A Reference Architecture for web browsers: Part II, A pattern for Web Browser Content Renderer

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

Currently, most software developments are 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 a *web browser*, which allows access to Internet services, data or operations that a system delivers. Within the browser's main components, a *rendering engine* is in charge of obtaining a convenient data structure as an output for a *browser process*. We developed a Web Browser Communication pattern that describes the infrastructure to allow the communication between a web client (or a web browser) and a server in the Internet. In this paper, we describe the component in charge of the rendering for a obtained web resource within the web browser, named here as **Web Browser Content Renderer**. In this work we have described this component as a pattern, to describe how a browser's *rendering engine* works and interacts with other subsystems. Patterns combine experience and good practices to obtain models that can be used for new designs, to compare and select systems/applications or to teach others. The audience to which our paper is focused are browser developers, web application developers, researchers and teachers, being the first two the most important.

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.

A Reference Architecture (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 aim of a RA is to provide a guide for developers, to develop architectures for concrete versions of some system or to extend such system. We are trying to describe the browser architecture as a RA composed of architectural patterns. The pattern diagram [Buschman et al. 1996] in Figure 1 shows the relationships between our created patterns and we can see how different models relate to each other, where rounded rectangles represent patterns and the arcs indicate dependencies between patterns; for example, Interception of Traffic describes traffic threats.

Since we started to build our Reference Architecture (Figure 1), we are extending it by creating new architectural patterns for our RA and misuse patterns to describe threats and 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 SRA is the application of a systematic methodology from [Fernandez et al. 2006; Fernandez et al. 2011; Fernandez et al. 2016], which can be 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 Browser Content Renderer** pattern is presented (See Figure 1 to see how it fits with our other patterns). We have already documented a **Web Browser Communication** pattern [Silva

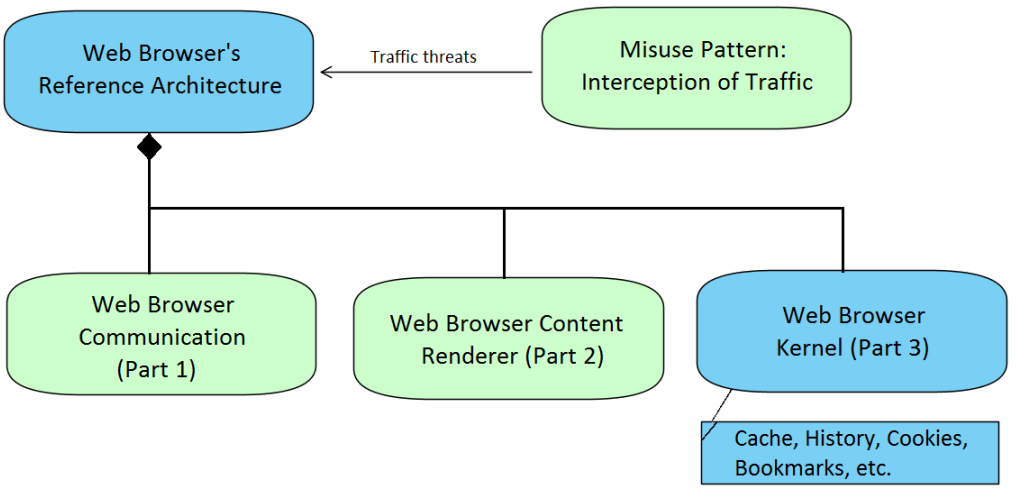

Fig. 1: Pattern Diagram of our current work on the Reference Architecture for the Web Browser. Finished patterns in green and still developing in blue.

et al. 2016b] as our foundation and in this paper we are extending our RA/SRA with this new pattern. The patterns from our RA and SRA were conceived while examining current web browsers like Google Chrome, Internet Explorer and Mozilla Firefox, since these are the most used nowadays. The audience to which our paper is focused are browser developers, web application developers, researchers and teachers, being the first two the most important, since they are the responsible for the implementation and correct use of secure mechanisms.

