The Security Architecture of the Chromium Browser

Adam Barth
UC Berkeley

Collin Jackson
Stanford University

Charles Reis
University of Washington

Google Chrome Team
Google Inc.

ABSTRACT

Most current web browsers employ a monolithic architecture that combines "the user" and "the web" into a single protection domain. An attacker who exploits an arbitrary code execution vulnerability in such a browser can steal sensitive files or install malware. In this paper, we present the security architecture of Chromium, the open-source browser upon which Google Chrome is built. Chromium has two modules in separate protection domains: a browser kernel, which interacts with the operating system, and a rendering engine, which runs with restricted privileges in a sandbox. This architecture helps mitigate high-severity attacks without sacrificing compatibility with existing web sites. We define a threat model for browser exploits and evaluate how the architecture would have mitigated past vulnerabilities.

1. INTRODUCTION

In the past several years, the web has evolved to become a viable platform for applications. However, most web browsers still use the original monolithic architecture introduced by NCSA Mosaic in 1993. A monolithic browser architecture has many limitations for web applications with substantial client-side code. For example, a crash caused by one web application takes down the user's entire web experience instead of just the web application that misbehaved [22]. From a security point of view, monolithic web browsers run in a single protection domain, allowing an attacker who can exploit an unpatched vulnerability to compromise the entire browser instance and often run arbitrary code on the user's machine with the user's privileges.

In this paper, we present the security architecture of Chromium, the open-source web browser upon which Google Chrome is built. Chromium uses a modular architecture, akin to privilege separation in SSHD [19]. The browser kernel module acts on behalf of the user, while the rendering engine module acts on behalf of "the web." These modules run in separate protection domains, enforced by a sandbox that reduces the privileges of the rendering engine. Even if an attacker can exploit an unpatched vulnerability in the rendering engine, obtaining the privileges of the entire rendering engine, the sandbox helps prevent the attacker from reading or writing the user's file system because the web principal does not have that privilege.

There have been a number of research proposals for modular browser architectures [8, 28, 5, 7] that contain multiple protection domains. Like Chromium's architecture, these proposals aim to provide security against an attacker who can exploit an unpatched vulnerability. Unlike Chromium's architecture, these proposals trade off compatibility with existing web sites to provide architectural isolation between web sites or even individual pages. The browser's security policy, known as the "same-origin policy," is complex and can make such fine-grained isolation difficult to achieve without disrupting existing sites. Users, however, demand compatibility because a web browser is only as useful as the sites that it can render. To be successful, a modular browser architecture must support the entire web platform in addition to improving security.

Chromium's architecture allocates the various components of a modern browser between the browser kernel and the rendering engine, balancing security, compatibility, and performance. The architecture allocates high-risk components, such as the HTML parser, the JavaScript virtual machine, and the Document Object Model (DOM), to its sandboxed rendering engine. These components are complex and historically have been the source of security vulnerabilities. Running these components in a sandbox helps reduce the severity of unpatched vulnerabilities in their implementation. The browser kernel is responsible for managing persistent resources, such as cookies and the password database, and for interacting with the operating system to receive user input, draw to the screen, and access the network. The architecture is based on three design decisions:

1. The architecture must be *compatible* with the existing web. Specifically, the security restrictions imposed by the architecture should be transparent to web sites. This design decision greatly limits the landscape of possible architectures but is essential in order for Chromium to be useful as a web browser.

2. The architecture treats the rendering engine as a *black box* that takes unparsed HTML as input and produces rendered bitmaps as output (see Figure 1). In particular, the architecture relies on the rendering engine alone to implement the same-origin policy. This design decision reduces the complexity of the browser kernel's security monitor because the browser kernel need only enforce course-grained security restrictions. For example, the browser kernel grants the ability to upload a file to an entire instance of the rendering engine, even when that privilege is only needed by a single security origin.

3. The architecture should *minimize user security decisions.* Users dislike constant security prompts and occasionally make insecure decisions [10]. Whenever

possible, the browser should attempt to make the correct security decisions without resorting to yet another security dialog box.

The architecture does not prevent an attacker who compromises the rendering engine from attacking other web sites (for example, by reading their cookies). Instead, the architecture aims to prevent an attacker from reading or writing the user's file system, helping protect the user from a drive-by malware installation.

To evaluate the security of Chromium's architecture, we examine the disclosed browser vulnerabilities in Internet Explorer, Firefox, and Safari from the preceding year. For each vulnerability, we determine which module would have been affected by the vulnerability, had the vulnerability been present in Chromium. We find that 67.4% (87 of 129) of the vulnerabilities would have occurred in the rendering engine, suggesting that the rendering engine accounts for a significant fraction of the browser's complexity.

Not all rendering engine vulnerabilities would have been mitigated by Chromium's architecture. Chromium's architecture is designed to mitigate the most severe vulnerabilities, namely those vulnerabilities that let an attacker execute arbitrary code. If an attacker exploits such a vulnerability in the rendering engine, Chromium's architecture aims to restrict the attacker to using the browser kernel interface. We find that 38 of the 87 rendering engine vulnerabilities allowed an attacker to execute arbitrary code and would have been mitigated by Chromium's architecture. These account for 70.4% (38 of 54) of all disclosed vulnerabilities that allow arbitrary code execution.

To evaluate the security benefits of sandboxing additional browser components, we examined the arbitrary code execution vulnerabilities that would have occurred in the browser kernel. We find that 72.7% (8 of 11) of the vulnerabilities result from insufficient validation of system calls and would not have been mitigated by additional sandboxing. For example, one such vulnerability involved the browser improperly escaping a parameter to ShellExecute when handling external protocols. Although counting vulnerabilities is an imperfect security metric [25], these observations lead us to believe that Chromium's architecture suitably divides the various browser components between the browser kernel and the rendering engine.

