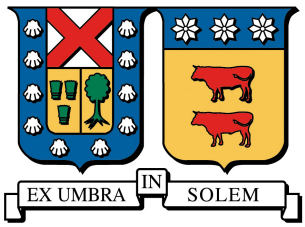
Universidad Técnica Federico Santa María
Departamento de Informática
Valparaíso, Chile



?????

Paulina Andrea Silva Ghio

Memoria para obtar al título de: Ingeniera Civil Informática

Profesor Guía: Raúl Monges
Profesor Correferente: Javier Cañas

22 de septiembre de 2015

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.

Al tener sistemas que se interconectan con el Web Browser, tanto Stakeholder como Desarrolladores deben estar al tanto de los posibles riesgos que su proyecto enfrenta. La falta de educación de Seguridad en los Desarrolladores de un proyecto, la poca y dispersa documentación de los navegadores (así como su estandarización), podría llegar a ser un flanco débil en el desarrollo de grandes architecturas que dependen del Browser para realizar sus servicios. Una Arquitectura de Referencia del Web Browser, utilizando Patrones Arquitecturales, podría ser una base para el entendimiento de los mecanismos de seguridad y su Arquitectura, que interactua con un sistema Web mayor. Ésto mismo, entregaría una unificación de ideas y terminología, al dar una mirada holística, sin tener en cuenta detalles de implementación tanto del Browser como el sistema con el que interactua.

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

The Web Browser is known as one of the most used applications - or *killer app* - and also was the first introduced when the Internet was created (1990). Which is why, it's significant maturity level is above in comparison with other developements and can assure a certain level of *trust* whenever it is used as a client with other systems.

[1]Universidad Técnica Federico Santa María

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Contexto General

Entre 1989 y 1990, Tim Berners-Lee acuñó el concepto de *World Wide Web* y con ésto realizó la construcción del primer *Web Server*, *Web Browser* y las primeras páginas *Web*. Mucho antes que aparecieran los grandes sistemas que ahora conocemos, el *Web Browser* permitía navegar páginas estáticas y realizar una serie de acciones limitadas a las tecnologías de ese tiempo. En la actualidad el *browser* es la herramienta predilecta por todos, desde comprar tickets para una película, realizar reuniones por videoconferencia y muchas otras tareas que invitan a nuevas formas de interactuar y comunicar.

Durante la conocida *guerra de navegadores*, en la decada de los noventa, los browser tuvieron solo el objetivo de poder adquitir la mayor cantidad de usuarios posibles, entregando mejores funcionalidades que sus competidores. Debido a esto era habitual encontrar muchos parches que solucionaban problemas de seguridad, dada la cantidad de errores de programación y casi nula estructura del navegador, dado que no había una pronta preocupación por el diseño. Además la nula documentación que existía debido a la gran competitividad, muchas veces hacía que las extensiones hechas por *third-parties* creaban más agujeros de seguridad, que nuevas funcionalidades. El inicio del Software Libre u *Open Source*, cambió el escenario y las circunstancias pero aún así, existen navegadores propietarios que no exponen la arquitectura de sus aplicaciones.

En el último tiempo el mercado de los *Web Browser* ha crecido bastante (Figura 1.1), principalmente debido a la robustez que éstos poseen y a la cantidad de años que llevan desarrollándose en la industria del Desarrollo de Software. Los navegadores más conocidos son: Google Chrome o su versión Open Source Chromium, Firefox, Internet Explorer, Opera y Safari; siendo los primeros 3 el enfoque de este trabajo.

