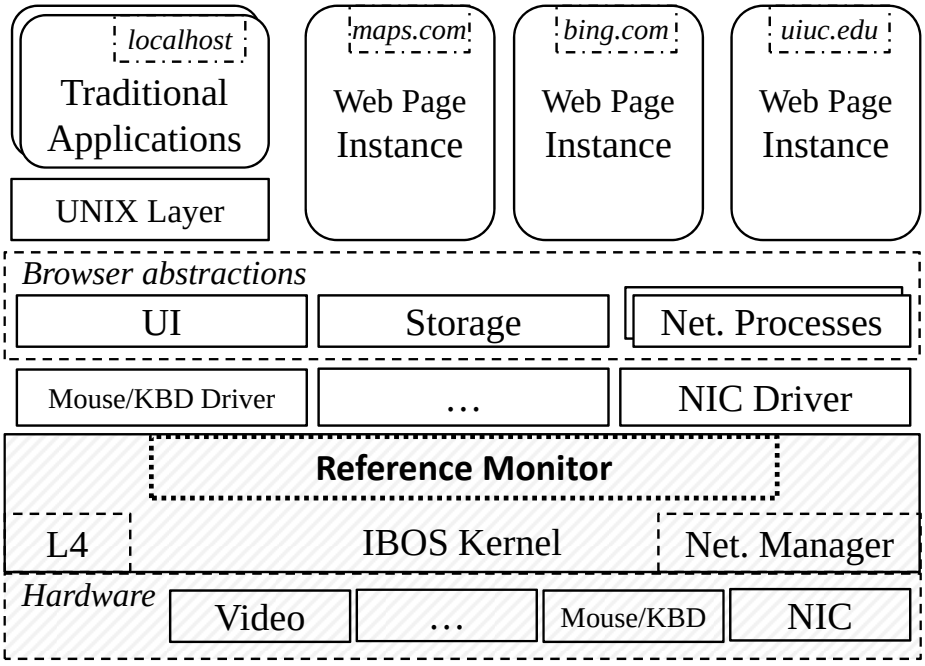

Figure 3.1: Overall IBOS architecture. Our system contains user-mode drivers, browsers API managers, web page instances, and traditional processes. To manage the interactions between these components, we use a reference monitor that runs within our IBOS kernel. Shaded regions make up the TCB.

---

content. Web apps use these abstractions directly to implement browser functionality, and traditional applications (traditional apps) use a UNIX layer to access UNIX-like abstractions on top of these browser abstractions.

The IBOS kernel Our IBOS kernel is the software TCB for the browser and includes resource management functionality and a reference monitor for security enforcement. The IBOS kernel also handles many traditional OS tasks such as managing global resources, creating new processes, and managing memory for applications. To facilitate message passing, the IBOS kernel includes the L4Ka::Pistachio [9] message passing implementation and MMU management functions. All messages pass through our reference monitor and are subjected to our overall system security policy. Section 3.3 describes the policies that the IBOS kernel enforces and the mechanisms it uses to implement these policies.

Network, storage, and UI managers The IBOS network subsystem handles HTTP requests and socket calls for applications. To handle HTTP requests, *network processes* check a local cache to see if the request can be serviced via the cache, fetch any cookies needed for the request, format the HTTP data into a TCP stream, and transform that TCP stream into a series of Ethernet frames that are sent to the NIC driver. *Socket network processes* export a basic socket API and simply transform TCP streams to Ethernet frames for transmission across the network. Only traditional apps can access our socket network processes. The IBOS kernel manages global states, like port allocation.

The IBOS storage manager maintains persistent storage for key-value data pairs. The browser uses the storage manager to store HTTP cookies and HTML5 local storage objects, and the basic object store includes optional parameters, such as `Path` and `Max-Age`, to expose cookie properties to the reference monitor. The storage manager uses several different namespaces to isolate objects from each other. Web apps and network processes share a namespace based on the origin (the `<protocol, domain name, port>` tuple of a uniform resource locator) that they originate from, and web apps and traditional apps share a "localhost" namespace, which is separate from the HTTP namespace. All other drivers and managers have their own private namespaces to access persistent data.

The IBOS UI manager plays the role of the window manager for the system. However, rather than implement the browser UI components on top of the traditional window motif, we opted for a tabbed browser motif. Basic browser UI widgets, called the browser chrome, are displayed at the top of the screen. IBOS displays web pages in tabs and the user can have any number of tabs open for web apps. There is a tab for basic browser configuration and administration, and a tab that is shared by traditional apps. If traditional apps wish to implement the window motif, they can do so within the tab. The main advantage of our browser-based motif is that it enables IBOS to bypass the extra layers of indirection traditional window managers put between applications and the underlying graphics hardware, exposing browser UI elements and events directly to the IBOS kernel. We discuss the security implications of our design decision in more detail in Section 3.3.8.

Web apps, traditional apps, and plugins The IBOS system supports two different types of processes: web page instances and traditional processes. A web page instance is a process that is created for each individual web page a user visits. Each time the user clicks on a link or types a uniform resource locator (URL) into the address bar, the IBOS kernel creates a new web page instance. Web page instances are responsible for issuing HTTP requests, parsing HTML, executing JavaScript, and rendering web content to a tab. Traditional processes can execute arbitrary instructions, and the key difference between a web page instance and a traditional processes is that the IBOS kernel gives them different security labels, which the kernel uses for access control decisions. Web page instances are labeled with the origin of the HTTP request used to initiate the new web page, and traditional processes are labeled as being from "localhost." These two processes interact via the storage subsystem since both types of processes can access "localhost" data.

In general, plugins are external applications that browsers use to render non-HTML content. One common example of a plugin is the Flash player that enables browsers to play Flash content. In IBOS, plugins run as traditional processes, except that they are launched by the browser and the system gives them access to browser states and events through a standard plugin programming interface, called the NPAPI [2].

3.3 IBOS security policies and mechanisms

Our primary goal is to enforce browser security policies from within our IBOS kernel. This section describes the mechanisms that the IBOS kernel uses to enforce the SOP. We also discuss policies and mechanisms for enforcing UI interactions, and we describe a custom policy engine that lets web sites further restrict current policies.

3.3.1 Threat model of IBOS

Our primary goal is to ensure that the IBOS kernel upholds our security policies even if one or more of the subsystems have been compromised. In our threat model, we assume that an attacker controls a web site and can serve arbitrary data to our browser, or that the system contains a malicious

traditional app. We also assume that this malicious data or traditional app can compromise one or more of the components in our system. These susceptible components include all drivers, browser API managers, web page instances, and traditional processes. Once the attacker takes control of these components, we assume that he or she can execute arbitrary instructions as a result of the attack. We focus on maintaining the integrity and confidentiality of the data in our browser. In other words, we would like the user to be able to open a web page on a trusted web server, and interact with this web page securely, even if everything on the client system outside of our TCB has been compromised. Availability is an important, but separate, aspect of browser security that we do not address in this dissertation.

In our system we trust the layers upon which we built IBOS. These layers include the IBOS kernel and the underlying hardware. Like all other browsers, IBOS predicates security decisions based on domain names, so we trust domain name servers to map domain names to IP addresses correctly. Compromising any of these trusted layers compromises the security of IBOS.

3.3.2 IBOS work flow

This section describes a web page instance making a network request to help illustrate the security mechanisms that IBOS uses.

Figure 3.2 shows the flow of how a web page instance fetches data from the network. The user visits a page hosted at `uiuc.edu` and this web page includes an image from `foo.com`. To download the image, (1) the web page instance will make an HTTP request that the IBOS kernel forwards to an appropriate network process. The network process forms a HTTP request, which includes setting up HTTP headers, (2) fetching cookies from the storage subsystem, (3) requesting a free local TCP port to transform this request into TCP/IP packets and Ethernet frames, and (4) sending it to network manager. The network manager notifies the Ethernet driver which (5) programs the NIC to transmits the packet out to the network. When the NIC receives a reply for the request, (6) it notifies the Ethernet driver. The driver subsequently (7) notifies the network manager, which (8) forwards the packet to the appropriate network process. The network process then parses the data and (9) passes the resulting HTTP reply and data to the original

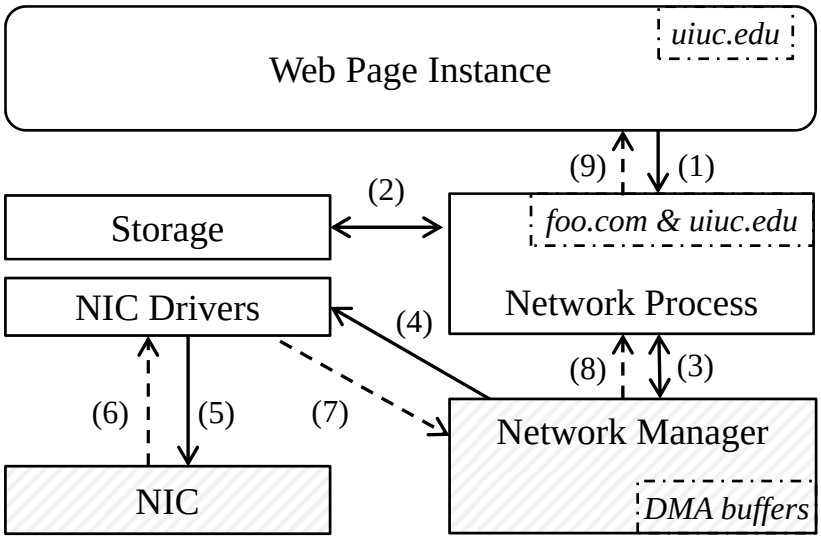

Figure 3.2: IBOS work flow. This figure enlarges the right half of Figure 3.1 and shows how our IBOS subsystems interact when a web page instance from `uiuc.edu` issues a network request to `foo.com`. Subsystems are shown in boxes and solid and dotted arrows represent IBOS messages for outgoing and incoming data respectively. The reference monitor (which is not shown here) checks all these messages to enforce security properties.

---

web page instance.

3.3.3 IBOS labels

To enforce access control decisions, the IBOS kernel labels web page instances, traditional processes, and network processes. IBOS labels specify the resources that a process can access or messages it can receive. Each web page instance has one label, which is the origin of the main HTML document. Each traditional process is labeled as being from "localhost" when they are created. Each network process has an origin label for the network resources it handles and has an origin label for the web page instances that are allowed to access it. IBOS labels the processes upon creation, and keeps the labels unchanged throughout the processes' life-cycle.

An important point is that the IBOS kernel infers the origin labels for web page instances and network processes automatically by extracting related information from the messages passed among them. By inferring labels rather than relying on processes to label themselves, the IBOS kernel ensures that

it has the correct label information, even if a process is compromised.

The *newUrl* and *fetchUrl* IBOS system calls are the two requests that cause the kernel to label processes. The *newUrl* system call is used by web page instances and the UI manager use to navigate the browser to a new URL. The *newUrl* system call consists of two arguments: a URL and a byte array for HTTP POST data. When the IBOS kernel receives a *newUrl* request it will create a new web page instance and set the label for this web page instance by parsing the origin out of the URL argument of the *newUrl* request. When servicing *newUrl* requests, the IBOS kernel will reuse old web page instances (to reduce process startup times), but only when the origin labels match for the old web page instance and the URL argument.

Web page instances use the *fetchUrl* system call to issue HTTP and HTTPS requests to fetch network objects, such as images. The *fetchUrl* system call has two arguments: a URL and HTTP header information. When a web page instance issues a *fetchUrl* system call, the IBOS kernel uses the origin of the web page instance (set by the original *newUrl* call) and the origin of the *fetchUrl* URL argument to find a network process with these same labels, or creates a new network processes and labels it accordingly if an existing network process cannot be found.

More details about how we use these labels for access control decisions are described in the remainder of this section.

3.3.4 Security invariants

For all of our subsystems, we use *security invariants* that are assertions on all interactions between subsystems that check basic security properties. The key to our security invariants is that we can extract security relevant information from messages automatically, and provide high assurance that the system maintains the security policy without having to understand how each individual subsystem is implemented. Using these security invariants, we remove from the TCB almost all of the components found in modern commodity operating systems, including device drivers.

The ideal security invariant is complete, implementation agnostic, executes quickly, and requires only a small amount of code in the IBOS kernel. A complete invariant can infer all of the states needed to ensure the high-level secu-

rity policy, and an implementation agnostic invariant can infer states without relying on the specific implementation of individual subsystems. The IBOS kernel evaluates invariants in the kernel and inline with messages, so security invariants should execute quickly and require little code to implement. In our design we strive to make the appropriate trade offs among these properties to improve security without making the system slow or increasing our TCB significantly. The base security invariant we have is:

SI 0: *All components can only perform their designated functions.*

For example, the UI subsystem can never ask for cookie data or the storage manager cannot impersonate a network process to send synthesized attack HTTP data to a web page instance.

3.3.5 Driver invariants

The two driver invariants the IBOS kernel enforces are:

SI 1: *Drivers cannot access DMA buffers directly.*

SI 2: *Devices can only access validated DMA buffers.*

In our approach, we use a split driver architecture where we separate the management of device control registers from the use of device buffers (SI 1). For example, our Ethernet driver never has access to transmit or receive buffers directly. Instead, it knows the physical addresses where the IBOS kernel stores these buffers, and it programs the NIC to use them. By separating these two functions we can interpose on the communications between them to ensure that IBOS upholds browser security policies, even if an attacker completely compromises a shared driver.

Using this split architecture, processes fill in device-specific buffers for DMA transfers, and the IBOS kernel infers when drivers initiate DMA transfers to ensure that the driver instructs the device to use a verified DMA buffer (SI 2). Fortunately, DMA buffers tend to use well-defined interfaces, like Ethernet frames for Ethernet drivers, so the IBOS kernel can readily glean security relevant information from these DMA buffers before the device accesses them. Unfortunately, the interface between drivers and devices

is device-specific, so the IBOS kernel must have a small state machine for each device to properly infer DMA transfers. However, we found this state machine to be quite small for the devices that we use in IBOS.

In IBOS we implement a driver for the e1000 NIC, a VESA BIOS Extensions driver for our video card, and drivers for the mouse and keyboard.

3.3.6 Storage invariants

The primary invariant we strive to enforce in the storage manager is:

SI 3: *All of our key-value pairs maintain confidentiality and integrity even if the storage stack itself becomes compromised.*

To enforce this invariant, our IBOS kernel encrypts all objects before passing them to the storage subsystem. To encrypt data, the IBOS kernel maintains separate encryption keys for all of the namespaces on the IBOS system. These namespaces include separate namespaces for HTTP cookies based on the domain of the cookie, separate namespaces for web page instances based on the origin of the page, separate namespaces for each of our subsystems, and a separate namespace for all traditional apps. When the IBOS kernel passes a request to the storage manager it will append the security labels, a copy of the key from the key-value pair, and a hash of the contents to the payload before encrypting the data and passing it to the storage subsystem. When the IBOS kernel retrieves this data, it can decrypt the data and check the labels and integrity of the information. By using encryption, the IBOS kernel does not need to implement security invariants for any of our storage drivers, and our storage subsystem is free to make data persistent using any mechanisms it sees fit, such as the network (like in our implementation) or via a disk-based storage system.

Our current implementation does not make any efforts to avoid an attacker that deletes objects or replays old storage data. For web applications this limitation has only a small effect because the cookie standards do *not* require browsers to keep cookies persistently and because web applications often limit the lifetime of cookies using expiration dates, which are also part of the cookie standard. However, if this limitation did become problematic, we could apply the principles learned from distributed or secure file systems to provide stronger guarantees.

3.3.7 Network process invariants

Our IBOS kernel maintains five main invariants for network processes:

SI 4: *The kernel must route network requests from web page instances to the proper network process.*

SI 5: *The kernel must route Ethernet frames from the NIC to the proper network processes.*

SI 6: *Ethernet frames from network processes to the NIC must have an IP address and TCP port that matches the origin of the network process.*

SI 7: *HTTP data from network processes to web page instances must adhere to the SOP.*

SI 8: *Network processes for different web page instances must remain isolated.*

To help enforce these invariants, IBOS puts all network processes in their own protection domains. If a web page instance makes a HTTP request, the kernel will extract the origin from the request message and either route this request to an existing network process that has the same label, or it will create a new network process and label the network process with the origin of the HTTP request. Likewise, the kernel inspects incoming Ethernet frames to extract the origin and TCP port information, and routes these frames to the appropriately labeled network process. By putting network processes in their own protection domains, the kernel naturally ensures that network requests from web page instances and Ethernet frames from the NIC are routed to the correct network process (SI 4) (SI 5).

To ensure that the NIC sends outgoing Ethernet frames to the correct host, the IBOS kernel checks all outgoing Ethernet frames before sending them to the NIC to check the IP address and TCP port against the label of the sending network process (SI 6). Also, the IBOS kernel checks cookies before passing them to the network process to ensure that all of the origin labels adhere to cookie standards. By performing these checks, the IBOS kernel ensures that the NIC sends outgoing network requests to the proper host and that the request can only include data that would be available to the server anyway.

To enforce the SOP, the IBOS kernel inspects HTTP data before forwarding it to the appropriate web page instance and drops any HTML documents from different origins (SI 7). To inspect data, the kernel uses the content sniffing algorithm from Chrome [15] to identify HTML documents so the kernel can check to make sure that the origin of HTML documents and the origin of the web page instance match. This countermeasure prevents compromised web page instances from peering into the contents of a cross-origin HTML document, thus preventing the compromised web page instance from reading sensitive information included in the HTML document.

To help isolate web page instances from each other, we also label network processes with the origin of the web page instance (SI 8). This second label is used only for network access control decisions and does *not* affect the cookie policy, which is predicated on the origin of the network request. To access network processes, the origin of the web page instance must match the origin of this second label. By using this second label, the IBOS kernel isolates network requests from different web page instances to the same origin. As a result of this isolation, a web page instance that is served a malicious network resource (e.g., a malicious ad [77]) that compromises a network process remains isolated from other web page instances. If an attacker can compromise a network process, IBOS limits the damage to the web page instance that included the malicious content.

3.3.8 UI invariants

The three UI invariants that the IBOS kernel enforces are:

SI 9: *The browser chrome and web page content displays are isolated.*

SI 10: *Only the current tab can access the screen, mouse, and keyboard.*

SI 11: *The URL of the current tab is displayed to the user.*

The key mechanisms that our UI subsystem uses to provide isolation are to use a frame buffer video driver and page protections to isolate portions of the screen (SI 9). Our video driver uses a section of memory, called a frame buffer, for writing to the screen. Processes write pixel values to this frame buffer and the graphics card displays these pixels. Although our mechanism

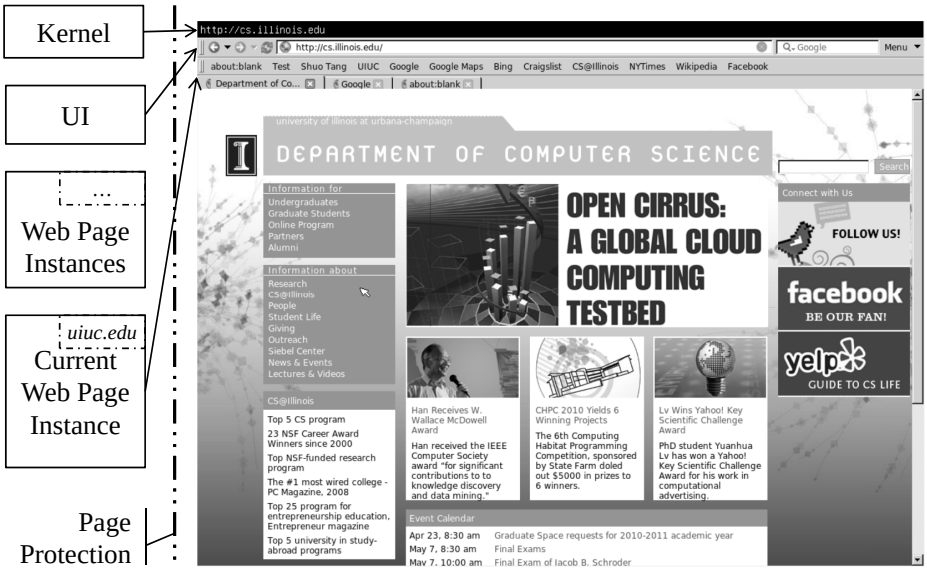

Figure 3.3: IBOS display isolation. This figure shows how IBOS divides the display into three main parts: a bar at the top for the kernel, a bar for browser chrome, and the rest for displaying web page content. The IBOS kernel enforces this isolation using page protections and *without* relying on a window manager.