By separating the browser two protection domains, one representing the user and another representing the web, Chromium's security architecture mitigates approximately 70% of critical browser vulnerabilities that let an attacker execute arbitrary code. The remaining vulnerabilities are difficult to mitigate with additional sandboxing, leading us to conclude that the architecture extracts most of the security benefits of sandboxing while maintaining performance and compatibility with existing web content.

Organization. Section 2 defines a threat model for browser exploits. Section 3 details Chromium's architecture. Section 4 describes the sandbox used to confine the rendering engine. Section 5 explains the browser kernel API used by the sandboxed rendering engine. Section 6 evaluates the security properties of the architecture. Section 7 compares Chromium's architecture with other browser architectures. Section 8 concludes.

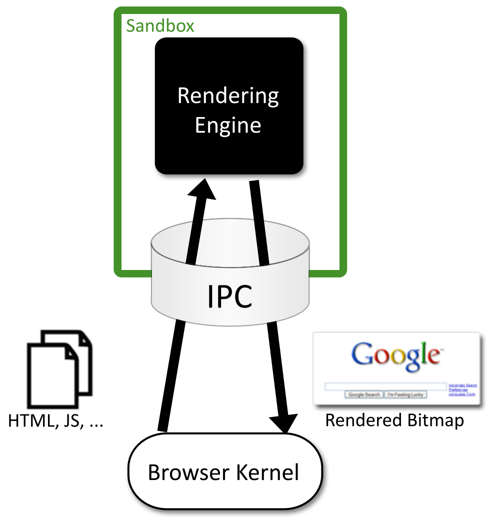

Figure 1: The browser kernel treats the rendering engine as a black box that parses web content and emits bitmaps of the rendered document.

2. THREAT MODEL

In order to characterize the security properties of Chromium's architecture, we define a threat model by enumerating the attacker's abilities and goals. The security architecture seeks to prevent an attacker with these abilities from reaching these goals. We can use this threat model to evaluate how effectively Chromium's architecture protects users from attack.

Attacker Abilities. We consider an attacker who knows an unpatched security vulnerability in the user's browser and is able to convince the user's browser to render malicious content. Typically, these abilities are sufficient to compromise the user's machine [21]. More specifically, we assume the attacker has the following abilities:

1. The attacker owns a domain name, say `attacker.com`, that has not yet been added to the browser's malware blacklist [20]. The attacker has a valid HTTPS certificate for the domain, and controls at least one host on the network. These abilities can all be readily purchased for approximately $5.

2. The attacker is able to convince the user to visit his or her web site. There are a number of techniques for convincing the user to visit `attacker.com`, such as sending out spam e-mail, hosting popular content, or driving traffic via advertising. It is difficult to price this ability, but, in a previous study, we were able to attract a quarter of a million users for approximately $50 [1].

3. The attacker knows, and is able to exploit, an unpatched arbitrary code execution vulnerability in the user's web browser. For example, the attacker might know of an unpatched buffer overflow in the browser's HTML parser [18], an integer overflow in the regular expression library [15], or a buffer overflow in the bookmarks system [16].

In-Scope Goals. Chromium's architecture focuses on preventing the attacker from achieving three high-value goals:

- **Persistent Malware.** The attacker attempts to install malicious software on the user's computer. For example, the attacker might attempt to install a botnet client [6] that receives commands over the network and participates in coordinated attacks on the user or on network targets. In particular, the attacker attempts to install persistent malicious software that survives the user closing his or her browser.

- **Transient Keylogger.** The attacker attempts to monitor the user's keystrokes when the user interacts with another program. Such system-wide keyloggers are often used to steal user passwords, credit card numbers, and other sensitive information. To achieve this goal, the attacker's keylogger need not survive the user closing the browser.

- **File Theft.** The attacker attempts to read sensitive files on the user's hard drive. For example, the attacker might attempt to read the system's password database or the user's financial records. File theft is an important concern for enterprise users whose machines often contain large amounts of confidential information.

If an attacker is able to achieve one or more of these goals, he or she has the ability to cause serious harm to the user. For example, an attacker who is able to install malware is no longer constrained by the browser's security policy and often said to "own" the user's machine. Chromium's architecture aims to prevent an attacker with the above abilities from achieving these goals.

Out-of-Scope Goals. There are a number of other attacker goals for which Chromium's *architecture* does not provide additional protection. Chromium includes security features that help defend against these threats, but these features rely on the rendering engine to enforce the browser security policy.

- **Phishing.** In a phishing attack, the attacker tricks the user into confusing a dishonest web site with an honest web site. The confused user supplies his or her password to the dishonest web site, who can then impersonate the user at the honest web site. An attacker who exploits an unpatched vulnerability can create a convincing phishing site by corrupting a window displaying the honest site.

  Chromium has a number of security features to help mitigate phishing attacks. For example, the browser's location bar highlights the web site's domain name, aiding users in determining whether they are viewing an honest or a dishonest web site. The browser also black-lists known phishing sites, showing a full-page warning if the user visits a known phishing site. Additionally, the browser displays additional security user interface elements if the site has an extended validation certificate. Many of these security features can be found in other browsers and are orthogonal to the design of Chromium's architecture.

