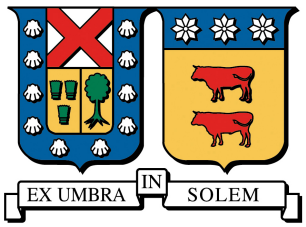
Universidad Técnica Federico Santa María
Departamento de Informática
Valparaíso, Chile



A Security Reference Architecture for Web Browser - Unification of Threat and Defense Concepts associated

Paulina Andrea Silva Ghio

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree:
Master of Science (MSc.) in Informatics Engineering

Guide Professor: Raúl Monges
X Professor: Javier Cañas

July 6, 2015

Agradecimientos

Resumen

El Web Browser es una de las aplicaciones más usadas - *killer app* - y también una de las primeras en aparecer en cuanto se creó el Internet (Década de los 90). Por lo mismo, su nivel de madurez con respecto a otros desarrollos es significativo y permite asegurar ciertos niveles de confianza cuando otros usan un Web Browser como cliente para sus Sistemas.

Actualmente muchos desarrollos de software crean sistemas que están conectados a la Internet, pues permite agregar funcionalidades al sistema y facilidades para sus *Stakeholders*. Esto lleva a depender de un cliente web, cómo un *Web Browser* que permita el acceso a los servicios, datos u operaciones que el sistema entrega. Sin embargo, la Internet influye en la superficie de ataque del nuevo sistema que se implementó, y lamentablemente tanto Stakeholders como muchos desarrolladores no están al tanto de los riesgos a los que se enfrentan.

En esta Memoria presentada al Departamento de Informática (DI) de la UTFSM[1] Casa Central, se al incursionará en el ámbito de la seguridad del Web Browser, con el objetivo de obtener documentos formales que servirán como herramientas a personas que Desarrollen Software y hagan un fuerte uso del Navegador para las actividades del sistema desarrollado.

Abstract

[1]Universidad Técnica Federico Santa María

Contents

Chapter 1

Introduction

1.1 General Context

In 1989 and 1990, Sir Tim Berners-Lee created the *World Wide Web* concept and also the first *Webserver*, *Web Browser* and the first *Web Pages*. Before today's complex systems appeared, the *Web Browser* allowed to access static pages and do some restricted actions with the technology of that time. Currently the *Browser* is the favorite tool for everybody to access the Internet, it lets you buy tickets for a movie, do videoconferences and so more tasks which invite to new ways of interacting and communicate.

In recent times the *Web Browser* Market has grown a lot, this is mainly due to its robust construction and the quantity of years they have been developing in the Software Development Industry. The most known *Web Browsers* are: Google Chrome/Chromium, Firefox, Internet Explorer, Opera and Safari; been the first 3 of them the subjects used in this work.

The *Web 2.0* began with the intensive use of AJAX technologies, and so this has allowed a new kind of symbiosis between the user, the *Browser* and the *Web Server*, which communicates with each one. Recently, they have changed the name to *Web 3.0*, for the extensve of use of Artificial Intelligence and Recommendation Systems to create new kinds of media content for the user.

1.2 The Problem

Every Software Development team acts differently and are not equal. For each new project it is neccesary to see what type of process will the team use, what people will be

part of the team work, which economical conditions will be exposed the project, which are the *Stakeholder* behind the project, and some many more variables that could be critical to the success of the team. According to that, systems could be too simple or otherwise too, which could lead to require certains Methodologies that could ensure that all *Functional REquirements* as *Non-Functional Requirements* of the system to develop are met. However, a recurrent problem that still exist in most Software built has a lot of *flaws* and *errors*, which generates vulnerabilities that could be exploited by attackers. Mainly this is due Developers build complex systems without taking care of Security in the first pahses of the Software Development.

Un fenómeno en la literatura llamado *Zero-day attack*, se refiere al momento clave donde un atacante explota una vulnerabilidad - hasta ese momento desconocida - de algún sistema (importante o no), y que si no es parchado lo antes posible puede comprometer no solo a sistemas, si no también a los usuarios que hacen uso de éste. Junto con lo anterior muchas veces ocurre que aunque se corrijan estos nuevos ataques, no todos los sistemas que podrían llegar a necesitar del mismo parche para protegerse del ataque, realizan la actualización y su adecuada configuración para así protegerse de una posible amenaza que explote la vulnerabilidad recientemente encontrada. Si bien un **Zero-day Attack** es un evento que podría no ocurrir tan repetidamente, dado que se produce por el largo estudio llevado por el atacante, sobre el sistema a vulnerar, existen otras formas de comprometer a un sistema. Muchas veces al desarrollar sistemas, se prefiere utilizar API's[1] de otros sistemas para poder incluir funcionalidades ya implementadas, fomentando así el Reuso de piezas de Software. Si bien lo anterior es una buena práctica, si el sistema no cuenta con las medidas de seguridad necesarias, estas piezas podrían ser causa de amenazas de seguridad que terminarían por corromper el sistema y en consecuencia podría causar una pérdida monetaria a los *Stakeholders*.

En general lo expuesto anteriormente ejemplifica perfectamente lo que tienen que lidiar los equipos de trabajo en proyectos de Desarrollo de Software, cuando dentro de sus preocupaciones la seguridad queda como un trabajo extra y no como parte del desarrollo completo. Bien es sabido que un proyecto en producción que presente problemas que involucren a varias entidades, el costo asociado puede llegar a ser altísimo [**?**], sin olvidar que podría llegar a afectar la Confidencialidad, Integridad y Disponibilidad de los datos de los involucrados con el sistema [**?**]. Por esto mismo, es imperante que sean entendidos, desde el comienzo, los *concerns* de los *Stakeholders* y los Requerimientos de Seguridad asociados, y que además todos los involucrados los entiendan perfectamente. La literatura que habla de la construcción de *Secure Software* o Software Seguro, indica que los practicantes del Desarrollo de Software deben entender, en gran medida, los problemas de seguridad que podrían llegar a ocurrir en sus sistemas. No basta con saber como unir las piezas, no basta con que cada pieza de por si sea segura, si los componentes del sistema no actuan de

[1] Application Programming Interface

forma coordinada probablemente éste no será seguro [?], dado que la seguridad es una Propiedad Sistémica que necesita ser vista de manera holística y al inicio del proceso.

1.3 Motivation

Con la aparición de la *Web 2.0 y 3.0*, con el uso de *AJAX*, inteligencia artificial y sistemas de recomendación, permitieron nuevas formas de interacción entre usuarios y sistemas, lo que causó que el browser fuera usado extensivamente en los nuevos Desarrollos de Software dado que:

- Permite disminuir los costos de construir un programa Cliente (desde cero) para el usuario del sistema.
- Actualmente la Seguridad implementada en los *Web Browser* es bastante buena, dado que existen grandes compañias que se aseguran de ello (Google, Microsft, Mozilla entre las más conocidas).
- El *browser* es una herramienta indispensable. La mayoría de los sistemas que lo usan en la vida cotidiana son de tipo: *online banking*, declaración de impuestos, promoción de empresas o tiendas, compras, y mucho más.

Sin embargo los sistemas que dependen del uso del *Browser*, deben de tener en cuenta las posibles amenazas de seguridad a las que se enfrentarán por el solo hecho de usarlo. Para un proyecto de gran envergadura, sería un error no tener en consideración los posibles peligros que trae el uso del *Browser*, y es el deber de todo integrante del equipo de Desarrollo tener el conocimiento de la seguridad del Cliente Web. El entendimiento de la estructura subyacente del Web Browser podría asegurar que las personas que trabajen en el desarrollo, comprendan los *trade-off* al momento de diseñar un Software que necesite la colaboración del Navegador Web [?, ?, ?].

En [?] menciona que en cursos de Ingeniería de Software los estudiantes no aprenden mucho sobre Principios de Diseño en Seguridad, ni técnicas que permitan una segura implementación de código, a menos que lo necesiten en algún momento. Más aún, la falta de este tipo de conocimiento puede hacer creer que la seguridad es un requerimiento que puede o no ser tomado en cuenta al comienzo del Desarrollo. En este trabajo el enfoque es otro, la seguridad es una propiedad sistémica que debe ser tomada en cuenta desde el inicio del sistema [?, ?, ?, ?].

Este trabajo tiene una motivación principal. Ésta es ayudar a quién lo necesite con el conocimiento necesario para entender el funcionamiento y construcción del Cliente, el Web Browser, los beneficios detrás de la Seguridad implementada en el Browser y de los peligros existentes de los que nos protegen. De esta manera se espera que

alguien que lea este trabajo, tanto Estudiantes como Desarrolladores de Softwares, obtengan el conocimiento necesario al momento de trabajar junto con el Navegador Web al realizar un Desarrollo de Software que dependa de éste.

1.4 Proposal

1.5 Hypothesis

1.5.1 Hypothesis statement

The hypothesis proposes:

It is possible to define a Security Reference Architecture for Web Browser which abstracts and captures principal structural aspects and its behaviors, to express known Misuse and Security Patterns related.

1.5.2 Objectives

General

Specific

1.6 Validation

As a Reference Architecture is not implementable, neither a Security Reference Architecture; both are abstract models and cannot be evaluated with respect to security or performance through experimentation or testing. A Reference Architecture and Security Reference Architecture are similar to a pattern and it has a similar use, it is a paradigm to guide implementation of new systems or evaluation of existing systems.

Their evaluation is based on how well they represent the relevant concepts of the systems they describe, how well they handle abstract threats, how complete they are, how precise they are, how they can be applied to the design or evaluation of systems, and how useful they are for other relevant functions. Their final validation comes from experts and practitioners who can find them useful and convenient to build concrete architectures.

In the particular case of this thesis work, validation is also achieved by being reviewed by experts in the field of patterns and computing systems, being accepted, presented and published in several international conferences. The individual papers were strongly discussed with the purpose of improve them before publish them.

1.7 Methodology

We used the following methodology to work on this Thesis, so a reproduction can be done by other interested party.