---

makes heavy use of the software rastering available in Qt Framework [5], our experiences and anecdotal evidence from the Qt developers shows that software rastering can perform roughly as fast as native X drivers running on Linux [8]. The key advantage of our approach is that the IBOS kernel can use standard page-protection mechanisms to isolate portions of the screen. Although our current implementation does not support hardware acceleration, we believe that our techniques will work because the IBOS kernel can interpose on standardized acceleration hardware/software interfaces, such as OpenGL and DirectX.

To provide screen isolation, we divide up the screen into three horizontal portions (Figure 3.3). At the top, we reserve a small bar that only the IBOS kernel can access. We use the next section of the screen for the UI subsystem to draw the browser chrome. Finally, we provide the remainder of the screen to the web page instance. To ensure that only one web page instance can write to the screen at any given time, we only map the frame buffer memory region into the currently active web page instance and we only route mouse and keyboard events to this currently active web page instance (SI 10).

To switch tabs, the UI subsystem notifies the IBOS kernel about which tab is the current tab, and the IBOS kernel updates the frame buffer page table entries appropriately. However, a malicious UI manager could switch tabs arbitrarily and cause the address bar and the tab content to become out of sync (e.g., shows a page from `attacker.com`, but claims the page comes from `uiuc.edu`). One alternative we considered for this UI inconsistency was interposing on mouse and keyboard clicks to infer which tab the user clicked on, and also performing optical character recognition on the address bar to determine the address that the UI manager is displaying. However, tracking this level of detail would require far too much implementation specific information and would require the IBOS kernel to track additional events like a user switching the order of tabs.

Our approach for the IBOS kernel is to use the kernel display area to display the URL for the currently visible web page instance (SI 11). The kernel derives the URL from the label of the currently visible web page instance, providing high assurance that the URL the kernel displays matches the URL of the visible web page instance without tracking implementation specific states and events in the UI manager. Although this security invariant appears simple, it is something that modern web browsers have had trouble getting right [24].

3.3.9 Web page instances and `IFRAME`s

The IBOS kernel creates a new web page instance each time a user clicks on a link or types a new URL in the address bar. To enforce the SOP on `IFRAME`s, we run cross-origin `IFRAME`s in separate web page instances. This separation allows us to fully track the SOP using kernel visible entities. To facilitate communication between web page instances and the `IFRAME`s that they host, we marshal `postMessage` calls between the two.

Our current display isolation primitives are coarse grained and we rely on the web page instance to manage cross-origin `IFRAME` displays even though `IFRAME`s run in separate protection domains. However, current display policies allow web page instances to draw over cross-origin `IFRAME`s that they host, so this design decision has no impact on current browser policies. One potential shortcoming of this display management approach is that compro-

mised web page instances can read the display data for embedded `IFRAME`s. Fortunately, many sites with sensitive information, like `facebook.com` and `gmail.com`, use frame busting techniques [60] to prevent cross-origin sites from embedding them, which the IBOS kernel can enforce.

3.3.10 Custom policies

Our main focus of this project is being able to enforce current browser policies from the lowest layer of software. However, we also want to create an architecture that exposes enough browser states and events to enable novel browser security policies. Attacks such as XSS operate within traditional browser policies and can be difficult to prevent without relying on the HTML or JavaScript engine implementations. Although our architecture cannot prevent XSS, our goal is to prevent these types of attacks from causing damage.

One mechanism we implement in IBOS is to give a web server the ability to create its own more restrictive security policy to prevent attacks from sending sensitive information to third-party hosts. In our custom policy, we allow web sites to specify a server-side policy file that IBOS retrieves to restrict network accesses for a web page instance, similar to Tahoma manifests [28]. For example, assume that a bank website located at `http://www.bank.com` creates a policy file at `http://www.bank.com/.policy` that specifies the online bank system can only access resources from `www.bank.com` or `data.bank.com`. IBOS retrieves the policy file and automatically applies a more restrictive policy for the online bank web application. This restrictive policy prevents an attacker from sending stolen information to a third-party host, providing an additional layer of protection for the web application.

3.4 Implementation

The implementation of IBOS is divided into three parts: the IBOS kernel, IBOS messaging passing interfaces, and IBOS subsystems. The IBOS kernel is implemented on top of the L4Ka::Pistachio microkernel and runs on X86-64 uniprocessor and SMP platforms. We modified L4Ka to improve its support for SMP systems. The IBOS kernel schedules processes based on a static priority scheduling algorithm.

The IBOS kernel provides three basic APIs (i.e., `send()`, `recv()`, and `poll()`) to facilitate message passing. Applications use `send()` and `recv()` for communication and call `poll()` to wait for new messages. The IBOS kernel intercepts all messages and automatically extracts the semantics from them, like creating a new web page instance or forwarding cookies to network processes. Then the kernel inspects the semantics to make sure they conform to all security invariants and policies that we described in previous sections.

The IBOS subsystems implements APIs for web browsers and traditional applications. They are built on top of an IBOS-specific uClibc [7] C library, lwIP [31] TCP/IP stack and the Qt Framework. The web browser also uses an IBOS-specific WebKit to parse and render web pages.

To support traditional apps, we use our uClibc and Qt implementations to provide access to browser abstractions using the UNIX-like abstractions of the C runtime, and GUI support from Qt. We use a few Qt sample programs for testing and we implement one plugin. Our plugin is a PDF viewer that uses the Ghostscript PDF rendering engine with bindings for Qt.

3.5 Summary

In this chapter, we presented IBOS, an operating system and web browser co-designed to reduce drastically the trusted computing base for web browsers and to simplify browsing systems. To achieve this improvement, we built IBOS with browser abstractions as first-class OS abstractions and removed traditional shared system components and services from its TCB. With our new architecture, we showed that IBOS enforced traditional and novel security policies, and we argued that the overall system security and usability could withstand successful attacks on device drivers, browser components, or traditional applications.

CHAPTER 4

THE OP2 SECURE WEB BROWSER

In this chapter, we present a standalone secure browser architecture that can be used on top of commodity operating systems. When a specialized operating system like IBOS is not desired, we argue that the architecture we propose in this chapter could give a user the maximum protection for web browsing.

4.1 Introduction

Flaws in the design and architecture of today's web browsers allow the trend of web-based exploitation to continue. Modern web browser design continues to support the original model of browser usage where users viewed several different static pages and the browser itself was the application. However, recent web browsers have become a platform for hosting web apps, where each distinct page (or set of pages) represents a logically different application, such as an email client, a calender program, an office application, a video client, a news aggregate, etc. The single-application model provides little isolation or security between distinct applications hosted within the same browser, or between different applications aggregated on the same web page. A compromise occurring on any part of the browser, including plugins, results in a total compromise of all web-based applications running within the browser.

In our previous work we decomposed a browser into smaller subsystems to separate the security policy enforcement of a browser from the implementation of the browser with the OP web browser [46]. OP decomposes the browser into five main subsystems: the web-page subsystem, a user-interface (UI) subsystem, a network subsystem, a storage subsystem, and a browser kernel. The subsystems all communicate using message passing, and all mes-

sages pass through the browser kernel, allowing the browser kernel to enforce many browser access control decisions. By applying these basic operating system principles to the OP design, we provided stronger isolation between web apps when compared to the state of the art at the time.

After the design and implementation of OP there has been significant effort by both the industry and research communities to design secure web browsers. After OP, two additional web browsers were designed to improve the state of theart in browser security: Google Chrome and Gazelle [18, 104]. OP, Chrome, and Gazelle share many common features and each browser was designed from the ground up with security goals in mind. In this chapter, we present the design of the second version of OP, called OP2, that combines techniques from OP, Chrome, and Gazelle.

Our goals in the design of OP2 are:

- *Security*: By drawing on the contributions of all three browsers we hope to achieve a more secure web browser.
- *Performance*: Architectural decisions can impact the speed of the web browser, the browser architecture should have low overhead compared to other, modern web browsers.
- *Compatibility*: Browser design should have a minimal impact on the compatibility of the browser with today's web pages.
- *Few lines of code*: OP2 should require as few lines of code as possible to keep the design simple and easy to reason about. A small code base also enables it to be used by the research community and simplifies maintenance.

4.2 The OP2 architecture

OP2 shares a similar architecture of the original OP web browser as shown in Figure 4.1. Our browser still consists of five main subsystems: the web page instances, a network component, a storage component, a user-interface (UI) component, and a browser kernel. Each of these subsystems run within separate OS-level processes. And we use OS-level sandboxing techniques

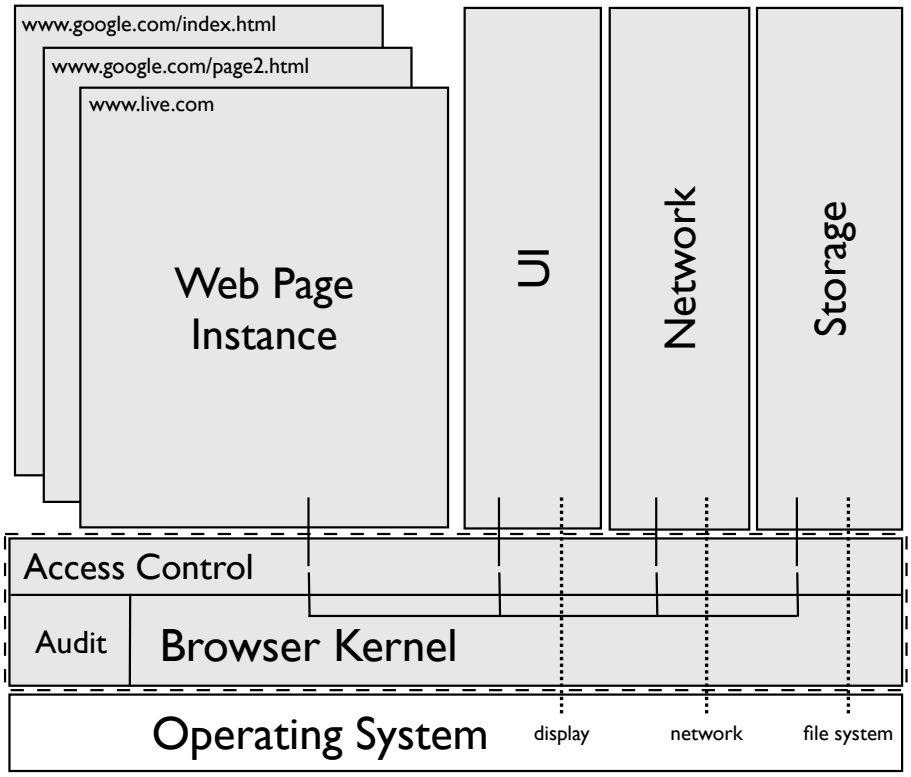

Figure 4.1: Overall architecture of the OP2 web browser. Our web browser contains five main subsystems: browser kernel, storage subsystem, network subsystem, user-interface subsystem, and web page instances; each of these subsystems run within separate OS-level processes.

to limit the interactions of each subsystem with the underlying operating system. The browser kernel manages the communication between each subsystem and between processes, and the browser kernel manages interactions with the underlying operating system.

4.2.1 Threat model of OP2

Similar to OP, the OP2 web browser was designed to operate under malicious influence. We consider attacks that originate from a web page, potentially targeting any part of the browser. We assume the attacker can have complete control over the content being served to the web browser. A browser compromise could be any sort of attack provided in this manner, with an attack that results in code execution being the most capable form of exploit.

The OP2 web browser does not address attacks that operate within modern browser security policy, such as XSS and CSRF. Our goal is to prevent bugs in the web browser application from degrading the overall security of the browser. For example, memory corruption bugs in the JavaScript interpreter and logic errors within a browser plugin are both types of browser bugs that OP2 attempts to mitigate. Techniques that secure web applications and provide greater protection against XSS, CSRF, and other web app attacks (e.g. in Chapter 5) are orthogonal to this work and can be used in conjunction with the OP2 web browser.

We trust the layers upon which OP2 is built. Namely, we trust the underlying operating system and libraries to enforce isolation for our subsystems. As with other current browsers, we trust DNS names for labeling our security contexts. If an attacker compromises any of these entities, the security of our browser is at risk.

4.2.2 Kernel architecture

OP has a minimalistic browser kernel design that facilitates message passing between browser processes. The browser kernel is a microkernel by design and has dedicated processes that provide access to system resources. In addition to message passing, the browser kernel uses an access control policy to restrict messages between browser processes and implements all of the

security policy for OP.

OP2 uses the original OP design for the browser kernel because the microkernel approach simplifies our implementation of the browser kernel while still providing the necessary information for our access control policies. We have found that the additional latency from message passing required in a microkernel design adds very little overhead because of added parallelism, as shown in Section 7.2.2. We do perform some simple optimizations in the browser kernel to enhance the speed of OP2, which we discuss in Section 4.4.

4.2.3 Process models

To be consistent with the prior description of OP, we call all of the processes responsible for executing content in a single page the web page instance, Figure 4.1 shows the logical grouping of processes into a web page instance in OP2.

The standard operation of OP uses a different web page instance for each page. In OP, each web page instance consists of a JavaScript, HTML, VNC, and multiple plugin processes.In OP2, however, we have chosen to combine each of the components in a web page instance into a single process with plugins remaining in a separate process. Navigation between pages causes new web page instances, and each tab is backed by a separate web page instance. New windows that are created by JavaScript are run in separate web page instances as in the original OP design. This change in architecture from OP to OP2 enables OP2 to reuse a substantial amount of WebKit [6] with minimal modifications to support the OP2 architecture. From a security perspective, the browser kernel still exposes the same set of system calls to each web page instance and is able to enforce modern browser security policy.

We change little of the functions of other subsystems in OP2. Our UI subsystem is designed to isolate content that comes from web page instances, and provide common utilities for web browsing such as address bar, bookmarks, and navigation buttons. Any time the web browser needs to store or retrieve a file, it is accomplished through the UI to make sure the user has an opportunity to validate the action using traditional browser UI mechanisms. This decision is justified, since users need the flexibility to access the file system to download or upload files, but our design reduces the likelihood

of a UI subsystem compromise. Providing user interaction for file operation prevents drive-by download attacks as well as arbitrary uploading of files from the user's file system.

We still provide components for accessing the file system or the network since the web page instance cannot access them directly. The storage component stores persistent data, such as cookies, in an SQLite database. SQLite stores all data in a single file and handles many small objects efficiently, making it a good choice for our design, since it is nimble and easy to sandbox. A separate network subsystem implements the HTTP protocol and downloads content on behalf of other components in the system.

4.2.4 Displaying page content

OP uses VNC for rendering in one process and displaying the rendered content in another. This allowed OP to achieve isolation between the rendering engine and the user interface, a desirable property to prevent parsing and rendering bugs from interfering with the user's interaction with the browser.

OP2 eliminates the VNC process for display to combine as many of the processes in a web page instance as we can. In place of VNC based rendering, OP2 uses native window reparenting supported by the window manager (i.e. Xorg, X11, or Windows). Window reparenting is fast and requires no redirection of rendered content between processes, so it is as fast as displaying windows natively. It is also simple and directly supported by the Qt toolkit. The primary disadvantage is that the web page instance requires direct interaction with the window manager. Direct interaction with the window manager limits OP2 to specific operating systems and exposes the window manager to potentially compromised browser components.

4.2.5 Cross-origin `IFRAME`

Display security refers to the browser's ability to isolate different-origin content inside of a single page. In OP – the browser is able to isolate plugins in a separate process and quarantine vulnerabilities and bugs in plugins. Gazelle introduced frame isolation that provides isolation for cross-origin frames in web pages [104]. Gazelle's mechanism works by creating a new web page

instance for the frame if it belongs to a different origin than that of the including page.

In OP2 we adopted the Gazelle design and isolate cross-origin frames and plugins using separate web page instances. In OP2, if a frame is encountered by the web page instance, a new web page instance is automatically created. Then, using window reparenting techniques, the frames are placed in the correct position on the page. Like Gazelle, OP2 isolates frames and uses content sniffing [15] to prevent a compromised web page instance from reading cross-origin HTML content. We cannot prevent a compromised web page instance from requesting sensitive cross-origin scripts, CSS, or images, since this is allowed within same-origin policy and available to a uncompromised web page instance. Gazelle has similar limitations, with the added ability to render cross-origin images in separate web page instances.

This policy does differ from popular browsers, such as Firefox and Internet Explorer, and since this policy is more restrictive it can cause incompatibilities in the display of a page.

4.3 Improving compatibility

OP2 has improved compatibility for web pages compared to OP. While some benefit comes from our use of the WebKit HTML and JavaScript engines, we have devoted significant effort to make OP2 compatible with many different web pages. In order to do this we have included additional support for JavaScript to use our messaging API. For example, the XMLHttpRequest object is able to use the OP2 network component through the messaging API. We have also provided access to cookies within our storage component from JavaScript again by using the messaging API.

Plugins have also received significant compatibility work. Currently we support any plugin that uses the Netscape Plugin API (e.g., Flash, Java runtime, and Silverlight). Our previous implementation required customized support for each plugin to redirect calls into our messaging infrastructure. We recognize there are potential compatibility problems with plugin policies that place restrictions on plugins such as Flash and are currently working on a more permissive policy. Loosening the plugin security policy can open the door for potential exploit, and we are currently working on developing new

browser policies that provide greater compatibility for plugins and evaluating the cost to compatibility [45].

4.4 Optimizations in OP2

The original version of OP performed approximately as fast as Firefox 2 [46]; however, since then browsers have seen substantial enhancements that improve performance. In our preliminary implementation of OP2 we found that OP2 added measurable overhead when compared to other WebKit-based browsers. The problem was that WebKit improved performance so significantly that the latency we added to certain key operations, such as downloading and displaying content for a new web page, caused a noticeable delay in the overall page load latency time.

The fundamental issues in the original version of OP2 that caused increased overhead were using new processes for each individual web page instance and serializing slow operations, such as windows reparenting. By using new processes for each individual web page we had to pay the process initialization cost each time a user visited a new web site, and we precluded ourselves from using any WebKit optimizations that assumed process reuse, such as in-memory object caches. By serializing window reparenting we added overhead directly to page load latency times.

To reduce this overhead, we optimized our web page instance management and parallelized slow serial operations. Specifically, we improved the load time of web pages in OP2 by using process pre-creation, parallelizing window manager operations, caching previously-used processes, and loading frames in parallel. The key to our optimizations is that we improve performance without compromising the security assurances of our OP2 architecture.

4.4.1 Process pre-creation

OP2 creates a new web page instance for each page and each web page instance requires a new process. During normal use, multi-process browser architectures such as OP, Chrome, and Gazelle all use more processes than their monolithic ancestors. As a result, the time it takes to initialize a new process can add overhead. To avoid this added overhead, we pre-create web

page instances during idle times and use these pre-created web page instances to service new web page requests, thus avoiding process initialization overhead for new web page instances.