- **Origin Isolation.** Chromium's architecture treats the rendering engine as representing the entire web principal, meaning an attacker who compromises the rendering engine can act on the behalf of any web site. For example, an attacker who exploits an arbitrary code execution vulnerability can obtain the cookies for every web site and can read all the passwords stored in the browser's password database. If the attacker is not able to exploit an unpatched vulnerability, the usual browser security policy prevents the attacker from reading cookies or passwords from host names that are not under his or her control.

- **Firewall Circumvention.** The same-origin policy is designed to restrict an attacker's network access from within the browser [9]. These restrictions are intended to protect confidential resources behind organizational firewalls. However, an attacker who exploits an unpatched vulnerability can bypass these restrictions and can read HTTP responses from internal servers by making use of the browser's URL requesting facilities. The ability to request arbitrary web URLs follows the compatibility and black-box design decisions in order to support stylesheets and image tags.

- **Web Site Vulnerabilities.** Chromium's architecture does not protect an honest web site if the site contains cross-site scripting (XSS), cross-site request forgery (CSRF), or header injection vulnerabilities. To be secure against web attackers, these sites must repair their vulnerabilities. Chromium supports `HttpOnly` cookies [13], which can be used as a partial mitigation for XSS.

3. CHROMIUM'S ARCHITECTURE

Chromium's architecture has two modules: a rendering engine and browser kernel. At a high level, the rendering engine is responsible for converting HTTP responses and user input events into rendered bitmaps, whereas the browser kernel is responsible for interacting with the operating system. The browser kernel exposes an API that the rendering engine uses to issue network requests, access persistent storage, and display bitmaps on the user's screen. The browser kernel is trusted to act on behalf of the user, whereas the rendering engine is trusted only to act on behalf of the web.

- **Rendering Engine.** The rendering engine interprets and executes web content by providing default behaviors (for example, drawing `<input>` elements) and by servicing calls to the DOM API. Rendering web content proceeds in several stages, beginning with parsing, building an in-memory representation of the DOM, laying out the document graphically, and manipulating the document in response to script instructions. The rendering engine is also responsible for enforcing the same-origin policy, which helps prevent malicious web sites from disrupting the user's session with honest web sites.

  The rendering engine contains the bulk of the browser's complexity and interacts most directly with untrusted web content. For example, most parsing takes place in the rendering engine, including HTML parsing, image decoding, and JavaScript parsing. These components are complex and have a history of security vulnerabilities. To interact with the user, the local machine, or

Rendering Engine	Browser Kernel
HTML parsing	Cookie database
CSS parsing	History database
Image decoding	Password database
JavaScript interpreter	Window management
Regular expressions	Location bar
Layout	Safe Browsing blacklist
Document Object Model	Network stack
Rendering	SSL/TLS
SVG	Disk cache
XML parsing	Download manager
XSLT	Clipboard

Both
URL parsing
Unicode parsing

Table 1: The assignment of tasks between the rendering engine and the browser kernel.

the network, the rendering engine issues calls to the browser kernel API. The rendering engine runs inside a sandbox that restricts access to the underlying operating system (see Section 4).

- **Browser Kernel.** The browser kernel is responsible for managing multiple instances of the rendering engine and for implementing the browser kernel API (see Section 5). For example, the browser kernel implements a tab-based windowing system, including a location bar that displays the URL of the currently active tab its associated security indicators. The browser kernel manages persistent state, such as the user's bookmarks, cookies, and saved passwords. It is also responsible for interacting with the network and intermediating between the rendering engine and the operating system's native window manager. To implement its API, the browser kernel maintains state information about the privileges it has granted to each rendering engine, such as a list of which files each rendering engine is permitted to upload. The browser kernel uses this state to implement a security policy that constrains how a compromised rendering engine can interact with the user's operating system.

The assignment of browser components to modules is driven by security, compatibility, and performance, but some assignments are due to historical artifacts. For example, the browser kernel is responsible for displaying JavaScript `alert` dialog boxes, whereas `<select>` drop-down menus are displayed by the rendering engine. Some features, such as the cookie database, are implemented in the browser kernel because of its direct access to the file system. Other features, such as regular expression parsing and execution, are implemented by the rendering engine because they are performance-sensitive and have often been the source of security vulnerabilities [23].

As shown in Table 1, the rendering engine is responsible for most parsing and decoding tasks because, historically, these tasks have been the source of a large number of browser vulnerabilities. For example, to display a web site's shortcut icon in the browser's user interface, the browser kernel retrieves the image from the network but does not attempt to decode it. Instead, the browser kernel sends the image to the rendering engine for decoding. The rendering engine responds with an uncompressed bitmap of the icon, which the browser kernel then copies to the screen. This seemingly convoluted series of steps helps prevent an attacker who knows an unpatched vulnerability in the image decoder from taking control of the browser kernel.

One exception to this pattern is the network stack. The HTTP stack is responsible for parsing HTTP response headers and invoking a `gzip` or `bzip2` decoder to inflate HTTP responses with these `Content-Encoding`s. These tasks could be allocated to the rendering engine, at the cost of complicating the implementation of the network stack and lowering performance. As another example, both the browser kernel and the rendering engine parse URLs because URL handling is ubiquitous in a web browser.

Process Granularity. Roughly speaking, Chromium uses a separate instance of the rendering engine for each tab that displays content from the web, providing fault tolerance in the case of a rendering engine crash. Chromium also uses the rendering engine to display some trusted content, such as the interstitial warnings for HTTPS certificate errors and phishing sites. However, these rendering tasks are performed by a separate instance of the rendering engine that does not handle content obtained from the web. The main exception to this pattern is the Web Inspector, which displays trusted content but is rendered by a rendering engine that contains web content. Chromium uses this design because the Web Inspector interacts extensively with the page it is inspecting.

