A pattern for a Web Content Renderer

Paulina Silva, Universidad Técnica Federico Santa María
Raúl Monge, Universidad Técnica Federico Santa María
Eduardo B. Fernandez, Florida Atlantic University

Currently, most software development is focused in creating systems connected to the Internet, which allows to add functionality within a system and facilities to their *Stakeholders*. This leads to depend on a *web client*, such as *Web Browser*, which allows access to services, data or operations that the system delivers. Within the browser's main components, a rendering engine is in charge of obtaining a convenient data structure or binary (depends on the resource) as an output for the browser process; who at the same time will send this output to the operating system. If the output is a web page, it will be shown to the user as a graphical representation; if not, the information obtained by the resource will be saved in a format to the user's convenience. However, the Internet influences the attack surface of the system, and unfortunately many stakeholders and developers are not aware of the risks to which they are exposed. The lack of security education among software developers and the scarce and scattered documentation for browsers (and standardization) could become a big problem in large architectural developments that depend on browsers to perform their services. A Reference Architecture (RA) of the *Web Browser*, using Architectural Patterns, could be a starting point for understanding its security mechanisms and architecture, which interact with a bigger web system. This architecture would unify ideas and terminology, giving a holistic view of implementation independent details for both the browser and the system it communicates with. To our best knowledge, we have not found documentation that could unify the mentioned ideas. We have developed a Browser Infrastructure pattern that describes the infrastructure to allow the communication between a Web Client and a Server in the Internet, and now we have extended it. In this paper we describe the component in charge of the rendering of a web resource within the web browser, named here as Web Content Renderer. Since Google Chrome's appearance, the structure of a Browser has evolved while thinking how to make the browser faster, simple and more secure.

Additional Key Words and Phrases: Browser, Web Client, Browser Renderer, Reference Architecture, Pattern

1. INTRODUCTION

Patterns are encapsulated solutions to recurrent problems and define a way to express requirements and solutions concisely, as well as providing a communication vocabulary for designers [Gamma et al. 1994; Buschman et al. 1996]. The description of architectures using patterns makes them easier to understand, provides guidelines for design and analysis, and can define a way of making their structure more secure.

The aim of a Reference Architecture is to provide a guide for developers, who are not security experts, to develop architectures for concrete versions of the system or to extend such system. We describe the Browser ARchitecture as a Reference Architecture (RA) conformed of architectural patterns. An RA is created by capturing the essentials of existing architectures and by taking into account future needs and opportunities, ranging from specific technologies, patterns and business models. The pattern diagram [Buschman et al. 1996] in Figure 1 shows relationships between patterns and we can see how different models relate to each other, where round rectangles represent patterns and the arcs indicate dependencies between patterns.

We started to build a Reference Architecture in a previous work (Figure 1), and now we are trying to extend it by creating new architectural patterns for our RA, misuse patterns to describe threats and finding security patterns to build a Security Reference Architecture (SRA). We are currently building a SRA, which is a Reference Architecture where security services have been added in appropriate places to provide some degree of security for a specific system. The basic approach we will use to build a Security Reference Architecture is the application of a systematic methodology from [Fernandez et al. 2006; Fernandez et al. 2011; Fernandez et al. 2015], which can be

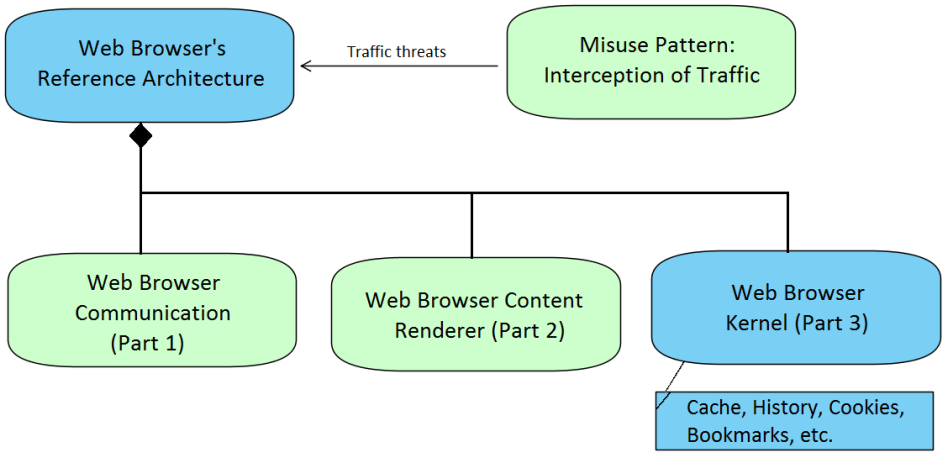

Fig. 1. Pattern Diagram of our work. Finished patterns in green and still developing in blue.

used as a guideline to build secure web browser systems and/or evaluate their security levels. By checking if a threat, expressed as a misuse pattern, can be stopped or mitigated in the security reference architecture, we can evaluate its level of security.