2. WEB BROWSER CONTENT RENDERER PATTERN

Intent

The Web Browser Content Renderer describes the architecture for converting the web resources obtained by a browser into a convenient data structure [Google 2014] or graphical representation, such as a bitmap.

Example

Once a resource is obtained by the web browser, it is neccesary to interpret it (if the resource is a web page) or save it temporarily or indefinitely in the web browser's host. If it is a web page, the web browser needs to parse it, interpret it and obtain a convenient structure for displaying it 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 an implicit request for interaction with a resource and sending them to the corresponding URL address for that resource or interaction.

Problem

Retrieved resources, such as files, data, images, etc., from providers may be in a specific format; the challenge is to find a way to make them visible on the screen of the user's browser. In this case, if appropriate 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 different formats or syntax errors, like invalid HTML5.

—*Isolation*: If a request is completely processed, 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 other resources in unexpected ways.

—*Heterogeneity*: It does not matter the type of provider with which the browser communicates, it should be possible to interact with any 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.

Solution

Provide the web browser, with the functions needed to understand responses and then obtain a confortable format for the host's Graphics Processor Unit (GPU) to display it to the browser's user. Our solution is directly related to our **Web Browser Communication** pattern [Silva et al. 2016b] , where we show the browser's components and the possible interactions between them when a resource is requested. The **Web Browser Content Renderer** is a specialization of a **Controlled Process**, and it will communicate with the **Browser Kernel** of the web browser to do its job, which is rendering a resource.

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** . When a response from a provider needs to be rendered and shown to the user, the **Frame Constructor** will receive from the **Browser Kernel** the content of the resource. The **HTML/XML Parser** is in charge of parsing a web resource and obtain an object representation of it, called a **DOM Tree** (DOM: Document Object Model). 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 be displayed. 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** [1]. **CSS Style Rules** can also change in the same way, since script languages, like Javascript, can change styles dynamically. A **Rendered View** is the graphical representation or data structure representing a resource obtained from the **Browser Kernel** . A **Render Tree** is a dynamic structure that links the **DOM Tree** and **CSS Style Rules** together, to create a **Rendered View** or **bitmap**.

Glossary

—**Frame Constructor:** in charge of putting together the results obtained by the parsing process of HTML and CSS files, as well of executing at the right time the Javascript code that will manipulate the DOM Tree and the CSS Styles rules.

—**Rendered View:** a graphical representation or data structure of a Render Tree, where this latter was built 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].

[1]Since most browsers support Javascript as its scripting language, we have decided using this name