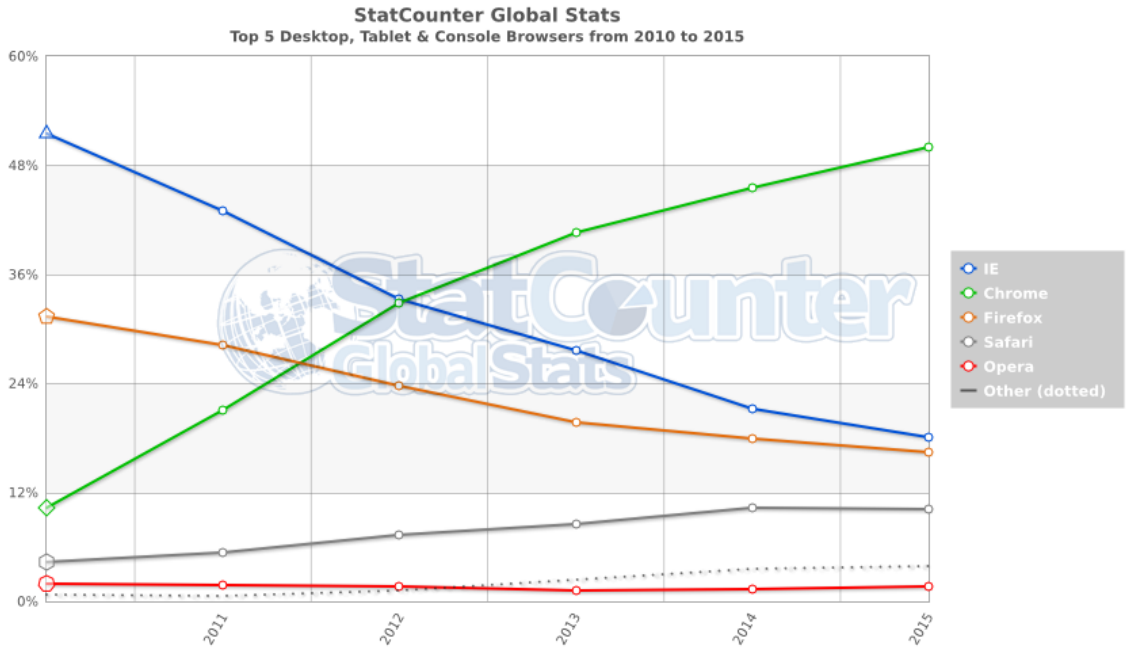


Figura 1.1: Porcentaje de uso de Navegadores [1]

La *Web 2.0* se inició con el uso intensivo de tecnologías como AJAX, y ésto ha permitido una nueva simbiosis entre el usuario, el Web Browser y el Servidor o Web Server que se comunican entre sí. El Navegador Web es una herramienta indispensable para todo tipo de tareas computacionales como comunicacionales, su existencia a penetrado completamente en las labores diarias de todos nosotros. En este mismo instante, la Web a evolucionado nuevamente obteniendo un nuevo nombre: *Web 3.0*, donde se realiza el uso de inteligencia artificial y sistemas de recomendación para generar nuevos tipos de contenido media para el usuario.

1.2. El Problema: Desarrollo de Software y Seguridad

Ningún Desarrollo de Software es igual al anterior. Por cada nuevo proyecto que surge es necesario ver qué tipo de proceso es el que se usará, qué personas serán parte del grupo de trabajo, qué condiciones económicas estará expuesto, qué *Stakeholder* están pendientes de que el Proyecto salga exitoso y un sin numero de variables, no menos importantes a considerar. Por lo tanto dependiendo de lo anterior, los sistemas podrían llegar a ser simples o muy complejos. En consecuencia, se hace necesario te-

ner ciertas metodologías que aseguren que se cumplan con todos los Requerimientos Funcionales como No-Funcionales del Sistema a construir. Sin embargo un problema que existe recurrentemente, es que la mayoría del Software construído contiene numerosos **defectos** y **errores**, generando así **vulnerabilidades** que son encontradas y explotadas por los atacantes, generandp un compromiso parcial o total del sistema [3]. Lo anterior sucede frecuentemente por que los sistemas no son desarrollados teniendo en cuenta la seguridad [4, 5, 6].

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*. 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 [7], sin olvidar que podría llegar a afectar la Confidencialidad, Integridad y Disponibilidad de los datos de los involucrados con el sistema [8]. 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. Según [6] la falta de preocupación en temas de seguridad en desarrollos de software no tiene una raíz principal, diversos factores como: la falta de estudios de seguridad en las mallas curriculares de las Universidades, pocos fondos para la investigación, la falta de iniciativa y precoupación desde la industria, el exceso de confianza de los desarrolladores, etc., son los causantes de las futuras vulneraciones en sistemas críticos.

La literatura que habla de la construcción de *Secure Software* o Software Seguro, indica que los practicantes de 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 forma coordinada probablemente éste no será seguro [9], dado que la seguridad es una Propiedad Sistémica que necesita ser vista de manera holística y al inicio del proceso.