1. Understanding the Conceptual Framework of related concepts.
2. Capturing and studying the State of Art about the Web Browser, focused specially on security.
3. Identify concepts, actors, compoenents, interactions and functions were identified and unified.
4. Create Architectural Patterns which defines compoenents and responsabilities, with the goal of being unified in a Reference Architecture.
5. Create Misuse and Security Patterns by using the Reference Architecture.
6. The proposed Architecture and Security Application of the same needs to be validated. This is done by submitting a paper with them and then being accepted, presented and published in several international conferences wherein pattern and computer science experts reviewed them.

1.8 Estructura del Documento

El presente documento trata del trabajo de Memoria que se divide en las siguientes partes:

- En el capítulo **??**...
- Luego de tener un extenso conocimiento de lo que actualmente es conocido como **Web Browser**, el capítulo **??**

Chapter 2

Framework Conceptual

2.1 Definiciones Básicas

Para empezar este estudio es necesario introducir ciertas nociones y lenguaje que se usarán durante todo el documento. Estos conceptos son usados en la Seguridad y Desarrollo de Software, y son extendibles para lo que se verá en este estudio.

- Seguridad - *Security*:
  Es una Propiedad que podría tener un sistema, donde asegura la protección de los recursos e información, en contra de ataques maliciosos desde fuentes externas como internas. La Seguridad también involucra controlar que el funcionamiento de un sistema sea como debería ser, y que nada externo o interno genere un error.

- Error - *Error*:
  Es una acción de caracter humano. Éste se genera cuando se tienen ciertas nociones equivocadas, que causan un Defecto en el Sistema o Código.

- Defecto - *Defect*:
  Es una caracterítica que se obtiene a nivel de Diseño, cuando una funcionalidad no hace lo que tiene que realmente hacer. Según la IEEE CSD o *Center for Secure Design* [**?**], un defecto puede ser subdividido en 2 partes: falla o **flaw** y **bug**, donde la primera tiene que ver con un error de **alto nivel**, mientras que un bug es un problema de implementación en el Software. Una falla es menos notoria que un bug, dado que ésta es de caracter abstracto, a nivel de diseño del Software.

- Falla - *Fail*:
  Es un estado en que el Software Implementado no funciona como debería de ser.

- Vulnerabilidades - *Vulnerability*:
  Es una debilidad inherente del sistema que permite a un atacante poder reducir el nivel de confianza de la información de un sistema. Una vulnerabilidad convina 3 elementos: un **defecto** en el sistema, un **atacante** tratando de acceder a ese defecto y la **capacidad** que tiene el atacante para llevarlo a cabo. Particularmente las vulnerabilidades más críticas son documentadas en la *Common Vulnerabilities and Exposures* (CVE) [**?**].

- Superficie de Ataque - *Attack Surface*:
  Es el conjunto de todas las posibles vulnerabilidades que un sistema puede tener, en un cierto momento, para una cierta versión del sistema, etc.

- Amenaza - *Threat*
  Es una acción/evento que se aprovecha de las vulnerabilidades del sistema, debilidades, para causar un daño, y que dependiendo del recurso al que afecte el daño puede o no ser reparable.

- Ataque - *Attack*
  Es el éxito de la amenaza en el aprovechamiento de la vulnerabilidad (explotación de ésta), de tal forma que genera una acción negativa en el sistema y favorable para el atacante.

- *Exploit*:
  Usar una pieza de software para poder llevar a cabo un ataque sobre un objetivo, intentando **explotar** la vulnerabilidad de éste. Este tipo de acción permite en consecuencia obtener control en el sistema computacional, en donde la vulnerabilidad permitió su acceso.

- Ingeniería Social - *Social Engineering*
  El acto de manipular a las personas de manera que realicen acciones o divulguen información confidencial. El termino aplica al acto de engañar con el propósito de juntar información, realizar un fraude, u obtener acceso a un sistema computacional. La definición anterior encontrada en Wikipedia es extendida por el autor del libro "The Social Engineer's Playbook" [**?**], donde agrega que además la Ingeniería Social involucra el hecho de manipular a una persona en realizar acciones que finalmente no son para beneficiar a la víctima. Un ataque de éste tipo también puede llegar a ser realizado tanto **cara a cara**, como de forma indirecta. Pero el autor del libro indica que siempre hay un **contacto** previo con la víctima.

- Confidencialidad - *Confidentiality*
  Característica o propiedad que debe mantener un sistema para que la información privilegiada de alguna entidad que depende de tal sistema, no sea develada a nadie más que al que le pertenece la información.

- Integridad - *Integrity*
  Característica o propiedad que asegura que la información no será modificada/alterada nada más que por la entidad a quién le pertenece y con el previo consentimiento de éste.

- Disponibilidad - *Availability*
  Característica o propiedad que permite que la información esté disponible para quién lo necesite, en el momento que sea. La imposibilidad de obtener data en un cierto instante de tiempo, conlleva a la perdida de esta propiedad.

- *Phishing*
  Técnica de Ingeniería Social. Mediante el uso de correo elentrónico, links (url's), acortamiento de urls y otras herramientas, se busca que una victima visite un sitio o aprete un link de manera que se de la **autorización explicita** del usuario para descargar código malicioso o enviar datos a un servidor malicioso. El objetivo de esta técnica es poder obtener información valiosa de la victima o relizar algún daño en el cliente web.

- *Malware*
  Software creado para realizar acciones maliciosas en la data o sistema de un usuario. Puede ser instalado tanto de forma discreta como indiscreta, siendo la segunda opción causada a través de un ataque previo a cierta vulnerabilidad que permitió la instalación del malware, sin el consentimiento del usuario privilegiado del sistema.

- *Man-in-the-Middle*
  Ataque que causa una pérdida en la Confidencialidad de la información que es revelada. La causa de este ataque puede ser tanto:

  - Por técnicas de Ingeniería Social, entregano un certificado malicioso que el usuario acepta con o sin intención.
  - A través de vulnerabilidades del sistema que debieron ser explotadas antes para causar el ataque MiTM.

2.2 Browser

2.2.1 Arquitectura Cliente/Servidor

La web emplea lo que se conoce como una Arquitectura Cliente-Servidor, donde la comunicación entre ambas entidades se basa mediante mensajes de *request-response* o solicitud-respuesta. Con el tiempo la forma en que se comunican estos programas

a cambiado, desde iniciar solicitudes de forma secuencial e independiente, hasta solicitar asíncronamente varias peticiones. La evolución que ha tenido el cliente web ha permitido una mejor experiencia para el usuario, pero que conlleva ciertos riesgos que es necesario que el que usa el Browser sea consciente. De la misma manera que podemos afectar a un servidor a través de las solicitudes, las respuestas que el servidor envía al cliente pueden tener consecuencias graves [**?**].

2.2.2 HTTP: Hypertext Transfer Protocol

El Protocolo de la capa de Aplicación conocido como HTTP fue creado en los años 90 por el **World Wide Web Consortium** y la **Internet Engineering Task Force**, define una sintaxis y semántica que utilizarían los software basados en una arquitectura Web para comunicarse. El protocolo sigue un esquema de pregunta-respuesta o *request-response*, donde un cliente solicita un recurso que el servidor posee, y el servidor entrega una respuesta de acuerdo al recurso solicitado. La forma en que se localiza un recurso es mediante la dirección URL o *Uniform Resource Locator*

HTTP Headers

HTTP es la implementación de la capa de aplicacón del modelo OSI que sigue todo dispositivo que desea conectarse a la Internet. Los headers o cabeceras que utiliza este protocolo permiten configurar la comunicación entre un *Web Server* y un cliente web, en este caso con el Browser. Estos headers indican **dónde** debe ir el mensaje y **cómo** deben ser manejados los contenidos del mensaje. En cada petición o *request* del Navegador, éste debe especificarlos para que el servidor pueda enteder las peticiones; de la misma manera, el servidor enviará cabeceras que el cliente también debe entender. Algunos *headers* son necesarios y hasta obligatorios, para algunos servidores, y en otros da lo mismo como vayan.

- Content Security Policy: Es un mecanismo de defensa crea exclusivamente para la defensa de ataque de tipo XSS o *Cross-Site Scripting*. La misión de éste es definir bien la línea entre intrucciones y contenido, donde la primera se refiere a código que se debe ejecutar. Para que sea posible utilizar este mecanismo es necesario agregar al header del servidor, para la *request* del cliente, el header Content-Security-Policy o X-Content-Security-Policy, donde se indica la localización de donde los scripts pueden ser obtenidos o *loaded* y además pone restricciones a estos mismos scripts.
- Secure Cookie Flag: El propósito de este header es de instruir al Browser de nunca mandar una *cookie* sobre un canal no seguro, solo debe ser realizado por HTTPS. Esta medida permite asegurar que una cookie tampoco será enviada

por canales mixtos, donde al inicio de la comunicación HTTPS y luego vuelve a HTTP.

- HttpOnly Cookie Flag: Una opción para las *cookies* que permite inhabilitar el acceso al contenido de una cookie por medio de scripts. Esta opción originalmente fue pensada para evitar ataques XSS.
- X-Content-Type-Options: Un servidor que manda la directiva nosniff para este header, obligará al Browser a renderizar la página así como lo dice el header content-type. La idea de este header es poder limitar la ejecución del tipo objeto que pide el browser.
- Strict-Transport-Security: Obliga al navegador a que la comunicación con el servidor sea realizada por un tunel válido HTTPS, de manera que la comunicación sea completamente segura.
- X-Frame-Options: este header previene que se realize un *framing* de la página, es decir, esta opción evita que la página sea mostrada a través de un <iframe>. Este control permite especialmente mitigar ataques de *Clickjacking*, donde el usuario es engañado a través de lo que se muestra en la ventana del navegador.

