2008 IEEE Symposium on Security and Privacy

Secure web browsing with the OP web browser

Chris Grier, Shuo Tang, and Samuel T. King
Department of Computer Science
University of Illinois at Urbana-Champaign
Email: {grier, stang6, kingst}@uiuc.edu

***Abstract*—Current web browsers are plagued with vulnerabilities, providing hackers with easy access to computer systems via browser-based attacks. Browser security efforts that retrofit existing browsers have had limited success because the design of modern browsers is fundamentally flawed. To enable more secure web browsing, we design and implement a new browser, called the OP web browser, that attempts to improve the state-of-the-art in browser security. Our overall design approach is to combine operating system design principles with formal methods to design a more secure web browser by drawing on the expertise of both communities. Our overall design philosophy is to 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 our new browser security features.**

To show the utility of our browser architecture, we design and implement three novel security features. First, we develop novel and flexible security policies that allows us to include plugins within our security framework. Our policy removes the burden of security from plugin writers, and gives plugins the flexibility to use innovative network architectures to deliver content while still maintaining the confidentiality and integrity of our browser, even if attackers compromise the plugin. Second, we use formal methods to prove that the address bar displayed within our browser user interface always shows the correct address for the current web page. Third, we design and implement a browser-level information-flow tracking system to enable post-mortem analysis of browser-based attacks. If an attacker is able to compromise our browser, we highlight the subset of total activity that is causally related to the attack, thus allowing users and system administrators to determine easily which web site lead to the compromise and to assess the damage of a successful attack.

To evaluate our design, we implemented OP and tested both performance and filesystem impact. To test performance, we measure latency to verify OP's performance penalty from security features are be minimal from a users perspective. Our experiments show that on average the speed of the OP browser is comparable to Firefox and the audit log occupies around 80KB per page on average.

I. Introduction

Current web browsers provide attackers with easy access to modern computer systems. According to a recent report by Symantec [48], over the last year 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 plugins over the same period of time including high-profile bugs in the Java virtual machine [10], the Adobe PDF reader [38], the Adobe flash player [8], and Apple's QuickTime [39]. Unfortunately, attackers actively exploit these bugs according to several recent reports [50], [36], [41], [48].

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 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-based applications, 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 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-based applications running within the browser.

Efforts to provide security in this evolved model of web browsing have had limited success. The *same origin*[1] *policy* – which states that scripts and objects from one domain should only be able to access other scripts and objects from the same domain – is one security policy most browsers try to implement. However, different browsers have varying interpretations of the same-origin policy [24], and the implementation of this principle tends to be error prone due to the complexity of modern browsers [12]. Furthermore, the same-origin policy is too restrictive for use with browser plugins and as a result browser plugin writers have been forced to implement their own ad-hoc security policies [7], [47], [35]. Plugin security policies can contradict a browsers overall security policy, and create a configuration nightmare for users since they have to manage each plugin's security settings independently.

Current research efforts to retrofit today's web browsers help to improve security, but fail to address the fundamental design flaws of current web browsers. One project, *MashupOS* [49], proposes new abstractions to facilitate improved sharing among multiple principles hosted in the same web page. Another project, *Script Accenting* [12], encrypts scripts from different domains to improve enforcement of the same-origin policy. Both provide scripts with fine-grain isolation within the same web page. However, these mechanisms both run within current web browsers (Internet Explorer) and are only as secure as the browser they run within, which currently

[1]An origin is defined as the domain, port, and protocol of a request.

is not very secure. Sandboxing systems, such as Tahoma [17], prevent browsers from making persistent changes to the system and isolate distinct web-based applications using a virtual machine monitor (VMM). This type of persistent-state restriction can be problematic if users legitimately want to store persistent state and allowing users to store and execute downloaded files gives attackers an avenue into the system. Plus, sandboxing at the web-application level can be too coarse grained since it fails to isolate different scripts and objects within the same web application. Combining current fine-grained isolation techniques with sandboxing systems does not provide a complete solution since it would still rely heavily on the underlying browser itself.

This paper describes the design and implementation of the OP[2] web browser that attempts to address the shortcomings of current web browsers to enable secure web browsing. 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.

To show the utility of our browser architecture, we design and implement three novel security features. First, we develop a novel and flexible security policies that allows us to include plugins within our security framework. Our policy removes the burden of security from plugin writers, and gives plugins the flexibility to use innovative network architectures to deliver content while still maintaining the confidentiality and integrity of our browser, even if attackers compromise the plugin. Second, we use formal methods to prove that the address bar displayed within our browser user interface always shows the correct address for the current web page. Third, we design and implement a browser-level information-flow tracking system to enable post-mortem analysis of browser-based attacks. If an attacker is able to compromise our browser, we highlight the subset of total activity that is causally related to the attack, thus allowing users and system administrators to determine easily which web site lead to the compromise and to assess the damage of a successful attack.

To the best of our knowledge, the contributions of this paper are as follows:

- We present the design and implementation of a new browser architecture that facilitates the development of novel and flexible browser-level security policies.
- We are the first to enforce plugin security policies explicitly from the browser, and we are the first to cope with compromised plugins while still maintaining a high-level of overall browser security.
- We show how operating system principles can be combined with formal methods as a practical methodology for browser design and implementation.
- We are the first to develop techniques for performing post-mortem analysis of browser-based attacks.

[2]OP comes from Opus Palladianum, which is one technique used in mosaic construction where pieces are cut into irregular fitting shapes.

II. The OP browser design and implementation

This paper describes the design and implementation of the OP web browser that improves the security of web browsing; we have three primary goals. First, we should prevent browser-based attacks from happening. Next, although we hope to prevent many attacks, inevitably our browser will contain vulnerabilities so we should contain these attacks and limit the damage that can be done by a successful compromise. Finally, even if we prevent some attacks and contain others, attackers may be able to cause damage to infected systems, so we should provide the ability to recover from successful attacks.

In this section we describe the design of our OP web browser that attempts to achieve these goals. First we discuss our threat model and the principles that guide our design, then we discuss our overall architecture, and finally we describe the individual components that make up our browser. In Sections III, IV, and V we describe in detail the specific security features we implement within our browser that illustrate our ability to achieve our overall security goals.

A. Threat model and assumptions

We designed the OP web browser to operate under malicious influence. We consider attacks that originate from a web page and could potentially target any part of the browser. We assume that the attacker could have complete control over the content being served to the web browser. A browser compromise could be any sort of attack provided in this way; an attack that results in code execution is the most capable form of attack.