4.4.2 Parallelizing window manger operations

As discussed in Section 4.2.4, OP2 uses window manager reparenting to combine the display from multiple processes. During our testing of OP2 we found that this design decision requires around 0.51 seconds to complete. In order to eliminate this time from the overall page load time, we perform the window reparenting asynchronously and allow the web page instance to load while the window manager reparents the window. Although this optimization does not eliminate the cost of reparenting windows, it does mask the cost while the browser loads the page.

4.4.3 Process caching

WebKit uses an in-memory cache to provide fast retrieval of objects, such as scripts and stylesheets. This object cache improves the page load latency times by reusing web page resources shared between pages on subsequent visits. Since OP2 creates new processes for each web page, our original design was unable to use the object cache that WebKit provides.

To take advantage of the in-memory object cache, we implemented a process cache that reuses old web page instances for new web pages. When the browser navigates away from a web page, the previous web page instance could be killed and cleaned up by the browser. However, rather than removing these old web page instances, we add them to a cache that the browser kernel uses to service new web page instances. Each time the browser visits a new page, the browser kernel will first check the process cache to see if there are any web page instances that can be reused. If the browser kernel finds a suitable web page instance, then it is used for the new request, thus enabling the use of the in-memory object cache. If no suitable web page instances are found, then it will use a web page instance from the pre-created process pool for the new request.

One design decision we had to make was determining how to calculate

cache hits for our process cache. One way we could have calculated cache hits was by matching the origin of the request and the origin of cached web page instances. For example, if there was a web page instance in the process cache that was used for `docs.google.com` and the browser issued a request for `gmail.google.com`, then it would reuse the `docs.google.com` web page instance since both are from `google.com`. This approach has the advantage of improving the hit rate by matching pages liberally, but has the disadvantage of carrying around the state from the previous page, potentially leaking information to the new web page instance. Instead, our approach is to calculate cache hits based on the full URL of the request. This approach has the advantage of carrying around state for only the exact web page that was requested, thus minimizing potential information leakage, but will have a lower hit rate than an origin-based scheme.

4.4.4 Parallelizing frames

Section 4.2.5 describes how OP2 performs frame isolation and display protection. In addition to improving security, isolating frames and running each frame in a separate web page instance could potentially improve performance. Since frames from different origins run in separate processes, OP2 naturally parallelizes the execution and rendering of frames. This additional parallelism can mask the latency added by our frame isolation techniques and might be able help improve performance on today's multicore systems.

4.5 Implementation

To implement OP2 we choose to use the Qt framework [5], and WebKit [6]. We implement all the subsystems using C++, and use SELinux [63] as an option for sanboxing them. To enable window reparenting, we use XEmbed protocol provided in X11 in Linux platform. To have better maintainability, we choose to extend the framework of Qt and WebKit instead of modifying their internal implementation. For example, WebKit's Qt port uses a default `QNetworkAccessManager` object to provide network capability. Instead of changing the network module in WebKit, we provide a subclass of `QNetworkAccessManager` that uses OP2's network subsystem. By do-

ing this, we achieve minimal patch to original Qt and WebKit code base. If WebKit or Qt is upgraded, we would easily be able to adopt the new features into OP2. The whole set of source code of OP2 is available at `http://code.google.com/p/op-web-browser/`

4.6 Summary

In this chapter we have described the OP2 web browser and the different elements that make our browser secure and practical. By following the six principles we discussed in Chapter 1, we are able to create a web client capable of withstanding attacks. Specifically, we partition the browser into smaller subsystems and make all communication between subsystems simple and explicit. At the core of our design is a small browser kernel that manages the browser subsystems and interposes on all communications between them to enforce browser security policies.

The OP2 web browser is responsive to user interaction and implements features that make it compatible with current web pages. We have demonstrated that, by design, the OP2 web browser is not vulnerable to many forms of browser attacks while not limiting the full functionality of the browser.

CHAPTER 5

FORTIFYING WEB APPS AUTOMATICALLY

The browsing systems we described in previous chapters provide secure foundations for web browsing. Most of the architectural efforts focus on isolating and containing vulnerabilities in web apps. In this chapter, we discuss browser-based mechanisms that try to add protection to the web apps themselves.

5.1 Introduction

The Web has become a popular platform for building web apps and providing convenient and diverse services for users. One contributing factor in this rise in popularity is the features that browser developers add to browsers. Unfortunately, these new features have also created new avenues for attack. For example, web app developers can use frames (or `IFRAME`s) to compose web apps out of gadgets from different websites, but attackers can use frames to embed legitimate web apps inside of attack pages to trick users via "clickjacking" [49].

Browser developers have implemented security features to help web developers improve the security of web apps. Three examples of recent browser security features are `HttpOnly` cookies [4] that enable web developers to specify cookies that should be inaccessible from JavaScript, `X-Frame-Options` [60] to enable web developers to prevent their pages from being framed, and `JSON.parse()` [3] to enable web developers to deserialize JavaScript Object Notation (JSON) text safely and without executing JavaScript code.

However, web developers have been slow to use these new browser security features [82, 90]. We surveyed the Alexa top 100 websites [10], and found that these security mechanisms are not used widely:

- There are at least 34 websites that do not set the `HttpOnly` attribute on their credential cookies.
- Only 11 websites use `X-Frame-Options` to prevent their main or login pages from being framed.
- Of the 16 websites that use JSON within five seconds after the page loads, and only four of them use `JSON.parse()` to deserialize JSON text.

In this chapter we present ZAN – a browser-based system that fortifies web apps by applying new security mechanisms to existing web apps automatically. Our key insight is that the browser often has enough information to determine when new security features could be applied to existing web applications. This information can come from detecting common patterns in the code that web developers write or from identifying fundamental features of key web-app objects, like cookies. Our goal is to add simple mechanisms to narrow the attack surface or mitigate the damage of a web-based attack without requiring input from users or additional effort from web developers. We also aim to minimize incompatibilities induced by our system.

In general ZAN works by interposing on key states and events within the browser to detect candidates for applying stronger security mechanisms automatically. For example, ZAN inspects all cookies set by the web server to detect certain key words (e.g., "token" or "session") or randomness (e.g., hashed values) web apps tend to use within authentication cookies. When ZAN detects a combination of these conditions, it sets the `HttpOnly` attribute for these cookies to make them inaccessible from JavaScript, preventing authentication cookie theft via XSS attacks.

Our contributions are:

- We design and implement ZAN, a browser-based system for applying new security features to existing web apps.
- We show that web apps often contain enough information to allow ZAN to infer opportunities for applying new security mechanisms to legacy web apps.

5.2 Design

In this section, we discuss the threat model that ZAN considers, and the high-level design of ZAN. We also describe the websites we study for ZAN.

5.2.1 Threat model of ZAN

Our primary goal is to narrow the attack surface of web apps or to mitigate the damage of a successful attack. In our threat model, we assume that an attacker controls a malicious website and can serve sophisticated crafted web apps to the user, or that the attacker could escape sanitization processes to inject malicious scripts into legitimate web apps. We focus on non-memory based attacks and assume that the browser ZAN uses is faithful. Attacks that take control over a browser are still possible, such as a scenario involving buffer overflow. In these cases, it would require a better design and architecture of the browser system to improve the overall system security as we will present in later chapters. It is also possible that the attacker could completely compromise a trusted website, rendering ZAN – a client-side system – ineffective. This is a separate aspect of web security that we do not address in this dissertation.

5.2.2 Design principles

As shown in Figure 5.1, ZAN works as a module in browsing systems by interposing on key states and events to infer opportunities to apply security mechanisms. In general, ZAN is feasible because web apps often present enough information at the client side to filter out important objects or identify existing protection logics. For example, cookies that contain authentication tokens are likely obfuscated and have special names. A JavaScript-based defense workaround, once proposed, is always copied and pasted into multiple websites (e.g., frame busting we will discuss in Section 5.4).

In designing and implementing ZAN's different mechanisms, we follow the these principles:

1. *Use only information at the client side.* We already see slow adoption of new secure mechanisms in web app development. We hope a pure

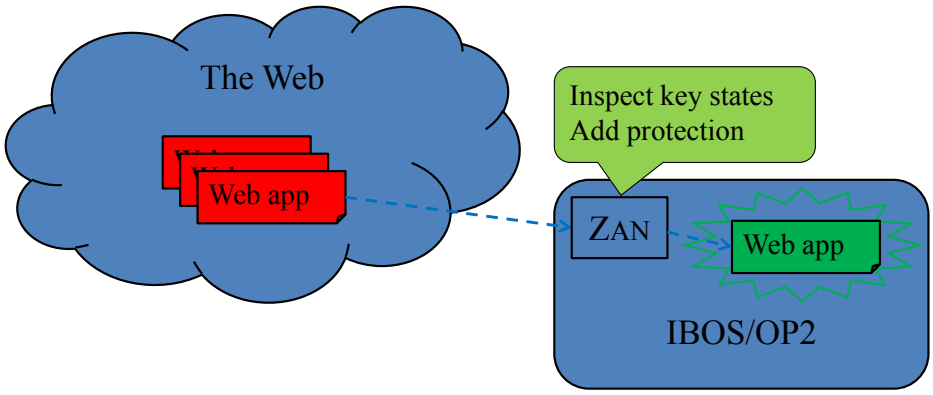

Figure 5.1: Deployment of ZAN. ZAN works by interposing on key states and events of web apps within the browser to detect candidates for adding protection automatically.

---

client-side solution could relieve the burden of web app developers.

2. *Make the mechanisms simple.* Browsers are already complex artifacts, it is desired to ensure the new mechanisms simple and efficient to avoid introducing new vulnerabilities.

3. *Maintain compatibility of the web apps.* It is necessary not to break the Web. A protection mechanism that results in loss of function is practically useless.

5.2.3 Websites used for testing

For ZAN, we use a consistent set of websites as test cases. We first pick the top 100 websites according to Alexa [10]. Because shopping and banking websites contain valuable data, we also include the top six websites in each category according to Alexa. Of course, if one of these is also in the top 100 websites, we skip it. We also use the top four web-mail sites according to comScore.

In this set of websites, some offer several services using a single domain and employ different security mechanisms among those services. To avoid ambiguity, we choose to analyze the main service, or the front page one, when we refer to a top website. For example, Google offers searching, calendar, documents, and many more within `google.com`. But we always refer to

Google search when we talk about `google.com`. In the remaining of this chapter, *top websites* are always used to refer to the 116 websites described in this subsection unless explicitly stated otherwise.

5.3 Case study: `HttpOnly` cookie

The first security augment that ZAN enables is adding `HttpOnly` attributes to credential cookies automatically. We begin the discussion with cookie related issues as it is chronologically the first available feature among the three we cover in this chapter. And `HttpOnly` is the earliest and most wildly used one among the three security mechanisms.

5.3.1 Cookies

A cookie, or an HTTP cookie, is a piece of text stored on a user's computer by his or her browser, typically consisting of one or more name-value pairs that the server and client pass back and forth. Cookies were invented by Lou Montulli at Netscape in 1994 to facilitate electronic commerce applications [58]. Initially developed as a method for implementing reliable virtual shopping carts, cookies were later pervasively used as the *de facto* way of authenticating users to web sites and storing the login information so that a web user does not have to keep entering their username and password each time he or she visits a same web site. Cookies can also be used to store identifiers so that web servers can track what the users have done during the visit.

In today's Web, part of cookie manipulation, like other computation, is pushed to the client side. For example, in Facebook's user sampling and tracking module, the session identifier is generated using JavaScript in the browser. A web app could also use `document.cookie` to set a cookie and then try to read it back to test if the browser has cookie support. As web apps continue to provide more versatile features, they also need a way for access local storage. Before HTML5 local storage [101], web developers chose to use cookies for storing data on the client.

5.3.2 Attacks on cookies

Since cookies often contain time sensitive information, such as credentials, and are relatively easy to access using client-side scripts, cookie theft is a common result in Web-based attacks, such as XSS. For example, in a successful XSS attack, the attacker from `attack.com` could easily steal victim's cookie using the following script:

```
var url
  = 'http://attack.com/stole.cgi?text='
  + escape(document.cookie);
var img = new Image();
img.src = url;
```

In the malicious script, the attacker uses `document.cookie` to retrieve the content of the list of cookies that the user has for the page, and embeds it into the query payload of a fake URL pointed to the attacker's web site. The attacker then creates a JavaScript image object on the fly and set its source path to the fake URL. As a result, this list of cookies is sent to the `attack.com` server.

Cookies also pose privacy threat [86] and enable the CSRF attack [113]. Mitigation methods are feasible but beyond the scope of this dissertation [16].

5.3.3 Alleviating cookie theft

An intuitive way to stop cookie theft in Web-based attacks is to address XSS. However, despite of many efforts of preventing XSS, such as client-side approaches [41, 81], server-side approaches [98], or hybrid client-server approaches [48, 70], XSS remains the top vulnerability [94].

There are also other ways of alleviating cookie theft. One could use HTTP authentication instead of a cookie-based approach. As the authentication information is not available to JavaScript in the cookies, nothing could be revealed to an attacker with an XSS attack. Also, one could also tie session cookies to the IP address that the user originates from and only permit that IP to use the cookies, rendering the stolen cookie useless in most situations. But it is possible that an attacker could spoof the IP address, or is behind

the same Network Address Translation (NAT) firewall or web proxy, thus breaking down the protection.

A declarative method, `HttpOnly`, was also been proposed. First introduced in 2002 in Internet Explorer 2.0 [4], the `HttpOnly` cookie attribute has been implemented in all major browsers. If the optional `HttpOnly` flag is included in the HTTP response header for a cookie, the cookie cannot be accessed by client-side scripts. As a result, the browser would not reveal authentication cookies to the attacker in an XSS attack if they are properly tagged with `HttpOnly` flags.

Surprisingly, the `HttpOnly` attribute has not been throughout deployed in today's Web, even though it was invented almost 9 years ago. For top websites, our survey shows that of the 93 websites that we are able to obtain accounts for and login to, 39 still have not incorporated `HttpOnly`.

5.3.4 Applying `HttpOnly` automatically

Fortunately, credential cookies often exhibit certain characteristics. We studied the 54 websites that use `HttpOnly` cookies. In some cases, cookies with `HttpOnly` are not necessarily used for authentication, but at least should not be accessed by client-side scripts. Still, analysis on the whole set would give a close enough estimation of characteristics for the login cookies. The preliminary experiments show that they generally exhibit three key properties:

- They tend to have English phrases related to authentication in their names such as token, session, and so on.
- Their values exhibit greater randomness than non-`HttpOnly` cookies.
- They use relatively long strings for their values compared to the cookies without `HttpOnly`.

Table 5.1 shows the distribution of meaningful English phrases in the names of `HttpOnly` cookies. It shows that at least 54% of them use phrases related to authentication, indicating we could use cookie name as one hint to decide if a cookie should be tagged as `HttpOnly`.

A well-known way to measure the randomness of a string is to calculate its entropy. There are many entropy models, and in Zan we use the Shannon

Phrase	**Count**
*sid$	122
auth	23
session	21
∧nid$	18
token	14
*sess$	11
other	174
Total	383

Table 5.1: Common phrases for `HttpOnly` cookie names. This table shows phrases (case insensitive) used as the names of `HttpOnly` cookies in top websites, where * means any combination of characters, while ∧ and $ means the begin and end of string respectively.

Property	`HttpOnly`		**Non-`HttpOnly`**	
	Aver.	**Stdev.**	**Aver.**	**Stdev.**
Entropy	4.00	1.60	2.83	1.82
Length	102	128	48	83

Table 5.2: Characteristics of `HttpOnly` and non-`HttpOnly` cookie values. This tables shows the average and standard deviation of entropy and lengths of `HttpOnly` cookie values and non-`HttpOnly` ones in the top websites that use `HttpOnly`.

Entropy Equation defined as:

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_b p(x)$$

where $p(x)$ is the *probability mass function* of a character x appeared in the string X [87].

We show the average and standard deviation of our entropy calculations in Table 5.2. `HttpOnly` cookies have an average of 1.17 more bits of entropy than cookies without the `HttpOnly` attribute. Table 5.2 also shows that `HttpOnly` cookies on average have 54 more characters than non-`HttpOnly` ones. These are not coincidences. It is common that credential cookies are encrypted using hashing algorithms such as MD5 or SHA1. At the same time, web sites need to use long enough strings for session tokens in order to avoid collision among different users.

Using data provided in Table 5.2, we can use a standard *Gaussian distribution classifier* to decide if a cookie resembles a credential one. The classifier we use in ZAN is

$$\tau = \frac{\mu_h - \mu_{nh}}{\sigma_h + \sigma_{nh}} \sigma_{nh} + \mu_{nh}$$

where μ_h and σ_h are the mean and standard deviation for `HttpOnly` cookies respectively, while μ_{nh} and σ_{nh} are the mean and standard deviation for non-`HttpOnly` ones. τ defines the delineation between `HttpOnly` cookies and non-`HttpOnly` cookies, so we then can assume that a cookie with entropy greater than 3.45 bits or containing 70 or more characters is likely one that should be tagged with `HttpOnly`.

Based on above information, we developed an algorithm to automatically detect credential cookies. The algorithm is defined as:

```
1  if (origin == JS || hasHttpOnlyAttr())
2        return;
3  for c in (the list of cookies)
4    if (c.name is common phrase)
5      if (entropy(c.value) > 3.45)
6        || len(c.value) > 70)
7        c.httponly = true;
8    else
9      if (entropy(c.value) > 3.45
```

```
10        && len(c.value) > 70)
11        c.httponly = true;
```

When a list of cookies is passed in with an HTTP request, we apply the algorithm to their name-value pairs. First, we only examine network cookies (lines 1 and 2). For cookies that are set by JavaScript, we skip the algorithm because they are by definition not `HttpOnly` cookies. Meanwhile, when `HttpOnly` is already present, we honor the web developer's decision and ignore the rest of the algorithm (lines 1 and 2).

Next for each cookie in the list, we check if it uses a common phrase, which is presented in Table 5.1, for its name. For the one that uses a common phrase, we assume it is most likely a credential cookie and use a relatively loose classifier on the entropy and length of its value. We would tag cookie with `HttpOnly` as long as *either* its entropy or length falls into the range of a credential cookie. For the one that does not use common phrase, we use a relatively tight standard. Only when *both* its entropy and length meet the bar of credential cookie, we opt to apply `HttpOnly` on it.

The overall algorithm is conservative to some extent as we will show later in the evaluation (Section 7.1). We choose to be conservative because setting `HttpOnly` on too many cookies would sometimes affect the compatibility and usability of web apps. If a cookie that is supposed to be used by JavaScript has been set with `HttpOnly`, the web app could function incorrectly. Meanwhile, missing a single credential cookie is not a serious problem as long as the attacker is not able to retrieve the complete set of authentication cookies.

5.4 Case study: frame-based attacks

In this section we discuss the state-of-the-art in frame based attacks and defense techniques. We also describe our algorithm for providing more complete defenses against frame-based attacks.

Back in the era of Netscape Navigator in early 1990s, the HTML `FRAME` element was introduced to allow web developers to delegate a portion of their document's visual display to another entity. These frames can then be navigated to independent documents, which can delegate their share of the screen further to sub-frames. The `FRAME` tag was inflexible and was replaced

by the more versatile `IFRAME` tag, which was introduced by Internet Explorer in 1997.

`IFRAME`s enable modern browsers to display one web document inside another at an arbitrary position, creating a complex frame hierarchy. Browser present only the URL of the main, or top-level, frame to the users in the address bar. Consequently, it is infeasible for an average user to distinguish sub-frames from other parts of a page. This inconsistency, coupled with flexible display overlaying mechanisms available in browsers, creates the opportunities for frame-based attacks.