Plug-ins. In Chromium's architecture, each plug-in runs in a separate host process, outside both the rendering engines and the browser kernel. In order to maintain compatibility with existing web sites, browser plug-ins cannot be hosted inside the rendering engine because plug-in vendors expect there to be at most one instance of a plug-in for the entire web browser. If plug-ins were hosted inside the browser kernel, a plug-in crash would be sufficient to crash the entire browser.

By default, each plug-in runs outside of the sandbox and with the user's full privileges. This setting maintains compatibility with existing plug-ins and web sites because plug-ins can have arbitrary behavior. For example, the Flash Player plug-in can access the user's microphone and webcam, as well as write to the user's file system (to update itself and store Flash cookies). The limitation of this setting is that an attacker can exploit unpatched vulnerabilities in plug-ins to install malware on the user's machine.

Vendors could write future versions of plug-ins that operate within Chromium's sandbox, to provide greater defense against plug-in exploits. Chromium also contains an option to run existing plug-ins inside the sandbox. To do so, run the browser with the `--safe-plugins` command line option. This setting is experimental and might cause instability or unexpected behavior. For example, sandboxed plug-ins might not be able to update themselves to newer versions.

4. THE SANDBOX

To help defend against an attacker who knows an unpatched vulnerability in the rendering engine, Chromium's architecture runs each rendering engine in a *sandbox.* This sandbox restricts the rendering engine's process from issuing some system calls that could be used to achieve the attacker's goals from Section 2.

Goals. Ideally, the sandbox would force the rendering engine to use the browser kernel API to interact with the outside world. Many DOM methods, such as `appendChild`, simply mutate state within the rendering engine and can be implemented entirely within the rendering engine. Other DOM methods, such as `XMLHttpRequest`'s `send` method, require that the rendering engine do more than just manipulate internal state. An honest rendering engine can use the browser kernel interface to implement these methods. The goal of the sandbox is to require even a compromised rendering engine to use the browser kernel interface to interact with the user's file system or the outside world.

Implementation. Currently, Chromium relies on Windows-specific features to sandbox the rendering engine. Instead of running with the user's Windows security token, the rendering engine runs with a restricted security token. Whenever the rendering engine attempts to access a "securable object," the Windows Security Manager checks whether the rendering engine's security token has sufficient privileges to access the object. The sandbox restricts the rendering engine's security token in such a way that the token fails almost every such security check.

Before rendering web content, the rendering engine adjusts the security token of its process by converting its security identifiers (SIDs) to "`DENY_ONLY`," adding a restricted SID, and calling the `AdjustTokenPrivileges` function. The rendering engine also runs on a separate desktop, mitigating the lax security checking of some Windows APIs, such as `SetWindowsHookEx`, and limiting the usefulness of some unsecured objects, such as `HWND_BROADCAST`, whose scope is limited to the current desktop. Additionally, the rendering engine runs in a Windows Job Object, restricting the rendering engine's ability to create new processes, read or write to the clipboard, or access `USER` handles. Other researchers have advocated similar approaches [11]. For further details about the Chromium sandbox, see the design document [4].

Limitations. Although the sandbox restricts the ability of a compromised rendering engine to interact with the operating system, the sandbox has some limitations:

- **FAT32.** The FAT32 file system does not support access control lists. Without access control lists, the Windows security manager ignores a process's security token when granting access to a file stored in a FAT32 file system. The FAT32 file system is rarely used on modern hard drives but is used on many USB thumb drives. For example, if a user inserts a USB thumb drive that uses a FAT32 file system, a compromised rendering engine can read and write the contents of the drive.

- **Misconfigured Objects.** If an object has a NULL security descriptor, the Windows security manager will grant access without considering the accessing security token. Although NULL security descriptors are uncommon, some third-party applications create objects with NULL security descriptors. On the NTFS file system, this limitation is largely mitigated because the sandbox forces Windows to check that the rendering engine has access to all parent directories as well as the target file. The sandbox enforces this rule by removing the rendering engine's privilege to "bypass traverse checking."

- **TCP/IP.** Theoretically, the rendering engine could create a TCP/IP socket on Windows XP because the low-level system calls to open a socket do not appear to require handles or to perform access checks. In practice, though, the usual Win32 library calls for creating a socket fail, because those APIs require handles which the rendering engine is unable to obtain. We have attempted to build a proof-of-concept but are as yet unable to open a socket from within a sandboxed process. On Windows Vista, the relevant system calls perform access checks based on the current security token.

5. THE BROWSER KERNEL INTERFACE

The sandbox restricts the rendering engine's ability to interact directly with the underlying operating system. To access operating system functionality, such as user interaction, persistent storage, and networking, the rendering engine relies on the browser kernel API. In providing functionality to the rendering engine, the browser kernel must be carefully designed not to grant more privileges than are necessary. In particular, the browser kernel interface is designed not to leak the ability to read or write the user's file system.