In this work, a Web Content Renderer Pattern is presented (Figure 1). We already built a Browser Infrastructure Pattern as our foundation and now we are extending it [Silva et al. 2016b]. The audience to which our paper is focused are Browser developers, Web Application developers, researchers and IT students, being the first two the most important, since they are the responsible for the implementation and correct use of secure mechanisms.

2. WEB CONTENT RENDERER PATTERN

Intent

The Web Content Renderer describes the architecture for processing the web resources obtained by a browser, into a confortable data structure [Google 2014] or graphical representation, such as a bitmap, which will be displayed to the screen to the user.

Example

Once a resource is obtained by the web browser, it is neccesary to interpret it if the resource is a web pag or save it temporaly or indefinetely in the web browser's host. If it is a web page, the web browser needs to parse it, interpret it and obtain a confortable structure for displaying the resource on the host's screen.

Context

A browser is used to access services or information in the Internet. A browser starts by accepting a URL from a user (explicit request) or interaction with a resource (implicit request) and sending it to the corresponding IP address. It also receives an answer to this request. Then, web browsers need to parse and interpret the resource to obtain a graphic representation of the resource for the user.

Problem

Requests for resources, such as files, data, images, etc., from providers/servers may be in a specific format, the challenge is to find a way to make them visible on the user's browser. In this case, if appropiate tools are not available, the resource cannot be

helpful because it cannot be seen correctly. How can the host provide these functions? The solution to this problem must resolve the following forces:

— *Transparency*: The user should not be concerned about how a request is manipulated to get its result on the screen.
— *Stability*: The browser must be capable of displaying the resource even in the prescence of syntax errors, like invalid HTML5.
— *Isolation*: If a request is done, this is with the Browser User permission and should not cause an undesirable behavior with other obtained resources. Also, if an error exists in one resource this should not affect or modify others in unexpected ways.
— *Heterogeneity*: It does not matter the type of provider with which the browser communicates, it should be possible to interact with whatever type, and it should be possible to show appropriately the content of the obtained resource.
— *Timeliness*: The obtained resource should be displayed within a reasonable time frame; otherwise, a user of the browser would have a bad experience.

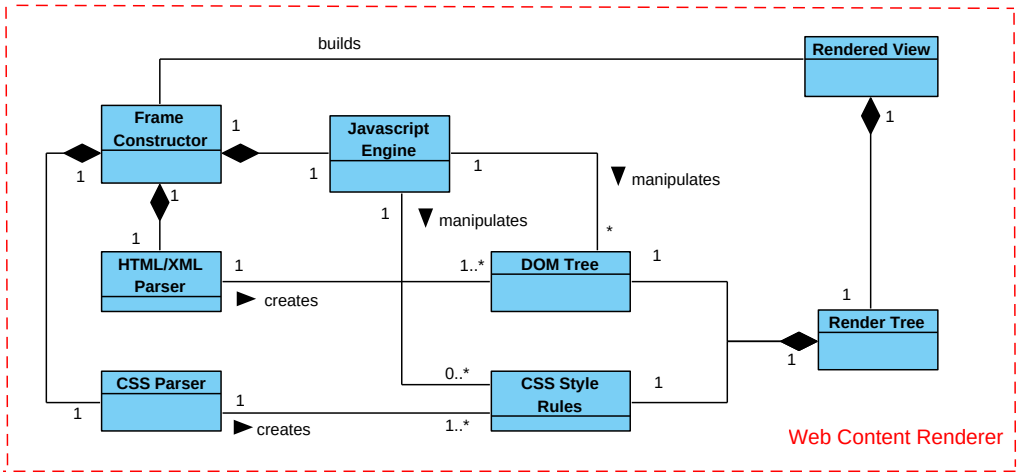

Fig. 2. High-level Components of the Web Content Renderer.

