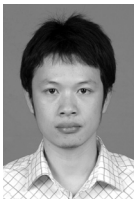
CHRIS GRIER, SHUO TANG, AND
SAMUEL T. KING

building a more secure Web browser

Chris Grier is a PhD student in the Electrical and Computer Engineering Department at the University of Illinois at Urbana-Champaign. He is interested in building secure software systems.

grier@uiuc.edu



Shuo Tang is a PhD student in the Computer Science Department at the University of Illinois at Urbana-Champaign. His research focuses on operating systems and system security.

stang6@uiuc.edu

Sam King is an Assistant Professor in the Computer Science Department at the University of Illinois at Urbana-Champaign. His primary research interests are in operating systems and security.

kingst@uiuc.edu

THE MODERN WEB BROWSER HAS evolved from a relatively simple client application designed to display static data into a complex networked operating system tasked with managing many facets of online experience. Support for dynamic content, multimedia data, and third-party plug-ins greatly enriches the browsing experience at the cost of increased complexity of the browser itself, resulting in a plague of security vulnerabilities that provide hackers with easy access to systems. To address the root of this problem, we designed and implemented the OP Web browser. We have partitioned the browser into smaller subsystems, isolated each subsystem, and made all communication between subsystems simple and explicit. Finally, we have used formal methods to prove the correctness of the communications between subsystems and the ability to limit the effects of compromised subsystems.

According to a recent report by Symantec [15], during the first half of 2007 Internet Explorer had 93 security vulnerabilities, Mozilla browsers had 74 vulnerabilities, Safari had 29 vulnerabilities, and Opera had 9 vulnerabilities. In addition to these browser bugs, there were also 301 reported vulnerabilities in browser plug-ins over the same period of time, including high-profile bugs in the Java virtual machine [3], the Adobe PDF reader [10], the Adobe Flash Player [2], and Apple's QuickTime [11]. Unfortunately, according to several recent reports [9, 12, 15, 16], attackers actively exploit these bugs.

The flawed design and architecture of current Web browsers make this trend of exploitation likely to continue. Modern Web browser design still has roots in the original model of browser usage in which users viewed static pages and the browser itself was the application. However, recent Web browsers have evolved into a platform for hosting Web-based applications, where each distinct page (or set of pages) represents a logically different application, such as an email client, a calendar program, an office application, a video client, or a news aggregate. The single-application model 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 within any part of the browser, including plug-ins, results in a total compromise of all Web-based applications running within the browser. This compromise may include all parts of the system that the user running the browser has access to, up to and including the operating system itself.

Efforts to provide security in this evolved model of Web browsing have had limited success. The same-origin policy, where the origin is defined as the domain, port, and protocol of a request, states that scripts and objects from one domain should only be able to access other scripts and objects from the same domain. This is the one security policy most browsers try to implement. However, modern browsers have different interpretations of the same-origin policy [6], and the implementation of this principle tends to be error-prone because of the complexity of modern browsers [4]. The same-origin policy is also too restrictive for use with browser plug-ins, and as a result browser plug-in writers have been forced to implement their own ad hoc security policies [1, 8, 14]. Plug-in security policies can contradict a browser's overall security policy and create a configuration nightmare for users, since they have to manage each plug-in's security settings independently.

Given the importance of Web browsers and the lack of security in current approaches, our goal is to design and implement a secure Web browser. More precisely, we want to prevent as many attacks as we can with reasonable cost, limit the damage that the remaining attacks can do, recover swiftly from successful attacks, and learn how to prevent them in the future.

This article describes the design and implementation of the OP Web browser, which attempts to address the shortcomings of current Web browsers to enable secure Web browsing. OP comes from Opus Palladianum, which is a technique used in mosaic construction where pieces are cut into irregular fitting shapes. In our design we break the browser into several distinct and isolated components, and we make all interactions between these components explicit. At the heart of our design is a browser kernel that manages each of our components and interposes on communications between them. This model provides a clean separation between the implementation of the browser components and the security of the browser, and it allows us to provide strong isolation guarantees and to implement novel security features.