5.4.1 Frame-based attacks

Frame-based attacks were first reported 2008 when Hansen and Grossman introduced the term "clickjacking" [49]. In a clickjacking attack, the attacker chooses a clickable region on the target website that the user is currently authenticated on (e.g., a "like" button in a Facebook page). To perform the attack, a malicious website will load a page from the victim website inside an `IFRAME`, using Cascading Style Sheets (CSS) to make it transparent. At the same time, this transparent clickable element is placed on top of some visible, fake, but interesting clickable gadget (e.g., click to win a free iPad). As a result, the user would "like" an attacker chosen page in Facebook instead of unrealistically winning a free iPad when he or she clicks it. Evidence shows that major websites such as Facebook, Twitter have already suffered from clickjacking attacks [19, 99]. There are also other variants of this same basic attack that use similar mechanisms to induce users to click on a page unwittingly.

As the Web evolves, the capability of frame-based attacks also improves. In recent research, Paul Stone demonstrated the next generation clickjacking by showing four new techniques [91]. In the new attack scenarios, the attacker could potentially use the drag-and-drop API in HTML5 and some social engineering to inject text into fields of victim, which could be used to sent fake emails from a user's account. An attacker could also extract content from the enclosed frame, which could be used to steal sensitive information such as passwords, or tokens that are used to authenticate a session and guard against cross-site request forgery (CSRF) [113] attacks. Stone also showed

that it is possible to use this new technique to achieve login detection that is used to facilitate CSRF or other clickjacking attacks.

5.4.2 Preventing framing

Fortunately, while a number of different techniques have been discovered to carry out frame-based attacks, they can mostly be defeated by some correctly used methods.

Frame busting was the first technique that was suggested to counter clickjacking attacks [49]. Frame busting often refers to a snippet of JavaScript code included in a web app that intends to prevent this web app from being included in a sub-frame. A simple example of frame busting is shown here:

```
if (top.location != self.location)
  top.location = self.location;
```

Typically, it includes a condition statement to detect if the web app is embedded in a frame. If so, the next statement acts as the countermeasure to break out and load the web app in place of the web site that is framing it. Unfortunately, this JavaScript-based approach is not always effective, and there is a list of ways to defeat it as described by Rydstedt *et al.* [82]. For example, a malicious site may try to use the `onbeforeunload` Document Object Model (DOM) event to prevent a framed site from navigating to a different URL, or merely disable scripting in the framed web page.

Another option for preventing web apps from being framed is the `X-Frame-Options` introduced by Internet Explorer 8 [60], which now widely implemented in all modern browsers. `X-Frame-Options`, as a declarative method, provides a clear and robust approach to avoid unsolicited framing. `X-Frame-Options` can be used either in a HTTP response header of a web page or as a HTML "http-equiv" META tag in the web page itself. `X-Frame-Options` has two options: (1) *DENY* - the browser prevents the page from rendering if it will be contained within a frame; and (2) *SAME-ORIGIN* - the browser blocks rendering only if the origin of the top-level browsing context is different than the origin of the content containing the `X-Frame-Options` directive. We also observe a third option – *ALLOW* – used by some `IFRAME`ed advertisements . We posit that it is used to advise the browser to allow the embedding in any case. Although more robust

Defense	Front page	Login page
Frame busting	19	34
`X-Frame-Options`	7	14

Table 5.3: Frame busting and `X-Frame-Options` usage among top websites.

than frame busting, `X-Frame-Options` also has a potential pitfall. When `X-Frame-Options` is included in the HTTP header, a web proxy could strip it, leaving the page unprotected.

However, like the `HttpOnly` attribute, anti-framing mechanisms are not sufficiently incorporated in top websites. Some websites use frame busting code and a few have started to use the new `X-Frame-Options` feature (Table 5.3).

5.4.3 Zan frame defense

As discussed above, both frame busting and `X-Frame-Options` have shortcomings and they are also poorly incorporated in top websites. To better counter frame-based attacks, we implement the following algorithm for each `IFRAME` in Zan.

```
1  if(hasXFrameOptions())
2    return;
3  state = init;
4  for s in (all JS statements):
5    if(state == init && isFrameDetect(s))
6      state = nav;
7    if(state == nav && isTopFrameNav(s))
8      injectXFrameOption();
```

When `X-Frame-Options` is present in a web app, we honor whatever the web developer sets and ignore the rest of the algorithm (lines 1 and 2).

Next, the algorithm detects conditional statements that are predicated on detecting framed pages (line 5). Fortunately, frame detection code tends to exhibit some fairly simple patterns. For the 34 websites we found with frame busting code, the frame detection patterns we found are shown in Table

Type	Number
`top != self`	17
`parent.frames.length != 0`	2
`parent.frames.length > 0`	2
`top.location != self.location`	4
`window.self != window.top`	3
`top.location != location`	3
`window.top != window`	2
`top.location != window.location`	1

Table 5.4: Conditional statements used for detecting framing. This table shows the distribution of the frame detection code found in the 34 websites shown in Table 5.3. `self != top` is put in the same category of `top != self`. And `!=` is considered the same as `!==`. Whitespace is ignored.

Type
`top.location = loc`
`top.location.href = loc`
`top.location.replace(loc)`
`parent.location.href = loc`

Table 5.5: Frame busting navigation countermeasures. This table shows the four navigation countermeasures used when framing is detected that we observed from websites that deploy frame busting code.

5.4. To find frame detection code, the `isFrameDetect()` function inspects JavaScript statements to check for one of the patterns listed in Table 5.4.

However, using frame detection code alone would induce false positives because these basic patterns are also used for functionality other than frame busting in web apps. To reduce false positives, we only inject `X-Frame-Options` if we also detect a countermeasure navigation statement (line 7). To find countermeasure navigation statements, our `isTopFrameNav()` function searches JavaScript statements for one of the patterns shown in Table 5.5. In these navigation countermeasures, loc could be any URL that the web author wants to use for replacing the top-level frame. The patterns we detect are less diverse than what we observe for frame detection. A recent study suggests that other countermeasures as well [82], but our evaluation indicates these four work well for top websites.

When ZAN detects frame detection code and countermeasure navigation statements, we apply `X-Frame-Options` with the SAMEORIGIN option to framed web apps (line 8), preventing them from being displayed as frames inside web pages that are from different origins.

This heuristic algorithm will not detect all frame busting code and it could detect frame busting code when there is none, but it is simple, efficient, and accurate for the websites that we examine (see Section 7.1 for more details). The algorithm also has some flexibility built in. Browser developers could adjust the number of frame detection or frame navigation statements the algorithm searches for to trade-off more aggressive security against compatibility.

One alternative and more aggressive defense could be to apply `X-Frame-Options` to any websites that have username and password fields, thus protecting login sites from frame-based attacks. However, we did not evaluate this more aggressive defense in this dissertation.

5.5 Case study: secure JSON parsing

Before the era of "Web 2.0", a full page re-load was required to update information on a webpage. The problem with this approach is that it is neither efficient or elegant. In terms of efficiency, the server was required to send a full version of the page to the client even for minimal content modifications, and in terms of elegance it forced visual resets of the screen requiring the client to wait while the refresh occurred. As such, websites needed a way to update information on the page less obtrusively, thus, Ajax was developed to enable asynchronous communication between the client and server. Ajax originally used the XMLHttpRequest object to transfer `XML` formatted data. Recently JSON has become an `XML` replacement in Ajax because it is simple and can be easily encoded into several popular programming languages.

5.5.1 JSON and exploiting JSON

JSON is a subset of the object literal notion of JavaScript. Originally specified by Douglas Crockford in RFC 4627, JSON is now supported in all major

browsers as part of JavaScript. JSON can be used in client-side scripts to facilitate easy data exchange with servers. Below is an example of a simple JSON string:

```
{"employee": [
  {"name": "Alice", "sex": "female"},
  {"name": "Bob", "sex": "male"}]}
```

In this example, the JSON string represents an object that contains a single member "employee", which contains an array containing two objects, each containing "name", and "sex" members.

JSON is often used together with XMLHttpRequests to enable the browser to exchange data asynchronously with the server. If an XMLHttpRrequest returns the above JSON string stored in a variable called `jsonText`, it can be converted into a JavaScript object using the JavaScript `eval()` function, which invokes the JavaScript compiler as shown in the following code snippet:

```
jsonObject = eval('(' + jsonText + ')')
```

Since JSON is a proper subset of JavaScript, the compiler will correctly parse the text and produce an object structure. For example, one can use `jsonObject.employee[0].name` to access the first employee's name. To avoid ambiguity in JavaScript's syntax, it is also recommended that the JSON text be wrapped in parentheses as shown in the statement above.

However, since `eval()` invokes the complete JavaScript compiler, it could execute any JavaScript program besides JSON, leading to potential security vulnerabilities. Typically in a web app, JSON is used over XMLHttpRequest, which is commonly restricted to communicate only with the origin that the web app comes from. Thus, the source is trusted. However, if the server does not provide correct JSON encoding, or it embeds user supplied content in JSON text without rigorous sanitization, it could deliver problematic JSON text to the client that could contain malicious scripts (e.g., CVE-2007-3227). The `eval()` function would then execute the script, resulting in an XSS attack. Assume in the above example, the name of an employee is provided by the user. If the user could enter a malicious script such as `", "arb": alert(document.cookie), "": "` instead of a real name,the resulting JSON text would become:

```
...
  {
    "name": "",
    "arb": alert(document.cookie),
    "": "",
    "sex": "female"
  },
...
```

When evaluated, the web page displays an alert showing the cookies for the active session. This threat has already been reported for real world websites such as in Google's personalized homepage and can be used for more serious script injection attacks [83].

5.5.2 Native JSON

To minimize script injection via JSON parsing, it is suggested that web developers use regular expressions to validate the data prior to invoking `eval()`. However, browser developers added a new function, `JSON.parse()`, as a safer and more robust alternative to `eval()` that parses JSON text without executing scripts.

`JSON.parse()`, which only recognizes JSON text, rejects all possible embedded malicious scripts. Additionally, `JSON.parse()` only parses JSON text that adheres to the JSON standard and will reject any malformed JSON text. Fortunately, browsers implement functions for converting JavaScript data structures into JSON text. These serialization and deserialization routines are well supported in most recent browsers as *Native JSON*. However, web developers have been slow to adopt this new security feature as well.

5.5.3 Automating `JSON.parse()` adoption

To prevent script injection via JSON, Zan inspects all strings passed into the JavaScript `eval()` function:

```
1  s = fixupEvalString(evalString);
2  if(s.startWith("({") && s.endWith("})"))
```

```
3    return zanParse(s);
4  if(s.startWith("([") && s.endWith("])"))
5    return zanParse(s);
```

If the algorithm detects a string that looks like a JSON object, it will pass that string to the `JSON.parse()` function automatically. Our logic for detecting JSON objects checks the beginning and end of the eval string to find the "({" and "})" (line 2) or the "([" and "])" (line 4) strings respectively. These checks will find cases where developers use JSON objects but an attacker was able to pass unsanitized JavaScript into the JSON object, thus thwarting this type of script injection attack.

For the websites we examined, we found a few cases where web developers use JSON, but the JSON text was not formatted according to the strict JSON grammar. To ensure that these almost-valid JSON strings can still pass through we used a modified `JSON.parse()` parser for parsing JSON text with a slightly updated grammar to handle these cases (lines 3 and 5). Specifically, we allow single quotes in addition to double quotes and we accept name strings that omit enclosing quotes. Additionally, we have a function that fixes up eval strings to make our detection logic easier by remove some whitespace to ensure that the JSON brackets and braces make it to the beginning and the end of the eval string (line 1). With these modifications, the Zan algorithm parses all JSON objects we observed in our tests.

Although this algorithm is simple, efficient, and effective, there are a few cases where it could fail. A web developer could use a JSON object that deviates from the JSON standard, but is still detected by our algorithm as JSON, resulting in a failed parse. We found a few cases of this type of deviation and correct for it by updating our JSON grammar, but other similar instances are also possible. Another problematic scenario is when an attacker replaces JSON text altogether with malicious JavaScript, which we would pass to `eval()`, missing the attack.

5.6 Case study: DOM-based XSS prevention

In a DOM-based XSS attack a malicious payload is injected into the vulnerable web application during execution by the browser. To prevent DOM-based

```
<HTML>
  <HEAD><TITLE>Hello</TITLE></HEAD>
  <BODY>
    Hi
    <SCRIPT>
      var pos = document.URL.indexOf("name=") + 5;
      var len = document.URL.length;
      var name = document.URL.substring(pos,len);
      document.write(name);
    </SCRIPT>
  </BODY>
</HTML>
```

Figure 5.2: A simple web app with DOM-based XSS vulnerability.

XSS attacks, we have implemented data flow tracking in the web page instance and developed suitable mechanism to prevent untrusted text from being executed as JavaScript.

In Figure 5.2 we show a simple web application with DOM-based XSS vulnerability. In the example the JavaScript parses the URL to obtain the visitor's name and then embeds the name into the page's HTML. Assuming this web application is hosted at `http://www.domxss.com`, a URL such as `http://www.domxss.com/#name=Alice` would work as developer intended and print "Hi Alice" in the web page. However, an attack can be constructed using a URL such as `http://www.domxss.com/#name=<script> alert(document.cookie)</script>` causing the web app to write the script tag into the document and then execute it. The attacker can tricking users into visiting a malicious URL like this to exploit the web app.

5.6.1 Taint tracking

In our system, objects originated from untrusted sources are marked as tainted and taint information is propagated as the web application interacts with tainted data. Our system propagates taint information at the JavaScript object level and we have extended both JavaScript and HTML engines to support taint tracking. In the JavaScript engine, we have added a field to every JavaScript object to indicate if an object is tainted. To

propagate taint, we handle three types of operations on JavaScript objects: assignments, logic or arithmetic operations, and string manipulation. In an assignment, the left operand becomes tainted if the right operand is tainted. For logic or arithmetic operations, the result is tainted if any of the operands are tainted. For string manipulation, the resulting string is tainted if it contains content from tainted sources. For example, any substring of a tainted string is tainted as is the lower case conversion of a tainted string. Similar to Yip et al. [111], our system does not track of implicit data flows.

In addition to the JavaScript engine, our system also needs to track data flow inside the HTML engine since JavaScript objects can be stored in the DOM tree and later retrieved by other JavaScript. To solve this problem, our system taints the DOM nodes where tainted JavaScript objects are stored and upon retrieval, taints the JavaScript objects that interact with the tainted DOM node.

5.6.2 Preventing DOM-based XSS

With the taint tracking capability, ZAN is able deploy new security policy in the client side to prevent DOM-based XSS.

To prevent DOM-based XSS attacks, DOM objects that could be controlled by an attacker are marked as tainted. Objects that we consider tainted are: `document.URL`, `document.referrer`, `document.location`, and `window.location`. This policy forbids the JavaScript engine from executing tainted text by using propagated taint information. If JavaScript source is constructed from tainted DOM objects, the interpreter will refuse to execute the tainted input. Using the example shown in Figure 5.2, a URL containing a script payload (`http://www.domxss.com/#name=<script>alert(document.cookie)</script>`) will result in the script text written to the page being tainted. When the JavaScript engine is invoked to execute the contents of the script tag, our policy prevents it from executing. We evaluated this policy and proved that this policy is able to prevent all the attacks we examined while not introducing any incompatibilities in other web applications.

5.7 Implementation

ZAN is implemented on top of OP2, which provides a clear and robust architecture for implementing the security features we describe in Chapter 5.

For the cookie protection algorithm, we modify the cookie subsystem in OP2 to enable our algorithm. OP2 uses a clear message passing mechanism for cookie access and ZAN interposes the messages passed into the cookie subsystem. Each time the cookie subsystem receives a list of cookies from an HTTP response, ZAN applies the algorithm described in Section 5.3 to the cookies, and tags `HttpOnly` attribute appropriately. There is no need to modify the cookie read procedure, because the cookie subsystem then prevents `HttpOnly` cookies from being accessed by client-side script.

For the frame-based attack defense, we interpose the HTML parser (which all static JavaScript source code also need to go through first) used in WebKit. ZAN uses simple string pattern matching routines to detect the existence of frame busting code using the algorithm proposed in Section 5.4, and then applies the same anti-framing method that `X-Frame-Options` implementation uses in WebKit. Login pages alway contain password fields, if needed, we detect those fields to as the hint to apply `X-Frame-Options` to them.

For automatic `JSON.parse()` adoption, we intercept the `eval()` calls by modifying its implementation in WebKit according to our safe JSON method described in Section 5.5.

For DOM-based XSS prevention, we modify the WebKit's HTML and JavaScript engines. Basically, we add a bit field for taint information in every DOM and JavaScript object. To propagate taint information, we instrument all assignments, arithmetic or logical operations, and string manipulation in JavaScript, as well as all types of C++ constructors related to those objects behind the scene. We then intercept all the strings passed to JavaScript engine, and stop the execution if one is tainted.

5.8 Discussion

A complete security solution is always desired, but is extremely hard to achieve for a complex system like the Web that consists of hundreds of thousands of various web apps. In this section, we discuss the lessons we learn

through designing and implementing ZAN, and also articulate some issues related to the approach of ZAN.

5.8.1 Generality

The insight behind ZAN is that client side software can infer the intentions of web app developers on important objects and special code logics. But is the approach of ZAN generic enough to apply to other features besides the four we discussed in this section?

The development of web app security is mostly a history of patching and fixing bugs. Web apps usually did not have a clear mind of the model of security threat at the first place when introducing new features, such as JSON deserialization and cross-origin frame communication [17]. Later, they would realize the problems and replace the old workaround with more robust and secure methods such as `JSON.parse()` and `postMessage`. Nonetheless, old approaches always exhibit common patterns, enabling potential detection and automatic replacement such as the secure JSON parser we described in this chapter. While automated `postMessage` would be harder to implement, it would not be totally infeasible.

In addition, we could analyze the tendency of use JavaScript-based defense. It is very common that after one entity (e.g., a website or a security researcher) discovers a piece of JavaScript code that could add defense to unprotected web apps, every other website follows on to copy and paste the same code snippet into their code base, or at least use similar approaches. So when a more reliable mechanism is supported later for the same security vulnerability, we could possibly detect these outdated JavaScript workarounds, such as the frame busting techniques we discussed.

Overall, there is no guarantee that ZAN's approach is generically applicable to all the security mechanisms for web app security. However, we argue that when a mechanism exhibits similar characteristics as the ones we discussed, it is possible to employ ZAN's approach to automate its application.

5.8.2 Use of heuristics

ZAN includes the use of heuristics for security. While server-side and hybrid approaches have full control of the web apps and could deploy more sound defense mechanisms, we already observe that web app developers are slow to do so. Pure client-side approaches do not require extra work in the server side, but have to recover or infer implicit information using methods such as heuristics.

One problem with using heuristics is that they are not 100 percent accurate, leading to a critique of this approach. Two typical concerns are: 1) whether the approach adds or improves protection for all possible candidates; 2) whether the approach results in incompatible web apps.

Of the two concerns, we argue that the second one is more important. A security mechanism resulting in loss of function in a web app is undesirable. Consequently, one has to preserve compatibility. However, it is not impossible that a heuristics-based mechanism can be both effective and compatible. There are real world examples of heuristics-based client-side protection. For example, Internet Explorer 8 uses heuristics to implement its XSS filters [81].

5.8.3 Deployment and evaluation

When introducing a new mechanism to the Web, we have to think about how to deploy it, whether it will cause new problems, and how to evaluate its impact.

We argue that ZAN can be incorporated into modern web browsers gradually. Browsers with and without ZAN capability in general will not exhibit fudementally different interpretation of a same web app. ZAN does not rely events and states that can be multiplexed for different purposes. For exampe, credential cookies should only be used for authentication between browsers and servers. It is a bad practice to incorporate script manipulation of them into a web app. It is possible that an attacker could inject frame busting code to try to cause denial-of-service by confusing ZAN. But in most cases, a web app is not embedded in a cross-origin frame. Even if ZAN detects the frame busting code, it will not prevent the rendering of the web app.