User Interaction. Commodity operating systems expose an interface that lets applications interact with the user, but these interfaces are often not designed to be used by untrusted applications. For example, in the X Window System, the ability to create a window on an X server also implies the ability to monitor all of the user's keystrokes [2]. The browser kernel mediates the rendering engine's interaction with the user to help enforce two security constraints:

- **Rendering.** Instead of granting the rendering engine direct access to a window handle, the rendering engine draws into an off-screen bitmap. To display the bitmap to the user, the rendering engine sends the bitmap to the browser kernel, and the browser kernel copies the bitmap to the screen. This design adds a single video memory to video memory copy to the usual drawing pipeline, which has a similarly small performance impact to double buffering, and clips the renderered bitmap to the browser window's content area.[1]

- **User Input.** Instead of delivering user input events directly to the rendering engine, the operating system delivers these events to the browser kernel. The browser kernel dispatches these events according to the currently focused user interface element. If focus resides in the browser chrome, the input events are handled internally by the browser kernel. If the content area has focus, the browser kernel forwards the input events to the rendering engine. This design leverages the user's intent (which interface element is in focus) to restrict which user input events can be observed by a compromised rendering engine.

[1]In the initial beta release of Google Chrome, the browser kernel also exposes an API for drawing menus for the `<select>` element that can be used to draw over arbitrary regions of the screen.

Persistent Storage. The sandbox is responsible for ensuring that the rendering engine cannot access the user's file system directly. However, the rendering engine does require some access to the user's file system to upload and download files.

- **Uploads.** Users can upload files to web sites using the file upload control. When the user clicks the form control, the browser displays a file picker dialog that lets the user select a file to upload. If the browser kernel did not restrict which files the rendering engine could upload, an attacker who compromised the rendering engine could read an arbitrary file on the user's file system by uploading the file to `attacker.com`.

  Instead of confirming each file upload with a dialog box, which would run contrary to minimizing user security decisions, Chromium uses the DarpaBrowser's "powerbox" pattern [28], treating the user's selection of a file with a file picker dialog as an authorization to upload the file to an arbitrary web site. The browser kernel is responsible for displaying the file picker dialog and records which files the user has authorized for which instances of the rendering engine. Similarly, dragging and dropping a file onto the browser's content area grants the associated rendering engine the permission to upload that file. These authorizations last for the lifetime of the rendering engine, which is often shorter than the lifetime of the entire browser because the browser kernel spawns new instances of the rendering engine as the user opens and closes tabs.

- **Downloads.** When downloading a file, a web site is permitted to write to the user's file system. Rather than writing to the file system directly, the rendering engine uses the browser kernel API to download URLs. Left unchecked, a compromised rendering engine could abuse this API to compromise the integrity of the user's file system. To help protect the file system, the browser kernel directs downloads to a designated download directory. Additionally, the browser kernel blacklists certain kinds of file names that the rendering engine could use to elevate its privileges, including reserved device names [14], file names with `.local` extensions [12], and shell-integrated file names, such as `Desktop.ini`.

Networking. Rather than accessing the network directly, the rendering engine retrieves URLs from the network via the browser kernel. Before servicing a URL request, the browser kernel checks whether that the rendering engine is authorized to request the URL. Web URL schemes, like `http`, `https`, and `ftp`, can be requested by every instance of the rendering engine. However, the browser kernel prevents most rendering engines from requesting URLs with the `file` scheme, because a compromised rendering engine could read the user's hard drive by requesting various `file` URLs. Chromium is able to render HTML documents stored in the local file system if requested by the user (for example, by typing a `file` URL in the address bar). However, these documents are rendered in a dedicated rendering engine.

	Browser	Renderer	Unclassified
Internet Explorer	4	10	5
Firefox	17	40	3
Safari	12	37	1

Table 2: Total Number of Browser CVEs by Chromium Module

6. SECURITY EVALUATION

It is difficult to evaluate the security of a system empirically because determining whether a system is secure requires considering all possible attacks. Instead of reasoning about all possible attacks, we examine recent security vulnerabilities in web browsers and evaluate whether those vulnerabilities, if they had existed in Chromium, would have allowed attackers to achieve the goals listed in Section 2. After analyzing CVEs statistically, we present a case study of one security vulnerability and explain how the vulnerability was mitigated by Chromium's architecture.

6.1 Browser CVE Analysis

To evaluate the extent to which Chromium's architecture protects users from security vulnerabilities, we analyze all browser security vulnerabilities that were patched between July 1, 2007 and July 1, 2008 for Internet Explorer, Firefox, and Safari. We classify each vulnerability, identified by its Common Vulnerabilities and Exposure (CVE) identifier, by what an attacker could gain by exploiting the vulnerability and by which module in Chromium's architecture would have contained the vulnerability had the vulnerability been present in an implementation of the architecture.

During this period, Internet Explorer patched 19 vulnerabilities, Firefox patched 60 vulnerabilities, and Safari patched 50 vulnerabilities. These counts cannot be compared directly because each browser has its own methodology for reporting bugs. For example, most security updates to Firefox contain one or two CVEs for "crashes with evidence of memory corruption," but these CVEs often represent 20 or 30 separate bugs (i.e., Bugzilla IDs). Also, closed source browser vendors are not required to obtain CVEs for vulnerabilities that are discovered and fixed privately [25].

Complexity. First, we classify each browser vulnerability by module (see Table 2). We use the relative number of vulnerabilities for each module as a rough estimate of the relative complexity of that module. If a module has had a greater proportion of vulnerabilities in the past, we assume that the module is likely to contain a greater proportion of future vulnerabilities. These classifications are somewhat subjective because there is often not a precise match between vulnerabilities in other browsers and modules in Chromium's architecture.

The classification for Safari is the most precise because Apple classifies Safari vulnerabilities by component (and Safari components roughly correspond with components in Chromium). The number of rendering engine vulnerabilities patched in Safari might be over-represented relative to the browser kernel vulnerabilities because we discovered some of these ourselves while working on the WebKit component of the rendering engine, which is shared between Safari and Chromium.