Building the OP Browser

In our current design [5] we break the browser into several distinct and isolated components, and we make all interactions between these components explicit. At the heart of our design is a browser kernel that manages each of our components and interposes on communications between them. This model provides a clean separation between the implementation of the browser components and the security of the browser, and it allows us to provide strong isolation guarantees and to implement novel security features. This architecture stands in stark contrast to current browser designs, which place all components in a single process and contain multiple paths for making security-critical decisions [4], making it difficult to reason about security.

DESIGN PRINCIPLES

Overall we embrace both operating system design principles and formal methods techniques in our design. By drawing on the expertise from both

communities we hope to converge on a better and more secure design. Four key principles guide the design for our Web browser:

- Have 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 easier.
- Have strong isolation between distinct browser-level components and defense in depth. 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.
- Design components to do the proper thing, but monitor them to ensure they adhere to the design. Delegating some of the security logic to individual components makes the browser kernel simpler while still providing enough information to verify that the components faithfully execute their design.
- Maintain compatibility with current technologies. We do our best to avoid imposing additional burdens on users or Web application developers—our goal is to make the current browsing experience more secure.

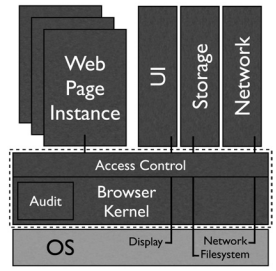
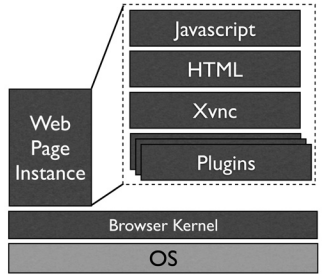


FIGURE 1: OVERALL ARCHITECTURE OF OUR OP WEB BROWSER

FIGURE 2: BREAKDOWN OF AN INDIVIDUAL WEB PAGE INSTANCE



OP BROWSER ARCHITECTURE

Figure 1 shows the overall architecture of OP. Our browser consists of five main subsystems: the Web page subsystem, a network component, a storage component, a user-interface (UI) component, and a browser kernel. Each of these subsystems runs within a separate OS-level process, and the Web page subsystem is broken into several different processes. The browser kernel manages the communication between the subsystems and between processes, and it also manages interactions with the underlying operating system.

We use a message-passing interface to support communications between all processes and subsystems. (See our recent paper [5] for a full listing of our message-passing interface.) These messages have a semantic meaning (e.g., fetch an HTML document) and are the sole means of communication

between different subsystems within our browser. They must pass through the browser kernel, and the browser kernel implements our access control mechanism, which can deny any messages that violate our access control policy.

We also use OS-level sandboxing techniques to limit the interactions of each subsystem with the underlying operating system. Each subsystem has a unique set of sandboxing rules specifically tailored to the individual component. For example, the Web page subsystem is denied access to the file system and the network, and the network subsystem is allowed to access the network, but not the file system. In our current design we use SELinux [7] to sandbox our subsystems, but other techniques would have been suitable for our purposes.

THE BROWSER KERNEL

The browser kernel is the base of our OP browser and it has three main responsibilities: manage subsystems, manage messages between subsystems, and maintain a detailed security audit log. To manage subsystems, the browser kernel is responsible for creating and deleting all processes and subsystems. The browser kernel creates most processes when the browser first launches, but it creates Web page instances on demand whenever a user visits a new Web page. Also, the browser kernel multiplexes existing Web page instances to allow the user to navigate to previous Web pages (e.g., the user presses the "back" button).

The browser kernel maintains a full audit log of all browser interactions. It records all messages between subsystems, which enables detailed forensic analysis of our browser if an attacker is able to compromise our system.

THE WEB PAGE SUBSYSTEM

When a user clicks on a link or is redirected to a new page, the browser kernel creates a new Web page instance. For each Web page instance we create a new set of processes to build the Web page. Each Web page instance consists of an HTML parsing and rendering engine, a JavaScript interpreter, plug-ins, and an X server for rendering all visual elements included within the page (Figure 2). The HTML engine represents the root HTML document for the Web page instance. The HTML engine delegates all JavaScript interpretation to the JavaScript component, which communicates back with the HTML engine to access any document object model (DOM) elements. We run each plug-in object in an OS-level process and plug-in objects also access DOM elements through the HTML engine. All visual elements are rendered in an Xvnc server, which streams the rendered content to the UI component where it is displayed.