Canales de comunicación en HTTP

Cuando se habla de HTTP usualmente ésto se relaciona con la comunicación que se lleva a cabo entre el cliente y servidor. Existen diversas formas para que ésto se lleve a cabo, las más conocidas son:

1. postMessage
2. XHR: XMLHttpRequest
3. WebSockets: Es una tecnología nativa del Navegador que permite abrir un canal de comunicación interactivo, responsivo y *full-duplex* entre el cliente y el servidor. Éste comportamiento permite tener *event-driven actions* rigurosas sin necesidad explícita de sondear el servidor en todo momento. Websockets intenta reemplazar las tecnologías *Push* basada en AJAX.
4. WebRTC: O mejor conocido como *Web Real-Time Communication*, es una API basada en la especificcación de la W3C, que utiliza las capacidades de Javascript y HTML5 (sin la utilización de plugins externos o internos) para transmitir audio, video y compartir archivos por medio de P2P. Ésta herramienta permite a los browsers comunicarse entre ellos a muy baja latencia y entrega un gran *bandwidth* para poder realizar comunicaciones media en tiempo real. Hasta el momento Google Chrome/Chromium y Firefox han implementado esta tecnología,

con el objetivo de: mejorar la experiencia de usuario al no necesitar plugins para ser usada, y entregar seguridad dado que impone el uso de encriptación en los datos.

2.2.3 SSL/TLS Encriptación en capa de Transporte

Si bien existe una medida de seguridad en los headers que se implementa en la capa de Aplicación por medio de HTTP, ésto no impide que otros puedan ver qué contienen los paquetes. La confidencialidad, autenticidad y el no repudio de lo que se envía, es un aspecto relevante cuando se está trabajando con sistemas con información crítica y confidencial. SSL (Secure Socket Layer) y TLS (Transport Layer Security) tienen el objetivo de proveer un canal confiable y privado de todo lo que se envía entre dos aplicaciones que se comunican, es decir, una seguridad *end-to-end*. TSL es el resultado de la estandarización de SSL por la Internet Engineering Task Force (IETF). SSL/TLS trabaja debajo del protocolo HTTP usando certificados de clave pública que permiten:

- Resolver parcialmente el problema de la autenticación de un usuario, al establecer un canal seguro y encriptado mediante el uso de certificados digitales.
- Identificar que la información enviada por los dos *endpoints* sea solo de ellos dos, agregando una firma al final del paquete usando la clave privada de la entidad que envía.
- Asegurar que todo lo que se envía sea visto sólo por las entidades que crean el canal de comunicación, a través de la codificación de los paquetes con las claves públicas de las entidades y su posterior decodicación con las respectivas claves privadas de cada uno.

El proceso que permite el inicio de una comunicación mediante SSL/TLS es:

1. Un usuario desea conectarse por el Browser a un Web Server.
2. Se inicia el proceso de *Handshake* entre el Browser y Servidor. Éstos dos se ponen de acuerdo en cómo se encriptará la comunicación (parámetros e información de los certificados) e intercambian una llave asimétrica.
3. El Navegador chequea la validez del certificado, ejemplo: revisa si está en una black list o está expirado o fue creado por una CA *Certificate Authority* confiable.
4. Si el servidor requiere un certificado por parte del cliente, el Browser le enviará el suyo. Esto permitirá tener una autenticación mutua entre las partes.

5. El Web Browser y el Servidor usan las llaves públicas del otro para poder acordar una clave simétrica, que es aquella que permitirá encriptar los mensajes. Sólo estas dos entidades conocerán tal clave.

6. El proceso de *handshake* termina y todo lo posterior se realiza encriptando los paquetes con la llave simétrica acordada por las partes.

Para que tanto SSL y TLS provean una conección segura, todos los componentes involucrados: cliente, servidor llaves y aplicación web deben ser seguros.

2.2.4 SOP: Same Origin Policy

Es un principio de seguridad implementado (hoy en día) por cada browser existente, su principal objetivo es restringuir las formas de comunicación entre una ventana y un servidor web. **Same Origin Policy** o **SOP** es un acuerdo entre varios manufacturadores de navegadores web cómo Microsoft, Apple, Google y Mozilla (entre los más importantes), en donde se definió una estandarización de cómo limitar las funcionalidades del código de scripting ejecutado en el browser del usuario.

Este importante concepto nace a partir del Modelo de Seguridad detrás de una Aplicación Web, al mismo tiempo que es el mecanismo más básico que el Browser tiene para protegerse de las amenazas que aparecen en el día a día, haciendo un poco más complicado el trabajo de crear un *exploit*. **SOP** define lo que es un **Origen**, compuesto por el **esquema**, el **host/dominio** y **puerto** de la URL. Esta política permite que un Web Browser aisle los distintos recursos obtenidos por las páginas web y que solo permita la ejecución de *Scripts* que pertenezcan a un mismo **Origen**. Inicialmente fue definido solo para recursos externos, pero fue extendido para incluir otros tipos de orígenes, esto incluye el acceso local a los archivos con el *scheme* **file://** o recursos relacionados al Browser con **chrome://**.

SOP puede distinguir entre la información que envía y recibe el Web Browser, y solo se aplicará la política a los elementos externos que se soliciten dentro de una página web (recepción de la información). Esta imposibilidad de recibir información de un **Origen** diferente al del recurso actual, permite disminuir la superficie de ataque (*Attack Surface*) y la posibilidad de explotar alguna vulnerabilidad en el sistema donde reside el Browser. Sin embargo, **SOP** no pone ninguna restricción sobre la información que el usuario puede enviar hacia otros. Sin **SOP** cualquier sitio podría accesar a la información confidencial de un usuario o de cualquier otro sitio. Por tanto es sencillo entender la razón de la existencia de **SOP**, se desea proteger la información del usuario, sus cookies, token de autenticación, etc. de las amenazas existentes en la Internet.

En [**?**] menciona que no existe una sola forma de **SOP**, si no que es una serie

de mecanismos que superficialmente que se parecen, pero al mismo tiempo poseen diferencias:

- **SOP** para acceso al **Document Object Model**: se dará permiso de modificar el **DOM** y sus propiedades solamente aquellos scripts que tienen el mismo dominio, puerto (para todos los browsers excepto Internet Explorer) y protocolo. Visto de otro modo, el mecanismo entrega una especie de Sandboxing para el contenido potencialmente peligroso y no confiable. Sin embargo éste no es suficiente, pues posee varias desventajas: el dominio es posible de cambiar a la conveniencia del atacante, limita las acciones a los desarrolladores lo que se traduce en que éstos tengan que buscar bugs que permitan liberarse de estas restricciones lo que incita a atacantes a aprovecharse de ésto.
- **SOP** para el objeto XmlHttpPRequest: para diferentes tipos de peticiones (GET, POST y otros) existen condiciones y suposiciones que hacen que se tome o no en cuenta el *request* del cliente, además del uso de una *whitelist* de las formas en que el header de la petición puede salir del browser.
- **SOP** para *cookies*: restringuiendo el uso de acuerdo su dominio, *path*, tiempo de uso, modificando o eliminado las cookies, e incluso protegiendo las cookies usando el *keyword: secure*. Sin embargo, desde su implementación las cookies han generado bastante problemas de seguridad.
- Y otros como: SOP para Flash, donde usa políticas para realizar peticiones fuera del dominio através de un archivo **crossdomain.xml**, **SOP** para Java y **SOP** para Silverlight, parecido al de Flash solo qe utiliza otros elementos.

Tanto para los atacantes como desarrolladores de Software, SOP puede llegar a ser bastante molesto. Para el primero, la respuesta es obvia, pero para el segundo está el problema de ¿cómo poder aislar los componentes no confiables o parcialmente confiables, mientras que al mismo tiempo se pueda tener una comunicación entre ellos de forma segura? Ejemplo de esto son los Mashup [**?**], que permiten juntar contenido de terceros en una misma página por medio de frames, etc.

Existen excepciones que permiten evitar el uso de SOP, pero como es de esperar esta vía puede ser mal usada por los atacantes en contra del usuario y de la Aplicación Web. Dentro de las excepciones están los elementos en HTML <script>, <img>, <iframe> y otros, que si bien permiten la comunicación entre diferentes orígenes, un mal uso de este puede causar grandes estragos, desde la eliminación de registros en una base de datos hasta la propagación de un gusano o virus.

Queda decir que si bien SOP entrega una capa de seguridad al usuario y a la Aplicación Web, contra cierto tipo de ataques (muchas veces del tipo de ataques de principiantes), esto no es suficiente. Es responsabilidad del desarrollador de Software

poseer las herramientas necesarias para asegurar la confidencialidad e integridad del sistema a través de otros métodos de seguridad.

2.2.5 Markup Languages

Un lenguaje de marcado sigue tradicionalmente un *Standard Generalized Markup Language*, de manera que entrega una semántica apropiada para representar o mostrar contenido, placeholders de aplicaciones y datos. Cada página mostrada por el navegador, sigue las instrucciones que el lenguaje de marcado le da al browser para mostrar el contenido. HTML y XML son los más conocidos en el mercado. Ambos lenguajes tienen sus especificaciones en la W3C o *World Wide Web Consortium*.

HTML: HyperText Markup Language

HTML [**?**], en especial la actual versión HTML5, es conocido por ser un *Simple Markup Language* o lenguage de marcado simple, usado principalmente para crear documentos de hypertextos que son posibles de portar desde una plataforma a otra, sin problemas de compatibilidad. Un documento HTML consiste de un árbol de elementos y texto, cada uno de esos elementos es denotado por un tag inicial y uno final; estos tags pueden ir anidados y la idea es no se superponen entre ellos. Un HTML User Agent o Browser consume el HTML y lo parsea para crear un árbol DOM, que es la representación en memoria del documento HTML. Una característica importante de este lenguaje de marcado es su flexibilidad ante los errores, esto es que en alguna ocasiones el programador perfectamente podría sobrarle un signo y HTML no le daría mayor importancia mientras no afecte a la estructura global de la página. Normalmente esta característica es aprovechada por los atacantes para insertar nuevos elementos html que ejecuten scripts que afectarían al navegador.