	Browser	Renderer	Unclassified
Internet Explorer	1	9	5
Firefox	5	19	0
Safari	5	10	0

Table 3: Number of Arbitrary Code Execution CVEs by Chromium Module

We are unable to classify several vulnerabilities, described below.

- One Internet Explorer CVE [17] did not contain enough information to determine which module would have contained the vulnerability. The four remaining unclassified vulnerabilities are in Internet Explorer's handling of ActiveX objects.
- We are unable to classify one Firefox vulnerability in Firefox's extension interface because Chromium's architecture does not yet contain an extension interface. The remaining two unclassified vulnerabilities are in Firefox's email handling infrastructure, which is not present in Chromium's architecture.
- The unclassified vulnerability in Safari was present in Safari's PDF viewer. However, Chromium does not contain a built-in PDF viewer.

Table 2 reveals that rendering engines account for the greatest number of disclosed vulnerabilities, suggesting that the rendering engine is more complex than the browser kernel. This observation is consistent with the line count heuristic for estimating code complexity. Chromium's rendering engine contains approximately 1,000,000 lines of code (excluding blank lines and comments), whereas the browser kernel contains approximately 700,000 lines of code (also excluding blank lines and comments).

Arbitrary Code Execution. Chromium's security architecture is designed to mitigate the impact of arbitrary code execution vulnerabilities in the rendering engine by limiting the ability of the attacker to issue system calls after compromising the rendering engine. Many of the vulnerabilities considered above are not mitigated by Chromium's architecture because they do not let an attacker read or write the user's file system. For example, one of the Firefox vulnerabilities let an attacker learn the URL of the previous page. While patching these vulnerabilities is important to protect the user's privacy (and sensitive information), these vulnerabilities are not as severe as vulnerabilities that let web sites to install and run malicious programs, such as botnet clients [21], on the user's machine.

If we restrict our attention to those vulnerabilities that lead to arbitrary code execution (see Table 3), we find that the rendering engine contained more arbitrary code execution vulnerabilities than the browser kernel. (As above, the four of the unclassified Internet Explorer vulnerabilities were related to ActiveX plug-ins and one contained insufficient information to determine the module.) Chromium's architecture helps mitigate these vulnerabilities by sandboxing the arbitrary code the attacker chooses to execute.

Of the vulnerabilities in the browser kernel that lead to arbitrary code execution, the majority (8 of 11) of these vulnerabilities were caused by insufficient validation of inputs to system calls and not by buffer overflows or other memory-safety issues. These vulnerabilities are unlikely to be mitigated by sandboxing more browser components because the browser must eventually issue the system calls in question, suggesting that other techniques are required to mitigate these issues.

Summary. Although "number of CVEs" is not an ideal security metric, this data suggests that Chromium's division of responsibilities between the browser kernel and the rendering engine places the more complex, vulnerability-prone code in the sandboxed rendering engine, making it harder for an attacker to read or write the user's hard drive by exploiting a vulnerability. Moreover, most of the remaining vulnerabilities would not have been mitigated by additional sandboxing, suggesting that assigning more tasks to the rendering engine would not significantly improve security.

6.2 Case Study: XML External Entities

Another method for evaluating Chromium's security architecture is to determine whether the architecture successfully defends against unknown vulnerabilities in the rendering engine. In this case study, we examine one vulnerability in detail and explain how the security architecture mitigated threats in the scope of our threat model but did not mitigate threats that are out of scope. This vulnerability is "unknown" in the sense that we discovered the vulnerability after implementing the sandbox and browser kernel security monitor. The vulnerability was fixed before the initial beta release of Google Chrome, but this section describes the state of affairs just after we discovered the vulnerability.

XXE. An XML Entity is an escape sequence, such as `&copy;`, that an XML (or an HTML) parser replaces with one or more characters. In the case of `&copy;`, the entity is replaced with the copyright symbol, ©. The XML standard also provides for external entities [3], which are replaced by the content obtained by retrieving a URL.