It is reported that existing pure client-side defenses such as XSS filters in Internet Explorer have introduced new vulnerabilities into the browsers [71].

As browsers are already complex artifacts, adding complex defense mechanisms would be problematic. However, we argue that all the mechanisms we demonstrate are simple (less than 100 lines of code each), and do not change the structure or content of a web page. Our secure JSON deserialization defense does change the JSON strings a little bit. But what ZAN tries to do it enforcing better format according to standards. Nevertheless, it is infeasible to prove that there is no new vulnerabilities added. The only way we can do is to keep ZAN simple.

It is generally hard to evaluate something on the scale of the Web. The Web consists of hundreds of thousands of different web apps that are not implemented in a consistent way. Automated testing is required to analyze a reasonably large number of websites. However, most websites employ mechanisms to prevent non-human users from using certain features, such as CAPTCHAs. Even established research that aims to study the Web in large scale tends to be ambiguous. For example, Singh *et al.* did not log into the websites they surveyed, compromising the confidence of the study (e.g., Facebook is certainly not the same complex web app when not logged into) [90]. For the evaluation we will present in Chapter 7, we argue that while ZAN might not be thoroughly tested, it presents encouraging results and a preliminary study of using these mechanisms.

5.9 Summary

In this chapter, we presented a generic approach for automatically adding new security features to existing web apps. ZAN accomplishes this by inspecting events and states in the browser to exploit opportunities for retrofitting legacy web apps with new security features. We presented three algorithms: `HttpOnly` cookie designation, which automatically restricts access to authentication cookies, `X-Frame-Options` specification, which denies the inclusion of web apps in `IFRAME`s, and `JSON.parse()`, which detects `eval()` calls on JSON text and parses them in safe routines. Each of these algorithms capitalize on unique details about applications to provide automated security mechanisms.

On key aspect of our approach is that our algorithms are simple. As browsers are complex artifacts, it is necessary to maintain this feature for

the development of practical systems. Despite their simplicity, our algorithms are effective at improving the security of several of the websites we evaluated. Furthermore, two of our algorithms, `HttpOnly` and `IFRAME` defense, are tunable and can be adjusted by browser developers to trade-off security against compatibility.

As web apps become increasingly popular, improving their security becomes paramount. Browser developers have been proactive in providing new security mechanisms, but web developers have either been too slow to adopt these new features or manage complex code bases that make it difficult to adapt legacy systems. ZAN is a system that can help improve the security of legacy web apps where web developers have neglected to use the security mechanisms available to them. We argue that ZAN presents a reasonable attempt to address this situation, and hope it could lead to future practical applications.

CHAPTER 6

USING FORMAL METHODS

Security evaluation is hard. Researchers sometimes turn to formal methods for help and hope to have a system theoretically proved. In this chapter, we present our work of modeling the high level design of the OP browser kernel and partly verify the correctness of some security properties we want to have in the browser. We also examine the possibility of formally verifying IBOS TCB.

6.1 Introduction

The security of web browsing is a pressing problem in modern computer systems. Even though tremendous efforts have been spent on web security, most of them rely on unverified assumptions about other untrustworthy components of the Web. Particularly, web browsers and the supporting operating systems, which were designed without analytical foundations, provide little formal assurance about the overall web security. In other words, security mechanisms are only as trustable as the browsing systems they run on. As the Web continues to evolve, reasoning about the security of the platform will increase in importance.

Formal models and tools have been useful in evaluating the security of file systems [109] and network protocols [22]. We believe that, with proper architectural support, formal methods can also be used to improve the design of browsing systems and provide high trustworthiness of the systems.

In this chapter, we present our initial work towards applying formal methods to web-based systems. Specifically, we first present the formulation of the OP web browser within the logical framework of rewriting logic and how use formal reasoning tools to verify model correctness, including the presence of attacks, successful compromises and access control. We show that

formal methods can help the design of secure web browser by verifying two example security invariants: 1) the address bar displayed within our browser user interface always shows the correct address for the current web page; 2) access control implemented in the browser kernel enforces the same-origin policy specified as invariants for browser-level components.

We also investigate the possibility of verifying the TCB of IBOS. Formally verified TCB for the browsing system means that for the first time we have a truly trustworthy foundation for secure web browsing. We discuss the challenge of verifying IBOS TCB and also provide some further goals once we have the verified IBOS kernel.

6.2 Maude background

In this dissertation, we use Maude as the tool for formal verification. Maude is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications. In particular, it supports very well object-oriented computation. Once having a proper model of a system implemented in Maude, we can use its model-checking capability to validate some of the assertions required in the system.

A simple example using Maude and model-checking of invariants is presented in the *Maude Manual* [27]. The example involves the model of a clock and uses Maude to search the state space for states with invalid hour values. The Maude model for the clock example is presented in Figure 6.1. This example illustrates several Maude features, though we only describe the ones relevant to the OP browser model we present later.

Figure 6.1 shows the Maude model named SIMPLE-CLOCK. The third line defines a sort, called `Clock`. A sort is similar to the `class` keyword in C++ and simply defines a category for later use. Line 4 in the figure contains the definition for an operation called `clock`; operations act on a sort and generally connect a sort (or set of sorts) to a different set of sorts. In the SIMPLE-CLOCK model, the sort `Int` and the sort `Clock` are related by the `clock` operation, which, given an integer `T`, produces a `Clock` object – `clock(T)` having time `T`. Rewrite laws begin with `rl` and describe transitions between states. The SIMPLE-CLOCK model has one rewrite law. This

```
1 mod SIMPLE-CLOCK is
2   protecting INT .
3   sort Clock .
4   op clock : Int -> Clock [ctor] .
5   var T : Int .
6   rl clock(T) => clock((T+1) rem 24) .
7 endm
```

Figure 6.1: A simple Maude example. This figure shows a Maude example from the *Maude Manual* (Version 2.3). This example describes a model for a 24-hour clock in Maude.

```
search in SIMPLE-CLOCK :
clock(0) =>* clock(T)
such that T < 0 or T >= 24 .
```

Figure 6.2: The search statement in Maude. This figure shows a search statement from the *Maude Manual* (Version 2.3) showing how to model-check the SIMPLE-CLOCK model invariant using Maude's search functionality.

---

rewrite law says that the clock operation increments the clock variable T and then takes the remainder after dividing by 24.

Once we define a model in Maude, we can use the `search` command to have Maude explore the state space and find states that match our search criteria. For the SIMPLE-CLOCK example we want to find states that violate an invariant, such as the clock's state being outside of the 0 to 24 range. Figure 6.2 shows the example search statement for the SIMPLE-CLOCK model. This search statement defines an initial state, `clock(0)`, and the condition to match when searching. The invariant is verified because no state violating it can be found.

The Maude model for the OP browser consists mostly of definitions of types and state for each component by defining sorts, operations, and state variables for each browser component. To define browser behavior, we use rewrite laws to show transitions between different internal states in the browser. Our invariants are specified as statements and we use the same search functionality in Maude to find matching states.

6.3 Validating browser design

We designed the OP web browser to include the use of formal methods to verify its correctness. To better support formal methods we use small, simple, and exposed APIs that allow us to model our system and reason about it. Using formal methods, we are able to provide greater assurance that we preserve our security goals during an attack and compromise.

We formulate the OP web browser within the logical framework of rewriting logic and use formal reasoning tools to verify model correctness, including the presence of attacks, successful compromises, and access control [26]. The reasoning engine we use is the Maude system [65]. We use the term "Maude" to refer to both the Maude interpreter and the language.

Once the browser model has been formally specified we can use Maude's search ability for model-checking to verify invariants over the finite state space we need to consider. The invariants of a browser system fall into two categories: program invariants and visual invariants. Program invariants for OP consist of the goals of the access control policy. These invariants are relatively easily gathered from the source code and concise specification of security policy. The visual invariants (e.g., preventing address bar spoofing) need extra effort to be mapped into program invariants. In this dissertation, we model these invariants, and we also translate browser compromise and built-in defenses into rewriting logic rules. As we explain in the following, OP's address bar logic and same-origin policy are specified by rewrite rules and equations in Maude, and we use model-checking to search for spoofing and violation of same-origin policy scenarios. The result of the search is a list of states that are violations of the invariants specified and the sequences of actions that lead to the invalid state. States that are violations of security invariants can assist in the development process by catching potential problems before they are exploited.

In this section we discuss how we use formal methods to improve the design of our browser. We discuss how we created our model, and describe how we model-check it to prove the absence of address bar spoofing attacks and to verify parts of our same-origin policy model.

6.3.1 Formal models and system implementations

There is often a gap between the formal model used to verify properties and the system implementation. While we recognize that this gap exists between our model and system, we feel that for our uses of formal methods, thanks to the executable nature of the Maude specification, the difference is small enough that we are able to use the results of model-checking to iterate on design and development. Since we implement each of the browser components separately and use a compact API for message passing, the model that we use to formally verify parts of our browser is very similar to the actual implementation. The model we have created is focused on message passing between components. We do *not* verify, for example, that the HTML parsing engine is bug-free; instead we verify that, *even if* the HTML parsing engine had a bug, the messages that a code execution attack could generate (potentially any message) would not force the browser as a whole into a bad state. To do this, each component is modeled in Maude and aspects of every component's internal state are included. Messages are the means for the browser's internal state to change.

Our application of formal methods helped us find bugs in our initial implementation. By model-checking our address bar model, we revealed a state that violated our specification of one address-bar visual invariant. The resulting state was actually due to a bug in our implementation, as we had not properly considered the impact of attackers dropping messages or a compromised component choosing to not send a particular message. Our model gives an attacker complete control over the compromised component, including the ability to selectively send some types of messages and not others. We used the result to fix our access control implementation, and we updated our model accordingly.

In the interest of space we have not included the entire Maude model. In the following sections we highlight parts of our model that we use to model-check same-origin and visual invariants. We have not specified all browser invariants in our model, as this is a first step in our venture into formally verifying an entire web browser.

```
< UI-ID : Frame | addrBar: URL, ... >
imsg(count, src, dst, IDENTIFIER, content)
< ... > ...
```

Figure 6.3: OP message specification in Maude. This figure shows the message specification in Maude. The first section of the specification is a class-like structure, starting with < and ending in >. UI-ID is the instance identifier of the type, Frame is the type, and after the pipe are the members of the type. The next line begins with `imsg` and is the constructor for the message type. The constructor takes the elements in parentheses and creates an object of a specific type. The `imsg` constructor creates an object of type Message.

6.3.2 Modeling the OP browser

Component-based systems can be modeled in Maude as multi-sets of entities, loosely coupled by a suitable communication mechanism. For OP, the entities are browser components, each with a unique identity, and the communication mechanism is the message-passing API. In the Maude version of our OP implementation, the states of OP are represented by symbolic expressions, and the state transitions are specified by rewrite rules describing the components' communication with each other and the state transformation. In this section we discuss our model for message passing and processing, user actions, and how to include browser compromise into this model.

Communication between components in OP is done through the message-passing interface, which is the communication mechanism modeled in Maude. The messages are expressed as entities in the multi-set of components. The message specification in Maude is shown in Figure 6.3. The messages are tagged with a count to make sure they are processed in the right order. Message ordering is preserved by the browser kernel, and in order to have ordering in the multi-set representation in Maude, a count attribute is introduced. A simple example illustrating our model of the message-passing interface and a corresponding state change is shown in Figure 6.4. This rule is responsible for updating the browser state, including the address bar of the user interface.

The browser state as a whole is represented by the objects corresponding to each of the components. This means that Maude represents a state as a grouping of the UI, network, plugin, and other subsystem states. Figure 6.4

```
< UI-ID : Frame | addrBar : URL, ... >
< MSG-ID : MsgCount | msg-to-process : N,  msg-to-send : M >
imsg(N, webAppId, UI-ID, MSG-SET-LOCATION-BAR, new-URL)
=>
< UI-ID : Frame | addrBar : new-URL, ... >
< MSG-ID : MsgCount | msg-to-process : s(N), msg-to-send : M >
```

Figure 6.4: The Maude rule corresponding to the state change in OP. This figure shows the Maude rule used to describe the state change due to a SET-LOCATION-BAR message being received. Notation here is similar to that of Figure 6.3. The first three lines are the current state and creation of the message to be processed. The remaining lines represent the state after the state change. The full browser state includes other components besides the UI and message queue.

---

shows an action that sets the location bar in the UI. The first three lines of Figure 6.4 describe the current browser state and include a message called MSG-SET-LOCATION-BAR using the `imsg` constructor. The browser state is rewritten, including in the UI a new address in the address bar (shown by new-URL), and the results of the rewrite are the last two lines of the figure. Rewrite rules such as these cause the Maude model to change state. Model-checking through `search` locates possible states that can occur as a result of these rules and satisfy an additional pattern.

Modeling user actions We also model the user actions in the browser system, such as clicking the "GO" button to request a new web page. The Maude model of the UI is very similar to the Java source code we wrote to implement the UI in the OP web browser. The Maude rule describing the message generation as a result of the "GO" button being clicked is listed in Figure 6.5. This Maude rule is especially descriptive of the original Java source; as we can see, the message created has the source set to UI-ID, a destination of KERNEL-ID, the message type of MSG-NEW-URL, and the URL that is the content of the message. This is precisely the same set of actions as in the Java code that implements the sending of the NEW-URL message. The first two lines of Figure 6.5 are the current UI and message queue state, plus the user action labeled "GO." The three lines following the `=>` marker are the new browser state, which include a new message being

```
< UI-ID : Frame | addrBar: URL, ... > GO
< MSG-ID : MsgCount | msg-to-process : N, msg-to-send : M >
=>
< UI-ID : Frame | addrBar : URL, ... >
imsg(M, UI-ID, KERNEL-ID, MSG-NEW-URL, URL)
< MSG-ID : MsgCount | msg-to-process : N, msg-to-send : s(M) >
```

Figure 6.5: Maude expression for the "GO" action. This figure shows how to use Maude to describe the "GO" UI button that causes a message to be sent. The first line represents the portion of the browser state for the Frame and the user action being performed, which in turn causes produces a new Frame state and the message with type set to MSG-NEW-URL.

---

generated by the `imsg` constructor.

Modeling browser component compromise Our model also includes potential attack paths. As an example of a component-level compromise, the attacker could take control of a web page subsystem instance and, using the message API, force the compromised component to send incorrect URL information to the UI component, resulting in address bar spoofing. Setting the address bar to a different location than the page contents is primarily useful for phishing attacks. Using access controls, we prevent such attacks from being successful. In Maude, we express the compromise of a component as additional rules that generate messages and trigger message passing and processing like ordinary rules.

6.3.3 Model-checking address bar invariants

Cases that allow the address bar in the browser to mismatch the page content were examined for Internet Explorer in work by Chen et al. [24]. They searched for violations of invariants specified for GUI elements in Internet Explorer under normal operation. We are able to verify a similar result for the OP browser using our formal model of the message-passing interface and our security policy. The key difference in our approach is that our proof holds even in the presence of a fully compromised web page instance.

To model-check and find cases of address bar spoofing, we must define a good browser state. Once we have an expression for good browser states, we

```
< UI-ID : Frame | AddrBar : S1:String, NavWebApp : WebApp1:Int,...>
< WebApp2:Int : WebApp | Content : S2:String, ... >
such that (WebApp1:Int == WebApp2:Int) /\ (S1:String =/= S2:String)
```

Figure 6.6: Maude expression for checking address bar spoofing. This figure shows a Maude expression describing the condition checked for address bar spoofing. This condition is used as a test for bad browser states. The first line is the current state of the browser, specifying the UI and ID for an instance of the web page subsystem. The last line is the comparison, which checks that the URLs associated with the address bar and web page subsystem are different, indicating a state where the address bar is spoofed.

---

can use Maude to search for the bad ones. We define a good state as a state where the content of the currently navigated web page matches with the URL shown in the address bar. The Maude expression describing spoofing is shown in Figure 6.6. When we use the model-checking search tool to search from an initial state, consisting of all the components of OP and some user actions, the results show that there is no logic error leading to the address bar spoofing scenarios. We also make sure that the address bar cannot be spoofed once the web page subsystem is compromised, showing that the access control logic can defend the browser against possible attack sequences. This result verifies that, if the browser kernel and UI are trusted, no sequence of messages can violate our address bar invariant, even if an attacker compromises a web page instance. This result does not provide guarantees for the display of the web page content. For example, a compromised rendering engine could display incorrect content. Our model-checking gives us high assurance that such an exploit will not be able to affect the address bar or other UI elements and will remain contained inside the rendering engine.

6.3.4 Model-checking the same-origin policy

Our implementation of the same-origin policy for the OP web browser controls access to all browser components. We use model-checking to verify that the same-origin policy cannot be violated by a single component being compromised. Although our model focuses on interactions with plugins, other components with similar interactive capabilities can benefit from the result. We model a compromised web page subsystem and plugin, and verify that

the access control implemented in the browser kernel enforces the same-origin policy specified as invariants in our model.

Plugins and JavaScript are able to interact with each other through the scriptable plugin extension to the Netscape Plugin API, and we support such interaction in OP. Enforcing the same-origin policy for these components is done in the same manner as our other security policies for plugins in the browser kernel. The simple message API keeps the state space small enough for model-checking to be tractable when considering all the possible actions by different browser components. We have proved a few different invariants. For example, we have proved that a plugin from one domain cannot send a message to a plugin (or web page instance) from another domain, and vice versa.

Introducing binary compatibility for the Netscape Plugin API would increase the size of the message API for plugins, although our access controls still remain in the browser kernel. Once OP is binary compatible with the Netscape Plugin API, we should be able to adapt the model and verify that the same-origin policy is upheld with the added complexity.

6.4 Can we verify IBOS TCB?

We have already shown that the architecture of IBOS can reduce drastically the TCB for browsing system. While IBOS already presents a huge step towards secure web browsing, we argue that the security guarantee can be further improved if the TCB of IBOS can be formally verified.

In seL4 [57], the authors develop the whole framework for converting formal specification to C implementation and guarantee that there is no programming error such as divided by zero and null pointer dereference in the OS kernel. It is not feasible or reasonable for us to develop a similar framework to verify the IBOS TCB, as the one for seL4 took more than 20 person years. While in this section, we examine the feasibility of verifying IBOS TCB, hoping to spur future work.

In IBOS, our goal is to minimize the TCB for web browsers and to simplify browser-based systems. To quantitatively evaluate our effort, we count the LOC in the IBOS TCB and compare it against the TCBs for Firefox and ChromeOS. IBOS supports fewer hardware architectures, platforms, device

System	LOC
IBOS	**42,044**
IBOS Kernel	8,905
L4Ka::Pistachio	33,139
Firefox on Linux	**> 5,684,639**
Firefox 3.5	2,171,267
GTK+ 2.18	489,502
glibc 2.11	740,314
X.Org 7.5	653,276
Linux kernel 2.6.31	1,630,280
ChromeOS	**> 4,407,066**
Chrome browser kernel 4.1.249	714,348
GTK+ 2.18	489,502
glibc 2.11	740,314
ChromeOS kernel & services (May 2010)	2,462,902

Table 6.1: LOC of TCBs for IBOS, Firefox, and ChromeOS. This table shows the estimation of LOC of TCBs for IBOS, Firefox on Linux, and ChromeOS. LOC counts are also shown for some major components that are included in the TCB.

drivers and features, such as browser extensions, than Firefox running on Linux and ChromeOS. For a fair comparison, we only count source code that is used for running above Linux and on the X86-64 platform. Also, we omit all device drivers from our counts except for the drivers we implement in IBOS.