We trust the layers upon which OP is built. Namely, we trust the underlying operating system and Java virtual machine (JVM) to enforce isolation for our subsystems. Like 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.

B. 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:

1) *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.
2) *Strong isolation between distinct browser-level components and defense-in-depth.* Providing isolation between browser-level components reduces the likelihood

Subsystem	File system access	Network access
UI	allowed	denied
Web page	denied	denied
Storage	limited	denied
Network	denied	allowed
Browser kernel	allowed	allowed

TABLE I
SUMMARY OF OS-LEVEL SANDBOXING FOR EACH OP SUBSYSTEM.

of unanticipated and unaudited interactions, and allows us to make stronger claims about general security and the specific policies we implement.

3) *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.
4) *Maintain compatibility with current technologies.* We try to avoid imposing additional burdens on users or web application developers, our goal is to make the current browsing experience more secure.

C. OP browser architecture

Figure 1a 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 run within separate OS-level processes, and the web page subsystem is broken into several different processes. The browser kernel manages the communication between each subsystem and between processes, and the browser kernel manages interactions with the underlying operating system.

We use a message passing interface to support communications between all processes and subsystems (see the Appendix for a listing of all messages). 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 that can deny any messages that violate our access control policy. We discuss our access control policy in detail in Section III.

We use OS-level sandboxing techniques to limit the interactions of each subsystem with the underlying operating system. In our current design we use SELinux [33] to sandbox our subsystems, but other techniques like AppArmor [21], Systrace [40], or Janus [20], would have been suitable for our purposes. Table I summarizes the limitations put on each of our subsystems.

D. 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).

All messages between subsystems and processes pass through the browser kernel. The browser kernel implements message passing using OS-level pipes, and it maintains a mapping between subsystems and pipes. This mapping allows the browser kernel to avoid source subsystem spoofing since the browser kernel can accurately identify the subsystem connected to a pipe when it receives a message.

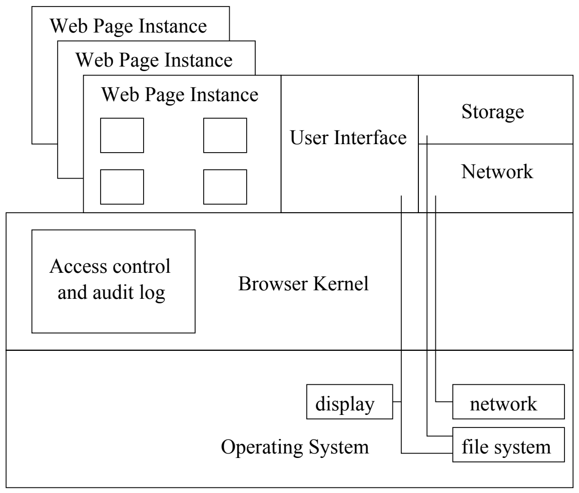
To simplify our implementation, the browser kernel is a single threaded, event-driven component and all messages have a unique message ID and a global order. This global order helps make reasoning about security properties easier and reduces many possible race conditions.

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

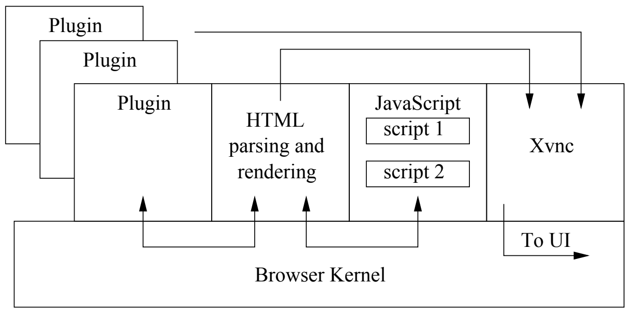
E. The web page subsystem

Each web page instance represents an individual web page. 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, plugins, and an X server for rendering all visual elements included within the page (Figure 1b). 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 plugin object in an OS-level process and plugin 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.

One design decision we make is to use an existing HTML parsing and rendering engine instead of building our own. In our first design iteration we built our own HTML parsing and rendering engine based on classes provided in Sun's Java runtime. The advantage of this approach is that we could use the type safety properties of Java to provide stronger isolation between individual items within HTML documents (e.g., DOM nodes). However, we found it was difficult to render correctly even simple web pages because of buggy HTML handling and cascading style sheets; thus, we decided to use an existing HTML engine instead.



(a) Overall architecture of our OP web browser



(b) Overall architecture of a web page instance

Fig. 1. Overall architecture of our OP 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. Within an individual web page instance (b), each subsystem runs in a separate OS-level process, and each plugin instance runs within a separate OS-level process. All of the processes communicate through the browser kernel, except for the HTML rendering engine and the plugins which communicate directly with a Xvnc server. The Xvnc server renders the elements locally and streams them to the UI component, via the browser kernel, where they are displayed for the user.

In our current design we use the KHTML HTML parsing and rendering engine [29]. It has the advantage of rendering today's web pages beautifully, but the disadvantage of being implemented in an unsafe programming language (C++). Relying on an unsafe programming language for our HTML engine is problematic because we rely on the HTML engine to tag JavaScript code and browser plugins with the proper source domain. We use domains in our security policies to isolate different scripts and objects on the same web page; the HTML engine sets the domain and the JavaScript component and the browser kernel enforce isolation between different entities. To lessen the impact of this shortcoming we still allow KHTML to mark the source domain for JavaScript code and browser plugins, but we check them using our Java-based HTML parser. However, our HTML parser handles today's HTML poorly, so this check produces a silent warning in our audit log rather than halting the web page instance or notifying the user when OP detects a violation.

Determining how to include a JavaScript interpreter and plugins was a relatively easy design decision. For JavaScript we use the Rhino JavaScript interpreter [37]. Rhino is a high-quality JavaScript interpreter written in Java, which gives us strong isolation between different JavaScript instantiations. Unlike the other components of the web page instance we use a single OS-level process to handle all JavaScript interpretations. We justify this decision since we rely on the Java Virtual Machine (JVM) to provide the necessary level of isolation between script objects. For browser plugins we use existing plugins written in unsafe languages since there are too many plugins for us to re-write them. Plugins already have well-defined interactions with the rest of the browser so we break each plugin instance into a separate OS-level process to provide the necessary level of isolation.