XML: eXtensible Markup Language

Este lenguaje de marcado tiene una estrecha relación con HTML, pero a diferencia de este último tiene una sintáxis y semántica más rígida ya que sigue al pie de la letra un lenguaje libre de contexto. Este tipo de lenguaje es ideal para el transporte de data entre *web Services* o interacciones **RPC**, dado que no hay forma de como malinterpretar la data.

2.2.6 CSS: Cascading Style Sheets

Es un lenguaje usado junto a HTML o XML para definir la capa de presentación de las páginas web que el navegar renderiza al usuario. La W3C se encarga de la especificación de las hojas de estilos para que los browser sean capaces de interpretar bajo estándares y aseguren ciertos niveles de calidad. Una hoja de estilo se compone de una lista de reglas. Cada regla o conjunto de reglas consiste en uno o más selectores y un bloque de declaración, más los estilos a aplicar para los elementos del documento que cumplan con el selector que les precede.

2.2.7 DOM: Document Object Model

Es una *API* independiente del lenguage y multiplataforma para HTML válido y bien formado, que define la estructura lógica de un documento que permite ser accedido y manipulado. DOM es una especificación que permite a programas Javascript modificar la estructura del contenido de una página dinamicamente. Esto permite que una página pueda cambiar sin la necesidad de realizar nuevas peticiones al servidor y sin la interacción del usuario. Posteriormente la *W3C* [**?**] formó el *DOM Working Group* y con ello se creó la especificación a través de la colaboración de muchas empresas y expertos. La arquitectura de esta *API* se presenta en la Figura 2.1, donde el *Core Module* es donde están las interfaces que deben ser implementadas por todas las implementaciones conformes de DOM. Una implementación de DOM puede ser construida por uno o más módulos dependiendo del host, ejemplo de esto: la implementación de DOM en un servidor, donde no es necesaria la implementación de los módulos que manejen los triggers de eventos del mouse.

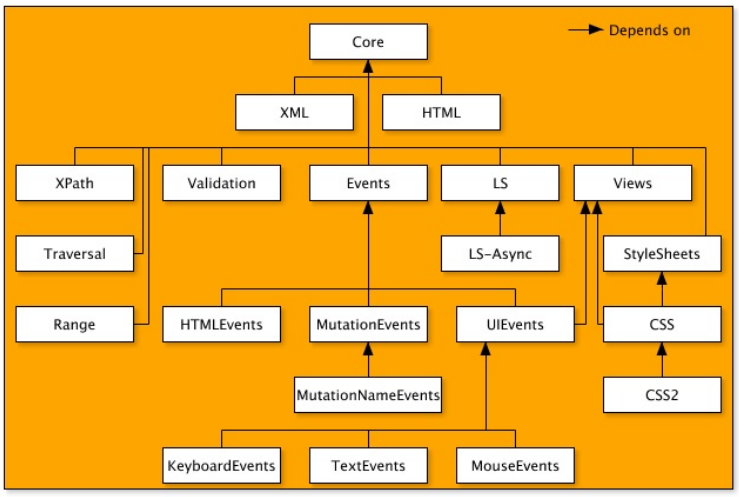


Figure 2.1: Arquitectura de DOM [**?**]

La interfaz de *DOM* fue definida por el **OMG IDL** y fue construida para ser usada en una gran variedad de ambientes y aplicaciones. El documento parseado por DOM se transforma en un gran objeto, tal modelo captura la estructura del documento y el comportamiento de éste, además de otros objetos de lo que puede estar compuesto y las relaciones entre ellos. Cada uno de los nodos representa un elemento parseado del documento, el cuál posee una cierta funcionalidad e identidad. La estructura de árbol del DOM construido puede llegar a ser gigantezca, y almacena más de un árbol por cada documento que parsea.

2.2.8 Javascript, VBScript y otros

Ambos son lenguajes de scripting orientados a objetos. Javascript fue desarrollado por Netscape mientras que VBScript fue desarrollado por Microsft para Internet Explorer, los dos siguen el estandar del lenguage de scripting **ECMAScript**. Dado que VBScript no era usado por muchos y no tenía soporte para otros navegadores más que Internet Explorer, Microsoft decidió abandonarlo.

Muchos piensan que JavaScript es un lenguage interpretado, pero es más que eso. Javascript es un lenguage de **scripting dinámico** (por tanto no tipificado) que soporta la construcción de objetos basados en **prototipos**. Esto quiere decir que a diferencia de un lenguage de programación orientado a objetos como Java, un lenguage orientado a prototipos no hace la distinción entre clases y objetos (clase instanciada), son simplemente objetos. Y cómo tal al ser construido con sus propidades iniciales, es posible poder agregar o remover propiedades y métodos de forma dinámica (durante el runtime) tanto a un objeto como a la clase.

Javascript puede funcionar tanto como un lenguage de programación procedural o como uno orientado a objetos. Firefox usa una implementación en C de Javascript llamada *Spider Monkey*, Google Chrome/Chromium tiene un motor de JavaScript llamado *V8* e Internet Explore no usa realmente JavaScript si no que *JScript* (hace lo mismo que las otras implementaciones solo que difiere en el sistema operativo que utiliza) que en este caso se llama *Chakra*.

Si bien es posible comprender que JavaScript posee increíbles posibilidades para la creación de *RIA* (Rich Internet Applications), en [**?**] se muestra que puede llegar a ser un fracaso si es que no se toman en cuenta ciertas vulnerabilidades inherentes al lenguage. Estas vulnerabilidades que pueden llegar a ser criticas, a menudo permiten a un comunicante comprometer completamente a la otra parte. La misma naturaleza de JavaScript que permite la modificación en runtime de los objetos, puede llegar a ser aprovechada de esta situación; en la cita toma por ejemplo la comunicación entre los elementos de un *Mashup*.

2.2.9 Geolocalización

Cada Browser posee una API que permite obtener la data de la localización del host donde el browser está alojado. Ésta es obtenida ya sea del GPS, si es un dispositivo móvil, como de la triangulación de la señal del celular, localización de IP del movil o *access point*.

2.2.10 Speedy o Protocolo SPDY

Es un protocolo de red abierto desarrollado por Google en el 2009, para el transporte de contenido Web. A modo general utiliza técnicas de *multiplexing*, compresión y prioritización, sin embargo depende bastante de las condiciones del sitio web y su despliegue en la red. SPDY manipula el tráfico en el protocolo HTTP para disminuir el tiempo de carga de las páginas web, al mismo tiempo que cuida la seguridad de los datos. Este protocolo modifica la forma en que las peticiones y respuestas HTTP son enviadas a la internet (por el cable); SPDY es considerado una especie de tunel. Sin embargo, cuando la versión 2 de HTTP esté completa SPDY será deprecada. Implementaciones de este protocolo se dan en: Google Chrome/Chromium, Internet Explorer, Firefox, Safari, Opera y Amazon Silk.

2.2.11 WebWorkers

Ésta tecnología permite la creación de *threads* en el browser para separar las tareas de éste, dejando algunas en el *background* para incrementar el rendimiento total de la carga de las páginas web. Existen 2 tipos: una que es compartida por todo aquello de un mismo **Origen** y otra que se comunica hacia atrás a la función que la creó. Esta API entrega al desarrollador más flexibilidad, pero que sin duda los atacantes también aprovechan bastante.

2.2.12 CORS: Cross-Origin Resource Sharing

Cómo lo define su nombre es un mecanismo (especificación) que permite al cliente realizar request entre sitios de diverso *Origen*, ignorando el **SOP**. *CORS* define una forma en que el Browser y el Servidor Web puedan interactuar para determinar si permitir o no el request a otro origen. Un Browser utiliza SOP para restringuir los request de la red y prevenir al cliente de una Aplicación Web ejecutar código que se encuentra en un origen distinto, además de limitar los request HTTP no seguros que podrían tratar de generar un daño. CORS extiende el modelo que el Browser maneja e incluye:

- Un header en la respuesta/response del servidor solicitado llamado *Access-Control-Allow-Origin*, donde se debe escribir el origen que tendrá acceso a los recursos solicitados al servidor. Si el valor de la respuesta del servidor coincide con el *origen* de quién lo solicitó, se podrá realizar el uso del recurso en el navegador, de lo contrario se generará un error.
- Otro header llamado *Origin* pero esta vez en el request de la solicitud, para permitir al Servidor hacer cumplir las limitaciones en las peticiones de distinto origen.
- En algunos casos un browser deberá agregar el header *Access-Control-Allow-Methods*, ya que el servidor no responderá de vuelta si no es así. Esto permite limitar la superficie de ataque en el servidor.

Existen ciertos métodos en HTTP que necesitan realizar un *pre-vuelo* antes de ser ejecutados, si la response del servidor es afirmativa luego se enviará el request original con el método que se debió confirmar su utilización. Para el caso de los métodos GET y POST, los más usados, este pre-vuelo no es necesario y se puede enviar el request inmediatamente.

La gran diferencia de CORS con cualquier otro método de que permita hacer request hacia un origen distinto, es que el Browser por default no enviará ningún tipo de información que permita identificar al user. De esta manera se puede disminuir considerablemente las amenazas en la confidencialidad, pues el atacante no podrá hacerse pasar por un usuario del que no tiene información. Casi todos los navegadores web, a diferencia de Internet Explore [**?**], realizan sus solicitudes a servidores de diverso origen por medio de la interfaz *XmlHttpRequest*, en el caso de Internet Explorer esta se llama *XDomainRequest*.

2.2.13 Sandboxing

La idea es encapsular el área de mayor probabilidad de ataque en un espacio aislado, minimizando la superficie de ataque de un software. Sandboxing no es una técnica tan nueva, han existido sistemas que ya lo han incorporado. Ésta protección puede ser aplicada dependiendo del diseño del software, algunos ocupan Sandbox a nivel del sistema operativo como otros que ocupan al nivel del *engine* de Javascript. En el caso especial del Browser, esta técnica es construida en el nivel más alto posible para un programa de usuario, lo que permite la separación de privilegios entregados por el sistema operativo al browser y los subprocesos que corren dentro de éste. El atacante que se enfrente a un browser que tenga este mecanismo de defensa, tendrá que realizar primero un *bypass* encontrando una vulnerabilidad en el sandboxing del browser. Existen diferentes técnicas para Sandboxing, todo depende del diseño del Browser.