Table 6.1 shows the result of LOC counts in the TCBs for these three systems, measured by SLOCCount [106]. For Firefox and ChromeOS, our counts are conservative because we only count the major components that make up the TCB for each system – there are likely more components that are also in the TCBs for these systems. IBOS TCB has only around 42K LOC, which is about two orders of magnitude smaller than the other two systems. And we argue that, since most of the components of IBOS are out of its TCB, the TCB size would not increase significantly even if we add more features to the system. With a proper framework like the one in seL4 [57], it is feasible to formally verify the entire IBOS TCB.

Once the IBOS TCB is verified, we can use it as the foundation to provide high-level security assurance of the browsing system, i.e., prove the desired security invariants in the whole system. Most of the security invariants pre-

sented in Chapter 3 could be verified in a similar way as we did for the OP web browser. Here, we discuss two of them as examples.

- *UI invariant.* The user interface of a browse sometimes also present security information of a web app. For example, as we discussed before, the browser should make sure that the address bar accurately displays the URL of the top-level frame. In IBOS, we also include the feature of display isolation between web page instances, browse chrome, and kernel display area. By proving this feature, we can provide high assurance that no browser-level component can interfere the display of another in the system.

- *Storage invariant.* IBOS includes the use of encryption of storage content, eliminating the threat from an untrusted file system. Nevertheless, we still need to verify the IBOS kernel provides enough checks to ensure that the cookies can only be sent to the right place. For example, a web page instance should only be able to access cookies within the same domain (or within the same subdomain). By verifying this, IBOS could guarantee that a compromised web app cannot steal other's cookies.

6.5 Summary

In this chapter, we presented some initial attempts of using formal methods to improve the security of web browsing. First, we showed that the microkernel-like architecture of the secure web browser we proposed enables application of formal verification. We showed that it is feasible to model different browser components as state machines, and express messages by means of rules that drive state transitions. By using Maude's model-checking capability, we were able to verify address bar dispaly consistency and part of the same-origin policy.

Secondly, we investigated the possibility of verifying IBOS TCB. Generally, with a formally proved IBOS TCB, we can achieve even higher security guarantee of the system. We also discussed some of the high-level security properties we could validate once we have the verified IBOS TCB.

The Web is growing in features and complexity, making it hard to ensure the trustworthiness of web-based systems. Web browsers have become the *de facto* operating system for hosting web-based applications, making it vital and pressing to ensure their security. Building the formal foundation of browser systems is rewarding, yet not an easy task. Of course, our model and discussion do not capture the entire web platform and its issues. However, we believe our study takes an important step in this direction.

CHAPTER 7

EVALUATION

This chapter describes our evaluation of IBOS, OP2 and Zan. In our evaluation, we analyze the security of IBOS and OP2 by looking at recent bugs in comparable systems and counting vulnerabilities that IBOS and OP2 are susceptible to. We also revisit the example attacks we discussed in the beginning of Chapter 3, and we measure the performance of those systems.

7.1 Security analysis

This section describes our security analysis. In our evaluation, we analyze the architectural impact of IBOS and OP2 for defending known vulnerabilities. we also test the efficacy of the defense mechanisms of Zan.

7.1.1 Browser vulnerabilities

To evaluate security improvements that OP2 and IBOS make for browsers themselves, we compared how well OP2 and IBOS could contain or prevent vulnerabilities found in Google's Chrome browser. For this evaluation, we obtained a list of 295 publicly visible bugs with the "security" label in Chrome's bug tracker. Out of the 295 bugs, 42 cause denial-of-service such as a simple crash or 100% CPU utilization. IBOS does not address denial-of-service or resource management currently. An additional 78 are either invalid, duplicate, not actually security issues, or related to features that IBOS does not have, such as browser extensions. For the remaining 175 bugs, we examined each of them to the best of our knowledge and classified them into the following seven categories and compared how Chrome, OP2, and IBOS handle those cases. Generally speaking, while OP2 has finer-grain modularization, OP2 and Chrome share a very similar design. From the architectural

Category	Example	Num.	Chrome/OP2	IBOS
			Contained	Contained or eliminated
Memory exploitation	A bug in layout engine leads to remote code execution	82	71 (86%)	79 (96%)
XSS	XSS issue due to the lack of support for ISO-2022-KR	14	12 (87%)	14 (100%)
SOP circumvention	XMLHttpRequest allows loading from another origin	21	0 (0%)	21 (100%)
Sandbox bypassing	Sandbox bypassing due to directory traversal	12	0 (0%)	12 (100%)
Interface spoofing	Two pages merge together in certain situation	6	0 (0%)	6 (100%)
UI design flaw	Plain-text information leak due to autosuggest	17	0 (0%)	0 (0%)
Misc	Geolocation events fire after document deletion	22	0 (0%)	3 (14%)
Overall		175	83 (46%)	135 (77%)

Table 7.1: Browser vulnerabilities. This table shows the number of Chrome vulnerabilities that Chrome itself contains, OP2 contains, and IBOS contains or eliminates.

prospective, they have identical ability to contain attacks in most cases. In the following paragraphs, when we show Chrome is able to contain a type of vulnerability, it indicates OP2 could handle the case as well unless explicitly stated otherwise.

Memory exploitation: an attacker could use a memory corruption bug to deploy a remote code execution attack. For Chrome, if the bug is in its rendering engine, Chrome contains the attack. However, bugs in the browser kernel give attackers access to the entire browser. For IBOS, bugs in either the rendering engine or other service components are contained as they are all out of the TCB.

XSS: browsers rely on careful sanitization and correct processing of different encodings to prevent XSS attacks. For both Chrome and IBOS, it is infeasible to eliminate XSS attacks, but they both contain the attacks in the affected web apps.

SOP circumvention: Chrome runs contents in frames from different origins in a single address space and uses scattered "if" and "else" statements to enforce the same-origin policy. This logic can be sometime subverted. In IBOS, we run iframes in different web page instances to provide strong isolation and check cross-origin access in the IBOS kernel.

Sandbox bypassing: Chrome uses sandboxing techniques, such as SELinux, to limit the rendering engine's authority. However, rule-based sandboxing is complex and can be bypassed in some scenarios. In IBOS, we designed browser abstractions to restrict the authority of each subsystem, which are immune to this kind of problem naturally.

Interface spoofing: browsers are sometime vulnerable to visual attacks in which a malicious website can use complex HTTP redirection or even replicate the "look and feel" of victim websites to deploy phishing. Chrome uses a blacklist-based filter to warn users of malicious websites. In IBOS, the IBOS kernel separates the display of different web page instances and uses the labels of web page instances to display the correct URL in the top of the screen to give the user a visual cue of which website he or she is visiting.

UI design flaw: some security concerns arise because of careless implementation, such as showing users' passwords in plain text. Both Chrome and IBOS are vulnerable to this type of problem.

Misc: some vulnerabilities could not easily be classified and mostly have low security severity. This is the category for those remaining bugs.

In Table 7.1, we show the detailed results of the analysis of the 175 vulnerabilities, broken down by the classifications above. We examined each of them to determine whether Chrome contains the threats in the affected components, and whether IBOS contains or eliminates the attacks. The table shows IBOS successfully protects users from 135 of the 175 vulnerabilities (77%).

The largest portion of bugs are browser implementation flaws that cause memory corruption and allow remote code execution. Chrome does a fairly good job containing most of them when they are in the rendering engine. However, Chrome is unable to contain exploits in the browser kernel. A good example is a bug in the HTTP chunked encoding module in the browser kernel, which opens the possibility for a remote attacker to inject code. In IBOS, the TCP/IP and HTTP stack is pushed out of the TCB, and is replicated and isolated according to browser security policies. Thus, IBOS is able to contain this bug. The three memory corruption bugs IBOS could not contain were from bugs in Chrome's message passing system. Because the IBOS message passing logic resides within our TCB, we counted these bugs as bugs that IBOS would have missed.

7.1.2 OS and library vulnerabilities

To evaluate the security impact of IBOS's reduced TCB, we obtained a list of 74 vulnerabilities found in the Linux kernel, X Server, GTK+, and glibc this year so far (as of Sep. 18, 2010) [1] to see how the IBOS architecture handles them. Out of the 74 vulnerabilities, 20 are related to unsupported hardware architectures and devices, and 26 cause denial-of-service, which is out-of-scope for this dissertation. For the remaining 28, we classify them based on the subsystem the vulnerability lies in to determine if IBOS is susceptible to these vulnerabilities.

Table 7.2 shows IBOS is able to prevent 27 of 28 vulnerabilities (96%). The only vulnerability we miss is a memory corruption vulnerability in the e1000 Ethernet driver. Normally IBOS is *not* susceptible to bugs in device drivers, but this particular bug resulted from the driver not accounting properly for Ethernet frames larger than 1500 bytes, and this type of logic is what our NIC verification state machine uses, so we counted this bug against IBOS.

Affected Component	Num.	Prevented
Linux kernel overall	21	20 (95%)
File system	*12*	*12 (100%)*
Network stack	*5*	*5 (100%)*
Other	*4*	*3 (75%)*
X Server	2	2 (100%)
GTK+ & glibc	5	5 (100%)
Overall	28	27 (96 %)

Table 7.2: OS and library vulnerabilities. This table shows the number of vulnerabilities that IBOS prevents.

7.1.3 IBOS motivation revisited

In the beginning of Chapter 3, we listed some examples of attacks that an attacker can use to still cause damage to modern secure web browsers by exploiting code in their TCB. We revisit these examples again to argue that IBOS can prevent them.

A compromised Ethernet driver cannot access the DMA buffers used by the device. Even if an attacker exploits the Ethernet driver, he or she still cannot tamper with network packets because the driver does not have access to DMA buffers and because the IBOS kernel validates all transmit and receive buffers that the driver sets.

A compromised storage module has little impact on data confidentiality and integrity. The IBOS kernel encrypts all data with secret keys that only the IBOS kernel has access to. Stored objects are tagged with a hash and origin information so that the IBOS kernel is able to detect tampered data. The only thing a compromised storage module can do is delete objects.

A compromised network stack is constrained as well. In IBOS, every network process runs a complete network stack. A compromised network process cannot send users' data to a third party host as the IBOS kernel ensures it can only communicate with the expected host. Network processes do have the ability to modify or replay HTTP requests, but the web server might have a mechanism to defend against replay attacks.

A Compromised window manager cannot affect other subsystems in IBOS. In IBOS, the role of window manager is simplified to only draw the browser chrome. It can change some potentially sensitive information, such web page titles. However, the IBOS kernel displays the URL of the current tab in the

	Efficacy	Coverage	Comp.
Cookie protection	1/1 scenario	31/39 sites	n/a
Frame defense	6/6 scenarios	34/34 sites	114/116 sites
Auto `JSON.parse()`	3/3 attacks	12/12 sites	116/116 sites
DOM-based XSS prev.	10/10 attacks	14/14 sites	116/116 sites

Table 7.3: Defense efficacy, coverage, and compatibility of Zan. This table shows how the three Zan algorithms perform when tested on top websites.

kernel display area, providing users with some visual cues as to the provenance of the displayed web content.

7.1.4 Efficacy of Zan

In this section we simulate the attacks that we designed Zan to prevent, and show the protection result in the second column of Table 7.3. Overall, our algorithms improve the security in at least one way for 57 different sites in our top websites set.

Cookie protection. To evaluate how effectively our algorithm detects credential cookies, we apply it to 33 websites that have login cookies but have not incorporated `HttpOnly` flags. To simulate an attack we log in to each of the websites. Then, we delete all of the cookies that Zan marks as `HttpOnly`. After erasing the `HttpOnly` cookies we attempt to continue our login session. If the website kicks us back to the login page we consider this a successful defense because it implies that at least one of the cookies Zan identified was needed for authentication at the website. Thus, if an attacker had stolen the cookies via JavaScript then the set of cookies they would have access to would not allow authenticated requests.

For 31 of the 33 websites that we tested, Zan ended the session after we deleted the `HttpOnly` cookies. The two websites that were able to continue the session despite the deleted cookies were `4share.com` and `imageshack.us`. For these sites Zan identified cookies, but we were able to continue interacting with the site as an authenticated user despite our deleted cookies. Interestingly, the cookies we discarded had names "PHPSESSION" and "JSESSION", but our sessions persisted despite their removal.

Frame defense. To test our frame defense we try to frame a website that ZAN injects `X-Frame-Options` into and we confirm that ZAN prevents framing. We verify that the five attack scenarios that are applicable to WebKit [82] can be mitigated in ZAN because ZAN does not rely on the correct execution of frame busting code.

Automatic `JSON.parse()`. To test our JSON parsing defense we manually craft three attacks that simulate the script injection vulnerabilities we describe in Section 5.5. In all of our tests ZAN detected the JSON text and ran it through the `JSON.parse()` parser, which correctly failed to parse the text.

DOM-based XSS prevention. We have tested our protection mechanism against publicly disclosed attacks and examples of DOM-based XSS attacks. We test against five documented XSS attacks disclosed for popular services. We also generate five synthetic attacks based on examples used to demonstrate DOM-based XSS attacks. For both sets of attacks our policy successfully prohibits the execution of the injected script.

7.1.5 ZAN's coverage and compatibility impact

In this section we evaluate websites where the additional protection provided by ZAN would improve security. Based on manual analysis of top websites we identify websites that would benefit from the protections each of our algorithms provide. Then, we visit these websites with ZAN and measure how many of them ZAN was able to detect and apply the appropriate defenses. For our tests we visit the site and stay for five seconds after the page finishes loading to give the XMLHttpRequests time to use JSON objects. we also analyze cases where ZAN changes the behavior of a website by enabling security features. We show the coverage and compatibility impact in the third and fourth column of Table 7.3 respectively.

Cookie protection. To test the coverage of our `HttpOnly` algorithm, we logged into websites to test ZAN's ability to set `HttpOnly` cookies. As we describe in Section 5.3, we train our algorithm on the 54 websites that use

`HttpOnly` cookies. After this training, we applied our algorithm to the remaining 39 websites that we had login credentials for.

ZAN was able to apply `HttpOnly` to 33 of websites automatically. There are 6 websites for which our algorithm could not detect credential cookies, mostly due to either low entropy or short value and irregular cookie names. `Craigslist.com` had credential cookies using seemingly common *Login* name, but this name was infrequent in our data set, so we did not detect this cookie. Of the 33 websites that ZAN did set `HttpOnly` cookies for, the algorithm identified authentication cookies for 31 of them, as we described in Section 7.1.4.

The fundamental way of evaluating the usability impact of our cookie protection algorithm is to examine if one cookie that is designed to use in JavaScript has been incorrectly tagged with `HttpOnly` by ZAN. However, without full knowledge of the usage of each cookie, it is infeasible to carry out a complete quantitative analysis. Instead, to have a close estimation, we opt to visit the top 10 websites and use their representative services. Our tasks include, but are not limited to, doing a Web search, sending and receiving email, editing a Wikipedia document, sending messages to Facebook friends, posting on Twitter, customizing Yahoo! homepage, publishing a blog, and commenting a popular video in YouTube. In all of these user interactive experiments, we performed each task without any problems, indicating that ZAN does not affect popular websites. Moreover, the whole algorithm can be tweaked to be more aggressive or conservative.

Frame defense. Our frame defense and automatic `JSON.parse()` algorithms were able to cover all of the potential opportunities to apply stronger defenses. For our frame defense we visited the 34 websites in our set of top websites that have frame busting code, and ZAN applied `X-Frame-Options` to all 34 sites. Interestingly all of the websites in our test that do use `X-Frame-Options` also include frame busting code. Thus, had an HTTP proxy stripped the `X-Frame-Options` from the HTTP response header then ZAN would still protect the site.

Fundamentally the `X-Frame-Options` mechanism is different from frame busting. `X-Frame-Options` prevents a cross-origin frame from being loaded whereas frame busting is controlled by the programmer and usually (but not always) results in navigation of the top-level frame. Thus, the effects of

`X-Frame-Options` are going to be different than if the frame busting code ran for all of the websites in our experiments. However, the `X-Frame-Options` option provides a more robust mechanism for preventing framing, which is fundamentally what frame busting is trying to do.

To test our frame defense we ran two experiments. First, we visited all of the top websites in our data set and we measured ZAN's effects on any of the `IFRAME`s included in these 116 sites. Then, we visited the 82 websites in our top websites sets that did *not* have frame busting code and we framed them to see if ZAN applied `X-Frame-Options` incorrectly.

ZAN maintains compatibility with all of the `IFRAME`s included in our top websites and 80 of the 82 websites that do not include frame busting code. The first website where ZAN injects a `X-Frame-Options` incorrectly is `en.wikipedia.org/Main_page`. Wikipedia does contain a frame busting code, but this code is protected by an if statement. This if statement enables Wikipedia to frame bust its login page, but not the main page. Since we use string pattern matching instead of relying on complex control flow analysis in ZAN, we are unable to eliminate this false positive. The other case is similarly caused by a if statement in `cnn.com`. CNN maintains a blacklist of websites and only executes its frame busting code when it is framed in one of them. For the same reason as in Wikipedia, ZAN applies the `X-Frame-Options` defense even though the domain we use for the top-level frame is not in the blacklist.

Automatic `JSON.parse()`. For our automatic `JSON.parse()` algorithm we visited the 12 websites in our set of top websites that deserialize JSON text using `eval()` and ZAN ran all 48 JSON objects on these sites through the `JSON.parse()` parser correctly.

For our experiments, ZAN's automatic use of the `JSON.parse()` function did not affect any of the 116 websites we visited. In other words, all eval strings were processed identically in ZAN when compared to processing them with `eval()`. However, we recognize that we did have to add to the grammar of our `JSON.parse()` parser a little bit to maintain this compatibility and it is possible that websites outside of our data set could induce false positives, but our evidence suggests that our techniques would be robust.

DOM-based XSS prevention. The mechanism that prevents DOM-based attacks does not cause any compatibility problems. We test it on the top 116 websites, and there is no case that our system prevent legitimate JavaScript from being executed. Our policy is also configured to generate warnings when tainted data is passed to the HTML parser and could result in a DOM-based XSS attack. We found that there are fourteen websites using input from unsafe sources, and Zan is able to generate warnings on all of them. Thirteen of them use JavaScript `escape()` method to sanitize the text. While for remaining one we were able to construct an attack to exploit the vulnerability found. This vulnerability is in a popular sports website and can be used to inject arbitrary JavaScript into the page. After generating a sample attack we confirmed that our policy prevents the attack, and reported the problem to its administrator.

7.2 Performance

To evaluate the performance implication of the three systems we build, we choose to measure their browsing experiences and compare with other web browsers. We use page load latency to represent browsing experience. Page load latency is defined as the elapsed time between initial URL request and the DOM `onload` event.

All experiments were carried out on a 2.33GHz Intel Core 2 Quad CPU Q8200 with 4GB of memory, a 320GB 7200RPM Seagate ST3320613 SATA hard drive and an Intel PRO/1000 NIC connected to 1000Mbps Ethernet. However, experiments for difference systems were conducted at different time, using different versions of Linux, Qt, and WebKit, which means the performance for the same website could vary. And only for OP2, we turned on disk cache for HTTP data, resulting in overall faster page loading.

7.2.1 IBOS performance

To evaluate the performance implication of IBOS's architecture, we compare its browsing experience to other web browsers running in Linux. For Linux, we used Ubuntu 9.10 with kernel version 2.6.31-16-generic (x86-64).