Solution

Provide the Web Browser, with the functions needed to understand requests. Our solution is directly related to our Web Browser Communication pattern [Silva et al. 2016b], where we talk about how the browser's components communicate between each other when a resource is requested.

Structure. Figure 2 shows a **Frame Constructor** in charge of building a **Rendered View** of the requested resource and communicating it to the **Browser Kernel**. The **HTML/XML Parser** is in charge of parsing a web resource and obtain an object representation of it, called a **DOM Tree**. A **CSS Parser** does the same job as the **HTML/XML Parser**, but creates **CSS Style Rules** to change visual aspects of the resource to display. The **DOM Tree** is a dynamic structure that can change anytime while the resource is needed, by using a scripting language that is interpreted

and executed by the **Javascript Engine**; **CSS Style Rules** can also change in the same way, since a script languages, like javascript, can change styles dynamically. A **Rendered View** is the grapgical representation or data structure representing a resource obtained from the **Browser Kernel**, where a **Render Tree** is the first step to obtain it. The **Render Tree** is a dynamic structure that links the **DOM Tree** and **CSS Style Rules** together.

Dynamics. Some use cases are the following:

— Load Resource (actor: Browser Kernel)
— Execute User action (actor: Browser Kernel)

We show in detail Load Resource below. (Figure 3):

Summary. A URL resource which can be obtained by using the HTTP protocol, as required by the Provider. The Browser Kernel will be in charge of sending a retrieved resource to the Controlled Process, which is instantiated as a Web Content Renderer for displaying the resource appropriately.

Actor. Browser Kernel

Preconditions. The Host must have one or more Browser Kernels, for different instanced web browsers. In addition to being connected to a network or the Internet. The Provider to which the browser will contact must also be available. The Web Content Renderer must consider that a web resource can be in many different formats, so it should be capable of displaying them accordingly.

Description

(1) First, the Browser Kernel sends a resource to the Controlled Process, which is instantiated as a Web Content Renderer. The Frame Controlled receives each request for rendering a new resource.
(2) The Frame Constructor uses its HTML/XML Parser to understand the resource. During the parsing process new resources will be needed such as images, plugins, content, etc., where each one of them will do a Load Resource use case. In parallel the tokenization process creates a dynamic structure called as the DOM Tree, which is returned to the Frame Constructor. The same process is done by a CSS Parser to obtain CSS Style Rules.
(3) If scripts are found on the resource while parsing it, the Javascript Engine has the job of parsing, interpreting and executing them. This action can change the DOM Tree and the CSS Rules Styles already built.
(4) Finally, the Frame Constructor is in position for building a confortable data structure or graphical representation of the resource to be rendered. The DOM Tree and CSS Styles Rules are linked together into a Render Tree, and then a process called Layout and Painting is called to get a bitmap [Google 2014; Mozilla 2000] for the Browser Kernel.

Alternative flows

— The HTML/XML Parser fails in getting the DOM Tree from the resource.
— The resource does not exist, so an **Error page** should be created.
— The execution of Javascript or the rendering process is cancelled.

Postconditions. The Browser Kernel obtains a data structure or graphical representation of the obtained resource.