Un sistema cualquiera conectado al internet y accesado por un usuario a través de un Navegador, estará en algún momento de su vida útil bajo amenazas que el Desarrollador debe estar al tanto. Al conocer los peligros es posible comunicar a los *Stakeholders*, de los posibles peligros a los que se enfrentan, aún cuando en el

[1] Application Programming Interface

equipo de Desarrolladores no exista un experto en seguridad. Bajo esta misma lupa, una Arquitectura de Referencia permitiría comunicar efectivamente los componentes, interacciones y comportamientos existentes entre el Browser y el sistema con el que se comunica, de tal manera que sería posible entender los posibles problemas que el Browser puede llegar a generar.

1.3. Motivación: ¿Por qué estudiar el Browser?

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 [10, 11, 12].

En [3, 6] mencionan que la cantidad de tiempo dedicado en temas de seguridad, por los estudiantes en áreas de la Información/Computación/Software es casi nula. Si bien existen diversos factores que pueden ser causa de este comportamientos en las universidades [6], la industria es un fuerte factor en la adopción de un cambio de mentalidad. Desarrollar un sistema crítico (o no) pero seguro, representa un gran desafío. No solo para los desarrolladores, si no también para los involucrados indirectamente, como los usuarios que navegan por el browser para hacer uso de éste sistema. Por lo mismo, es entendible que la seguridad sea un tema complicado de impartir, pero eso no significa que sea imposible. Nuestro trabajo tiene la intención de presentar un

trabajo conceptual, que permita entender más este atributo de calidad y que pueda ser ayudar a otros desarrollos ser más seguros.

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.

De nuestro conocimiento hasta el momento, no existe ninguna propuesta de Arquitectura de Referencia del Browser moderno, solo existe material desactualizado que no se adecua al paisaje actual. Creemos que una falta de conocimientos de seguridad con respecto al Browser, podría afectar de forma directa el desarrollo de aplicaciones que lo utilizan. Por esto mismo, una guía o compilado de información, semi-formal, que una los conceptos y componentes, como lo hace una Arquitectura de Referencia, podría enseñar a desarrolladores no expertos en seguridad, los peligros que existen. Nuestro trabajo explaya a la seguridad como una propiedad sistémica que debe ser tomada en cuenta desde el inicio del sistema [5, 13, 14, 9].

1.4. Contribuciones

Dada la falta de documentación para enseñar a desarrolladores no expertos en seguridad, acerca de los componentes, interacciones y comportamientos del Web Browser: El Objetivo General de esta Memoria es generar un cuerpo organizado de información sobre el Web Browser y su Seguridad, de tal manera que se pueda sistematizar, organizar y clasificar el conocimiento adquirido en un documento, con formato semi-formal, tanto para Profesionales como Estudiantes del área Informática que estén insertos en el área de Desarrollo de Software.

Este trabajo busca cumplir con los siguientes Objetivos Específicos:

- Comprender los conceptos relacionados al navegador web, sus componentes, interacciones o formas de comunicación, amenazas y ataques a los que puede estar sometido, como los también los mecanismos de defensa. Esto se realizará a través de un Estado del Arte sobre el Browser.
- Identificar actores, componentes, funciones, relaciones, requerimientos y restricciones del Navegador, para lograr abstraer una Arquitectura de Referencia (AR) a partir de documentación disponible en internet, blogs de desarrolladores, papers e iniciar un pequeño catálogo de Patrones de Mal Uso. Esto permitirá con-

densar el conocimiento obtenido en el punto anterior a través de documentos semi-formales, lo que permitirá generar una guía para comunicar los conceptos relevantes que pudieran afectar la relación existente entre un desarrollo de software y el navegador.

- Profundizar el conocimiento en ataques relacionados con métodos de Ingeniería Social.