We compare IBOS with Firefox 3.5.9, Chrome for Linux 4.1.249. We also

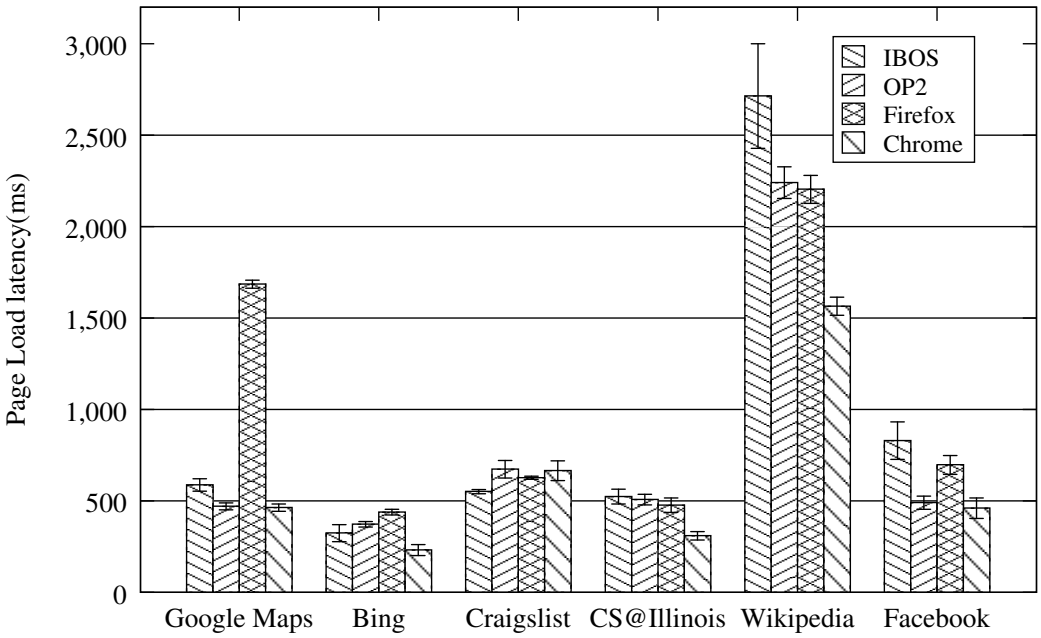

Figure 7.1: Page load latencies for IBOS and other web browsers. All latencies are shown in milliseconds.

compare it with OP2. Since OP2 uses almost identical design of browser components except for just one single network process, we could focus on the performance impact of our IBOS kernel architecture. In IBOS, we statically allocate processors for subsystems as follows: the kernel and device drivers run on CPU0, network processes run on CPU1, web page instances run on CPU2, and all other components run on CPU3. IBOS, OP2, and Chrome all use a same version of WebKit from February 2010 with just-in-time JavaScript compilation and HTTP pipelining enabled. For the WebKit-based browsers, we instrument them to measure the time in between the initial URL request and the DOM `onload` event. For Firefox, we use an extension that measures these same events. To reduce noise introduced by our network connection, we load each web site using a fresh web page/browser instance with an empty cache 15 times and report the average of the five shortest page load latency times.

In Figure 7.1, we present the page load latency times for six popular websites and show the standard deviations with the error bars. Overall, Chrome has the shortest page load latencies due to its effective optimization techniques. For `maps.google.com`, IBOS, OP2, and Chrome out-perform Fire-

fox, possibly due to optimization in the WebKit engine for this particular site. For `www.bing.com`, `sfbay.craigslist.org` and `cs.illinois.edu`, IBOS, OP2, and Firefox show roughly the same results. IBOS has the fastest loading time for `craigslist`. `Craigslist` is a simple web site with few HTTP requests and with a large number of HTML elements. We hypothesize that the small performance improvement is due to the simplified IBOS software stack.

Both `en.wikipedia.org/wiki/Main_Page` and `www.facebook.com` have more HTTP requests than any of the other sites, and we observe slower page load latencies for IBOS than for other browsers. For these experiments IBOS performs slower than OP2. Because we use the IBOS components in Linux, we believe that this performance difference occurs from overhead in the IBOS kernel. To test this hypothesis, we ran a number of micro benchmarks on the two systems and we believe that the overhead is due to contention for spinlocks in the L4 IPC implementation. The net effect of this contention is that heavy use of network processes requires heavy use of IPC, which adds latency to all IPC messages and slows down the overall system. However, the OP2 results for these experiments show that this slow down is not fundamental and can be fixed with a more mature kernel implementation.

Overall, the page load latency experiments show that even with a prototype implementation of IBOS, our architecture will not slow down the browsing speed significantly for the web sites we tested.

7.2.2 OP2 performance

To measure the latency introduced by OP2, we compare the load times of a few common pages with a simple WebKit-based browser – QtLauncher. QtLauncher runs as a single process, and uses exactly the same Qt Framework (4.6) and WebKit engine (r54749). To some extent, its performance could serve as the upbound for OP2.

Figure 7.2 shows the results of five different OP2 configurations in comparison to QtLauncher. Each website is loaded six times, and the loading times are averaged. In all tests we use a warmed up browser that has previously loaded the page to warm any caches. We have performed three optimizations to improve the performance of OP2 and evaluate each optimization as well

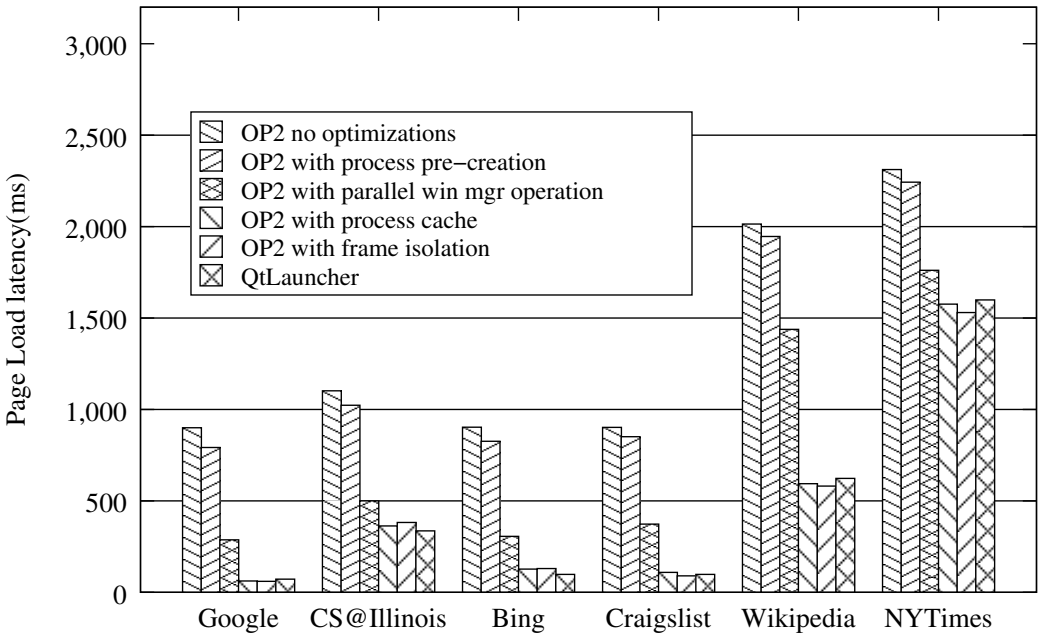

Figure 7.2: OP2 performance. This figure shows the page load latencies for a six popular pages using OP2 and QtLauncher. From the left bar to the right shows OP2 starting without any optimizations, and shows the effects of adding each optimization cumulatively, where the OP2 with frame isolation bar represents a fully-optimized OP2 browser. All tests have primed disk cache for HTTP data.

---

as the effects of frame isolation. Figure 7.2 shows how each optimization improves the overall OP2 performance.

Process pre-creation. Our first optimization is designed to eliminate the cost of creating new processes by creating web page instances during idle times in anticipation of future use. Currently we keep a web page instance pool containing a single web page instance. As shown by the second bar from the left in Figure 7.2, OP2 with only the process pre-creation optimization performs slower than QtLauncher in every test.

Parallelizing window manger operations. The second optimization we implemented performs the window reparenting in parallel with the page downloading and rendering. Window reparenting under X11 is costly, and this optimization provides the largest performance improvement. The third bar from the left in Figure 7.2 shows the impact of this optimization combined with process pre-creation. On average, parallelizing the window manager operations improves page load times by approximately a half second.

Process caching. Our last optimization reuses old web page instances if the same URL is loaded multiple times, limiting the costs of initializing a web page instance with the same content. This approach also enables using the in-memory object cache for OP2. Process caching uses a least recently used cache of web page instances that can be reused when a previously seen URL is loaded. The default cache configuration retains ten web page instances in the cache.

Combined with the other two optimizations, process caching improves the performance of OP2 to be as fast or faster than QtLauncher in all of our tests, as shown by the fourth bar from the left in Figure 7.2.

Frame Isolation. In Section 4.2.5 we discussed the design of OP2 that allows for the browser to automatically isolate frames. The fourth and fifth bars from the left in Figure 7.2 show a comparison of OP2 without and with frame isolation. For most of the pages tested we see little difference in the page load times. The result is a little different from an early experiment of OP2 [47], where OP2 shows greater performance improvement for `nytimes.com`. We believe that OP2 isolates cross-origin frames in separate web page instances allowing the use of additional CPUs to render frames in parallel. However, browser is complex and its performance is affected by various sources, such as web server efficiency, web page content, network latency, and rendering engine algorithm. It could be some change in one of those factors leading to the different result. For example, the new WebKit engine could lower the priority of `IFRAME` loading, delaying the loading of frames until after `onload` event. Nevertheless, superior performance is not the goal of this dissertation, and we would defer further investigation of this topic into future work.

As shown in Figure 7.2, OP2 without optimizations are always slower than QtLauncher, sometimes by more than 10x; however, with all optimizations enabled, OP2 is as fast as QtLauncher. Interestingly, by decomposing the browser into separate subsystems that run in different OS processes and enforcing additional security policies, we often add latency. For example, sending an HTTP request in QtLauncher and OP2 both require setting up HTTP headers, attaching appropriate cookies, connecting to the network, and sending the request. However, in OP2 the web page instance must also send an IPC message, through the browser kernel, to the network subsystem, adding latency directly to roughly equivalent functionality.

Running reasonable amount of processes for subsystems, however, could expose parallelism that is absent in current versions of WebKit, potentially enabling multicore systems to improve performance. For example, OP2 will overlap the handling of HTTP requests with the downloading, parsing, rendering, and painting of WebKit. We believe that this trade off of increased latency for increased parallelism is why OP2 is able to run as fast as Qt-Launcher on our multicore system. We recognize that this observation might indicate that WebKit could be made faster by introducing more parallelism, which could expose the latency added by OP2, but further study of this performance issue is beyond the scope of this dissertation.

Overall the results in Figure 7.2 indicate that with our optimizations OP2 does not introduce latency that would be detrimental to a user using OP2.

7.2.3 Zan performance

To evaluate the performance implication of the three defense techniques used in Zan, we compare its browsing experience to the unmodified OP2 web browser, both using Qt Framework 4.6 and WebKit r54749. For Linux, we used Ubuntu 10.04 with the x86-64 kernel.

We compare Zan with the unmodified OP2 web browser. In general, there is no noticeable performance impact during our daily use of OP2 when enabling Zan. Nevertheless, we measured the page load latency times on six web sites: `google.com`, `facebook.com`, `yahoo.com`, `live.com`, `baidu.com`, and `youtube.com`. We visited the front page for each of these sites six times, and present the average in Figure 7.3. We show that Zan did not add any measurable amount of overhead.

7.3 Summary

Security evaluation is hard, especially for newly proposed systems, as there is no standard benchmark available. Typical methods include: 1) automated testing or manual review of the code. But bugs always exist in spite of the effort spent; 2) formal verification of the implemention. But this approach itself is under research and there is no mature framework for commodity systems; 3) deploying into practice and drawing conclusions from real world

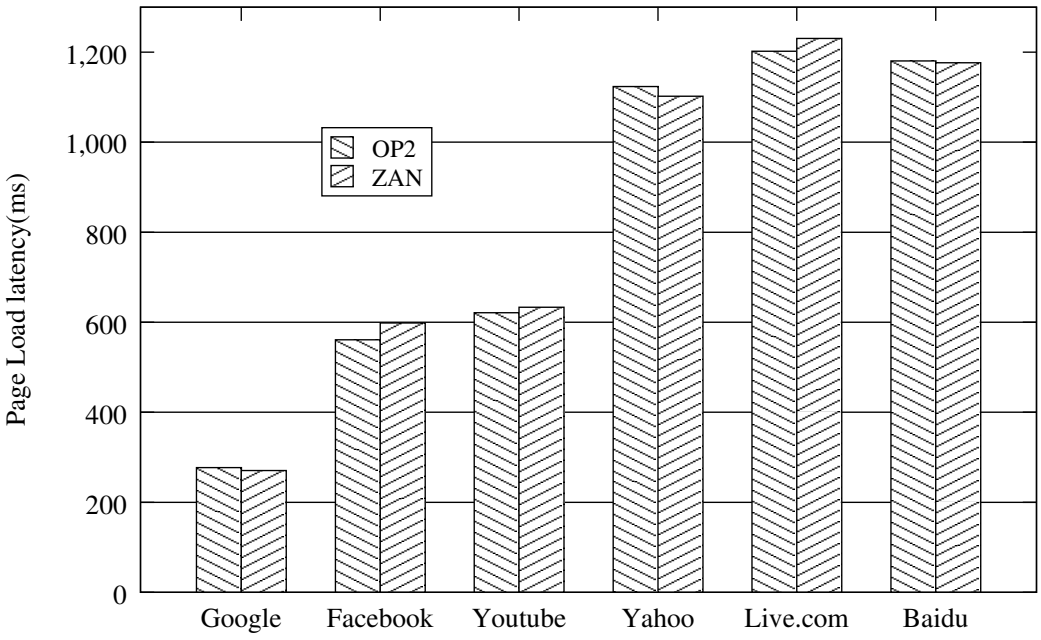

Figure 7.3: Performance impact of ZAN. This figure shows the page load latencies for unmodified OP2 web browser and ZAN. All latencies are shown in milliseconds.

---

experiences. This approach gives high confidence but is infeasible for a research project.

Instead, to evaluate the security of the new systems we build, we examine the features of existing vulnerabilities of similar systems. We then quantitatively analyze the architectural impact of our systems to each vulnerability: 1) whether it is structurally impossible; 2) whether it is addressed by a newly introduced mechanism 3) whether it still affects the new design?

Overall, we show that, by following the six principles we proposed in Chapter 1, we are able to build systems that could withstand most of the attacks we target to prevent, still maintain reasonable compatibility to existing web standards, and introduce little overhead to the overall browsing performance.

CHAPTER 8

FUTURE WORK

Though providing a great step towards secure web browsing, the systems we described in previous chapters are not yet the final solutions for the Web. However, we believe that it is feasible to solve some remaining issues by following the six principles we summarize in this dissertation. In this chapter, we discuss some of the future research directions.

8.1 Safe extensibility

In Chapter 3, we described the security invariants we introduced in IBOS. Security invariants are used as the assertions on all interactions between subsystems to check basic security properties. One key argument to security invariants is that we can extract security relevant information from messages automatically to enforce security policies. By doing so, we are able to provide high assurance of the overall system without having to understand how each individual subsystem is implemented.

The current approach of enforcing security invariants is not flexible. In our prototype, all the messages passed between subsystems are formalized according to a pre-defined format. To enable security invariants, we hard-code extraction logic and security policies in the IBOS kernel. This inflexible approach is sufficient as the first step for a research prototype, since we only support a limit number of hardware devices and network protocols.

As discussed before, the Web is fast evolving. A browsing system has to have the capability to adapt to this evolution. Meanwhile, to make IBOS as practical system, we inevitably need to support for more devices and different types of services. For example, we will have to support cameras, GPSes, and different network protocols. We hope to have a mechanism to enable more flexible security policies safely for these new features while avoiding including

significantly more lines of code in the IBOS TCB.

To overcome the scalability problem, we propose a set of generic abstractions to support flexible security policies in IBOS. The idea is mainly motivated by two previous projects – UDFs in Exokernel [56] and Proof-Carrying Code [72]. Both approaches support untrusted extensions to operating systems without sacrificing safety.

Similarly, IBOS should provide a set of generic abstractions to enable: 1) untrusted components to install mapping functions to allow automatically extraction of security relevant information from messages; 2) administrators or trusted web sites to install new security policies dynamically without changing the IBOS kernel implementation.

While allowing those extensions to IBOS kernel, we also need to make sure that the new logics are separated with the TCB of IBOS and would not affect other parts in the systems. We hope that this set of abstractions allow IBOS to accommodate the development of the Web and enable safe browser extensibility (e.g., Native Client-like plugins [110]), while retaining a small TCB.

8.2 Convergence of mobile apps and web apps

Recent years have witnessed a dramatic trend towards mobile computing driven by increasingly powerful mobile devices, such as smartphones, tablets. The Web also plays an important role in this shift. Though more capable than ever before, mobile devices still have limited computation power compared to their desktop counterparts. However, these mobile devices fit perfectly in the scenario of accessing the numerous resources in the Web – a thin client. Intuitively, the popularization of mobibe computing would also reciprocate a favor, stimulating further development of the Web.

Two roads, however, diverge on these shrimpy screens. One is the path to the web apps, the other leads to the native applications (or mobile apps). While web apps possess a series of advantages, such as platform agnosticism, free of installation and update, native support of sharing and collaboration, and easy to develop, they do suffer from issues like requirement of Internet access, subpar performance, limited access to hardware sensors (e.g., compass, gyroscope), and responsiveness of user interface. In addressing these

shortcomings of web apps, mobile apps come on the scene, serving as the complement.

Nevertheless, there are no fundamental differences between web apps and mobile apps. They often present the same information to the users despite of running on top of different abstractions. For example, some popular websites (e.g., `google.com`) publish mobile app versions only to facilitate the access to them. Some other websites just encapsulate their web content in the corresponding mobile apps, such as `aa.com`. And we also see from survey [33] that mobile apps incur the same security and privacy concerns as web apps.

Based on the above observation, we argue that we should treat the security of mobile apps in a similar way to web apps, i.e., we need to follow these six principles to build secure mobile operating systems.

Before we could research further into the security of apps in mobile platform, we need to answer a question – are the problems faced in mobile platform exactly the same as in desktop? The trend of mobile computing enables even more average users. Meanwhile, users are more likely to store sensitive information in their phones, such as contacts, private photos, and financial records. Moreover, mobile devices also expose new set of information that is not presented in desktops, such as geolocation. Consequently, the answer to the question should be "no". Unfortunately, current mobile operating systems, such as iOS and Android, are constructed just as the functionally reduced desktop operating systems, providing little specific security consideration. We hope future research could identify the problem, and propose correct solutions to the security of mobile apps and web apps.

8.3 Browser performance

Web browsing should be fast, but currently it is slow, especially in mobile platform. Modern computer systems have plenty of hardware to power fast applications – they contain on the order of gigabytes of memory, are connected to high-speed networks capable of sending and receiving megabytes each second, and have multiple cores running in the gigahertz range. However, web browsing fail to make use of these luxury computing resources.

The need of speed has become even stronger as recent studies show evidence of a positive correlation between the performance of web apps and

business value. For example, Google and Microsoft reported that a 200ms increase in page load latency times resulted in "strong negative impacts" and that delays of under 0.5 seconds "impact business metrics" [84].

Several recent projects have been devoted to speeding up web browsing. There are proposals for optimizing individual components [44], pipelining network transfers and computations between web servers and the browser [54, 73], and pushing computation to web servers [68]. Although these optimizations improve the performance of the web apps, none of them exploits the multi-core client systems that run on current desktops and future mobile platforms.

There are also research efforts of exploiting parallelism to improve browser performance such as parallel layout algorithms [13, 66]. However, these special cases only speed up web apps that make heavy use of specific features (e.g., cascading style sheets (CSS)). Unfortunately, years of sequential optimizations, the sheer size of modern browsers (e.g., Firefox has over 3 million lines of code), and the fundamentally single-threaded event-driven programming model of modern browsers make it challenging to refactor today's browsers into multi-threaded parallel applications.