F. The user interface, network, and storage subsystems

Our user-interface (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 [44]. 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 allows us to have stronger guarantees that potentially 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. Any time 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.

III. Security policy and enforcement

The OP browser is able to enforce different security policies through flexible access control; we implement three different security policies to explore the access controls that OP offers. In addition to implementing the ubiquitous same origin policy, we develop two novel policies designed to provide additional flexibility to plugins, while still providing the required security for the rest of the browser.

In this section we discuss the OP security policies. First we discuss browser plugins and some of the problems with current approaches, then the policies we implement with a focus on how each policy improves the security of browser plugins. In Section IV we describe how we use formal methods to show that our access controls are correct and maintain the required security policy through a compromised browser component.

A. Browser plugins

Browser plugins provide web browsers with the ability to view additional types of content. Most web browsers have at least one plugin installed (Adobe reports their Flash Player has been installed on 99% of Internet users desktops [6]) and the browser identifies which plugin to use for a particular type of content by the corresponding MIME type. For example, the MIME type “application/x-shockwave-flash” is handled by a flash capable movie player such as Adobe Flash Player [4]. Other popular plugins include Windows Media Player, Adobe Acrobat, Quicktime, RealPlayer and Java. Plugins provide a variety of functionality from playing music to just-in-time compilation of programming languages.

Web developers include plugins within in a web page by using the OBJECT or EMBED HTML tags. Figure 2 presents sample HTML used to include plugin content in a web page and shows the specification of MIME types for the included content. The first plugin referenced in Figure 2 includes a flash movie from YouTube. The flash movie is executed by the flash plugin and has the capability to play different video content and interact with the user. The second plugin embeds an instance of a Quicktime capable player to download and play a video. We refer to the code that is responsible for viewing a particular MIME type as the *plugin* and the content being downloaded and viewed as the *plugin content*.

Plugins complicate browser security because they are given unchecked access to browser internals, making it difficult for the browser to enforce security policies on plugins. Plugins are supplied to the browser in a binary format and usually loaded as a dynamically loaded library. Though plugins are provided with an API to interact with the browser [2], plugins run in the same address space as the browser, so they are free to modify browser structures as needed. Thus, a successful attack on a single plugin leads to a full browser compromise.

Currently plugin providers implement their own ad-hoc security mechanisms and policies for each different plugin, which causes security problems even for uncompromised plugins. Security policy goals for the browser are not necessarily reflected by the plugin security policy resulting in inconsistent accesses between the browser and and the plugin. Also, there can be differences in plugin policy between plugin implementations for the same content type. For example, different Flash players could allow different cross domain accesses based on their developers interpretation of Flash security policy.

Another aspect of per-plugin security policy is the complicated configuration presented to the user. For example, the Adobe Flash Player provides two different security mechanisms that require configuration. The first is the plugin’s local security settings accessed through an in-browser menu [7]. The second is a server side XML manifest governing cross domain accesses [5]. Since there are a large number of plugins available for modern browsers, requiring the user to configure each one separately is unlikely to be effective.

Providing a common security policy and policy decision point between plugins and the whole browser is important to address the security needs of plugins in modern web browsers.

B. Plugin security in OP

To address the shortcomings of current plugins, we design and implement a plugin architecture to provide security for plugins in the OP browser. The OP browser enforces security policy in the browser kernel. Consistent with all other policy decisions, any plugin related access control is done by the same security mechanisms enforcing policy for the rest of the browser.

To enforce security policy we interpose on message passing in the browser kernel. Each browser process is labeled with a security context (i.e. domain) depending on the security policy being used. We run each instance of a plugin in a separate process that is assigned its own label by the kernel. In order to correctly label each plugin process the browser kernel inspects messages that trigger the plugin to load content from a URL. This security label is then used to make decisions for other plugin and browser actions. The plugin can be denied access to browser resources; similarly, the rest of the browser can be denied access to plugin resources. Each pairwise communication channel between browser subsystems can have an access control module operate on the messages. Any security related state is maintained inside of the corresponding access control module. Our implementation provides a simple API for implementing different security policies.

C. Plugin security policies

In addition to developing a plugin architecture, we develop two novel security policies that specifically address the needs of plugin security while still providing enough flexibility to support common plugin usage.

1) Provider domain policy: Provider domain policy allows a plugin embedded in a page permissions associated with the source of the plugin content. Media sharing from sites

```
<object width="425" height="355">
  <param name="movie" value="http://www.youtube.com/v/oNsCaVC4Z0o&rel=1"/>
  <param name="wmode" value="transparent"/>

  <embed src="http://www.youtube.com/v/oNsCaVC4Z0o&rel=1"
  type="application/x-shockwave-flash" wmode="transparent" width="425" height="355"/>
</object>

<object
src="http://movies.apple.com/movies/paramount/iron_man/iron_man-tlr1_h.640.mov"
type="video/quicktime" width=640 height=288 autoplay="true"/>
```

Fig. 2. This HTML is an page excerpt downloaded from the www.uiuc.edu domain. OBJECT tags are handled by most web clients while EMBED tags are interpreted by some Mozilla browsers. The URLs that each plugin loads are specified by the SRC attribute. Other parameters for the execution of the plugin can be specified either with PARAM tags or attributes. The first set of tags references Flash content hosted at www.youtube.com while the second is a movie hosted at movies.apple.com. The HTML also demonstrates how a page can include content from multiple different sources by specifying a URL in the SRC.

like YouTube allow one site to host the video content and other content publishers to embed the video into their sites or blogs. Advertisements are provided by an advertising company and similarly embedded along with web page content. In both cases the web page creator has little control over the content inside an embedded area, especially if it includes plugin content. Our policy is designed to reflect the intent that a web page creator has when embedding videos and content across domain boundaries.

The provider domain policy sets the origin of the plugin to the site hosting the plugin content. When a page uses the OBJECT tag to include plugin content, the plugin is given permissions according to the domain of plugin content provider. If same-origin policy were applied instead, the browser would associate the plugin content with the domain of the page containing the OBJECT tag. The same HTML from Figure 2 can help to illustrate the difference. Same-origin policy would treat both plugins as if they came from the www.uiuc.edu domain, since that is where the web page is hosted containing the HTML. Our provider domain policy labels the two plugins differently. The first movie would be tagged with the domain www.youtube.com, and the second with movies.apple.com. The page hosting the content is given permissions according to the www.uiuc.edu domain. Label differences force separation between the content and prohibits the embedded content from altering the page or fetching any resources associated with the www.uiuc.edu domain. Each of the plugins embedded inside the page can access data associated with the corresponding domains. For example, the YouTube video can access cookies, make network connections and use other resources from www.youtube.com. This example shows how popular use of plugins can be met inside the constraints of browser security policy.