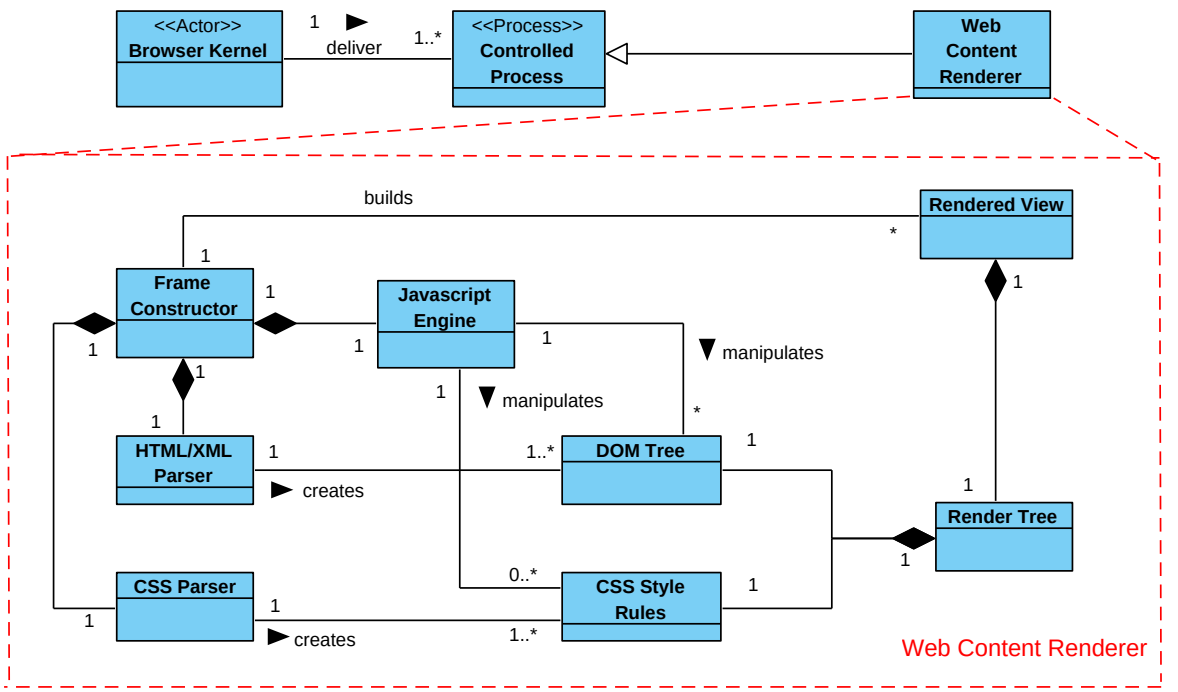

Fig. 2: High-level Components of the Web Browser Content Renderer.

Dynamics. Some use cases are the following:

—Load Resource (actor: Browser Kernel): which includes the rendering of a resource.

—Execute User action (actor: Browser Kernel): changes to the loaded resource done by the user.

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

Summary. The **Browser Kernel** sends a retrieved resource to the **Controlled Process** (Figure 2) , which is instantiated as a **Web Browser Content Renderer** for displaying the resource appropriately. The resource is obtained by the **Browser Kernel** from some URL.

Actor. Browser Kernel

Preconditions. In addition to being connected to a network or the Internet, the Host must have one or more **Browser Kernels**, each one instantiated as a different web browser. Also, the Provider which the browser will contact must be available.

Description

(1) The **Browser Kernel** delivers a resource to the **Controlled Process**, which is instantiated as a **Web Browser Content Renderer**. The **Frame Constructor** 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 perform a Load Resource use case. Step 4 of the sequence diagram in Figure 3 oversimplifies the request for secondary resources. Where in reality the **Web Browser Content Renderer** will parallelize loading of secondary resources, or would communicate in a controlled manner (passing through a Reference Monitor) with the **Browser Kernel** to other **Controlled Processes** to use plugins or extensions.

(3) In parallel the tokenization process creates a dynamic structure called the **DOM Tree**, which is returned to the Frame Constructor. The same process is done by a **CSS Parser** to obtain **CSS Style Rules**.

(4) If scripts are found while parsing the primary resource, 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.

(5) Finally, the **Frame Constructor** is in position for building a convenient 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, and an **Error page** will be created for prompting the user's new action.

—The execution of Javascript or the rendering process is cancelled. If the rendering process has not finished, the displaying of the resource will stop.