Particularmente se ha escogido como metodología base la dada por el autor del libro [9]. Una Architectura de Referencia (AR) tiene como objetivo el mismo descrito en [11, 12, 15], éste es el ayudar a los *implementors* o desarrolladores del software, a entender los *trade-off* cuando se diseñan nuevos sistemas, y puede ayudar a los mantenedores de estos sistemas a entender el código *legacy* detrás los navegadores que trabajan a mano a mano. Además una Arquitectura de Referencia permite comparar las diferencias en decisiones de diseño del Navegador y así poder entender los cambios realizados a lo largo del Desarrollo de un sistema. Junto con lo anterior, Íla AR permitirá tener una visión holística del sistema y mostrará las decisiones de alto nivel para asegurar la Seguridad del sistema. Por otra parte, los Patrones de Mal Uso o Uso Indebido, permitirán enseñar y comunicar las posibles formas en que tal sistema puede ser usado inapropiadamente.

En este trabajo se presentará nuestra Arquitectura de Referencia y X Patrones de Uso Indebido, que usarán la AR contruída para mostrar los componentes y mensajes que una amenaza puede realizar, con tal de lograr un ataque en el Browser. El uso de patrones nos permitirá abstraer compontentes y comunicaciones entre dichos sistemas, al mismo tiempo que vislumbrará los mecanismos de seguridad implementados. Los patrones son herramientas de gran valor, que permiten generar el entendimiento de los aspectos funcionales y pueden complementar con otros patrones relacionados para alcanzar una arquitectura más entendible. Estos patrones serán presentados usando el template POSA [16] y UML, para así modelar las interacciones entre los diversos componentes de la arquitectura.

1.5. Metodología

Este trabajo se realizará de la siguiente forma:

1. Introducción de un *Framework conceptual* para entender los conceptos relacionados.
2. Contruir un Estado del Arte sobre el Browser, así como también de los posibles peligros a los que se enfrenta.

3. Identificar los conceptos, actores, componentes, interacciones y funciones, en relación al tema principal.

4. Construir patrones de arquitectura que definan los componentes y responsabilidades, con el objetivo final de ser unidos en una Arquitectura de Referencia.

5. Contruir patrones de Mal Uso/Uso Indebido por medio del punto anterior.

1.6. 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 **??**

Capítulo 2

Marco Teórico - Conceptual Framework

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* [17], 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) [18].

- 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" [19], 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 informa-

ción 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 [20].

2.3. Procesos, Comunicación e Información de Estado

2.3.1. 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: Mecanismo de comunicación entre entre ventanas y frames, disponible en la API de HTML5, entre diversos dominios. El comando window.postMessage es usado para realizar llamadas entre diversos orígenes de forma segura. Si bien SOP normalmente denega este tipo de solicitud, si se hace de forma correcta es posible cominicarse con diversos orígenes por medio del uso de este comando.
2. XMLHttpRequest o XHR: [21] define una API que proporciona una guía de funcionalidad al cliente, para que éste pueda transferir datos del cliente al servidor. Dicho de otra manera, XHR es un objeto que permite la obtención de recursos

en la Internet. Soporta peticiones en HTTP o HTTPS, en general soporta toda actividad relacionada con un *HTTP request or response*, para los métodos definos.

3. WebSockets [22]: 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 [23]: 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.

2.3.2. 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.

2.3.3. 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.

2.4. Tecnologías usadas

2.4.1. 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 [24], 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.

2.4.2. 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.

2.4.3. 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* [2] 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.

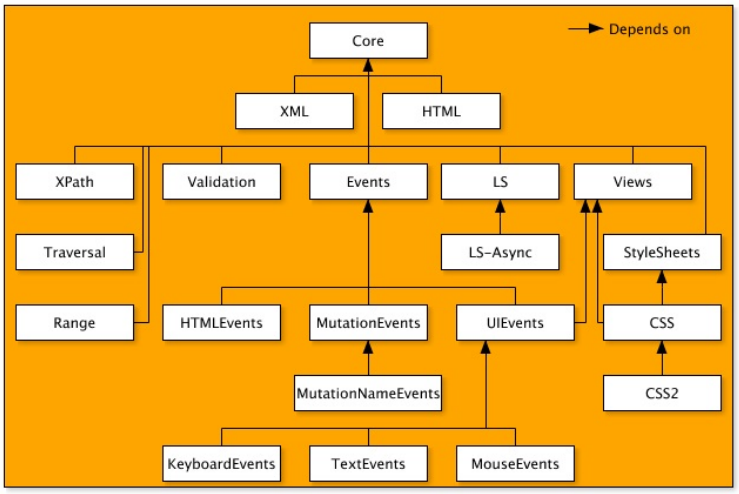


Figura 2.1: Arquitectura de DOM [2]

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.