The provider domain policy has important security implications. As content is shared between sites and included in blogs and websites, the authors of the pages do not need to be concerned with security when including cross domain content. Included content is isolated and users viewing websites including cross domain plugin content are safe from malicious and vulnerable plugins. This policy limits plugins included across domains and prohibits plugin content included in this way from accessing cookies, DOM elements and other browser components.

2) *Plugin freedom policy:* The plugin freedom policy provides additional flexibility to plugins by allowing additional outgoing network accesses while limiting access to page and user information contained in the browser. This policy is motivated by plugins such as Sopcast [3], a peer-to-peer video player, that needs additional flexibility for outgoing network connections.

The plugin freedom policy provides local plugin storage and unlimited network access at the cost of access to DOM elements and other browser components. To implement this policy, using the OP browser access controls we simply prohibit communication between the plugin and any browser components, except the network and storage subsystem. The storage subsystem provides a location for per-plugin storage that is allowed by this policy. Per-plugin storage allows plugins to have access to local settings and save files in a safe environment. All network communications are also allowed. Any other communications are prohibited in and out of the plugin. The rest of the browser subsystems are able to interact as normal and provided with standard same-origin protections.

The plugin freedom policy prevents some plugin content from functioning properly, though media sites such as YouTube, Apple Movie Trailers and others continue to function. The plugin content on these pages does not need to access any of the other browser components besides network resources. This policy is similar to current plugin operation in browsers with the scriptable API components removed. Removing interactions with other browser components prevents plugins from leaking client information across multiple sites if the plugin is exploited or the plugin content is malicious. Any plugins that interact with the content on pages will function incorrectly since any attempted Javascript execution will be prevented by the access controls.

IV. Formal Verification

We designed he OP web browser to include the use of formal methods to verify its correctness. To do 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 [15]. The reasoning engine we use is the Maude system [34]. 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 paper, 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. First we give a very brief overview of Maude, then we discuss how we created our model. Finally, we describe how we model check to prove the absence of address bar spoofing attacks and to verify parts of our same origin policy implementation.

A. Modeling using Maude

A simple example using Maude and model checking of invariants is presented in the *Maude Manual* [16]. 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 3. This example illustrates a number of Maude features though we only describe the ones relevant to the OP browser model we present later.

```
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
```

Fig. 3. A simple 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 .
```

Fig. 4. The search statement from the Maude Manual (Version 2.3) showing how to model check the SIMPLE-CLOCK model invariant using Maude's search functionality.

Figure 3 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 sorts) to a different set of sorts. In the SIMPLE-CLOCK model we connect the sort Int to the sort Clock. Operations do not define how Maude connects the two types, instead specifies the connection. Rewrite laws begin with `rl` and describe transitions between states. The SIMPLE-CLOCK model has one rewrite law. This 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 function 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 which violate an invariant, such as the clock's state being outside of the 0 to 24 range. Figure 4 shows the example search statement for the SIMPLE-CLOCK model. This search statement defines an initial state, 0, and the condition to match when searching.

The Maude model for the OP browser consists mostly of definitions of types and state for each component by defining sorts, operations and 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.

B. 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 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*

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

Fig. 5. 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 parenthesis and creates an object of a specific type. The imsg constructor creates an object of type Message.

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.

C. 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. We discuss our model for message passing and processing, user actions, and how 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 in Figure 5. 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 is in Figure 6. This rule is responsible for updating the browser state including 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 shows an action that sets the location bar in the UI. The first three lines of Figure 6 are the current browser state and include the creation of 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 simply locates states which are possible to have as a result of these rules and satisfy an additional expression.

1) Modeling user actions: We also need to model the user actions in the browser system, such as clicking the "GO" button to request a new web page. The Maude model 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 7. 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. The first two lines of Figure 7 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 generated by the `imsg` constructor.

2) 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, and 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.

D. Model checking address bar invariants

Determining cases that allow the address bar in the browser to mismatch the page content was examined for Internet Explorer in recent work by Chen *et al.* [11]. They search 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

```
< 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 >
```

Fig. 6. This is the Maude rule corresponding to the state change due to a SET-LOCATION-BAR message being received. Notation here is similar to that of Figure 5, the first 3 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.

```
< 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) >
```

Fig. 7. Maude expression for the "GO" UI button causing 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.