2.3 Arquitectura de Referencia o Reference Architecture (RA)

Una arquitectura de Referencia, de acuerdo a la *Open Security Architecture* o OSA[**?**], es considerado un elemento que describe un **estado de ser** y debe representar aceptadas buenas practicas. En [**?**] se explica que una RA es una arquitectura de software genérica y estandarizada, para un dominio particular e independiente de la plataforma o detalles de implementación. En ésta especifica la decomposición del sistema en subsistemas, las interacciones entre éstas partes y la distribución de funcionalidad entre ellas [**?**]. Una RA es una herramienta que permite facilitar el entendimiento de sistemas complejos y su apropiada construcción a sistemas reales. Si bien una RA es usada principalmente para capturar los *concerns* de los *Stakeholders* al comienzo de un Desarrollo de Software, también puede ser usada para educar al realizar la unión de ideas y terminologías usadas por diversos sistemas que se asemejen. Para describir la Arquitectura de Referencia nos hemos basado en los trabajos [**?**, **?**, **?**], usando patrones para la contrucción de la AR.

Actualmente no hay un consenso de lo que una AR debe contener, [**?**] describe un ejemplo e indica como debería de ser ésta con los siguientes elementos:

- Describir los Stakeholders que interactuan con el sistema y que poseen *concerns* de éste.
- Generar *views* usando UML y teniendo en cuenta un proceso *Rational Unified Process*: crear casos de uso, modelos de análisis y diseño, modelo de despliegue e implementación.
- Patrones de Arquitectura.
- Atributos de calidad deseables que el sistema debe garantizar. Es importante solo destacar aquellos realmente necesarios, dado que un sistema sobrecargado con ellos tampoco es conveniente.

La Arquitectura de Referencia debe ser en lo posible descrita de la forma más abstracta posible, pues su función guiar la construcción de arquitecturas concretas, sin tener en cuenta detalles de las tecnologías usadas.

Las ventajas y usos que se obtienen al construir una RA son:

- Comprender la estructura subyacente de un Web Browser y las interacciones que tendrá con otros sistemas.

- Proveer una base tecnológica modular y flexible. Al tener los subsistemas compartimentalizados es posible quitar y sacar piezas, que poseen interfaces similares, y de esa manera reusar lo otro sin tener que construir un sistema nuevo.
- Entrega una base para el desarrollo de otros Navegadores Web, sin explicar detalles de implementación.

En este trabajo el enfoque estará en el primer punto, donde se quiere entender las interacciones entre un desarrollo de Software y la utilización de las funcionalidades del Navegador. Dado que parte de la investigación es obtener Patrones de Mal Uso o Uso Indebido del Navegador Web, es primordial concebir una Arquitectura de Referencia que permita encontrar donde es posible aplicar Patrones de Seguridad para poder mitigar los malos usos del Browser [?]. A continuación se presenta la AR obtenida a partir de los Browsers más usados actualmente [?]: Google Chrome/Chromium, Internet Explorer y Firefox.

2.4 Desarrollo de Software Seguro y Diseño de Software Seguro

La filosofía detrás de *Secure Software Developmet* es que detrás de cada etapa de desarrollo del software, se tengan en cuenta los prinicipios de Seguridad: Confidencialidad, Integridad, Disponibilidad y Auditoría. Para cumplir este cometido es que se deben llegar a políticas y reglas que aseguren la Seguridad como una propiedad sistémica.

Varias comunidades tienen diferentes enfoques y técnicas de cómo asegurar la Seguridad en los sistemas, muchas pueden incluso tener similitudes y hasta trabajar juntas. En este trabajo, el enfoque tomado es aquél que busca entregar la propiedad de seguridad a través del entendimiento de un sistema a un alto nivel, identificando las amenazas durante la elicitación de requerimientos, de manera que se pueda extraer las posibles amenazas que podrían existir y utilizando elementos de diseño para hacer cumplir los principios de seguridad necesarios por el sistema; este enfoque es el que se dedica la comunidad de *Secure Software Design*.

Fernandez [?] sostiene que para construir un sistema seguro es necesario realizarlo de manera sistemática de tal manera que la seguridad sea parte del integral de cada una de las etapas del Desarrollo de Software - de inicio a fin. El enfoque que propone es ingenieril y por tanto es aplicable incluso para sistemas *legacy*, donde es posible hacer ingeniería inversa para comprobar si existen o no los requerimiento de seguridad implementados, de manera que permite generar un estudio con la intención de comparar y mejorar nuevos sistemas. En su libro [?] presenta una completa metodología para

construir sistemas seguros a partir del Diseño Orientado a Objetos, UML y patrones, a los cuales nombra como **Security Patterns**.

Como parte de la metodología propuesta, se plantea que para diseñar primero se deben entender las posibles amenazas a las que está expuesto el sistema. La identificación de Amenazas [**?**, **?**] es la primera tarea que presenta la metodología, que considera las actividades en cada caso de uso del sistema.

2.5 Patrones

Los Patrones encapsulan soluciones recurrentes a problemas y definen una forma de expresar los requerimientos y soluciones de una forma concisa, al mismo tiempo que proveen de un vocabulario común entre los diseñadores [**?**]. Un patrón encarna el conocimiento y experiencia de desarrolladores de software que puede ser reusado posteriormente en nuevas aplicaciones [**?**, **?**]. Los Patrones expresan las relaciones entre un contexto, un problema y una solución. Para un contexto dado, el patrón puede ser adaptado para encajar en diversas situaciones. La construcción de Patrones de Seguridad parte de la premisa anterior, éste permite construir sistemas seguros a través del uso de Patrones adaptados a las necesidades del sistema y preocupaciones de los *Stakeholders*. Por otra parte, una Arquitectura puede ser descrita a través de Patrones, permitiendo que haya un mejor entendimiento al momento de proveer con guías de diseño y análisis a desarrolladores.

Los patrones describen diseños recurrentes en un mediano nivel de abstraction y es poco probable que existan solos, es decir, existen en conjunto a otros patrones. Un patrón puede proveer una solución usando diagramas en UML, de manera que describen de forma precisa al sistema.

La Arquitectura de Referencia a confeccionar será realizada por medio de patrones y éstos serán descritos con el template creado por [**?**], llamado POSA, que contiene las siguientes secciones para describir un patrón: *Intent*, Contexto, Problema, Solución, Implementación, Usos comúnes, Consecuencias y Patrones relacionados.

2.6 Patrones de Seguridad

Los Patrones de Seguridad son aquellos que encarnan buenos principios de diseño que tienen en cuenta ciertos principios de seguridad, y que al ser aplicados en una metodología para el desarrollo de sistemas, es posible asegurar que el sistema aplique esos principios y en consecuencia generar un sistema seguro [**?**, **?**]. Estos patrones describen las maneras de detener o mitigar una posible amenaza de seguridad, especi-

ficando una solución a través de mecanismos de seguridad, para el contexto dado. Las soluciones propuestas deben resolver las fuerzas o *forces* indicadas por el patrón. Un uso importante de estos patrones es la ayuda que aportan a desarrolladores que no son expertos en seguridad; éstos permiten ayudarlos a implementar mecanismos que implementen los principios de seguridad necesarios.

2.7 Patrones de Mal Uso

Para diseñar sistemas seguros, se es necesario identificar las posibles amenazas que un sistema puede sufrir. Papers como [**?**, **?**, **?**, **?**] describen el desarrollo de una metodología completa para encontrar amenazas, a través del análisis de actividaddes de los casos de uso del sistema buscando como podría un atacante interno o externo socavar las bases de esas actividades. Es importante no confundir *Attack Patterns* con *Misuse Pattern*, pués claramente en [**?**, **?**] dejan explícito que un *Attack Patttern* es una acción que lleva a un mal uso o *misuse*, o acciones **específicas** que toman ventaja de las vulnerabilidades de un sistema, como por ejemplo un *buffer overflow*. A partir de los trabajos [**?**, **?**, **?**] se hace la unión de los conceptos de *Attack Patttern* para dar forma a la definición de *Misuse Pattern* [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]:

> Un patrón de mal uso o *Misuse Pattern* describe, desde el punto de vista del atacante, cómo un tipo de ataque es realizado (qué unidades usa y cómo), analiza las maneras de detener el ataque através de la enumeración de posibles Patrones de Seguridad que pueden ser aplicados, y describe cómo rastrear un ataque una vez que ha ocurrido por medio de una recolección y observación apropiada de datos forenses.

Sin embargo, cuando un sistema ya está diseño y construido, como es el caso del Web Browser, lo que va a importar es saber **cómo** los componentes del sistema, pueden ser usados por el atacante para alcanzar sus objetivos. Un *Misuse Pattern* o **Patrón de Mal Uso** describe, desde el punto de vista del atacante, cómo un tipo de ataque es realizado, indicando **qué** componentes usa y **cómo**. Además analiza las formas de detener el ataque a través de un listado de posibles *Security Patterns* o **Patrones de Seguridad** que pueden ser aplicados para esa situación, y describe cómo poder seguir el rastro de un ataque una vez que ha sido realizado con éxito en el sistema, a través de data forense. Además describe un contexto en dónde puede ocurrir el ataque.

Un catálogo de *Misuse Patterns* podría ser de gran valor en el Desarrollo de Sistemas que interactúan con el Navegador, pués provee a desarrolladores un medio para evaluar los diseños de sus sistemas, al analizar las posibles amenazas del Browser que pusieran afectar al software que está siendo construído.