2.4.4. 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 [25] 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*.

2.4.5. 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*.

2.4.6. 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. La API permite que el autor de una aplicación web, ejecute *trabajadores* que corren scripts en paralelo, la coordinación entre estos se logra a través del paso de mensajes. 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.

2.5. 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 [26] 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 [27], 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.

2.6. 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 [28], realizan sus solicitudes a servidores de diverso origen por medio de la interfaz *XmlHttpRequest*, en el caso de Internet Explorer esta se llama *XDomainRequest*.

2.7. 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.

2.8. Desafíos del Navegador

- Navegación a todo tipo de contenidos/compatibilidad: sin importar el esquema de la página web, el navegador debe ser capaz de presentar todo tipo de contenido al usuario. [29] asegura que los usuarios demandan compatibilidad, por que el browser es sólo útil mientras pueda mostrar las páginas.
- Navegación personalizada: el browser debe ser capaz de entregar la información a la Web Aplicación/sistema, para que identifique al usuario por detrás, de manera que la navegación sea personalizada.
- Navegación sin inconvenientes: el navegador debe ser capaz de aislar los errores que se presenten en algunas páginas, de tal manera que no molesten a las otras

páginas web que se ven al mismo tiempo.

- Seguridad: Los datos de los usuarios y el host donde se mantiene el Browser no deben ser expuestos a terceras partes.

2.9. Arquitectura de Referencia o Reference Architecture (RA)

Una arquitectura de Referencia, de acuerdo a la *Open Security Architecture* o OSA[30], es considerado un elemento que describe un **estado de ser** y debe representar aceptadas buenas practicas. En [15] 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 [31].

Actualmente no hay un consenso de como definir una AR o lo que debería contener [32], [15] 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.

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 [33].

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. Una 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.

[34] menciona que existe una falta de procedimientos para diseñar sistemáticamente una Arquitectura de Referencia, que sea al mismo tiempo fundamentada empíricamente. El mismo trabajo explica que mientras un Arquitecto de Sistema y un Experto de Dominio trabajen juntos, es posible diseñar AR ya sea desde *cero* o basado en artefactos arquitecturales ya existentes. Para una AR no construída desde el comienzo, la evaluación es menos crítica dado que la AR utiliza conceptos arquitecturales ya comprobados por expertos. Por lo tanto, la validación de ésta puede derivarse desde arquitecturas ya construídas, en este caso a partir de los browser que se encuentran en el mercado.

Para describir la Arquitectura de Referencia nos hemos basado en los trabajos [35, 33], usando patrones para la contrucción de la AR. Así como indica [31], es posible usar patrones arquitecturales para diseñar una Arquitectura de Referencia, con tal de obtener atributos de calidad deseados.

2.10. 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 [9] 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 [9] 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 [14, 13] es la primera tarea que presenta la metodología, que considera las actividades en cada caso de uso del sistema.

2.11. 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 [16]. Un patrón encarna el conocimiento y experiencia de desarrolladores de software que puede ser reusado posteriormente en nuevas aplicaciones [5, 9]. 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 [16], llamado POSA, que contiene las siguientes secciones para describir un patrón: *Intent*, Contexto, Problema, Solución, Implementación, Usos comúnes, Consecuencias y Patrones relacionados.

2.12. Patrones de Mal Uso

Para diseñar sistemas seguros, se es necesario identificar las posibles amenazas que un sistema puede sufrir. Papers como [13, 36, 14, 9] 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 [37, 9] 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 [36, 38, 39] se hace la unión de los conceptos de *Attack Patttern* para dar forma a la definición de *Misuse Pattern* [37, 40, 41, 42, 43, 44, 45, 46]:

> 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.