```
< 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)
```

Fig. 8. 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.

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 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 8. 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 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 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.

E. Model checking same origin policy

Our implementation of 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, such as Javascript, 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 the scriptable plugin extension to the Netscape Plugin API, and we support such interaction in OP. Enforcing 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 prove a few different invariants. For example we prove that a plugin from one domain can not send a message to a plugin (or web page subsystem) 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 same origin policy is upheld with the added complexity.

V. Analyzing browser-based attacks

Although we put significant effort into securing our OP web browser, attacks may still occur. One class of attacks that may occur are "social engineering" attacks where a user is fooled into performing an action that violates the security of the system. 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 [41]. A second class of attacks that may occur are web-based application bugs, such as cross-site request forgery attacks [27]. In these attacks a malicious web site can coax the browser into performing the actions of a site, such as transferring money using a banking application, without the users knowledge or consent. The problem with these classes of attacks is that they adhere to the browsers security policy and from the browsers perspective appear to be legitimate actions, making these types of attacks difficult to prevent.

Our goal is to allow users and system administrator to recreate the past to analyze browser-based attacks. To analyze an attack, users and administrators may want to perform two types of analysis. First, they may want to determine which web site initiated the attack so they can blacklist the web site and avoid visiting it again in the future. Second, they may want to track the effects of known-malicious web sites that they visited to determine if they were attacked, or to assess the damage of a successful attack.

One difficulty in analyzing browser-based attacks is that the activities of the attacker are intermingled with legitimate actions. Even if users and system administrators have a complete security audit log that can recreate arbitrary past states and events, most of the content in the audit log is from legitimate web usage. Thus, it is difficult to highlight the subset of browsing activity that is most likely to be part of the attack.

We designed our OP web browser to overcome these shortcomings to enable users and system administrators to better understand browser-based attacks. To highlight the activities of an attacker, we use browser-level dependency graphs to help visualize the attack. Browser-level dependency graphs are graphs of browser-level objects connected by causal events within our browser. A causal event is defined as any event where information flows from one object to another, thus forming a dependency from the source object to the sink object. For example, if a user clicks on a link, this forms a causal link from the web page instance that hosts the link to the new web page instance that the browser kernel creates in response. Using these connections, we generate dependency graphs of attacks to show where an attack came from and to show what effects an attack had on our system.

To give a more concrete example of a dependency graph, Figure 9a shows a graph for a successful browser-based attack from a real web site. For this graph we assume that we (as the user) know the web site *videozfree.com* is malicious (as was pointed out in a recent paper from Google [41])we want to check if we downloaded files from the site into our file system anytime in the past. Our analysis starts with the *videozfree.com* web page. From *videozfree.com* we clicked on a still image of a video and were sent to *clipsforadults.com*, which displayed an image that appeared to be a video plugin. When we clicked on the "play" button, it automatically prompted us to download a new codec to view the video, which we downloaded as the file *setup.exe*. For this experiment the attack was in the middle of several weeks worth of typical browsing, and our audit log contained 349,313 events and 1218 different web page instances, yet we were able to automatically extract this much smaller subset of the total information available. Also, even though *videozfree.com* was listed as the malicious web site, the actual download came from a different site (*zsvcompany.com*), and based on monitoring *videozfree.com* for several months this download site changes periodically making it hard to track using blacklists.

In this section we discuss our techniques for analyzing browser-based attacks. First we describe the objects we track, the dependency forming events that connect objects, and the dependency graphs we generate to facilitate analysis. Then, we describe an example that illustrates how analyze a cross-site request forgery attack.

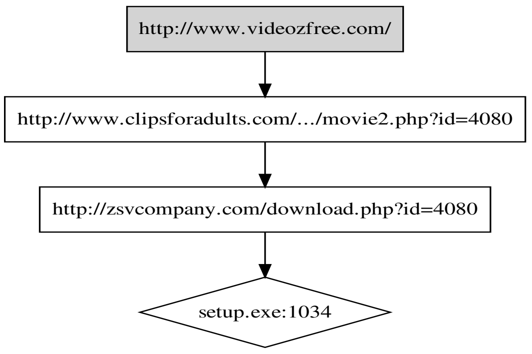
A. Intrusion analysis design

To track attacks using browser-level dependencies we need to define the objects we monitor and the events that connect these objects. When defining objects and events we have three main design considerations. First, we must decide at what level of granularity we should display our dependency graph. More coarse-grained dependency graphs will be smaller and easier to analyze manually, but may lack the fidelity to provide useful information about the attack. Second, we must decide at what level of granularity to track dependencies. In general more coarse-grained dependency tracking will be more efficient to implement, but could lead to false dependencies due to excessive tainting. Third, we must decide which events give the attacker the most direct control over the system and focus our analysis on these events. For example, we do not track the event where a web page instance sets the status bar in the UI. Certainly an attacker could use this as part of an attack, but is it likely to assist the attacker in fooling the user to do something else – such as visit a malicious web site – that we do track.

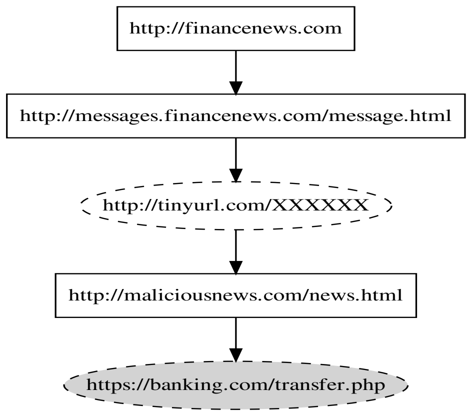
To strike a balance between analysis fidelity and the amount of information displayed to the user, we track two primary objects: web pages and files. Web page objects map directly to the web page instance subsystem within our browser. Web page objects are identified by the URL used to open the web page and we consider different web page instances with the same base URL to be different objects. Web page objects consist of HTML documents, images, JavaScript, plugins, etc. and even though we do record the interactions between these entities we chose to display them all as a single object to reduce the number of visual elements in our dependency graphs. File objects are file system objects, and we track these as they flow through our browser.

We track events that connect web pages together, and web pages with files. Whenever a web page creates a new web page (e.g., the user clicks on a link), the new web page depends on the web page that initiated the action. When a user downloads or uploads a file, we connect the file with the associated web page.

Although we designed our browser to make dependency-causing events explicit, we still have to put in effort to


http://www.videozfree.com/
http://www.clipsforadults.com/.../movie2.php?id=4080
http://zsvcompany.com/download.php?id=4080
setup.exe:1034

(a) Forward dependency graph for *videozfree* attack


http://financenews.com
http://messages.financenews.com/message.html
http://tinyurl.com/XXXXXX
http://maliciousnews.com/news.html
https://banking.com/transfer.php

(b) Backward dependency graph for *xsrf* attack