Chapter 3

Browsers existentes y Estado del Arte

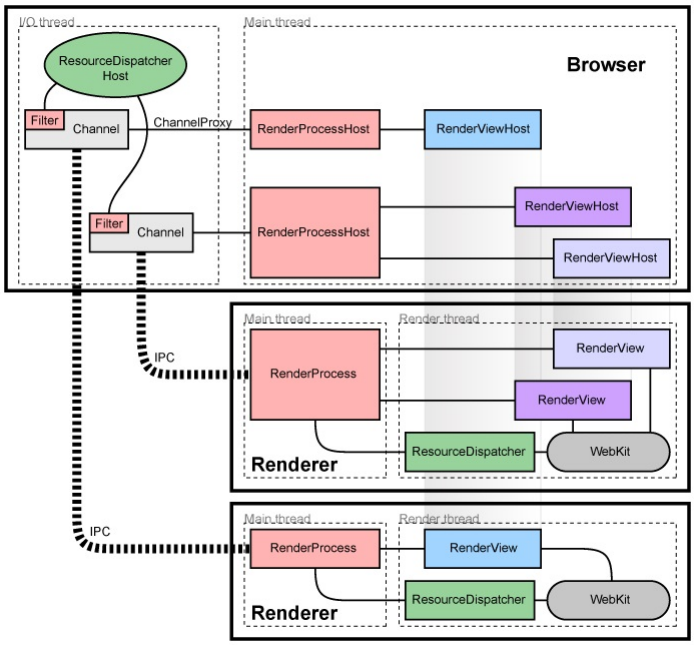
3.1 Navegadores Existentes

3.1.1 Google Chrome y Google Chromium

La misión de Google es organizar la información del mundo y lograr que sea útil y accesible para todo el mundo. Esta gran empresa partió como un búscador y rápidamente llegó a ser dueño de la mayor parte de búsquedas del mercado. Tiene servicios de almacenamiento en la nube, correo electrónico, *e-wallet* y otros más. Google ha sido responsable por la construcción del Navegador Web **Google Chrome** y *Google Chromium*. En el 2008, Google liberó gran parte del código de Chrome bajo el proyecto de nombre Google Chromium, el cuál es open-source que permitiría a desarrolladores *third-party*, estudiar el código fuente y ayudar en la implementación para las plataformas Linux y OS X. La diferencia entre Chrome y Chromium son: actualizaciones automáticas y Adobe Flash integrado. Así como se puede ver en [?] éste Navegador ha sido el último que ha salido y se llevado una gran parte del mercado [?].

La arquitectura de Chrome o Chromium se basa principalmente en dos módulos: el Browser Kernel y el Rendering Engine, cómo se puede ver en la Figura 3.1.1.

Én la documentación de Google Chromium [?], que es base para Google Chrome, afirma que la arquitectura soporta para cada tab un proceso nuevo, de manera de hacer al Browser más robusto y modularizar el sistema para evitar ciertas amenazas de seguridad. El proceso principal es llamado *Browser Process/Kernel/Engine* y se encarga de la *User Interface*, manejo de las tabs y los procesos de los *plug-in*. Cada tab es asociado a un Rendering Engine, éstos tienen restricciones de acceso (*Sandoboxing*) a los demas y al sistema, lo que permite que exista una protección de la memoria y un



control de acceso. En [?] se explica que el objetivo principal de esta arquitectura es poder mitigar ataques muy severos sin tener que sacrificar la compatibilidad con los sitios web ya existentes. Para lograr el objetivo Google ha ganado muchas lecciones de cómo realizar esto [?], pues explican que un gran desafío en la seguridad es proteger a los usuarios de los atacantes que se aprovechan de las vulnerabilidades y debilidades de los clientes web-browsers. En su arquitectura modular se puede ver que se intenta proveer una seguridad que evita afectar la compatibilidad con otros sitios. La arquitectura comentada se basa en dos decisiones de diseño: La arquitectura depende en el Rendering Engine para aquellos componentes de alto riesgo como JavaScript, el parser de HTML y la creación de DOM para hacer cumplir SOP; al estar rodeados por un Sandboxing hace que el Rendering Engine se comporte como una caja negra.

Google Chrome expone en [?] que existen ciertas lecciones que han ido utilizando para mejorar la calidad de su browser. Estas son:

- Reducción de las vulnerabilidades de seguridad, se basa en la aislación de ciertos componentes y la reducción de privilegios de ciertas tareas en el browser. La aislación lo lograron con la creación del Rendering Engine y el Browser Kernel, que tienen como objetivo proteger la data del sistema de archivos. Si bien esto puede no entregar muchos beneficios a una aplicación web, si lo hace en el usuario del browser.

- Reducir la ventana de vulnerabilidades, la actualización del browser se hace cada cierto tiempo de forma automática para así cubrir las vulnerabilidades que van

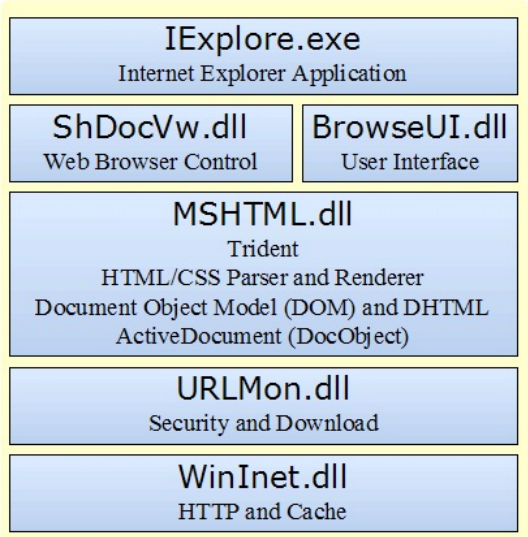
apareciendo.

- Reducción de la frecuencia de exposición, Google trabaja con StopBadware.org para entregar una mayor seguridad al descubrir nuevos tipos de ataques y vulnerabilidades relacionadas con el browser.

3.1.2 Internet Explorer

Internet Explorer es el navegador grafico predeterminado por Microsoft y que su primera versión 1.0 fue realizada en 1995. IE es una derivación de Spyglass Mosaic desarrolado por la NCSA (National Center for Supercomputing Applications). En primera instancia fue un navegador que podría ser obtebido si era comprado como complemento de *Microsoft Plus!* o mediante la versión *OEM* de Windows 95. Desde la tercera versión de IE, en 1996, que esta se lanzó de forma gratuita.

La arquitectura de este navegador es modular y permite al desarrollador por utilizar los recursos para crear diferentes funcionalidades, ejemplo de esto son: toolbars, Microsoft Active X controls, etc. En la Figura 3.1.2 [**?**] se puede ver los principales componentes de la architectura del browser mencionado. IE utiliza *COM* o *Component Object Model* una interfaz binaria standard para componentes de software introducida por Microsoft en 1993 y que permite una comunicación entre procesos/-componentes de software provenientes de la familia de software de Microsoft. *COM* es similar a otras tecnologías de interfaz de componentes de software (Component Software Interface Technologies) cómo CORBA y Java Beans. El uso de *COM* gobierna la forma la interacción de los componentes que se comunicann y permite que haya un reuso y extensibilidad de estos.



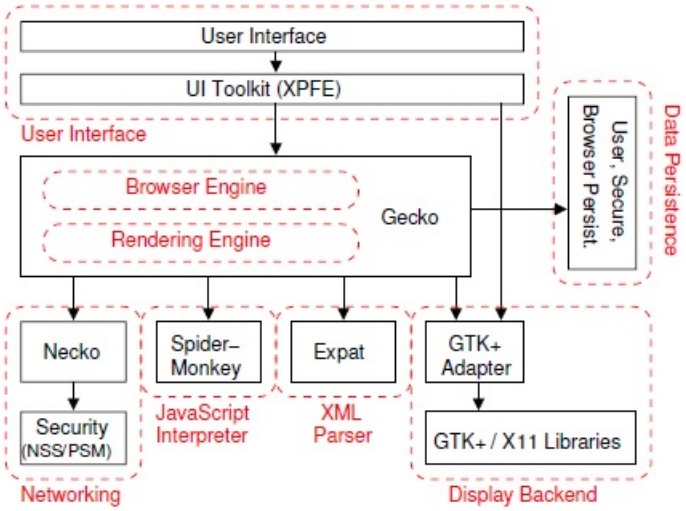
Rendering Engine

O también llamado MSHTML, es un rendering engine privativo, sin embargo es posible usarlo al usar librería de Windows **mshtml.dll**. Según [?] es un objeto OLE (Object Linking and Embedding) Active Document que representa el *layout* de Internet Explorer y permite mostrar graficamente las páginas por medio del *display* del host. Dentro de éste se manejan las Extenciones, el *engine* de Javascript y la librería que contiene la API para tareas de *networking*, además de proveer una capa de seguridad y manejar las descargas de archivos.

3.1.3 Firefox

Firefox fue creado a partir del navegador *Netscape* en 1998, actualmente la fundación Mozilla ha sido la que la ha mantenido, generando varias modificaciones desde su nacimiento. Las metas de diseño que Mozilla desee en el navegador son:

- Renderizado rápido de las páginas web.
- Fuerte apoyo a los estandares web como la W3C.
- Interoperabilidad en las diversas plataformas.



La arquitectura de este browser puede ser vista en la Figura 3.1.3 donde se pueden observar los siguientes componentes:

- La interfaz de usuario, puede ser reutilizada para otras aplicaciones.

- La persistencia de los datos, tanto de bookmarks como de data de bajo nivel como el *cache*.
- EL *Rendering Engine*, permite el renderizado de documentos HTML/XML aún cuando estos estén mal formados. Este Engine es capaz de renderizar la interfaz de la aplicación multi-plataforma.

La arquitectura de Mozilla se distingue de las demás en que la visualización especificada por la plataforma y la librería de *widgets* son usados directamente en el navegador, lo que minimiza el costo necesario para soportar diferentes plataformas.