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

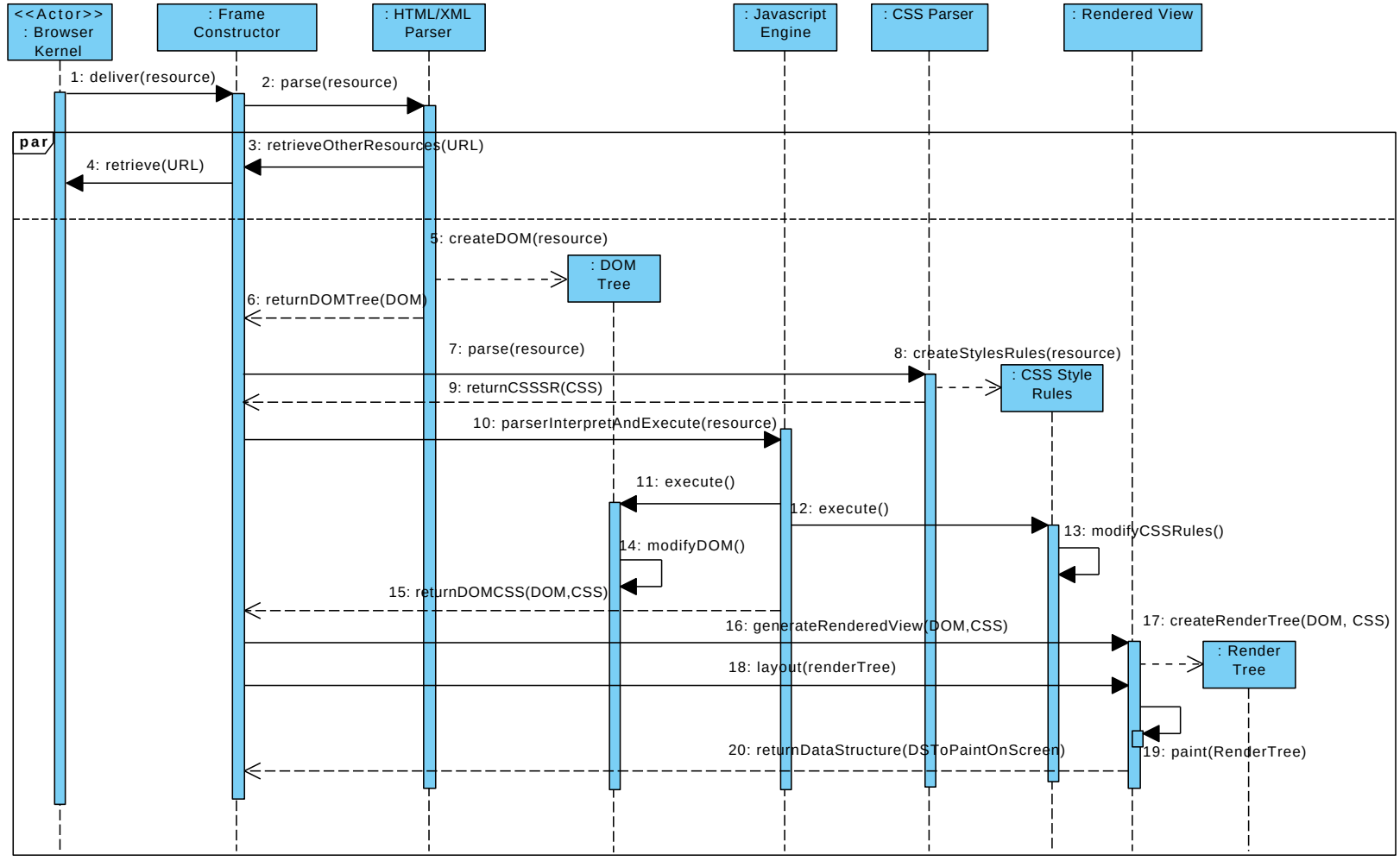

Fig. 3: Sequence Diagram: Load Resource.

Implementation

—Most browsers use Javascript for its scripting language. For those who do not use it, it is possible that the rendering engine is only in charge of parsing the resources without building dynamic structures like the DOM, CSS Styles Rules or the Render Tree.

—The separation of the components of the *Browser* in various processes, with different levels of access, comes from the more general concept called a Modular Architecture [Vrbanec 2013]. This enables the separation of concerns in the browser, which may give greater stability, isolation, security and speed. There are other software architectures used by browsers, but currently the use of a modular architecture is rising.

—Each Web Browser Content Renderer instance, is a separate process isolated from others [Goo 2015; 2009]. In this way, the Same Origin Policy (SOP) [W3C 2010] is enforced in each instance defined by the origin of the resources: by their 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, avoiding mutual interference. To enforce the SOP, Google Chrome, Firefox and Internet Explorer use different schemes [Crowley 2010; Reis and Gribble 2009; Jackson and Barth 2008].

—Google Chrome and Internet Explorer have their rendering engine implemented with a sandbox mechanism. This security feature makes every renderer instance to have restricted privileges on the system, so they cannot interact 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 exploit web page vulnerabilities.