Fig. 9. In our dependency graphs the squares are web pages, diamonds are files, ovals are URL requests, and the detection point is shaded. We show URL requests only when they are the detection point or when the server automatically redirects our request resulting in a web page instance with a different URL than the requested URL.

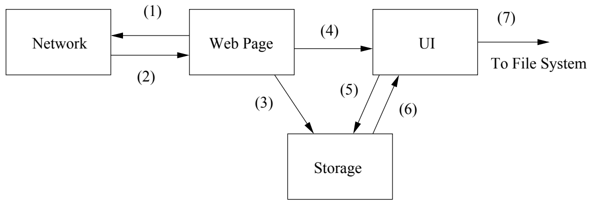


Fig. 10. This figure shows how our browser subsystems interact to download a file to the file system. First, the user clicks on a link which causes (1) the web page to request the file and (2) the network subsystem returns the downloaded content to the web page. Then (3) the web page stores the file in the storage manager and (4) notifies the UI that there is a downloaded file waiting. The UI then prompts the user and (5) retrieves the file from the storage manager (6). Finally, (7) the UI saves the file in the file system.

achieve the fine-grained dependency tracking to support the events we follow. For example, when a user downloads a file and stores it in the file system the downloaded file travels through many different subsystems (Figure 10), but we want to make the connection directly between the web page and the file without including the intermediate subsystems. When a user downloads a file they usually initiate this action by clicking on a link in a web page. The web page then makes a network request and then stores the file as a persistent object in our storage subsystem. Then, the web page notifies the UI that a downloaded file is waiting the storage subsystem, and the UI retrieves the file from the storage subsystem and saves it in the file system. We designed the storage and UI subsystems so that this data will pass through them without being modified, but an attacker that compromises one of these components could violate this assumption and result in dependencies that we miss since the attacker was able to affect data without using an explicit message. To prevent this type of unaudited dependency we monitor the storage and UI subsystems to verify that all objects and files that pass through them are unmodified. This simple invariant allows us to track these objects at a fine granularity without monitoring the internals of each subsystem. An attacker can still leak information using covert channels [32], but this would require the attacker to compromise two components, not just one. In our current implementation we do not perform this check for the network subsystem, but adding the additional invariant to verify network object integrity would be straightforward.

We apply the BackTracker [30] graph generation algorithm to create dependency graphs. The BackTracker graph generation algorithm starts with a single object, called a *detection point* and traverses the audit log to find the set of objects that are causally connected to the detection point. We use the backward graph generation algorithm [30] to find the origin of an attack (e.g., malicious web site) and the forward graph generation algorithm [31] to track the effects of an attack.

B. Example: cross-site request forgery

In this section we discuss an artificial cross-site request forgery attack based on a real attack described by Stamos and Lackey in their Black Hat presentation [46]. We show how our analysis techniques can help users understand this type of attack. Our example starts with a victim receiving a monthly banking statement and noticing a spurious transfer of $5000 and our goal is to figure out how this transfer occurred.

The starting point for our analysis is the specific HTTP request that resulted in the transfer of funds. We assume the victim can identify the specific network request that lead to the transfer (perhaps with the help of the bank). Starting with this request, we work our way backward to figure out what went wrong.

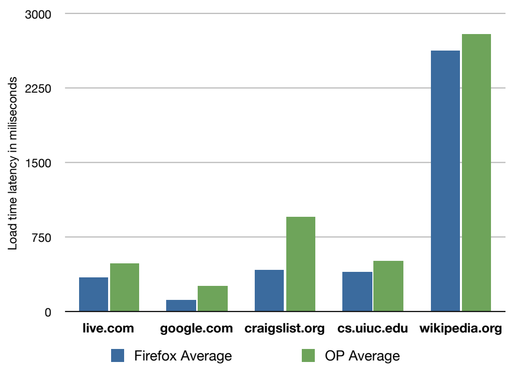
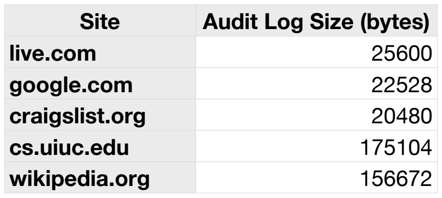


Fig. 11. Loading latencies for OP and Firefox.



Site	Audit Log Size (bytes)
live.com	25600
google.com	22528
craigslist.org	20480
cs.uiuc.edu	175104
wikipedia.org	156672

Fig. 12. Audit log size generated for a single visit.

Figure 9b shows the dependency graph for this attack. The request in question originated from the *maliciousnews.com* site. The victim arrived at the *maliciousnews.com* site by visiting a financial new site, *financenews.com* and going to message boards (*messages.financenews.com*) to look for "leaked news" about stocks of interest. The victim found a story they were interested in and clicked on a *tinyurl.com* link that redirected them to *maliciousnews.com.* The news story on the *maliciousnews.com* site contained an invisible inline HTML frame (iframe) that issued the request to transfer funds. This action was possible because of a design flaw in a banking application that made available to the iframe a login cookie that was created when the user logged into the banking application in a previous browsing session. This attack was *not* the result of a browser bug, the browser correctly enforced the same origin policy in this example. This type of attack is commonly referred to as a cross-site request forgery (xsrf) attack.

One omission from this dependency graph is the cookie connecting the *maliciousnews.com* site with the banking application. We filter out cookies because many web sites use cookies to track users across multiple unrelated sites, and including cookies results in large dependency graphs. We could have created a list of well-known tracking cookies and removed only these tracking cookies from our graphs, but we did not explore this alternative technique for this paper.

VI. Evaluation

In this section we evaluate the performance of OP and we present a qualitative security analysis of our browser.

A. Performance evaluation

To evaluate the OP web browser performance we measure page load times and to determine the file system impact from the extensive logging we examine the size of the audit logs for single page loads. Our goal when performing these evaluations is only to verify that the browser does not introduce unreasonable delays that are noticeable by the user. We also aim to check that the logs for a running browser are reasonable in size. All experiments were carried out on a 2.66GHz Intel Core 2 Duo with 2GB of memory and a 250GB serial ATA hard drive. The OS is Fedora Core 7, running the 64bit version of the Linux kernel 2.6.22. We use Firefox 2.0.0.12 as a base for performance comparisons.

To measure the latency introduced by our browser we compare the load times of a few common pages with those of Firefox. Figure 11 shows a list of the sites tested and the loading latency times. Each page is loaded 5 times, and the loading times are averaged. To measure the latency in Firefox we use an extension that monitors internal Firefox events to determine page load times. We monitor similar events inside of the OP browsers web page subsystem. Caching is disabled in Firefox for all tests. The results in Figure 11 indicate that we have not introduced latency that would be detrimental to a user using OP. The primary slowdown over Firefox is due to our implementation of Javascript.

We also examine the footprint of the audit log that is generated for each site. The audit log is stored as a single file consisting of all the events recorded during a page load inside of OP. After each test the audit log is cleared and the browser restarted to provide a clean start. As can be seen in Figure 12 the amount of space needed to record events is small for a single page. The size is largely dependent on the size of the pages being downloaded rather than data introduced for audit purposes. To quantify the likely storage needed for typical browsing, one of the authors used OP for typical browsing needs over several weeks. Over this period of time they created over 1000 web page instances and accumulated an audit log of 206MB, which is reasonable considering the low cost of storage.

B. Security analysis