3.2 Arquitectura de Referencia del Browser y Patrones

Método

El primer paso para realizar el estado del arte con respecto a este punto fue realizar una búsqueda ordenada y a través de string de búsqueda dentro de web engines y librerías digitales conocidas. Se utilizaron las siguientes plataformas para buscar documentación al respecto:

- Google, usando "google dorks" para filtrar resultados
- Google Scholar, usando string de búsqueda y operadores booleanos para filtrar resultados.
- IEEE Xplore Digital Library, usando su buscador basado en comandos y utilizando operadores booleanos para filtrar (de buscó tanto en metadata como en el texto completo).
- Direct Science, CiteSeerX, Springer Link y otras librerías digitales.

Para aquellos buscadores donde era posible usar operadores booleanos también se trató de filtrar el contenido para que pudiera mostrar resultados relacionados a: "Browser" y "Reference Architecture". Sin embargo los resultados fueron bastante pobres. Desafortunadamente hasta la fecha no existe mucho trabajo relacionado a la construcción de una Arquitectura de Referencia para el Browser. Dado que la búsqueda no entregó muchos resultados, se procedió a hacer un *forward snowballing* con el único paper que creemos entrega información similar a lo que se buscaba.

Lo encontrado

En Larrondo-Petrie et. al [?] se realiza un análisis orientado a objetos del dominio del web browser, con el fin de obtener un modelo de dominio, un modelo de objetos y un *feature tree* que describiera la estructura y funcionalidad entregada comúnmente por los Web Browser. Sin embargo éste estudio está bastante lejos de lo que se quiere hacer en este trabajo, sin embargo sirve para obtener un transfondo de lo que sucede en el Web Browser, aún cuando la información esté muy desactualizada.

En el trabajo de Grosskurth et al. [?, ?] se llega a una arquitectura de referencia de muy alto nivel en base a dos navegadores open-source: Mozilla y Konqueror. En esta arquitectura se identifican los siguientes subcomponentes: Interfaz Usuaria, Persistencia de Datos, Browser Engine, Rendering Engine, Networking, Interprete de Javascript, XML Parser y Display Backend. Una vez obtenida la parte conceptual, se inició una evaluación de ésta al comparar las arquitecturas concretas de cada browser open-source, obtenidas por una herramientas de ingeniería inversa, para ver que tanto el modelo conceptual era cercano a la realidad; los browsers usados para validar fueron: Epiphany, Safari, Lynx, Mosaic y Firefox. Si bien la arquitectura entregada entrega bastante información a alto nivel, no desarrolla más que esa capa de abstracción. Además explica cada subsistema con los elementos que se deberían encontrar y para que son usados, por ejemplo explica que en el Rendering Engine es el encargado de mostrar lo parseado del HTML o XML de la página web e indica la relación de éste con el Browser Engine.

En el documento [?] realizado en el año 2000, se describe la experiencia realizada al extender el rabajo del proyecto TAXFORM. Usando PBS, una herramienta de Ingeniería Inversa, se extrajo la arquitectura de software del Navegador Mozilla, con el objetivo de entender la estructuración de sus componentes. Si bien el trabajo ayuda a entender un poco la estructura detrás del navegador, este trabajo es muy antiguo y la versión más actual del navegador ha cambiado bastante. Además lamentablemente, el enfoque de este estudio no es intentar entender lo que hace cada subsistema, si no que es la implementación de la herramienta misma para obtener la arquitectura de software del browser seleccionado.

En [?] se propone un Browser llamado Anfel SOFT, donde gracias al uso de Inteligencia Artificial, crea agentes que permiten mejorar la experiencia del usuario. El trabajo asegura que el browser será capaz de aprender el comportamiento de navegación del usuario, y guiará al usuario en su navegación para que esta sea lo más efectiva posible. El paper obtiene los subsistemas que se pueden encontrar en un browser de la misma manera que lo realiza [?]. Si bien la arquitectura que muestra refleja parte de lo visto en los 3 browsers escogidos en este estudio, no da detalles acerca de cada subsistema identificado. Además la arquitectura de Referencia que entrega es la misma vista en [?, ?] y a pesar que identifica otros posibles componentes, no agrega nada nuevo.

Podemos ver en los trabajos ya mencionados, que la definición de Arquitectura de Referencia dista bastante de la definición dada por nosotros en el capítulo 2. En cada uno de ellos el trabajo ha sido a muy alto nivel y la descripción de los subcomponetes del sistema es mínima. Si bien explican las relaciones entre éstos, no dan un mayor entendimiento en como se comportan en ciertas situaciones. Desafortunadamente para esta memoria no existe mucha literatura sobre el desarrollo de una Arquitectura de Referencia del Browser, y de lo que hay el trabajo más actual es el realizado por [?] en el año 2009.

3.3 Evolución y Seguridad en el Browser

En [?] y [?] podemos notar que existe una mayor importancia en llegar a los componentes y las relaciones detrás del browser, y casi una nula mención de elementos de seguridad que permiten salvaguardar datos críticos o cómo protege al host de las amenazas. Podemos dar esta falta de conocimiento dado que en tales fehcas la cantidad de ataques de seguridad al Browser es mucho menos que en la actualidad

3.3.1 Estandarizaciones

3.3.2 Vulnerabilidades

3.3.3 Amenazas

3.3.4 Medidas de Mitigación o Mecanismos de Defensa

1. Safe Browsing API and Content-Agnostic Malware Prevention
   XXX

2. Sandoxing
   En el desarrollo de [?] se define un modelo de amenazas donde se enumeran las habilidades que debería de tener un atacante y los objetivos de estos, para así caracterizar y evaluar las propiedades de seguridad necesarias para evitar que los atacantes cumplan su objetivo. Una propiedad importante que hacen destacar en el estudio es cómo aislar ciertos procesos que pueden ser aprovechados por los atacantes y ofrece una forma para poder mitigar esto: Sandboxing. El Sandboxing de Google Chrome previene al atacante de leer o escribir en el sistema de archivos del usuario, dejando al Principal Web con los privilegios necesarios para parsear un HTML/XML y ejecutar código JavaScript. Sin embargo esta arquitectura no imposibilita al atacante a atacar otros sitios web si es que el

Rendering Engine fue comprometido, lo que puede convertirse en una amenaza muy grande para otros sitios web.

3. Actualizaciones Periódicas en Background
   XXX

4. Privacy Settings: Do Not Track and Third-Party Cookies
   XXX

5. SmartScreen
   XXX

6. Application Reputation / App Rep
   XXX

7. Content Security Policy
   XXX

8. HTTP Headers
   XXX

3.4 Sumario

Chapter 4

Arquitectura de Referencia del Browser

La arquitectura de referencia a construir en este trabajo tiene como objetivo catalizar el entendimiento de la estrecha relación que existe entre el [**?**] y los sistemas que serían construídos sobre la Internet. En vista a la poca, casi nula, documentación de la creación de Arquitecturas de Referencias del Navegador Web, la necesidad de hacer una AR para entender cómo la arquitectura de este sistema puede relacionarse con el futuro desarrollo de otros sistemas, llega a ser imperante. Al considerar al Web browser como un *concern* en el desarrollo de SoftWare puede ser una buena estrategia para evitar una gran perdida monetaria u organizacional. Se sabe que el Browser es un pieza de Software que ha sufrido varios cambios desde la década de los 90, por lo tanto entre los desarrolladores de ésta herramienta ya existen conveniones de qué elementos funcionan mejor. Por consiguiente, no es de extrañar que diferentes browsers estén construidos de formas muy similares, y en consecuencia puedan ser conceptualizados en una Arquitectura de Referencia que manifeste los componentes, mecanismos de comunicación y funciones de esta pieza de Software.

En este capítulo se presenta una Arquitectura de Referencia (AR) desarrollada principalmente a partir de la abstracción de las propuestas existentes hoy em día como: Google Chrome/Chromium, Internet Explorer y Firefox; nos basaremos en información del año 2014 hacia atrás. Primero se identificarán y analizarán los stakeholders, se identificarán los casos de uso relacionados a uno de estos stakeholders y se dará una descripción breve. Luego se presentarán patrones que formarán parte de nuestra AR. Éstos patrones serán descritos utilizando un template POSA [**?**] y notación UML para precisar cómo los componentes de la AR se relacionan.

4.1 Casos de Uso del Browser

4.1.1 Stakeholders (actores) y Concerns de estos

Se es necesario encontrar los actores (definido por sus roles) que participan en el uso y operación del Navegador, estos son:

Usuario Consumidor (UC)

Este es el principal stakeholder, pues de él depende que se realice el inicio de una petición para buscar una página web, recurso o servicio. Sin éste la utilidad del Web Browser es nula. El stakeholder al mismo tiempo podría ser una entidad no humana, como un plugin, extensión o instancia de una página web, que requiere hacer peticiones por medio de las interfaces del navegador, pero que al fin y al cabo cumplen con los deseo del usuario del host de mostrar el contenido.

Sistema Operativo (SO)

Crea el o los procesos necesarios para iniciar el Navegador, además de entregar un ambiente al Browser para que éste pueda funcionar adecuadamente. En base al enfoque de este estudio, ésta identidad se encarga de aplicar políticas de seguridad sobre el Browser cuando se necesiten realizar operaciones o el navegador desee crear nuevos procesos.

Proveedor de Servicios (PS)

Este puede ser un: Web Server, Web Aplicación, Servicio de actualización del Browser, etc. Su interacción con el Browser se limita a entregar contenido a éste, no a usarlo.

4.1.2 Casos de Uso

Casos de Uso para Usuario Consumidor

El usuario consumidor es el más importante stakeholder del web browser, pues sin él no habría razón para la existencia del browser; un servidor tampoco existiría dado el mismo principio. Bajo ésta entidad es que suceden la mayoría de los casos de uso del sistema al que realizaremos su AR. Al encontrar los casos de uso de éste

stakeholder veremos los concerns de éste, lo que nos permitirá entender las necesidades de seguridad para proteger al navegador.