In an Xml eXternal Entity (XXE) attack, the attacker's XML document, hosted at `http://attacker.com/`, includes an external entity from a foreign origin [26]. For example, the malicious XML document might contain an entity from `https://bank.com/` or from `file:///etc/passwd`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE doc [ <!ENTITY ent SYSTEM "/etc/passwd"> ]>
<html>
  <head><script> ... </script></head>
  <body>&ent;</body>
</html>
```

If vulnerable to XXE attacks, the browser will retrieve the content from the foreign origin and incorporate it into the attacker's document. The attacker can then read the content, circumventing a confidentiality goals of the browser's security policy.

libXML. Like many browsers, Chromium uses libXML to parse XML documents. Unlike other browsers, Chromium delegates parsing tasks, including XML parsing, to a sandboxed rendering engine. After implementing the sandbox, but prior to the initial beta release of Google Chrome, we became aware that the rendering engine's use of libXML was vulnerable to XXE attacks. As a result, the rendering engine was not preventing web content from retrieving URLs from foreign origins. Instead, the rendering engine was passing the requests, unchecked, to the browser kernel.

Using our proof-of-concept exploit, we observed that the browser kernel performed its usual black-box checks on the URLs requested by the rendering engine. If the external entity URL was a web URL, for example with the `http`, `https`, or `ftp` schemes, the browser kernel serviced the request, as instructed. However, if the external entity URL was from the user's file system, i.e. from the `file` scheme, then the browser kernel blocked the request, preventing our proof-of-concept from reading confidential information, such as passwords, stored in the user's file system.

Discussion. The XML external entity vulnerability illustrates three properties of Chromium's security architecture:

1. By performing complex operations, such as parsing, in the sandboxed rendering engine, Chromium's security architecture mitigated an unknown vulnerability. The sandbox helped prevent the attacker from reading confidential information stored in the user's file system.

2. The sandbox did not completely defend against the XXE vulnerability because the attacker was still able to retrieve URLs from foreign web sites. However, the security architecture does not aim to prevent an attacker who exploits a bug in the rendering engine from requesting URLs from arbitrary web sites. To block such requests and treat the rendering engine as a black box, the browser kernel would need to sacrifice compatibility (e.g., ban cross-site images).

3. Chromium's architecture mitigated the XXE vulnerability even though the vulnerability did not let an attacker execute arbitrary code. Although the architecture is designed to protect against an attacker who fully compromises a rendering engine, the architecture also helps mitigate less-severe vulnerabilities that lead to partial compromises of the rendering engine.

7. RELATED WORK

In this section, we compare Chromium's architecture to the architectures of other popular and proposed web browsers.

Monolithic. Traditionally, browsers are implemented with a monolithic architecture that combines the rendering engine and the browser kernel into a single process image. For example, Internet Explorer 7, Firefox 3, and Safari 3.1 each execute in a single operating system protection domain. If an attacker can exploit an unpatched vulnerability in one of these browsers, the attacker can gain all the privileges of the entire browser. In typical configurations of Firefox 3 and Safari 3.1, these privileges include the full privileges of the current user.

Internet Explorer 7 on Windows Vista can run in a "protected mode" [24], which runs the browser as a low-integrity process. Running in protected mode, the browser is restricted in its ability to write to the user's file system, but an attacker who uses an exploit to run arbitrary code can still *read* the user's file system and exfiltrate confidential documents.

The VMware browser appliance [27] hosts Firefox inside a virtual machine with limited rights. The virtual machine provides a layer of isolation that helps prevent an attacker who exploits a vulnerability in the browser from reading or writing the user's file system. The protection afforded by this architecture is coarse-grained in the sense that the browser is prevented from reading any of the user's files, even files the user wishes to upload to web sites (for example, to a photo-sharing site or to attach to email messages at a webmail site).

Modular. A number of researchers have proposed other modular browser architectures and have made different design decisions.

- **SubOS.** In SubOS [8], the authors leverage sub-process isolation features of an experimental operating system to divide a web browser into multiple modules. Instead of implementing the usual same-origin security policy, SubOS isolates web pages with different URLs, rendering SubOS incompatible with many web sites.

- **DarpaBrowser.** The DarpaBrowser [28] uses an object capability discipline to grant an untrusted rendering engine a limited set of capabilities necessary to render a web page. For example, the DarpaBrowser grants the rendering engine the capability to navigate to URLs contained in HTML hyperlinks but does not grant the engine the ability to navigate to any other URLs. This non-black-box architecture prevents the DarpaBrowser from being compatible with many web sites (e.g., those that navigate using JavaScript).

  The DarpaBrowser has high goals for security. In a threat model in which the rendering engine is completely compromised, the authors seek to render honest web sites without granting the rendering engine the capability to exfiltrate confidential information found on those web sites. This goal conflicts with compatibility because the web platform provides many avenues for exfiltrating data.

- **Tahoma.** Tahoma [5] runs each "site" in a separate protection domain, isolated using a virtual machine monitor. Tahoma defines a site by a manifest file that enumerates the URLs that the site wishes to be included in the same protection domain. Sites run in separate instances of a rendering engine and are unable to communicate with each other. The rendering engine includes the vast majority of browser components, including the cookie store, history database, network cache, and password database. The browser kernel, which runs outside the virtual machines, is responsible only for compositing the rendered output of the rendering engines onto the user's screen. The browser also limits the network connectivity of rendering engines by implementing a reverse proxy that mediates network requests from the rendering engines.

  The Tahoma architecture has strong isolation properties. Tahoma helps prevent an attacker who compromises one of the rendering engines from reading or writing files on the user's file system. To make use of these isolation features, a web site operator must opt-in by publishing a manifest file. After publishing a manifest, the web site operator need not use a standard rendering engine and can, instead, run arbitrary code inside the virtual machine. To help prevent attacker from abusing the privilege, Tahoma asks the user approve each web site, which is inconsistent with our "minimize security decisions" design decision. If the user incorrectly approves a malicious web site, that web site can steal confidential documents from within an organizational firewall or use the user's machine to send spam e-mail.