Our previous work indicates that task-level parallelism applied to the browser might be feasible [47]. We demonstrate that rendering logically independent `IFRAME`s in separate processes would result in performance gain. However, this approach is limited to the tasks that the browser developers identify ahead of time. And in practice, `IFRAME`s do not always load balance.

To systematically exploit parallelism to improve the speed of web browsing, we need to answer the following questions:

- How to automatically divide rendering of a single web app into several loosely dependent tasks?
- How to load balance these tasks to overcome the overhead of running them in different computing units?
- How to maintain the same look and feel and function of the original web app?

In answering the above questions, we hope to have a systematical approach that is able to speed up web browser in both desktop and mobile multi-core platforms.

8.4 Summary

There are two major directions to pursue future research. One is following the principles learned in this dissertation to build practical and extensible secure browsing systems in both desktop and mobile platforms.

The other is to research into the performance issues in today's browsing systems. With careful design, we hope to systematically parallelize the process of web browsing, taking advantage of current and future multi-core systems.

CHAPTER 9

CONCLUSIONS

In recent years, the Web has crawled into almost every aspect of human life. Meanwhile, the web browser has become the primary interface to today's computing systems, handling vital personal information and with implications for users' security and privacy. Despite many attempts to improve the security of web browsing, these efforts fail to address the fundamental issues in modern browsing systems.

In this dissertation, we proposed a set of design and architectural principles that should be followed when building secure browsing systems: 1) make security decisions at the lowest layer of software and make it simple; 2) enforce strong isolation between distinct browser-level components; 3) employ simple and explicit communication between components; 4) provide the right set of operating system abstractions; 5) maintain compatibility with current browser standards; 6) expose enough browser states and events to enable new browser security policies.

Following these principles, we presented IBOS, the first operating system and web browser co-designed to reduce drastically the TCB for web browsers and to simplify browsing systems. To achieve this improvement, we built IBOS with browser abstractions as first-class OS abstractions and removed traditional shared system components and services from its TCB. With the new architecture, we showed that IBOS could enforce traditional and novel security policies, and withstand attacks on device drivers, browser components, or traditional applications.

We have also described the OP2 web browser – a standalone secure browser architecture when a specialized browser operating system is not available. This architecture resembles a microkernel operating system. By employing a browser kernel to enforce security policies and explicit communications between isolated browser components, this new browser architecture is able to enable secure web browsing on top of commodity operating systems. We

have shown that by using an architecture that is designed to be secure we can also use formal method to validate its security properties.

In addition, we presented browser-based approaches that try to improve security further. We demonstrated four mechanisms – cookie protection, frame-based attack defense, secure JSON deserialization, and DOM-based XSS prevention. These mechanisms capitalize on unique details about web apps to provide automated security mechanisms at the client side.

Overall, we show that the principles we proposed are able to guide the design of secure systems that overcome the issues of existing browsing systems. We believe that the principles we defined and systems we built advance the state of the art of secure web browsing.

REFERENCES

[1] CVE - Common Vulnerabilities and Exposures (CVE). http://cve.mitre.org.

[2] Gecko plugin API reference. https://developer.mozilla.org/en/Gecko_Plugin_API_Reference.

[3] JSON in JavaScript. http://www.json.org/js.html.

[4] Mitigating cross-site scripting with HTTP-only cookies. http://msdn.microsoft.com/en-us/library/ms533046.aspx.

[5] Qt - A Cross-platform application and UI. http://qt.nokia.com/.

[6] The WebKit Open Source Project. http://webkit.org/.

[7] uClibc. http://www.uclibc.org/.

[8] Qt labs blogs: So long and thanks for the blit, 2008. http://labs.trolltech.com/blogs/2008/10/22/so-long-and-thanks-for-the-blit/.

[9] L4Ka::Pistachio microkernel, 2010. http://l4ka.org/projects/pistachio.

[10] Alexa. Alexa top 500 global sites. http://www.alexa.com/topsites.

[11] Anderson, J. P. Computer security technology planning study. Tech. rep., HQ Electronic Systems Division (AFSC), October 1972. ESD-TR-73-51.

[12] Apple Inc. About the security content of the iOS 4.0.2 update for iPhone and iPod touch, August 2010. http://support.apple.com/kb/HT4291.

[13] Badea, C., Haghighat, M. R., Nicolau, A., and Veidenbaum, A. V. Towards parallelizing the layout engine of firefox. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism* (Berkeley, CA, USA, 2010), HotPar'10, USENIX Association, pp. 1–6.

[14] Balduzzi, M., Egele, M., Kirda, E., Balzarotti, D., and Kruegel, C. A solution for the automated detection of clickjacking attacks. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2010), ASIACCS '10, ACM, pp. 135–144.

[15] Barth, A., Caballero, J., and Song, D. Secure content sniffing for web browsers or how to stop papers from reviewing themselves. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2009).

[16] Barth, A., Jackson, C., and Mitchell, J. C. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (2008), pp. 75–88.

[17] Barth, A., Jackson, C., and Mitchell, J. C. Securing frame communication in browsers. In *Proceedings of the 17th USENIX Security Symposium (USENIX Security 2008)* (2008), pp. 17–30.

[18] Barth, A., Jackson, C., Reis, C., and The Google Chrome Team. The security architecture of the chromium browser, 2008. `http://crypto.stanford.edu/websec/chromium/chromium-security-architecture.pdf`.

[19] BBC. Facebook "clickjacking" spreads across site, June 2010. `http://www.bbc.co.uk/news/10224434`.

[20] Bohannon, A., and Pierce, B. C. Featherweight Firefox: Formalizing the core of a web browser. In *Usenix Conference on Web Application Development (WebApps)* (June 2010).

[21] Bomberger, A. C., Frantz, W. S., Hardy, A. C., Hardy, N., Landau, C. R., and Shapiro, J. S. The KeyKOS nanokernel architecture. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures* (Berkeley, CA, USA, 1992), USENIX Association, pp. 95–112.

[22] Brumley, D., Caballero, J., Liang, Z., Newsome, J., and Song, D. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of the 16th USENIX Security Symposium (Security07* (2007).

[23] Cadar, C., Dunbar, D., and Engler, D. R. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 2004 Symposium on Operating Systems Design and Implementation (OSDI)* (2008), pp. 209–224.

[24] Chen, S., Meseguer, J., Sasse, R., Wang, H. J., and Wang, Y.-M. A systematic approach to uncover security flaws in GUI logic. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (May 2007), pp. 71–85.

[25] Chen, S., Ross, D., and Wang, Y.-M. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)* (2007), pp. 2–11.

[26] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Quesada, J. F. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science 285*, 2 (August 2002), 187–243.

[27] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. Maude manual (version 2.3), 2007.

[28] Cox, R. S., Hansen, J. G., Gribble, S. D., and Levy, H. M. A safety-oriented platform for web applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (May 2006), pp. 350–364.

[29] Criswell, J., Geoffray, N., and Adve, V. Memory safety for low-level software/hardware interactions. In *Proceedings of the Eighteenth Usenix Security Symposium* (August 2009).

[30] Douceur, J. R., Elson, J., Howell, J., and Lorch, J. R. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (2008), OSDI'08, pp. 339–354.

[31] Dunkels, A., Woestenberg, L., Mansley, K., and Monoses, J. lwIP embedded TCP/IP stack. http://savannah.nongnu.org/projects/lwip/, 2004.

[32] Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F., and Morris, R. Labels and event processes in the asbestos operating system. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2005), ACM, pp. 17–30.

[33] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (2010), OSDI'10.

[34] Engler, D. R., Kaashoek, M. F., and Jr., J. O. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 1995 Symposium on Operating Systems Principles* (December 1995), pp. 251–266.

[35] Erlingsson, U., Abadi, M., Vrable, M., Budiu, M., and Necula, G. C. Xfi: software guards for system address spaces. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 75–88.

[36] Feske, N., and Härtig, H. DOpE - a window server for real-time and embedded systems. In *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium* (Washington, DC, USA, 2003), IEEE Computer Society, p. 74.

[37] Feske, N., and Helmuth, C. A Nitpicker's guide to a minimal-complexity secure GUI. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 85–94.

[38] Garfinkel, T. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the 2003 Network and Distributed System Security Symposium (NDSS)* (February 2003).

[39] Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. A secure environment for untrusted helper applications. In *Proceedings of the 1996 USENIX Security Symposium* (July 1996), pp. 1–13.

[40] Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an Application Program. In *Proceedings of the 1990 USENIX Summer Conference* (1990).

[41] Google Inc. Chromium. `http://www.chromium.org/chromium-os`.

[42] Google Inc. Google Caja. `http://code.google.com/p/google-caja/`.

[43] Google Inc. Chromium OS, 2010. `http://www.chromium.org/chromium-os`.

[44] Google V8 Team. `http://code.google.com/p/v8`.

[45] Grier, C., King, S. T., and Wallach, D. S. How i learned to stop worrying and love plugins. In *Web 2.0 Security and Privacy* (May 2009).

[46] Grier, C., Tang, S., and King, S. T. Secure web browsing with the OP web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (May 2008), pp. 402–416.

[47] Grier, C., Tang, S., and King, S. T. Designing and implementing the OP and OP2 web browsers. In *ACM Transactions on the Web (TWEB)* (2011).

[48] Gundy, M. V., and Chen, H. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Proceedings of the Network and Distributed System Security Symposium* (February 2009).

[49] Hansen, R., and Grossman, J. Clickjacking, September 2008. `http://www.sectheory.com/clickjacking.htm`.

[50] Härtig, H., Hohmuth, M., Liedtke, J., Wolter, J., and Schönberg, S. The performance of μ-kernel-based systems. In *SOSP '97: Proceedings of the sixteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1997), ACM, pp. 66–77.

[51] Herder, J. N., Bos, H., Gras, B., Homburg, P., and Tanenbaum, A. S. MINIX 3: a highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev. 40*, 3 (2006), 80–89.

[52] Ioannidis, S., and Bellovin, S. M. Building a secure web browser. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (June 2001).

[53] Ioannidis, S., Bellovin, S. M., and Smith, J. Sub-operating systems: A new approach to application security. In *SIGOPS European Workshop* (September 2002).

[54] Jiang, C. Bigpipe: Pipelining web pages for high performance. `http://www.facebook.com/notes/facebook-engineering/bigpipe-pipelining-web-pages-for-high-performance/389414033919`, 06 2010.

[55] Jim, T., Swamy, N., and Hicks, M. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web* (2007), pp. 601–610.

[56] Kaashoek, M. F., Engler, D. R., Ganger, G. R., Briceño, H. M., Hunt, R., Mazières, D., Pinckney, T., Grimm, R., Jannotti, J., and Mackenzie, K. Application performance and flexibility on exokernel systems. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1997), ACM, pp. 52–65.

[57] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: formal verification of an os kernel. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), ACM, pp. 207–220.

[58] Kristol, D. M. Http cookies: Standards, privacy, and politics. *ACM Trans. Internet Technol. 1* (November 2001), 151–198.

[59] Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M. F., Kohler, E., and Morris, R. Information flow control for standard OS abstractions. In *SOSP '07: Proceedings of twenty-first ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2007), ACM, pp. 321–334.

[60] Lawrence, E. Combating clickjacking with X-Frame-Options, March 2010. `http://blogs.msdn.com/b/ieinternals/archive/2010/03/30/combating-clickjacking-with-x-frame-options.aspx`.

[61] Leslie, B., and Heiser, G. Towards untrusted device drivers. Tech. rep., UNSW-CSE-TR-0303, 2003.

[62] LeVasseur, J., Uhlig, V., Stoess, J., and Gotz, S. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 2004 Symposium on Operating Systems Design and Implementation (OSDI)* (December 2004).

[63] Loscocco, P., and Smalley, S. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the 2001 USENIX Annual Technical Conference FREENIX Track* (June 2001).

[64] Maone, G. NoScript - JavaScript/Java/Flash blocker for a safer Firefox experience!, 2008. `http://noscript.net/`.

[65] Meseguer, J. Conditional rewriting logic as a united model of concurrency. *Theoretical Computer Science 96* (1992), 73–155.

[66] Meyerovich, L. A., and Bodík, R. Fast and parallel webpage layout. In *Proceedings of the WWW 2010, Raleigh NC, USA* (2010).

[67] Moshchuk, A., Bragin, T., Gribble, S. D., and Levy, H. M. A crawler-based study of spyware on the web. In *Proceedings of the 2006 Network and Distributed System Security Symposium (NDSS)* (February 2006).

[68] Moshchuk, A., Gribble, S. D., and Levy, H. M. Flashproxy: transparently enabling rich web content via remote execution. In *Proceedings of the 6th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2008), MobiSys '08, ACM, pp. 81–93.

[69] Moshchuk, A., and Wang, H. J. Resource Management for Web Applications in ServiceOS. Tech. rep., Microsoft Research, May 2010.

[70] Nadji, Y., Saxen, P., and Song, D. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the Network and Distributed System Security Symposium* (February 2009).

[71] Nava, E. V., and Lindsay, D. Abusing internet explorer 8's xss filters. `http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf`, 2010.

[72] Necula, G. C. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1997), POPL '97, ACM, pp. 106–119.

[73] Nielsen, H. F., Gettys, J., Baird-Smith, A., Prud'hommeaux, E., Lie, H. W., and Lilley, C. Network performance effects of http/1.1, css1, and png. In *Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication* (New York, NY, USA, 1997), SIGCOMM '97, ACM, pp. 155–166.

[74] Okhravi, H., and Nicol, D. M. Trustgraph: Trusted graphics subsystem for high assurance systems. In *ACSAC '09: Proceedings of the 2009 Annual Computer Security Applications Conference* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 254–265.

[75] Palm Inc. webOS, 2010. `http://opensource.palm.com`.

[76] Perl.org. Perl taint mode. `http://perldoc.perl.org/perlsec.html`.

[77] Provos, N., Mavrommatis, P., Rajab, M. A., and Monrose, F. All your iFRAMEs point to us. In *Proceedings of the 17th Usenix Security Symposium* (July 2008), pp. 1–15.

[78] Provos, N., McNamee, D., Mavrommatis, P., Wang, K., and Modadugu, N. The ghost in the browser: Analysis of Web-based malware. In *Proceedings of the 2007 Workshop on Hot Topics in Understanding Botnets (HotBots)* (April 2007).

[79] Reis, C., Dunagan, J., Wang, H., Dubrovsky, O., and Esmeir, S. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of The 7th Symposium on Operating Systems Design and Implementation (OSDI)* (November 2006).

[80] Reis, C., and Gribble, S. D. Isolating web programs in modern browser architectures. In *Proceedings of the 2009 EuroSys conference* (2009).

[81] Ross, D. IEBlog : IE8 Security Part IV: The XSS Filter, 2008. `http://blogs.msdn.com/ie/archive/2008/07/01/ie8-security-part-iv-the-xss-filter.aspx`.

[82] Rydstedt, G., Bursztein, E., Boneh, D., and Jackson, C. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *in IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010)* (2010).

[83] Saxena, P., Hanna, S., Poosankam, P., and Song, D. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS)* (February 2010).

[84] Schurman, E., and Brutlag, J. Performance related changes and their user impact. Velocity: Web Performance and Operations Conference, 2009. `http://velocityconf.com/velocity2009`.

[85] Sekar, R. An efficient black-box technique for defeating web application attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium* (February 2009).

[86] Shankar, U., and Karlof, C. Doppelganger: Better browser privacy without the bother. In *Proceedings of the 13th ACM Conference on Computer and Communications Security* (2006), pp. 154–167.

[87] Shannon, C. E. A mathematical theory of communication. *The Bell System Technical Journal 27* (July, October 1948), 379–423,623–656.

[88] Shapiro, J. S., Smith, J. M., and Farber, D. J. EROS: a fast capability system. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles* (New York, NY, USA, 1999), ACM, pp. 170–185.

[89] Shapiro, J. S., Vanderburgh, J., Northup, E., and Chizmadia, D. Design of the EROS trusted window system. In *Proceedings of the 13th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2004), USENIX Association, pp. 12–12.

[90] Singh, K., Moshchuk, A., Wang, H. J., and Lee, W. On the incoherencies in web browser access control policies. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2010).

[91] Stone, P. Next generation clickjacking, April 2010. `http://www.contextis.co.uk/resources/white-papers/clickjacking/Context-Clickjacking_white_paper.pdf`.

[92] Swift, M. M., Annamalai, M., Bershad, B. N., and Levy, H. M. Recovering Device Drivers. In *Proceedings of the 2004 Symposium on Operating Systems Design and Implementation (OSDI)* (December 2004).

[93] Swift, M. M., Bershad, B. N., and Levy, H. M. Improving the reliability of commodity operating systems. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), ACM, pp. 207–222.

[94] Symantec Inc. Symantec internet security threat report april 2010. `http://www.symantec.com/business/theme.jsp?themeid=threatreport`.

[95] Symantec Inc. Symantec report on the underground economy, November 2008. `http://www.symantec.com/business/theme.jsp?themeid=threatreport`.

[96] Symantec Inc. Symantec global Internet security threat report: Trends for 2008, April 2009. `http://www.symantec.com/business/theme.jsp?themeid=threatreport`.

[97] Tan, L., Zhang, X., Ma, X., Xiong, W., and Zhou, Y. AutoISES: Automatically inferring security specifications and detecting violations. In *Proceedings of the 17th USENIX Security Symposium (USENIX Security '08)* (July-August 2008).

[98] Ter Louw, M., and Venkatakrishnan, V. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Proceedings IEEE Symposium on Security and Privacy* (May 2009), pp. 331–346.

[99] Twitter. Clickjacking blocked, February 2009. `http://blog.twitter.com/2009/02/clickjacking-blocked.html`.

[100] Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (February 2007).

[101] W3C. HTML 5. http://www.w3.org/TR/html5/.

[102] W3C. The iframe element. http://www.w3.org/TR/html5/the-iframe-element.html.

[103] Wang, H. J., Fan, X., Howell, J., and Jackson, C. Protection and communication abstractions for web browsers in MashupOS. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)* (October 2007).

[104] Wang, H. J., Grier, C., Moshchuk, A., King, S. T., Choudhury, P., and Venter, H. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the 2009 USENIX Security Symposium* (August 2009).

[105] Wang, Y.-M., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., and King, S. Automated Web Patrol with Strider HoneyMonkeys: Finding Web sites that exploit browser vulnerabilities. In *Proceedings of the 2006 Network and Distributed System Security Symposium (NDSS)* (February 2006).

[106] Wheeler, D. SLOCcount, 2009. http://www.dwheeler.com/sloccount/.

[107] Williams, D., Reynolds, P., Walsh, K., Sirer, E. G., and Schneider, F. B. Device driver safety through a reference validation mechanism. In *OSDI 08: Proceedings of the 8th symposium on operating systems design and implementation* (2008).

[108] Woodward, J. P. Security requirementes for systems high and compartemented mode workstations. Tech. rep., MITRE Corp., 1987. MTR 9992.

[109] Yang, J., Twohey, P., Engler, D., and Musuvathi, M. Using model checking to find serious file system errors. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)* (December 2004), pp. 273–288.

[110] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 79–93.

[111] Yip, A., Wang, X., Zeldovich, N., and Kaashoek, F. Improving application security with data flow assertions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 2009)* (October 2009).

[112] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D. Making information flow explicit in HiStar. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 263–278.

[113] Zeller, W., and Felten, E. W. Cross-site request forgeries: Exploitation and prevention. Tech. rep., Princeton University, October 2008. `http://www.freedom-to-tinker.com/sites/default/files/csrf.pdf`.

[114] Zhou, F., Condit, J., Anderson, Z., Bagrak, I., Ennals, R., Harren, M., Necula, G., and Brewer, E. Safedrive: safe and recoverable extensions using language-based techniques. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 45–60.