THE USER INTERFACE, NETWORK, AND STORAGE SUBSYSTEMS

Our UI subsystem is designed to isolate content that comes from Web page instances. The UI is a Java application and implements most typical browser widgets, but it does not render any Web page content directly. Instead the Web page instance renders its own content and streams the rendered content to the UI component using the VNC protocol [13]. By using Java and having the Web page instance render its own content we enforce isolation and add an extra layer of indirection between the potentially malicious content from the network and the content being displayed on the screen. This isolation and indirection allow us to have stronger guarantees that poten-

tially malicious content will not affect the UI in unanticipated ways. The UI includes navigation buttons, an address bar, a status bar, menus, and normal window decorations.

The UI is the only component in our system that has unrestricted access to the underlying file system. Anytime the Web browser needs to store or retrieve a file, it is done 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.

Since other components cannot access the file system or the network, we provide components to handle these actions. 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. The network subsystem implements the HTTP protocol and downloads content on behalf of other components in the system.

Security in the OP Browser

We drew on the expertise of the operating systems community to make our browser architecture well suited for security. Subsystems within the browser are first-class principals, and communication between subsystems is explicit and exposed, thus providing mechanisms suitable for implementing browser-based security. Next, we explore two areas: security policies for browser extensibility and formal methods for proving invariants about our browser.

BROWSER EXTENSIBILITY

Modern Web browsers support extensibility through two main mechanisms: browser plug-ins and browser extensions. Plug-ins are a browser mechanism for hosting additional applications within a Web page, usually to render non-HTML content such as multimedia files. For example, browsers render "application/x-shockwave-flash" content using a flash-capable movie player such as Adobe Flash Player.

Extensions are a browser mechanism for extending browser functionality. Extensions interpose on and interact with browser-level events and data and provide developers with the ability to add user-interface widgets to the browser itself. Three popular extensions are the Yahoo! toolbar, which provides easy access to the Yahoo! search engine, the Greasemonkey extension, which allows users to script common tasks such as filling in form data automatically, and the NoScript extension, which provides fine-grained control over which pages can run JavaScript, thus preventing untrusted pages from running potentially malicious scripts.

These mechanisms for browser extensibility introduce unique challenges from a security perspective. Common uses of plug-ins often contradict a browser's overall security policy, so plug-ins operate outside of current browser security policies. Extensions need flexibility to integrate tightly within the browser itself and often run with full privileges. For a browser to be secure one must support these rich features securely without compromising the flexibility of commonly used plug-ins and extensions. Two current browsers that support extensions, Firefox and Internet Explorer, opt to pro-

vide flexibility at the expense of security and have no mechanisms or policies for running extensions securely.

As a first step toward our greater goal of securing browser extensibility, we have integrated plug-in security policies within the OP Web browser. In addition to supporting the ubiquitous same-origin policy, we developed two novel plug-in policies designed to provide security for the browser even if an attacker successfully exploits a plug-in vulnerability. You can learn about these policies by reading pages five and six of our paper [5].

FORMAL METHODS

Large software artifacts are typically built and maintained by groups of developers who contribute thousands of lines of code over a period of several years. This process is error-prone despite the best intentions of the developers. Moreover, for the software to be useful over an extended period of time, it must adapt to changes in user requirements. These extensions may be made by the original developers, but they are more typically made by a different group of people who may work for a different company. Ideally, formal verification that the code is correct with respect to the security requirements of the system would be an essential part of initial software development as well as the subsequent revision process. For many reasons, however, formal verification is usually not a central component in software development. We address this problem for the OP Web browser by making formal methods a fundamental part of our overall design process.