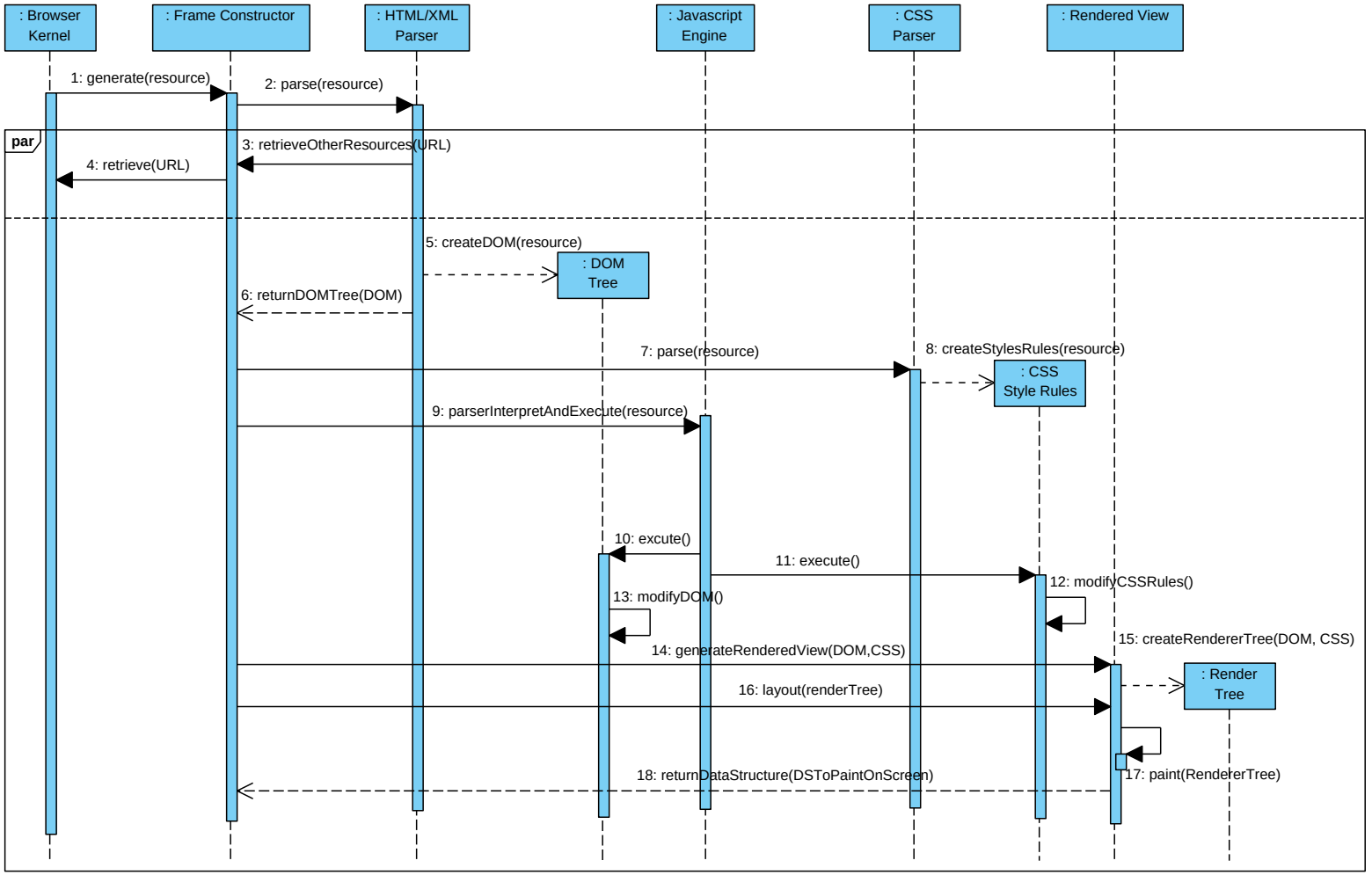

Fig. 3. Sequence Diagram: Load Resource.

Implementation

— The separation of the components of the *Browser* in various processes, with different levels of access is called a Modular Architecture [Vrbanec 2013]. This enables the separation of concerns in the browser, which gives greater stability, isolation, safety and speed. There are other software architectures used by browsers, but currently the use of a modular architecture is rising.
— Each Web Content Renderer instance, each one a separate process, is isolated [Goo 2015; 2009] from each other. This happens so the Same Origin Policy (SOP) [W3C 2010] is enforced in each instance defined by the **origin** of the resources: by its domain, scheme and port. The mentioned policy is the minimum security mechanism a browser has while requesting cross-origin resources, and divides the different kind of contents so they cannot interfere with each other. To enforce the Same Origin Policy, Google Chrome, Firefox and Internet Explorer use different schemes [Crowley 2010; Reis and Gribble 2009; Jackson and Barth 2008].
— Google and Internet Explorer have their rendering engine implemented with a sandbox mechanism. This security feature makes every renderer instance have low privileges in the system, so they cannot interacts in a dangerous way with the system. This defense mechanism enforces a least privilege policy, so the file system can be safe against threats that try to take advantage of web pages vulnerabilities that lead to browser attacks.