- **OP Browser.** Similar to Tahoma, the Opus Palladianum (OP) web browser [7] runs multiple instances of a rendering engine, each in a separate protection domain isolated using different trust labels in SE Linux. Unlike Tahoma, the OP browser uses a separate protection domain for each web page and reserves a number of components (including the JavaScript virtual machine, the network stack, and cookie storage) for other browser modules. In the OP architecture, the browser kernel is more akin to a micro-kernel: chiefly responsible for message passing.

  The OP browser runs each page in a separate protection domain. This design mitigates of unpatched vulnerabilities, but imposes a compatibility cost on inter-page scripting and third-party iframes. The OP browser's sandboxing of plug-ins is more restrictive than Chromium's `--safe-plugins` option, imposing a higher compatibility cost.

- **Internet Explorer 8.** Internet Explorer 8 runs tabs in separate processes, each of which runs in protected mode. This architecture, called "loosely coupled IE," is designed to improve the reliability, performance, and scalability of the browser: rendering engine crashes do not result in full browser crashes [29]. After the rendering engine crashes, each tab is re-opened in a new instance of the rendering engine.

8. CONCLUSIONS

Chromium's security architecture divides the browser into two protection domains, the browser kernel and the rendering engine. The sandboxed rendering engine is responsible for performing many complex, error-prone tasks, such as parsing HTML and executing JavaScript. As a result, the architecture helps protect the confidentiality and integrity of the user's file system even if an attacker exploits an unpatched vulnerability in the rendering engine.

Chromium's design differs from other modular browser architecture proposals. Our decision to be compatible with existing sites requires that the architecture support the full range of web platform features. Our black box approach for the rendering engine reduces the complexity of the browser kernel's security monitor by enforcing policies that are independent of the actual contents of documents being rendered. Chromium's architecture is also designed to minimize user security decisions, avoiding constant security prompts.

One difficulty in evaluating the security properties of Chromium's architecture is that the architecture is designed to help provide security even if the implementation has bugs. We cannot simply assume that all such vulnerabilities will arise in the rendering engine because the browser kernel, an essential piece of the trusted computing base, is also of significant complexity. To estimate where future vulnerabilities are likely to occur, we survey browser vulnerabilities from the past year and find that 67.4% (87 of 129) would have occurred in the rendering engine had they been present in Chromium. We also find that Chromium's architecture would have mitigated 70.4% (38 of 54) of the most severe vulnerabilities.

Of the arbitrary code execution vulnerabilities that would have occurred in the browser kernel, 8 of 11 are a result of insufficient validation of parameters to operating system calls. These vulnerabilities are difficult to mitigate with additional sandboxing because the browser must eventually issue those system calls to compatibly render web sites. These observations suggest that Chromium's architecture makes good use of the sandbox in its allocation of tasks between the browser kernel and the rendering engine.

To download an implementation of the architecture, visit `http://www.google.com/chrome/`. The source code of our implementation is available at `http://dev.chromium.org/`.

9. REFERENCES

[1] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *15th ACM Conference on Computer and Communications Security (CCS)*, October 2008.

[2] Rune Braathen. Crash course in X Windows security, November 1994. `http://www.ussg.iu.edu/usail/external/recommended/Xsecure.html`.

[3] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*, section 4.2.2. `http://www.w3.org/TR/REC-xml/#sec-external-ent`.

[4] The Chromium Authors. Sandbox, 2008. `http://dev.chromium.org/developers/design-documents/sandbox`.

[5] Richard S. Cox, Jacob Gorm Hansen, Steven D. Gribble, and Henry M. Levy. A safety-oriented platform for web applications. In *IEEE Symposium on Security and Privacy*, 2006.

[6] Neil Daswani, Michael Stoppelman, and the Google Click Quality and Security Teams. The anatomy of Clickbot.A. In *Proceedings of HotBots 2007*, 2007.

[7] Chris Grier, Shuo Tang, and Samuel T. King. Secure web browsing with the op web browser. In *IEEE Symposium on Security and Privacy*, 2008.

[8] Sotiris Ioannidis and Steven M. Bellovin. Building a secure web browser. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, June 2001.

[9] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting browsers from DNS rebinding attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, November 2007.

[10] Chris Karlof, Umesh Shankar, J. D. Tygar, and David Wagner. 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)*, November 2007.

[11] David LeBlanc. Practical Windows sandboxing, July 2007. `http://blogs.msdn.com/david_leblanc/archive/2007/07.aspx`.

[12] Microsoft. Dynamic-link library redirection. `http://msdn.microsoft.com/en-us/library/ms682600.aspx`.

[13] Microsoft. Migitating cross-site scripting with HTTP-only cookies. `http://msdn.microsoft.com/en-us/library/ms533046.aspx`.

[14] Microsoft. Naming a file. http://msdn2.microsoft.com/en-us/library/aa365247(VS.85).aspx.

[15] Mitre. CVE-2006-7228, 2006.

[16] Mitre. CVE-2007-3743, 2007.

[17] Mitre. CVE-2007-3893, 2007.

[18] Mitre. CVE-2008-3360, 2008.

[19] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*, August 2003.

[20] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. All your iFRAMEs point to us. In *Proceedings of the 17th USENIX Security Symposium*, July 2008.

[21] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, K. Wang, and Nagendra Modadugu. The ghost in the browser - analysis of web-based malware. In *Proceedings of HotBots 2007*, April 2007.

[22] Charles Reis, Brian Bershad, Steven D. Gribble, and Henry M. Levy. Using processes to improve the reliability of browser-based applications. Technical report, 2007. University of Washington Technical Report UW-CSE-2007-12-01.

[23] SecurityFocus. PCRE Regular Expression Library Multiple Security Vulnerabilities, 2007. http://www.securityfocus.com/bid/26346.

[24] Marc Silbey and Peter Brundrett. Understanding and working in protected mode internet explorer, 2006. http://msdn.microsoft.com/en-us/library/bb250462.aspx.

[25] Window Snyder. Critical vulnerability in Microsoft metrics, November 2007. http://blog.mozilla.com/security/2007/11/30/critical-vulnerability-in-microsoft-metrics/.

[26] Gregory Steuck. XXE (Xml eXternal Entity) attack, October 2002. http://www.securiteam.com/securitynews/6D0100A5PU.html.

[27] VMWare. Browser appliance. http://www.vmware.com/appliances/directory/browserapp.html.

[28] David Wagner and Dean Tribble. A security analysis of the Combex DarpaBrowser architecture, March 2002. http://www.combex.com/papers/darpa-review/.

[29] Andy Zeigler. IE8 and Loosely-Coupled IE, March 2008. http://blogs.msdn.com/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx.