# Secure web browsing - The OS approach

Taner Aydin
(taydin@mailbox.tu-berlin.de)

Seminar Internet Security
Technical University of Berlin

WS 2009/20010 January 14, 2010

**Abstract**

Many of the vulnerabilities of today's web browsers are a consequence of the enormously evolved web. There is a significant difference in how the web is used today as opposed to 15 years ago, when the first browsers were created. Back then browsers were designed to render static web pages with no or little media content. Today's web pages use a lot of different media content through plugins like Flash or Quicktime. Also they use a lot of dynamics and user generated content. So while the web has evolved to the so-called web 2.0, the architectural design of browsers remains the same as 15 years ago, creating a huge base for new kinds of attacks. Attacks like cross-site-scripting, phishing, cookie-theft and others pose a major threat today. This seminar paper evaluates different new browser technologies – concentrating on and starting with the OP browser – that specifically address the security issues in today's web landscape.

## 1 Introduction

In the early days of the world wide web and the first graphical browsers the task of a browser was fairly simple: Download a provided web document and render it correctly for the user. The assumed usage model of the web was that users navigate from one static site to another. The browser vendors at that time could have never foreseen the situation taking place today: One user running a browser with several open tabs, each of which is hosting different web applications like home banking, social networking, email and others. These applications are likely to contain highly sensible data. But there is no real isolation between the applications since in many browsers (e.g. Firefox, Opera) they run in the same address space. In Firefox plugins even run in the address space of the browser[1]. The effect is that a compromise of one web application (including plugins) can spread to other web applications currently opened, or even further the whole browser can be corrupted, providing hackers easy access to sensitive information like passwords or cookies. Security efforts of browser vendors include closing code vulnerabilities with patches and at least trying to guarantee the correct functioning of crucial security policies such as the *same origin policy*, which dates back to the first netscape browser[2]. But these upgrades have their limitations. First of all browser vendors have different interpretations of such a policy[3]. Secondly, and this is more fundamental, the architectural design of browsers is flawed, i.e. it is based on the old usage model of the web.

Browsers become more and more an application host for web applications, similar to operating systems hosting different concurrent applications. Google for instance builds on this new *web application model* with their Google Chrome OS[4], the Google Chrome

*Browser* being basically the only visible application, because all other applications are thought to be web applications that run within the browser.

Since a browser must now deal with web *applications* it becomes obvious that security considerations for applications in operating systems could also apply here. It is therefore a good idea to use operating system techniques and incorporate them in a new way of browser architecture. This is what the research team at the computer science department at the University of Illinois came up with. They designed and implemented a new browser, with a supposedly more secure architecture which is based on ideas from the field of operating systems. The next chapter deals with the so called OP browser[5], that was published in May 2008. Chapter 3 just very shortly describes related works which are afterwards evaluated together with OP in chapter 4.

# 2 OP browser

The OP browser was designed and implemented to demonstrate a new way of web browser architecture. The first iteration in implementing the browser used KHTML[6] as its rendering engine. In the instructions for building the browser[7] there is a hint that a new version of OP, named OP2, exists which uses WebKit[8] as the rendering engine. WebKit originally was a branch of the KHTML engine and now serves as foundation for other major browsers like Safari and Chrome[9].

The OP browser combines OS design principles with formal methods for verifying the correctness of the whole browser. The verification process will not be the focus here but will be described very shortly in its function. OP also comes with new security policies and means to analyze browser based attacks *after* they have happened.

OP has three goals: To prevent attacks from happening, contain possible attacks and limit their damage and lastly the ability to recover from succeeded attacks.

The main assumption is that an attacker has complete control over the content served to the browser making it possible to target a specific component of the browser. So one cannot rely on correct content. What OP does rely on is the underlying OS, the Java Virtual Machine and a correctly functioning DNS.

## 2.1 Design Principles

The key aspect of the OP browser architecture is its division into separate components, each of them responsible for a specific task only. All of those components communicate with each other through a message passing interface to make the browser as a system work.

There are four design principles of the OP browser that have guided the design.

**Simple and explicit communication** Communication between the components of the browser should be simple and explicit. This provides a clean separation between functionality and security. Fewer paths can be taken for an action, and fewer paths do make a verification easier.

**Strong isolation** A strong isolation between distinct browser components reduces the likelihood of unanticipated interactions and also allows stronger claims about general security.

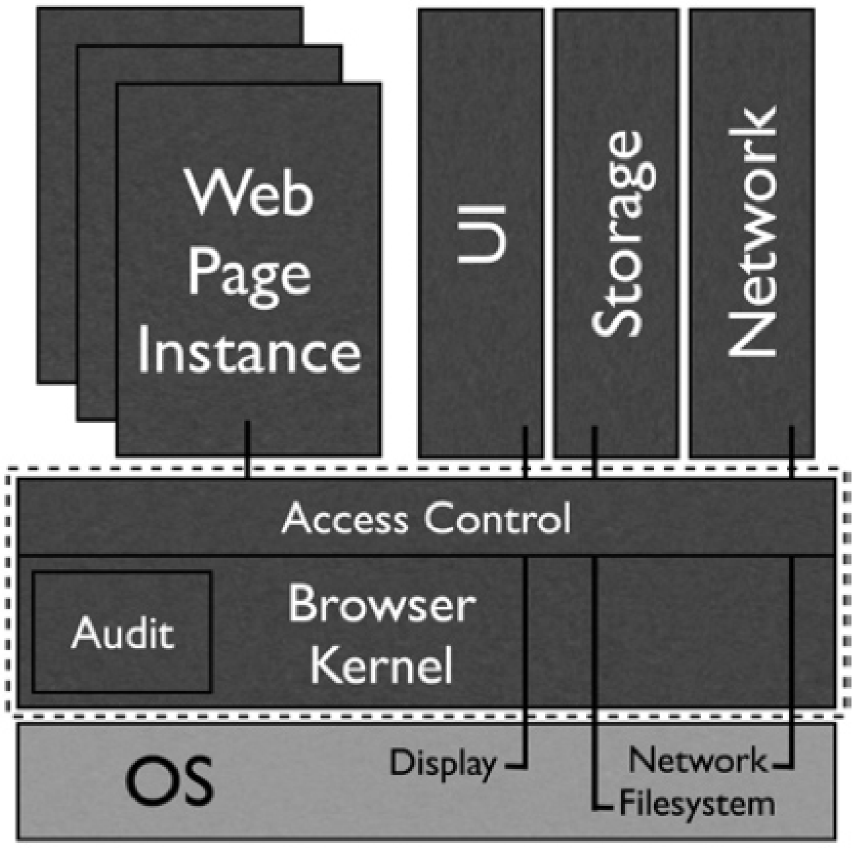
**Proper design of components** The components should be designed to do the proper thing. Also monitor the components to ensure they adhere to the design.

**Maintain compatibility with current technologies** Try not to complicate web application development, since that would contradict the original goal of making web browsing more secure.

## 2.2 Browser architecture

The browser architecture consists of five subsystems. Each subsystem runs within a separate address space. This way the wanted isolation between the different browser components is achieved. Additionally OS-level sandboxing techniques are used to limit the interactions of each subsystem with the OS itself. For this purpose SELinux[10] is used. The detailed tasks and duties of each subsystem are now described in detail.

**Figure 1:** The overall browser architecture[11]



### 2.2.1 Browser kernel

The browser kernel represents the base of the OP browser, similar to an OS kernel. It has three tasks: manage subsystems, manage all communication between each subsystem, and maintain a detailed security audit log.

It creates and deletes processes for the subsystems. Most of the processes are created while the browser launches, but as the user opens and closes new tabs or windows new processes have to be created for instance.

All messages between subsystem processes pass through the browser kernel (except for communication to the Xvnc server, which happens directly). The kernel uses OS-level pipes for message passing. For simplification reasons the kernel is written as a single-threaded, event-driven component and all messages have a unique message ID and a global order. The kernel can accept or deny any message that violate the access control policy.

For the security audit log the kernel records *all* messages between subsystems, which allows forensic analysis after an attack has happened.

### 2.2.2 Web page subsystem

Each web page is represented by a separate web page instance, which is implemented as a set of different processes with distinct address spaces. The kernel creates a new web page instance if the user clicks on a link or is redirected to a new web page. Each web page instance consists of 4 parts. These are the following.

**HTML engine** The HTML engine represents the root HTML document of the web page. It parses HTML and delegates JavaScript code found in the document to the JavaScript interpreter. It also tags JavaScript code and plugins with the source domain. This is useful for isolation of different content on the same web page, as will be described later on. This is a quite critical task, since the security and correctness of other parts of the system relies on the correct tagging. The rendering engines KHTML respectively WebKit are written in C++. Therefore the tagging result is afterwards checked by running a self-written Java-based HTML parser in order to eliminate the vector for memory corruption attacks on the rendering engine to achieve malicious tagging.

**JavaScript interpreter** The JavaScript interpreter handles all JavaScript code that is delegated from the HTML engine. To access DOM[12] elements it communicates back with the HTML engine. The JavaScript interpreter is again not built from scratch, instead the Rhino JavaScript interpreter[13] is used, which is written in Java. The good part about this is that the JVM automatically provides the needed isolation between the different JavaScript instantiations[5]. Therefore all JavaScript instantiations run within one address space.

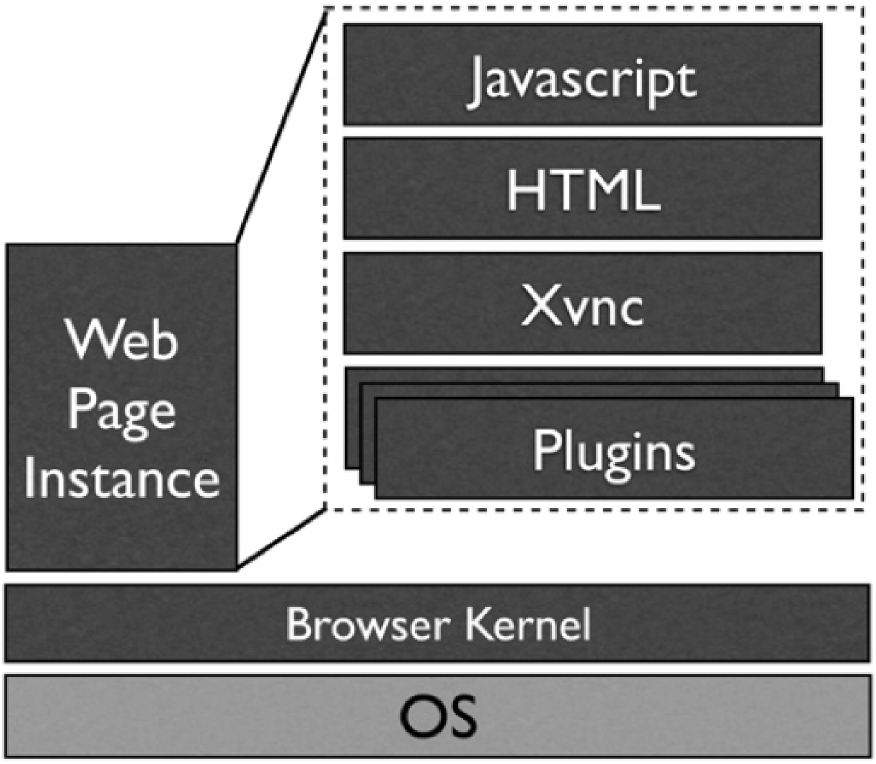
**Plugins** There are lots of different browser plugins for today's browsers. Some of them also written in unsafe languages, which do not prevent memory corruption attacks. OP puts each plugin instance in a separate address space to keep it isolated from the rest.

**X Server** All the content of a web page is rendered in an Xvnc server[14] which streams the rendered content to the UI component.

### 2.2.3 UI subsystem

The UI subsystem is a Java application, that implements the typical browser widgets, such as the typical buttons, an address bar and a browser history. To introduce another element of isolation it does not render the web page content itself. Instead it receives the rendered content from the web page subsystem via VNC. This way corrupt content that relies on the rendering to compromise the system does not affect the UI component at all. Such corrupt content could only compromise a web page instance, which is as stated above isolated from the rest of the system. This point is very important, because the UI subsystem is the only part of the browser that has unrestricted access to the underlying file system. Anytime the browser needs to store or load a file it is done through the UI component. This is of course because of the typical user initiated down- and uploads.

**Figure 2:** The web page subsystem architecture[11]



### 2.2.4 Network subsystem

No component of the OP browser has direct access to the network. To access it they must pass through the network subsystem. This implements the HTTP protocol and downloads and uploads data on behalf of all other components.

### 2.2.5 Storage subsystem

The same as for the network subsystem applies to the storage subsystem: No component of the browser has direct access to the file system, except for the UI component of course. All the other components must go through the storage subsystem. The storage subsystem stores persistent data, such as cookies, in an SQLite[15] database.

## 2.3 Security policy and enforcement

### 2.3.1 General issues with browser plugins

Browser plugins extend browsers with functionality to handle certain MIME-types which the browser itself is not able to handle. There are plugins for all different kinds of content types. The browser determines the MIME-type of loaded content and delegates the content to the particular plugin. Although plugins are provided with an API to interact with the browser, they still run in the same address space as the browser[1]. Thus they can, in the worst case, manipulate the complete browser as needed. Still, plugins have at least their own ad-hoc security mechanisms and policies. But this does not solve the problem, in fact it complicates browser security even further, since the plugin policy goals must not be coherent with the browser security goals[5].

### 2.3.2 Plugin security in OP

To overcome plugin security flaws OP does the same as for each other subsystem: Each plugin instance is run within a separate OS-level process and has to pass its messages through the browser kernel. The kernel can now apply security mechanisms by inspecting the messages. While inspecting messages that initially trigger a plugin to load content from an URL the kernel labels the plugin with a domain. Now the plugin can be denied access to any other component of the browser and vice versa in case it violates the security policy of the browser.

In general each pairwise communication channel between browser subsystems can have an access control policy that specifies which access between two specific subsystems is allowed. The OP browser implementation provides a simple API for that.

### 2.3.3 Plugin security policies

Besides the flexible access control to enforce different security policies and the omnipresent *same origin policy* which is also implemented in OP and even verified, there are two new developed security policies in OP that specifically address plugin security.

**Provider domain policy** One common implemented security policy in modern browsers is called the same origin policy. It ensures that a document or script loaded from a web site from domain $A$ cannot access or manipulate content from another web site from another domain $B$. One could think that this security policy is sufficient enough for plugins, but there is still a major problem. For instance, suppose a web developer creates a web site and embeds a YouTube video in it. With the same origin policy applied, the Flash plugin, together with its content, and the rest of the site are tagged with the same origin: the domain of the web developers web site. An attacker could conclusively inject malicious content on *youtube.com* but at the end affect the web page of the web developer, and get access to the DOM and cookies of the site.

The *provider domain policy* in contrast tags the plugin with the domain of the site that hosts the plugin *content*. In the example of a YouTube video on a web page, the origin of the embedded Flash plugin will be *youtube.com*. Thus the embedded video plugin is restricted to its real source.

**Plugin freedom policy** The plugin freedom policy aims at advanced flexibility for plugins like peer-to-peer video players. Such plugins especially need more flexibility for outgoing network connections. The policy provides local per-plugin storage and unlimited network access at the cost of access to DOM elements and other browser components. This policy is realized by prohibiting communication between the plugin and all browser components, except the network and storage subsystem, which is by the way similar to plugin operation in current browsers with the scriptable API components removed. Prohibiting the communication of a plugin with other browser components prevents them from leaking client information across multiple sites.

## 2.4 Analyzing browser based attacks

The creators of the OP browser try to remain realistic and assume that attacks are still possible. Therefore OP provides an analysis tool for browser based attacks. The analysis is based on the security audit log of the kernel. There are two types of analysis.

First, analyzing which sites initiated an attack so they can be blacklisted. Second, tracking the effects of known malicious web sites that were visited to determine if an attack happened and evaluate it.

The tool uses dependency graphs to illustrate the browsing history. Nodes in the graph can be web sites or files. Web sites are stored in the graph as they are visited. Different web page instances with the same URL are considered to be different nodes. Files are stored in the graph whenever actual files from the file system pass through the browser either by downloads or uploads.

### 2.5 Verification

The OP browser is modeled formally using Maude. Maude is a reflective language and system that supports rewriting logic specification[16]. Since OP consists of isolated components that interact via a compact message passing interface, the formal model and actual implementation are very similar. Once the browser is modeled Maude can be used to search the finite state space of the model and look for invalid respectively malicious states. The original paper describes some examples of verifying parts of the browser in more detail.

## 3 Related works

### 3.1 Gazelle

Gazelle is another, very similar, approach for secure web browsing[17], but it refines some aspects of OP. The authors of both the OP and Gazelle papers are partly the same. The Gazelle paper was published in February 2009.

**Basic architecture** The basic architecture of Gazelle is almost the same as in OP. Gazelle also has a browser kernel, that provides a message passing interface, and is run in a separate address space.

Gazelle's security model has one main aspect: Putting different principals (web pages) in separated protection domains (address spaces) in order to protect them from each other. This sounds quite similar to what OP does but it differs. The authors of Gazelle realized that the fine grained splitting in the web page subsystem of OP, where each web page instance is split up into different processes for the HTML engine and the JavaScript interpreter does not provide any more security but instead an unnecessary drawback. The communication between HTML engine and JavaScript interpreter, which is normally very intimate, for DOM manipulations for example, has to go through the browser kernel every time for each message. This imposes high IPC costs for script-intensive sites. But the point is, if a web page instance gets compromised it can be stopped completely and can be disposed. This way Gazelle is more effective when it comes to performance without losing security.

The protection domains of the different principals are defined by the same origin policy, namely the triple $(protocol, domain, port)$. This way the resources that need to be protected across principals, such as DOM and script objects, persistent state such as cookies, display and network communications, are all associated with their true origin. Different instances of the same principal are also run in separate protection domains. This means embedded frames, if from another origin, are also in a different protection domain. Also subdomains and their corresponding super domains are different principals resulting in different protection domains.

**Unified same origin policy across all resources** Gazelle uses a *unified same origin* policy across all types of resources in contrast to most current browsers where the same origin policy differs for different resources. For instance cookies are not associated with the triple $(protocol, domain, port)$ but with the URL including the path and without protocol information. The same origin policy is extended to all content types except scripts and style sheets. This means that included content in an plugin, image or Iframe is associated with its true source.

**Mixed HTTP and HTTPS content** In current browsers it is allowed for HTTPS sites to contain embedded HTTP content. They apply the same origin policy and since the protocol differs both contents will have different protection domains and don't interfere with each other. But there is still a problem. When in HTTPS content scripts or style sheets are delivered via HTTP, those scripts and style sheets run with the privileges of the HTTPS site. Gazelle prohibits this: HTTP-transmitted scripts or style sheets in HTTPS sites are not rendered.

**Subdomain treatment** Current browsers also allow that a site sets its document property *domain* to a suffix of its real domain. For example *www.google.com* can set its *domain* property to *google.com*. But then the site has no control over which other subdomains can access it. Gazelle doesn't allow subdomain sites to set the *document.domain* property.

## 3.2 Other current concepts

While OP and Gazelle are still research and the implementations explicitly meant for research interests, current browsers are still far away from the basic concepts. Still there is one considerable improvement.

There is a trend that especially WebKit-based browsers like Safari, Chrome and others do use a separate address space for plugins[18]. This is already a major step, since plugins mostly come from third-party vendors, which the browser vendors should not trust implicitly. This is one approach whose shortcomings will be evaluated in the evaluation section.

Additionally, Chrome does a big step further: it isolates tabs from one another by putting them into separate address spaces. Also Chrome splits the browser into components: the browser kernel and the rendering engine. Each tab is backed up by an instance of the rendering engine. Still, there are some crucial differences which are evaluated in detail in the Gazelle paper[17].

# 4 Evaluation

The OP browser aims for a completely new architecture for web browsers. The splitting into components with distinct address spaces that interact through a message passing interface and the interposing browser kernel are a new way to face the new web application model. Also the creators try to use as much safe programming languages as possible to reduce the likelihood of memory corruption attacks. The result is a browser where a corrupt webpage only compromises the web page subsystem. Attacks on the UI, network and storage subsystem are supposedly less likely since memory corruption likelihood is reduced with the use of Java. Also the kernel keeps internal state of the messages between the subsystems and all messages have a global order. This way the kernel can prevent out of order messages from being sent. Since the browser has a kernel, it is obvious that an attack on the kernel represents the most severe scenario. But according to the OP authors the browser kernel, with its 1K lines of C++ code, is small enough to make reasoning about its correctness simple[5].

The authors of Gazelle realized that the OP approach in splitting even the web page subsystem further into different processes does not add security benefits but instead performance overhead and therefore propose that the web page instance runs in one address space only.

Plugin compromise containment is done with both research browsers OP and Gazelle and also in some current browsers in use, such as Safari and Chrome. However the approaches differ a little bit. OP, Safari and Chrome place a plugin in a separate address space completely isolated from the rest of the browser. Gazelle puts the plugin together with HTML engine, JavaScript interpreter etc. in the address space of the web page subsystem, because the latter is already isolated from the rest of the browser and this way IPC costs are reduced.

In using the same origin policy both OP and Gazelle reduce the vector for cross-origin vulnerabilities, such as different kinds of cross-site-scripting attacks which have become a considerable threat to web browsing security. The browser kernel contains the logic to enforce this, namely the system calls and the message passing interface. The kernel simply forbids cross origin accesses if needed. If a web page instance in Gazelle or OP gets compromised the needed isolation is given to prevent a spread of the compromise on other instances. The same origin policy is also implemented in Safari, Chrome and almost every other browser. But there is one key advancement in OP and Gazelle.

Both OP and Gazelle implement the same origin policy in a way that it also applies correctly for content in embedded plugins on a web page. OP calls this *provider domain policy* and Gazelle leaves the name and just extends the policy. The plugin containment strategy ensures that an erroneous plugin with malicious content can not manipulate the site it is embedded in. This is a real important step in browser research since plugins have proven to be a significant source of vulnerabilities[19].

In conclusion, current browsers are already doing some advancements in the field of browser security, with Google's Chrome probably being the most known in public. But still there are some crucial security risks, why OP and Gazelle take one step further to make browsing as secure as possible.

## References

[1] Gecko plugin API. `https://developer.mozilla.org/en/Gecko_Plugin_API_Reference:Plug-in_Basics`, January 2010.

[2] Netscape Plugin API. `https://developer.mozilla.org/en/Gecko_Plugin_API_Reference:Preface`, January 2010.

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

[4] Chromium OS. `http://www.chromium.org/chromium-os`, January 2010.

[5] Chris Grier, Shuo Tang, and Samuel T. King. Secure web browsing with the OP web browser. *Proceedings of the 2008 IEEE Symposium on Security and Privacy (Oakland)*, May 2008.

[6] KHTML rendering engine. `http://developer.kde.org/documentation/library/kdeqt/kde3arch/khtml/`, January 2010.

[7] OP browser code. `http://code.google.com/p/op-web-browser/`, January 2010.

[8] WebKit project. `http://webkit.org/`, January 2010.

[9] Charles Reis and Steven D. Gribble. Isolating web programs in modern browser architectures. *Proceedings of Eurosys*, 2009.

[10] SELinux. `http://fedoraproject.org/wiki/SELinux`, January 2010.

[11] Chris Grier, Shuo Tang, and Samuel T. King. building a more secure web browser. *;login:*, 33(4):14–21, August 2008.

[12] Document object model. `http://www.w3.org/DOM/`, January 2010.

[13] Rhino JavaScript interpreter. `http://www.mozilla.org/rhino/`, January 2010.

[14] Xvnc. `http://xvnc.sourceforge.net/`, January 2010.

[15] SQLite. `http://www.sqlite.org/`, January 2010.

[16] Maude. `http://maude.cs.uiuc.edu/`, January 2010.

[17] Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. The multi-principal os construction of the gazelle web browser. Technical Report MSR-TR-2009-16, Microsoft Research, February 2009.

[18] Plugin architecture of Chromium. `http://dev.chromium.org/developers/design-documents/plugin-architecture`, January 2010.

[19] Microsoft security intelligence report (sir). Technical Report Volume 5, Microsoft, 2008.