Consequences

The Web Content Renderer pattern provides the following benefits:

— *Transparency*: All the process for rendering a resource is done without user interaction.
— *Stability*: Even if a resource (HTML for example) is invalid, the rendering process will try to get a visual representation. Other type of resource could be handled the same way or by requesting again the resource. Since each renderer has its on memory space, even if an error is found it will not affect the whole browser.
— *Isolation*: Resources are placed in different process (different memory spaces) and can only interact if the Same Origin Policy allows it. A Reference Monitor for the DOM Tree and Capabilities for the Javascript Engine are used to enforce this policy (Same Origin Policy - SOP). If a Web Content Renderer has an error while loading a resource, it will not affect other process that have instanced other Web Content Renderers that are part of the same browser.
— *Heterogeneity*: Because each web browser tries to follow the standards of the W3C [World Wide Web Consortium - W3C 1994], every page that follows these guidelines can be viewed, as well as other resources.
— *Timeliness*: Since a renderer engine will be in charge of rendering a particular page or pages (if bounded by the same origin policy as in Google Chrome), the time it will take to load several resources in differents renderer engines will be less compared if only one renderer is used for the same task.

This pattern has the following liabilities:

— Since many independent processes are used to create a Web Content Renderer for rendering a resource (depending on the scheme using the browser), it is possible that a lot of the host's resources are used to keep everything open. A modular architecture requires heavier data structure for multiprocessing and keeping all process working, if is compared with a monoprocess architecture with multithreading.
— Resources from providers which do not comply with the specifications of the W3C, will be displayed incorrectly by the Web Browser.

Example Resolved

With the given pattern it is now possible to display and navigate smoothly to all resources on the Internet we want. Independent on the type of web resource, it will be possible to display it appropriately in the browser.

Known Uses

— Google Chrome is based on a modular architecture, where each Renderer Process communicates with the Browser Kernel [Google Chromium 2008]. Blink is the implementation of its renderer engine and is based on WebKit.
— Gecko [Mozilla 2000] is the name for the Renderer in the monoprocess architecture Firefox, it is written in C++ and can be used to render the user interface of other applications such as Thunderbird.
— Internet Explorer, a proprietary browser, does not give much information about its structure or details of its implementation; [Crowley 2010] addresses Loosely-Coupled architecture [IE8 2008] and its components, like Trident its Rendering Engine, but without giving further details. Microsoft's new web browser, Microsoft Edge, integrates a new renderer too, called EdgeHTML [Microsoft 2015].

Related Patterns

— The Web Browser Infrastructure pattern represents the subsystem that represents the Web browser central component [Silva et al. 2016b].
— The Reified Reference Monitor [Fernandez 2013], describes how to enforce authorization rights when a subject requests access to a protected object or service and returns a decision (response).

3. CONCLUSIONS AND FUTURE WORK

A Web browser appears to be a medium complexity software for users and developers without security experience, but unfortunately this piece of software allows a variaty of attack vectors, to the user as well as the system with which interacts. Therefore it is important to understand its structure and how it interacts with internal and external Stakeholders.

A part of our Reference Architecture has been built through the abstraction of documentation through the Browser Infrastructure pattern. We created our first architectural pattern for the infrastructure of *Web Browser* to help others understand, holistically, the components, interactions and relationships of this system. Furthermore, it has been possible to characterize the Stakeholders and one of the most important use cases. From what we know, this is the second Reference Architecture for the *Browser*. The reference model obtained in [Grosskurth and Godfrey 2005] express the type of architecture used in the nineties until 2009 (approximately). However, our proposal represents the current implementation used in browsers: a **Modular Architecture**. The proposed work allows a better understanding of this system called web browser by using our partial Reference Architecture, which is also helpful to understand existing threats. Also, as it is not tied to specific implementations, it is possible to generalize some results to other browsers.

Future work is finishing the Reference Architecture for *Web Browsers*. Other patterns related to the Browser Infrastructure pattern will be obtained in order to complete the Reference Architecture, such as the Web Content Renderer (this paper) and Browser Kernel pattern.

We plan to build more [Silva et al. 2016a] misuse patterns for the Browser Infrastructure Pattern, to continue the study of the possible threats in the *Browser*, as a way to educate developers and stakeholders. At the same time, these patterns will allow

the construction of a Security Reference Architecture for the browser. In the same line, in addition to finding potential threats in the system, we need to find countermeasures or security defenses to prevent or foresee such threats through security patterns on the reference architecture built. An example of the type of work to be carried out can be seen in [Fernandez et al. 2015].

4. GLOSSARY

— **Frame Constructor:** in charge to put together the results obtained by the parsing process of a HTML and CSS file, as well to execute in the precise time the Javascript code that will manipulate the DOM Tree and CSS Styles rules obtained.
— **Rendered View:** a graphical representation of a Renderer Tree, from the DOM Tree and CSS Style rules obtained from a resource.
— **Bitmap:** a buffer of pixel values in memory (main memory or the GPU's video RAM) [Google 2014].

REFERENCES

2008. IE8 and Loosely-Coupled IE (LCIE) - IEBlog - Site Home. (2008). http://blogs.msdn.com/b/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx

2009. Security/ProcessIsolation/ThreatModel. (2009). https://wiki.mozilla.org/Security/ProcessIsolation/ThreatModel

2015. Site Isolation - The Chromium Projects. (2015). https://www.chromium.org/developers/design-documents/site-isolation

Frank Buschman, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. A system of patterns: pattern-oriented software architecture. (1996).

Matthew Crowley. 2010. *Pro Internet Explorer 8 & 9 Development: Developing Powerful Applications for The Next Generation of IE* (1st ed.). Apress, Berkely, CA, USA.

Eduardo B. Fernandez. 2013. *Security patterns in practice: designing secure architectures using software patterns*. John Wiley & Sons.

Eduardo B. Fernandez, M.M. Larrondo-Petrie, T. Sorgente, and M. VanHilst. 2006. A methodology to develop secure systems using patterns. *Chapter 5 in "Integrating security and software engineering: Advances and future vision"* (2006), 107–126.

Eduardo B. Fernandez, Raul Monge, and Keiko Hashizume. 2015. Building a security reference architecture for cloud systems. *Requirements Engineering* (Jan 2015). DOI:http://dx.doi.org/10.1007/s00766-014-0218-7

Eduardo B. Fernandez, Nobukazu Yoshioka, Hironori Washizaki, Jan Jurjens, Michael VanHilst, and Guenther Pernul. 2011. *Using Security Patterns to Develop Secure Systems*. IGI Global. DOI:http://dx.doi.org/10.4018/978-1-61520-837-1

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design patterns: elements of reusable object-oriented software*. Pearson Education.

Google. 2014. GPU Accelerated Compositing in Chrome - The Chromium Projects. (2014). https://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome

Google Chromium. 2008. Multi-process Architecture - The Chromium Projects. (2008). https://www.chromium.org/developers/design-documents/multi-process-architecture

Alan Grosskurth and Michael W. Godfrey. 2005. A reference architecture for web browsers. 661–664.

Collin Jackson and Adam Barth. 2008. Beware of finer-grained origins. *Web 2.0 Security and Privacy* (2008). http://seclab.stanford.edu/websec/origins/fgo.pdf

Microsoft. 2015. Introducing EdgeHTML 13, our first platform update for Microsoft Edge — Microsoft Edge Dev Blog. (2015). https://blogs.windows.com/msedgedev/2015/11/16/introducing-edgehtml-13-our-first-platform-update-for-microsoft-edge/

Mozilla. 2000. Gecko:Overview - MozillaWiki. (2000). https://wiki.mozilla.org/Gecko:Overview

Charles Reis and Steven D Gribble. 2009. Isolating web programs in modern browser architectures. *Proceedings of the fourth ACM european conference on Computer systems EuroSys 09* 25, 1 (2009), 219. DOI:http://dx.doi.org/10.1145/1519065.1519090

Paulina Silva, Raúl Monge, and Eduardo Fernandez B. 2016a. A Misuse Pattern for Web Browsers: Interception of traffic. *Proceedings of the 5th Asian Conference on Pattern Languages of Programs (AsianPLoP) 2016, Taipei, Taiwan* (2016).

Paulina Silva, Raúl Monge, and Eduardo Fernandez B. 2016b. A pattern for Web Browser Infrastructure. *Proceedings of the 5th Asian Conference on Pattern Languages of Programs (AsianPLoP) 2016, Taipei, Taiwan* (2016).

Tedo Vrbanec. 2013. The evolution of web browser architecture. (2013), 472–480.

W3C. 2010. *Same Origin Policy*. Web page. W3C. https://www.w3.org/Security/wiki/Same_Origin_Policy

World Wide Web Consortium - W3C. 1994. About. (1994). http://www.w3.org/Consortium/