Consequences

The Web Browser Content Renderer pattern provides the following benefits:

—*Transparency*: 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. Another type of resource could be handled the same way or by requesting again the resource. Since a Web Browser Content Renderer instance is separate process, each has its own memory space, even if an error is found while loading a resource it will not affect the whole browser and will be able to continue working.

—*Isolation*: Resources are placed in different processes (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 Browser Content Renderer makes an error while loading or processing a resource, it will not affect other processes that have instantiated the Web Browser Content Renderer pattern 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, independent of the type of browser.

—*Timeliness*: Since a renderer engine will be in charge of rendering a particular page or pages (if bound by the SOP as in Google Chrome), the time it will take to load several resources in differents renderer engines will be less compared to the worst case when 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 Browser 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 active. A modular architecture requires heavier data structures for multiprocessing and keeping all processes working, 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 content to the user and/or navigate smoothly through the obtained resources. Independent of the type of web resource, it will be possible to obtain an abstraction of it and 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 of 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, called EdgeHTML [Microsoft 2015].

Related Patterns

—The Web Browser Communication pattern presents the components of the web browser and how they communicate with each other when a resource is requested [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).

—The Client Rendering Pattern by Vilar et. al [Vilar et al. 2015] (which proposed five patterns to organize GUI widgets based on domain model meta data), resolves the problem of developers to compose widgets indirectly in the client layer of web Enterprise applications. The Rendering Engine, which is represented here as the Web Browser Content Renderer, is part of the architecture diagram for the Client Rendering Pattern.

3. CONCLUSIONS AND FUTURE WORK

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

A first part of our RA has been formulated with the **Web Browser Communication** pattern, and now we have presented our second architectural pattern: the **Web Browser Content Renderer**. These two patterns abstract the infrastructure of a web browser to help others understand holistically the components, interactions and relationships of this system. The proposed work allows a better understanding of browser's structure and behavior within our partial Reference Architecture, which is also helpful to understand existing threats in the browser. Furthermore, since a pattern is generic by definition, our solution is not tied to specific implementations and is possible to generalize some results to other browsers.

As a we are finishing a complete Reference Architecture for web browsers, another pattern related to the Web Browser Communication pattern will be obtained in order to complete the Reference Architecture, called the Browser Kernel pattern.

We plan to build more [Silva et al. 2016a] misuse patterns for the Web Browser Communication 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 to prevent or foresee such threats through security patterns on the reference architecture. An example of the type of work to be carried out can be seen in [Fernandez et al. 2016].

4. ACKNOWLEDGEMENTS

We thank our shepherd, Christopher Preschern, for his useful comments that significantly improved the quality of the paper. We also thank Elissaveta Gourova for supervising our paper shepherding. We would also like to gratefully thank to the participants of the EuroPLoP Writers Workshop: Tobias Rauter, Veli-Pekka Eloranta, Poonam Ponde, Shailaja Shirwaikar, Bogdana Botez, Andreas Sinnhofer, Roland H. Steinegger, Lukas Reinfurt. This work was partially supported by CONICYT (grant Fondecyt 1140408).

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. 2016. Building a Security Reference Architecture for Cloud Systems. *Requir. Eng.* 21, 2 (June 2016), 225–249. 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 B. Fernandez. 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 B. Fernandez. 2016b. A Reference Architecture for web browsers: Part I, A pattern for Web Browser Communication. *Proceedings of the 5th Asian Conference on Pattern Languages of Programs (AsianPLoP) 2016, Taipei, Taiwan* (2016).

Rodrigo Vilar, Delano Oliveira, and Hyggo Almeida. 2015. Rendering Patterns for Enterprise Applications. In *Proceedings of the 20th European Conference on Pattern Languages of Programs (EuroPLoP '15)*. ACM, New York, NY, USA, Article 22, 17 pages. DOI:http://dx.doi.org/10.1145/2855321.2855344

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. (1994). http://www.w3.org/Consortium/