We designed OP to prevent browser-based attacks, but compromises are still possible. We make heavy use of type safe programming languages to reduce the likelihood of memory corruption attacks and we use formal methods to help us reason about the security policies we implement. However, our implementation could contain exploitable vulnerabilities; in this section we discuss the impact of successful attacks on OP subsystems. In our analysis we consider a successful compromise of a single subsystem.

Successful attacks on the web page subsystem will have limited impact on the overall security of our browser. A compromised web page subsystem can only send messages to other subsystems, and has limited interactions with the local system. If the attack results in malicious messages being sent inside the browser, the browser kernel enforces security policy and forces the compromised subsystem to comply with local security policy. As a result, a successful attack on a web page subsystem only affects the single compromised web page instance, the browser kernel protects other web page instances being hosted by the browser.

Successful attacks on our UI, storage, and network subsystems are more severe than successful attacks on the web page subsystem. A compromised UI can tweak the user interface and access the file system, a compromised storage subsystem provides attackers with unchecked access to persistent browser storage (e.g., cookies), and a compromised network subsystem gives attackers the ability to make arbitrary network connections and to interpose on network traffic from all active web page instances. Attacks targeting these subsystems require sophisticated sequences of messages to be sent out. To mitigate this threat, our security policies maintain internal state while parsing messages and prevent spurious and out of order messages from being sent. Attacks may also be less likely since we make extensive use of Java to prevent memory corruption and the subsystems are simple and qualitatively easier to reason about than the components of the web page subsystem.

Successful attacks on the browser kernel are the most severe and lead to a full browser compromise. Since the browser kernel is trusted it can access all browser-level states and events. However, our browser kernel is simple (only 1221 lines of C++ code) and has limited functionality, which simplifies reasoning about correctness.

VII. Related work

In addition to the projects we already discussed in this paper, our work is related to previous studies in browser-based security.

Our OP web browser introduces a new architecture for building more secure web browsers. The most closely related works are Tahoma [17], the *Building a Secure Web Browser* project [22], and a recent position paper by Reis, *et al.* [43] which all propose new browser architectures. Tahoma shares many of the same design principles as OP, but our architecture differs in two key ways. First, Tahoma uses VMMs to provide isolation for different web-based applications, and they use a manifest to help craft their network policy. In contrast, we use OS-level mechanisms for isolation and we support existing web-based applications, and we track interactions at a more fine level of granularity, allowing us to explore novel policies such as our plugin policy. However, these two architectures are complementary: one could imagine OP using Tahoma to provide even stronger isolation. In the Building a Secure Web Browser project, the authors propose a new browser architecture that relies on the underlying OS policies (e.g., file system permissions) to enforce browser security. OP handles security policies in the browser kernel, giving us more flexibility. In a recent position paper by Reis, *et al.* the authors share many design principles and philosophies with OP, but this concurrent work is a position paper and does not include an implementation or evaluation.

A number of recent projects develop techniques for securing web-based applications [18], [14], [13], securing JavaScript [42], [51], [26], [9], supporting mashups [25], protecting privacy [24], [45], enforcing the same-origin policy [28], [12], adding new abstractions for improved sharing [49], and overcoming DNS rebinding attacks in browsers [23]. These projects are orthogonal to OP, our goal is to provide a more secure platform to implement these, and other, techniques.

The idea of sandboxing browsers was first introduced by Goldberg [20], and GreenBorder is a recent commercial product that sandboxes Internet Explorer [1]. We use this type of sandboxing in our OP 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 apply different sandboxing rules to each subsystem, giving us even more control over our browsers interactions with the underlying system.

We show how our browser can be used to analyze browser-based attacks; current approaches for analyzing intrusions will not work for browser-based attacks. Several recent projects [30], [31], [19] use OS-level dependency graphs to highlight the subset of activity on a system that is likely to be part of an attack. These techniques are effective for server-style workloads where servers isolate distinct sessions into different OS-level processes, thus allowing them to track malicious sessions using OS-level states and events. However, these techniques fall short when there are long-lived processes that handle multiple sessions because they taint conservatively entire OS-level objects and cannot separate out unrelated activities using OS-level events alone. In addition to OS-level techniques, researchers have been able to analyze browser-based attacks by using client-based honeypots [50], [36], [41] to crawl the Internet looking for malicious sites. For example, the HoneyMonkey project [50] batches many different sites together, and when they detect a malicious site they re-process each site in isolation to determine which one was responsible for the attack. These techniques work well for automated crawling experiments, but are not suitable for analyzing active browsers since they fail to capture and integrate the interactions of the user, something OP handles well.

VIII. Conclusions

We have described the OP web browser and the different elements that make our browser secure. We have shown that by using an architecture that is designed to be secure we can enforce security policies that are flexible enough to apply to browser plugins, while at the same time formally verifying important security properties.

The OP web browser is responsive to user interaction and implements features that make it compatible with current web pages. We include two plugins, supporting Flash compatible content as well as other multimedia content, Javascript and

Source subsystem	Destination subsystem	Message description	Includes return message
UI, HTML	Kernel	*New web page.* Tells the kernel to open a new web page instance.	No
HTML, JS, Plugin	Network	*Fetch URL.* Fetch an object from the network, return data, redirected URLs, and protocol metadata.	Yes
Kernel	HTML	*Set URL.* Sets the base URL for a newly created web page instance.	No
HTML, Javascript	UI	*Set status, location, title.* Sets the UI status bar, location address, and page title.	No
UI, Xvnc	Xvnc, UI	*Raw VNC data.* Mechanism for transmitting VNC data between the UI and the current Xvnc server. These are two separate one-way messages.	No
UI	Kernel	*Set current web page instance.* Gives the UI the ability to navigate the browsing history.	No
UI	Kernel	*Stop current page loading.* Notifies the kernel that the user wants to stop loading the current web page.	No
Kernel	UI	*New web page instance notification.* Notifies the UI of each new web page instance, the UI uses this information to track browsing history.	No
All	Storage	*Store and retrieve object.* Allows all subsystems to store and retrieve persistant data.	Yes
All	Storage	*Object acl add / remove user.* By default all objects are only accessable by the subsystem that created them, but owning subsystems can add additional readers to stored objects.	No
HTML	UI	*Object ready for download.* Web pages notfiy the UI when downloaded content (e.g., downloaded PDF files) is ready to be saved.	No
Network, HTML	Storage	*Store/retreieve/delete cookies.* Mechanism for the network and HTML (in response to Javascript) to manage cookies.	Yes
HTML, Plugin	Javascript	*Execute Javascript.* Mechanism for executing Javascript.	Yes
HTML	Javascript	*Set Javascript event handler.* Sets the event handling code for Javascript events.	No
HTML	Javascript	*Invoke Javascript event handler.* Invokes the Javascript handling code for a particular event.	Yes
HTML	Plugin	*Set URL*. Sets the base URL for a newly created plugin.	No
Javascript	HTML	*Access DOM element.* Provides access to DOM elements.	Yes
HTML	Plugin	*Call NPAPI function*. The browser makes a call into the plugin	Yes
Plugin	HTML	*Call NPAPI Function.* The plugin makes a call into the browser	Yes

Fig. 13. Message API for OP subsystems. In this figure we list the origin (or source) of the message and the destination subsystem, as well as a text description explaining the purpose of the message. Some messages include a separate return message that is a reply from the destination subsystem back to the source.

basic web page support, giving us a functional browser capable of enforcing security policies. We have also included plugins into our security model, which are difficult for current browsers to control and enforce policy upon. Using the system's implementation we have created and shown a formal model using Maude and model checked invariants describing the security of our browser. We have also shown how the OP web browser can assist in forensic examination of attacks that we are unable to prevent.

All of these elements build up the OP web browser security, creating a web client capable of withstanding attack. We have demonstrated that by design it is not vulnerable to many forms of browser attacks while not limiting the functionality of the browser.

Appendix

In Figure 13 we list our full message passing API. All messages include a header that lists the source ID, the destination ID, a global message ID, the message type, an optional message value, and a field for the length of the payload. For messages with a payload, the data follows directly after the header. The message data contains the message-specific data, such as a URL for a *fetch URL* message.

Acknowledgment

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 and Paul Debrowski for feedback on an early draft of our paper. Finally, we would like to thank the anonymous reviewers for their thorough and helpful comments and feedback. This research was funded in part by a grant from the Internet Services Research Center (ISRC) of Microsoft Research.

References

[1] Greenborder desktop dmz solutions. http://www.greenborder.com.
[2] Netscape pluing api. http://www.mozilla.org/projects/plugins/.
[3] Sopcast. http://www.sopcast.org/.
[4] Adobe. Adobe flash player. http://www.adobe.com/products/flashplayer/.
[5] Adobe. External data not accessible outside a macromedia flash movie's domain. http://www.adobe.com/go/tn_14213.
[6] Adobe. Flash player penetration. http://www.adobe.com/products/player_census/flashplayer/.
[7] Adobe. Flash player settings manager. http://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager.html.
[8] Adobe. Flash player update available to address security vulnerabilities. http://www.adobe.com/support/security/bulletins/apsb07-12.html.

[9] V. Anupam and A. Mayer. Security of web browser scripting languages: vulnerabilities, attacks, and remedies. In *Proceedings of the 7th conference on USENIX Security Symposium*, 1998.

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

[11] S. Chen, J. Meseguer, R. Sasse, H. J. Wang, and Y.-M. Wang. A systematic approach to uncover security flaws in gui logic. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, May 2007.

[12] S. Chen, D. Ross, and Y.-M. Wang. 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.

[13] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Oct 2007.

[14] S. Chong, K. Vikram, and A. C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *Proceedings of the 16th Usenix Security Symposium*, Boston, MA, USA, August 2007.

[15] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

[16] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude manual (version 2.3), 2007. http://maude.cs.uiuc.edu.

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

[18] U. Erlingsson, B. Livshits, and Y. Xie. End-to-end web application security. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS XI)*, May 2007.

[19] A. Goel, K. Po, K. Farhadi, Z. Li, and E. del Lara. The taser intrusion recovery system. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[20] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 1996 USENIX Security Symposium*, pages 1–13, July 1996.

[21] N. Inc. Apparmor linux application security. http://www.novell.com/linux/security/apparmor/.

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

[23] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting browsers from dns rebinding attacks. In *Proceedings of the 14th ACM conference on computer and communications security (CCS)*, 2007.

[24] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the 15th international conference on World Wide Web (WWW)*. ACM Press, 2006.

[25] C. Jackson and H. J. Wang. Subspace: Secure cross-domain communication for web mashups. In *Proceedings of the 16th International World Wide Web Conference (WWW)*, Banff, Alberta, Canada, may 2007.

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

[27] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *Proccedings of the IEEE International Conference on Security and Privacy for Emerging Areas in Communication Networks (Securecomm)*, August 2006.

[28] C. Karlof, J. Tygar, D. Wagner, and U. Shankar. Dynamic pharming attacks and locked same-origin policies for web browsers. In *Proceedings of the 14th ACM conference on computer and communications security (CCS)*, 2007.

[29] KDE. The konqueror web browser. http://www.konqueror.org/features/browser.php.

[30] S. T. King and P. M. Chen. Backtracking intrusions. In *Proceedings of the 2003 Symposium on Operating Systems Principles (SOSP)*, pages 223–236, October 2003.

[31] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching intrusion alerts through multi-host causality. In *Proceedings of the 2005 Network and Distributed System Security Symposium (NDSS)*, February 2005.

[32] B. W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, October 1973.

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

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

[35] Microsoft. Activex security: Improvements and best practices. http://msdn2.microsoft.com/en-us/library/bb250471.aspx.

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

[37] Mozilla. Rhino: Javascript for java. http://www.mozilla.org/rhino/.

[38] P. D. Petrkov. Pdf pwns windows. http://www.gnucitizen.org/blog/0day-pdf-pwns-windows.

[39] P. D. Petrkov. Quicktime pwns firefox. http://www.gnucitizen.org/blog/0day-quicktime-pwns-firefox.

[40] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 2003 USENIX Security Symposium*, pages 257–272, August 2003.

[41] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. 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.

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

[43] C. Reis, S. D. Gribble, and H. M. Levy. Architectural principles for safe web programs. In *Proceedings of the Sixth Workshop on Hot Topics in Networks (HotNets)*, Nov 2007.

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

[45] U. Shankar and C. Karlof. Doppelganger: Better browser privacy without the bother. In *Proceedings of the 13th ACM conference on computer and communications security (CCS)*, 2006.

[46] A. Stamos and Z. Lackey. Attaking ajax web applications. Presented at the 2006 Black Hat USA Conference.

[47] Sun. Java security architecture. http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc1.html.

[48] D. Turner. Symantec internet security threat report: Trends for january - june 07. Technical report, Symantec Inc., 2007.

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

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

[51] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, January 2007.