1. Usar datos privados de usuario (opcional): dependiendo del navegador, se podría indicar al usuario si quiere usar una cuenta externa o interna para acceder a información propia de usuario consumidor y mejor así la experiencia de usuario.

2. Realizar Petición/Request: El caso de uso más importante de éste sistema y en la que se basa la arquitectura cliente/servidor. Esta acción considera encontrar el recurso bajo la URI dada y la consecuente descarga del recurso. La descarga no es siempre necesaria, pues existe lo que se conoce como una peticiíon *preflight* donde solo se envía una petición para saber si es seguro realizar una petición/request. Entonces como una consideración a este caso de uso es que una petición/request no siempre conlleva una respuesta/response, por lo que hemos condensado estas dos (request/response) en un solo caso de uso.

3. Descargar recurso: Este caso de uso está incluído dentro de Petición/Request, pues para conseguir una página web es necesario la descarga del recurso.

4. Solicitar intervención: exiten oportunidades en que la realización de una petición/request conllevará a que el Browser pida permiso al UC para realizar otras acciones, ejemplo: instalar un plug-in, extensión, descargar un contenido que no se puede evaluar su seguridad. En los casos anteriores, una ventana de la interfaz gráfica pedira explícitamente la intervención del usuario para tomar una decisión.

Existen muchos más casos de uso relativos al stakeholder tomado, pero para este trabajo bastarán con los propuestos. Una vez que sabemos los roles involucrados y la forma en que interactuan con el sistema, será posible encontrar las amenazas contra el browser.

Casos de Uso para el Sistema Operativo

1. Operación CRUD en host: Acción realizada cuando el Browser obtiene una solicitud explícita o no (automática), del usuario para realizar acciones que involucren permitir un acceso desde el sistema operativo como: leer, escribir o modificar/eliminar algún recurso del host.

2. Recibir solicitud: En un browser es normal la comunicación entre los procesos que lo componen y el Sistema operativo por medio de un canal de comunicación, ejemplo de esto es cuando un plug-in se comunica con el proceso encargado de mostrar por pantalla lo que se hace en él.

Casos de Uso para el Proveedor de Servicios

1. Recibir Petición/Request: El PS siempre está escuchando en sus puertos habilitados a posibles peticiones de sus clientes. Una petición es recibida en un formato que el PS debe saber interpretar y en consecuencia realizar las acciones correspondientes que el cliente le pide. Por cada solicitud del cliente, en este caso el Web Browser, habrá una respuesta/response por parte del PS. Este caso de uso no se ejecuta, si dentro del header de uno de los paquetes está indicado que la petición fue hecha en modo *preflight*.

2. Operación CRUD: alguna petición podría terminar por realizar un cambio en el PS dado por alguna operación CRUD. Éstas opciones son la voluntad del usuario en el host, y que son transmitidas y realizadas indirectamente por el Web Browser.

Los casos de Uso descritos anteriormente se pueden divisar en la Figura 4.1

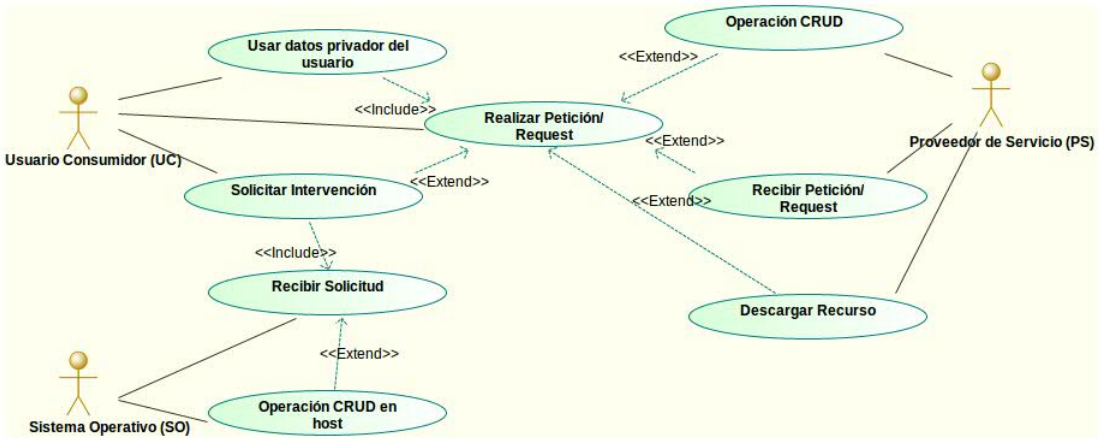


Figure 4.1: Diagrama de Caso de Uso del Web Browser

4.2 Patrón Broker/Frame/Browser Engine/Process (BP)

Intent

El patrón Broker/Frame/Browser Engine/Process o BP

Ejemplo

Contexto

Solución

Estructura

Dinámica

Consecuencias

Ejemplo Resuelto

Usos Comúnes

Patrones Asociados

4.3 Patrón Rendering Engine (RE)

Intent

El patrón Rendering Engine o RE representa a todos los componentes necesarios para realizar el parsing y layout de HTML y XML, interpretación de javascript, construcción del DOM y las interfaces limitadas para la comunicación entre procesos con el BP, para el posterior display de la información por pantalla usando Sistema Operativo (SO).

Ejemplo

Contexto

Solución

Estructura

Dinámica

Consecuencias

Ejemplo Resuelto

Usos Comúnes

Patrones Asociados

4.4 Patrón User Interface

Intent

Ejemplo

Contexto

Solución

Estructura

Dinámica

Consecuencias

Ejemplo Resuelto

Usos Comúnes

Patrones Asociados

4.5 Patrón Plugin

Intent

Ejemplo

Contexto

Solución

Estructura

Dinámica

Consecuencias

Ejemplo Resuelto

Usos Comúnes

Patrones Asociados

Chapter 5

Patrones de Mal Uso

En este capítulo se tiene como objetivos: obtener las amenazas que afectan al Web Browser y obtener Patrones de Mal Uso o Uso Indebido que las reflejen. Se realizará un análisis de amenazas basada en la metodología [**?**], que busca en base a las actividades de los casos de uso los posibles males usos que podrían realizar los atancates.

5.1 Identificando Amenazas

5.2 Template de Patrones de Mal Uso

Esta sección describe cada parte del template a usar para un Patron de Mal uso o Uso Indebido.

Nombre

El nombre del patrón debe corresponder al nombre genérico dado al tipo específico de ataque en los repositorios estandares de ataques, como por el usado por el CERT [**?**].

Intent o descripción básica

Una descripción corta del propósito del patrón (qué problema resuelve para el atacante).

Contexto

Describe el entorno genérico incluyendo las condiciones bajo a las cuales el ataque puede ocurrir. Esto puede incluir defensas mínimas presentes en el sistema, así como también vulnerabilidades típicas del sistema. El contexto puede ser especificado usando Diagramas de *Deployment* de las partes relevantes del sistema así como también Diagaramas de Secuencia o de Colaboración que explayen el uso normal del sistema. Un diagrama de clases podría mostrar la estructura relevante del sistema. Se especifican además precondiciones para que el ataque ocurra.

Problema

Desde la mirada del atacante, el problema es encontrar **cómo** atacar el sistema. Un problema adicional es cuando el sistema está protegido por mecanismos de defensa. Las **fuerzas o forces** indican qué factores pueden ser requeridos en orden de ejecutar el ataque y en cómo realizarlo; por ejemplo, qué vulnerabilidades pueden ser explotadas. Además, qué factores podrían evitar que el ataque se pueda llevar acabo o lo retrasen.

Solución

Esta sección describe la solución que resuelve el problema del atacante, ej: cómo el ataque puede alcanzar sus objetivos y los resultados esperados de éste. Diagramas en UML muestran los componentes del sistema involucrados. Diagramas de Secuencia o Colaboración muestran el intercambio de mensajes necesarios para cumplir con el ataque. Diagramas de actividad pueden añadir más detalle.

Componentes del sistema afectados (Dónde buscar evidencia)

Esta sección adicional al *template* original de un Patrón de Seguridad [**?**] es nueva, dado que tiene relación con el mal uso realizado. La solución debe representar todos los componentes que están involucrados en el ataque, pero no debe ser una extensa lista, solo los más importantes para prevenirlo o lo escencial para una examinación forense. Esto puede ser representado por un diagrama de clases, que puede ser un subset o un superset del diagrama del contexto.

Usos comúnes

Incidentes específicos en donde el ataque ha ocurrido son preferidos, pero para vulnerabilidades nuevas, donde el ataque aún puede que aún no se haya realizado, un contexto específico dónde el ataque pueda llevarse acabo es suficiente.

Consecuencias

Discute los beneficios y desventajas del patrón de mal uso o uso indebido desde el punto de vista del atacante. ¿Es acaso el esfuerzo y costo gastado comparable a los resultados obtenidos? Esta es una evaluación que debe ser realizada por el atacante al decidir realizar el ataque; Diseñadores deben evaluar el riesgo de sus activos usando algpún enfoque de análisis de riesgos. La enumeración incluye buenos y malos aspectos, que deben ser emparejados a las fuerzas o *forces*

Contramedidas y datos Forenses

Esta sección describe las medidas de seguridad necesarias para detener, mitigar, o rastrear este tipo de ataque. Esto implica la enumeración de los Patrones de Seguridad o *Security Patterns* que son efectivos contra este ataque. Desde un punto de vista Forense, se describe qué información puede ser obtenida en cada etapa rastreando el ataque y lo que puede ser deducido de los datos con el fin de identificar el ataque en específico. Finalmente, podría indicar qué información adicional debe ser recolectada en los componentes o unidades incolucrados para poder mejorar el análisis forense.

Patrones Similares

Discute otros patrones de mal uso o uso indebido con diferentes objetivos pero realizados de manera similar, o con objetivos similares pero realizados de otra manera.

Chapter 6

Discusión

Chapter 7

Conclusiones

7.1 Contribuciones

7.2 Trabajo Futuro

Appendix A

Anexos