Capítulo 3

Marco Teórico - (In) Seguridad en el Browser

En esta sección se presentan los posibles ataques que un Browser puede sufrir y que directamente podrían afectar al sistema con el que se comunica. Principalmente ahondaremos en los ataques en el Browser relacionados a las técnicas de Ingeniería Social [19]. El escenario actual de los ataques en el browser ha cambiado bastante, si es comparado a aquellos de la decada de los noventa. Cada día los Browsers son más robuztos y difíciles de explotar, por lo mismo, los ataques de tipo *drive-by downloads* o los basados en ejecución de código para vulnerar el sistema, cada vez son menores. Una nueva forma de ataques ha emergido y es al mismo tiempo, una forma más fácil de lograrlo, pues se basa en el engaño del usuario a realizar lo que el atacante desea. Una vez el usuario es engañado, el atacante puede lograr un control total tanto del Browser como del Host, sin haber tenido que vulnerar el sistema [**?**]. Desarrollos de sistemas críticos que interactuan a diario con diferentes usuarios en la red, deberían de ser los más preocupados de estos ataques pues atentan contra la confidencialidad, integridad y disponibilidad de los datos, tanto del usuario (personales) como los de los *Stakeholders* involucrados.

3.1. Social Engineering

[19] define este tipo de acción como: El acto de manipular una persona para realizar acciones que no son parte de los mejores intereses del *blanco o víctima* (la misma persona/organización/etc u otra entidad). Un ataque de éste tipo puede darse de diversas maneras, no dejando la posibilidad de un encuentro físico o digital con el que realiza el engaño. Un ataque basado en ingeniería social, es uno que se aprovecha del comportamiento humano y la confianza de la víctima. En el contexto del Web

Browser, el usuario engañado es la primera y última linea de defensa contra este tipo de ataques, pues un abuso en la confianza del usuario podría abrir las puertas al Host del Browser, logrando un daño tanto del usuario como con los sistemas externos con los que interactúa.

3.2. Ataques y Amenazas

Bibliografía

[1] Global Stats StatCounter. Top 5 desktop, tablet and console browsers, 2015.

[2] World Wide Web Consortium W3C. About.

[3] Karen M Goertzel, Theodore Winograd, Holly L McKinley, Lyndon J Oh, Michael Colon, Thomas McGibbon, Elaine Fedchak, and Robert Vienneau. Software security assurance: A state-of-art report (sar). Technical report, DTIC Document, 2007.

[4] J. Yoder, J. Yoder, J. Barcalow, and J. Barcalow. Architectural patterns for enabling application security. *Proceedings of PLoP 1997*, 51:31, 1998.

[5] Eduardo B Fernandez. A methodology for secure software design. In *Software Engineering Research and Practice*, pages 130–136, 2004.

[6] Bill Whyte and John Harrison. State of Practice in Secure Software: Experts' Views on Best Ways Ahead. IGI Global.

[7] Carnegie Mellon University Computer Emergency Response Team. Early identification reduces total cost (segment from cert's podcasts for bussiness leaders).

[8] Mike Hicks. Interview to **Kevin Haley** (from **Symantec**), 2014. Mike Hicks (Profesor of Software Security course in Coursera.org).

[9] Eduardo Fernandez-Buglioni. *Security patterns in practice: designing secure architectures using software patterns*. John Wiley & Sons, 2013.

[10] M.M. Larrondo-Petrie, K.R. Nair, and G.K. Raghavan. A domain analysis of web browser architectures, languages and features. In *Southcon/96. Conference Record*, pages 168–174, Jun 1996.

[11] Alan Grosskurth and Michael W. Godfrey. A reference architecture for web browsers. pages 661–664, 2005. URL: `http://grosskurth.ca/papers.html#browser-refarch`.

[12] Alan Grosskurth and Michael W. Godfrey. Architecture and evolution of the modern web browser. URL: `http://grosskurth.ca/papers.html#browser-archevol`. Note: submitted for publication.

[13] Eduardo B Fernandez, Michael VanHilst, Maria M Larrondo Petrie, and Shihong Huang. Defining security requirements through misuse actions. In *Advanced Software Engineering: Expanding the Frontiers of Software Technology*, pages 123–137. Springer US, 2006.

[14] Fabricio A Braz, Eduardo B Fernandez, and Michael VanHilst. Eliciting security requirements through misuse activities. In *Database and Expert Systems Application, 2008. DEXA'08. 19th International Workshop on*, pages 328–333. IEEE, 2008.

[15] Paris Avgeriou. Describing, Instantiating and Evaluating a Reference Architecture: A Case Study. *Enterprise Architect Journal*, 342(1):347, 2003.