In our work to date on verifying security properties of the OP browser we use formal methods as a useful and practical tool in our overall design process. We develop an abstract model of the browser components and exhaustively search through the execution state space using a model-checking framework to look for states that violate our specified security invariants. We verified our implementation of the same-origin policy and we verified an "address-bar visual invariant" that states the URL displayed in the address bar should always be the same as the URL of the displayed page.

There is often a gap between the formal model used to verify properties and the system implementation. Although we recognize that this gap exists between our model and our system, we feel that for our uses of formal methods 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 create 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, we model each component, 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 the 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 not to send a particular message. Our model gives an attacker complete control over the compromised component, in-

cluding 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.

This preliminary work on formal verification of our browser represents a first step toward our larger goal of full formal verification of the OP Web browser.

Conclusions

In this new era of Web-based applications and software as a service, the Web browser has become the new operating system. Unfortunately, current Web browsers are unable to cope with the complexity that accompanies this new role and have fallen subject to attack. In this article we showed how, by treating Web browsers like operating systems and by building them using operating system principles, we can make a first step toward a more secure Web browser.

We plan to have a version of the OP browser ready for download by the end of the summer.

ACKNOWLEDGMENTS

We would like to thank Jose Meseguer and Ralf Sasse for their valuable feedback on our use of formal methods. We would also like to thank Joe Tucek and Anthony Cozzie for discussions about the design of our browser, and Frank Stratton, Paul Dabrowski, Adam Lee, and Marianne Winslett for feedback on an early draft of our paper. This research was funded in part by a grant from the Internet Services Research Center (ISRC) of Microsoft Research.

REFERENCES

[1] Adobe Flash Player settings manager: http://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager.html.

[2] Adobe, Flash Player update available to address security vulnerabilities: http://www.adobe.com/support/security/bulletins/apsb07-12.html.

[3] AusCERT, Sun Java runtime environment vulnerability allows remote compromise: http://www.auscert.org.au/render.html?it=7664.

[4] S. Chen, D. Ross, and Y.-M. Wang, "An Analysis of Browser Domain-Isolation Bugs and a Light-weight Transparent Defense Mechanism," *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS),* 2007.

[5] C. Grier, S. Tang, and S.T. King, "Secure Web Browsing with the OP Web Browser," *Proceedings of the 2008 IEEE Symposium on Security and Privacy,* May 2008.

[6] C. Jackson, A. Bortz, D. Boneh, and J.C. Mitchell, "Protecting Browser State from Web Privacy Attacks," P*roceedings of the 15th International Conference on World Wide Web* (New York: ACM Press, 2006).

[7] P. Loscocco and S. Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System," *Proceedings of the 2001 USENIX Annual Technical Conference FREENIX Track*, June 2001.

[8] Microsoft, "ActiveX Security: Improvements and Best Practices": http://msdn2.microsoft.com/en-us/library/bb250471.aspx.

[9] A. Moshchuk, T. Bragin, S.D. Gribble, and H.M. Levy, "A Crawler-based Study of Spyware on the Web," *Proceedings of the 2006 Network and Distributed System Security Symposium (NDSS),* February 2006.

[10] P.D. Petrkov, Oday: PDF pwns Windows: http://www.gnucitizen.org/blog/0day-pdf-pwns-windows.

[11] P.D. Petrkov, Oday: QuickTime pwns Firefox: http://www.gnucitizen.org/blog/0day-quicktime-pwns-firefox.

[12] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu, "The Ghost in the Browser Analysis of Web-based Malware," *Proceedings of the 2007 Workshop on Hot Topics in Understanding Botnets (HotBots),* April 2007.

[13] T. Richardson, Q. Stafford-Fraser, K.R. Wood, and A. Hopper, "Virtual Network Computing," *IEEE Internet Computing,* 2(1):33–38, January 1998.

[14] Sun, Java Security Architecture: http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc1.html.

[15] D. Turner, Symantec Internet Security Threat Report: Trends for January–June 07, Technical Report, Symantec Inc., 2007.

[16] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King, "Automated Web Patrol with Strider Honeymonkeys: Finding Web Sites That Exploit Browser Vulnerabilities," *Proceedings of the 2006 Network and Distributed System Security Symposium (NDSS),* February 2006.