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

[17] IEEE Cyber Security. Avoiding the top 10 security flaws.

[18] The MITRE Coporation. Common vulnerabilities and exposures.

[19] Jeremiah Talamantes. The social engineer's playbook.

[20] Wade Alcorn, Christian Frichot, and Michele Orrù. *The Browser Hacker's Handbook*. John Wiley & Sons, 2014.

[21] 2014 Draft XMLHttpRequest Level 1. Xmlhttprequest specification.

[22] The WebSocket API. Websocket specification.

[23] WebRTC 1.0: Real time Communication Between Browsers. Webrtc specification.

[24] W3C Working Group. Html5 specification.

[25] Adam Barth, Collin Jackson, and William Li. Attacks on javascript mashup communication. In *Proceedings of the Web*, volume 2. Citeseer, 2009.

[26] Michal Zalewsk. Browser security handbook, part 2. Web page, Google, 2008.

[27] Adam Barth, Collin Jackson, and John C Mitchell. Securing frame communication in browsers. *Communications of the ACM*, 52(6):83–91, 2009.

[28] Bryan Sullivan and Vincent Liu. *Web application security*. McGraw-Hill, 2012.

[29] Adam Barth, Collin Jackson, Charles Reis, TGC Team, et al. The security architecture of the chromium browser, 2008.

[30] Open Security Architecture. Definitions by osa.

[31] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.

[32] A framework for analysis and design of software reference architectures. *Information and Software Technology*, 54(4):417–431, April 2012.

[33] Oscar Encina. Towards a Security Reference Architecture for Federated Inter-Cloud Systems. 2014.

[34] Matthias Galster and Paris Avgeriou. Empirically-grounded Reference Architectures: A Proposal. pages 153–157, 2011.

[35] Keiko Hashizume, Eduardo B Fernandez, and Maria M Larrondo-petrie. A reference architecturefor cloud computing. Submitted for publication. 2014.

[36] Eduardo Fernandez, Juan Pelaez, and Maria Larrondo-Petrie. Attack patterns: A new forensic and design tool. In *Advances in digital forensics III*, pages 345–357. Springer New York, 2007.

[37] E.B. Fernandez, N. Yoshioka, and H. Washizaki. Modeling misuse patterns. In *Availability, Reliability and Security, 2009. ARES '09. International Conference on*, pages 566–571, March 2009.

[38] N Yoshioka. A development method based on security patterns. *Presentation, NII, Tokyo*, 2006.

[39] Nobukazu Yoshioka. Integration of attack patterns and protective patterns. In *1st International Workshop on Software Patterns and Quality (SPAQu'07)*, page 45, 2007.

[40] Juan C Pelaez, Eduardo B Fernandez, and Maria M Larrondo-Petrie. Misuse patterns in voip. *Security and Communication Networks*, 2(6):635–653, 2009.

[41] Eduardo B Fernandez, Nobukazu Yoshioka, and Hironori Washizaki. A worm misuse pattern. In *Proceedings of the 1st Asian Conference on Pattern Languages of Programs*, page 2. ACM, 2010.

[42] Keiko Hashizume, Nobukazu Yoshioka, and Eduardo B Fernandez. Misuse patterns for cloud computing. In *Proceedings of the 2nd Asian Conference on Pattern Languages of Programs*, page 12. ACM, 2011.

[43] Jaime Muñoz-Arteaga, Eduardo B Fernandez, and Héctor Caudel-García. Misuse pattern: spoofing web services. In *Proceedings of the 2nd Asian Conference on Pattern Languages of Programs*, page 11. ACM, 2011.

[44] Eduardo B Fernandez, Ernest Alder, Richard Bagley, and Swati Paghdar. A misuse pattern for retrieving data from a database using sql injection. In *BioMedical Computing (BioMedCom), 2012 ASE/IEEE International Conference on*, pages 127–131. IEEE, 2012.

[45] Ali Alkazami and Eduardo B Fernandez. Cipher suite rollback: A misuse pattern for the ssl/tls client/server authentication handshake protocol. 2014.

[46] Oscar Encina, Eduardo B Fernandez, and Raúl Monge. A misuse pattern for denial-of-service in federated inter-clouds. 2014.