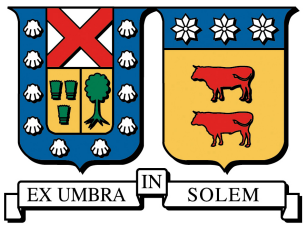
Universidad Técnica Federico Santa María
Departamento de Informática
Valparaíso, Chile



Desarrollo de un catálogo de Patrones de Mal Uso en el Web Browser.

Paulina Andrea Silva Ghio

Memoria para obtar al título de: Ingeniera Civil Informática

Profesor Guía: Raúl Monges
Profesor Correferente: Javier Cañas

2 de mayo de 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

Índice general

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.

En el último tiempo el mercado de los *Web Browser* ha crecido bastante, 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/Chromium, Firefox, Internet Explorer, Opera, and Safari; siendo los primeros 3 los que se trataran en este trabajo.

La *Web 2.0* se inició con el uso intensivo de tecnologías como AJAX, y ésto ha permitido una nueva simbiosis entre el usuario y el Web Browser. 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 y por consiguiente, se hace necesario tener 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, de tal manera que generan el compromiso del sistema completo [1].

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 [2], sin olvidar que podría llegar a afectar la Confidencialidad, Integridad y Disponibilidad de los datos de los involucrados con el sistema [3]. Por esto mismo, es

[1] Application Programming Interface

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 forma coordinada probablemente éste no será seguro [4], dado que la seguridad es una Propiedad Sistémica que necesita ser vista de manera holística y al inicio del proceso.

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 anemazas 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 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 [5, 6].

En [1] menciona que muchas veces 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 [7, 8, 9, 4].

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. Contribuciones

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.
- Construir una Arquitectura de Referencia del Web Browser e iniciar un pequeño catálogo de Patrones de Mal Uso o de Uso Indebido. Esto permitirá condensar 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 alguna entidad y el navegador.
- Clasificar los ataques y mecanismos de defensa (mitigación) de los navegadores Web.
- 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 [4]. Una Architectura de Referencia (AR) tiene como objetivo el mismo descrito en [5, 6], 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 2 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. Estos patrones serán presentados usando el template POSA [10] 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, especialmente sobre la seguridad de éste.
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

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* [11], 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) [12].

- 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" [13], 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 [14].

2.2.2. 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 [15] 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 [16], 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.

2.2.3. 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 [17], 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.

2.2.4. 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.

2.2.5. 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* [18] 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.

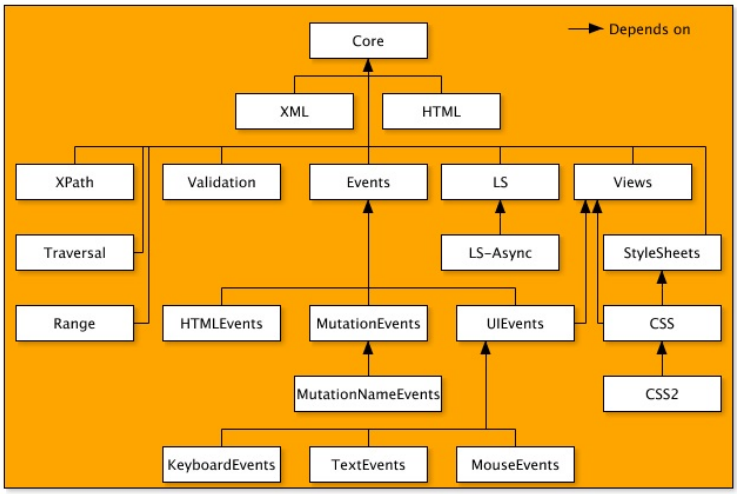


Figura 2.1: Arquitectura de DOM [18]

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.

2.2.6. 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 [19] 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*.

2.2.7. 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*.

2.2.8. 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 que utiliza las capacidades de Javascript y HTML5 para transmitir audio y video. 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.

2.2.9. 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.

2.2.10. 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 [20], realizan sus solicitudes a servidores de diverso origen por medio de la interfaz *XmlHttpRequest*, en el caso de Internet Explorer esta se llama *XDomainRequest*.

2.2.11. 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. 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 que los prinicipios de Seguridad: Confidencialidad, Integridad, Disponibilidad, 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 [4] 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 [4] presenta una completa metodología para construir sistemas seguros a partir de patrones de diseño, 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 [Bra08a] [Fer06c] es la primera tarea que presenta la metodología, que considera las actividades en cada caso de uso del sistema.

2.4. Arquitectura de Referencia o Reference Architecture (RA)

Una arquitectura de Referencia, de acuerdo a la *Open Security Architecture* o OSA[21], es considerado un elemento que describe un **estado de ser** y debe representar aceptadas buenas practicas. En [22] 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. Lo que debe incluir son los componentes principales y fundamentales del sistema, más la interacción entre éstas unidades. 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, sin tener que preocuparme de los detalles de implementación.

2.5. Patrones

2.6. Patrones de Seguridad

2.7. Patrones de Mal Uso

Para diseñar sistemas seguros, se es necesario identificar las posibles amenazas que un sistema puede sufrir. Papers como [4, 8, 23, 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 [24, 4] 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 [23, 25, 26] se hace la unión de los conceptos de *Attack Patttern* para dar forma a la definición de *Misuse Pattern* [24, 27, 28, 29, 30, 31, 32, 33]. Esta nueva definición indica entonces indica que:

> Un patrón de mal uso o *Misuse Pattern* describe, desde el punto de vista del atacante, qué tipo de ataque es realizadom ()

The misuse pattern describes, from the point of view of the attacker, how a type of at- tack is performed (what units it uses and how), analyzes ways of stopping the attack by enumerating possible security patterns that can be applied for this purpose, and describes how to trace the attack once it has happened by appropriate collection and observation of forensic data. It also describes precisely the context in which the attack may occur

Sin embargo, cuando un sistema ya está diseña 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.

Capítulo 3

Estado del Arte del Browser

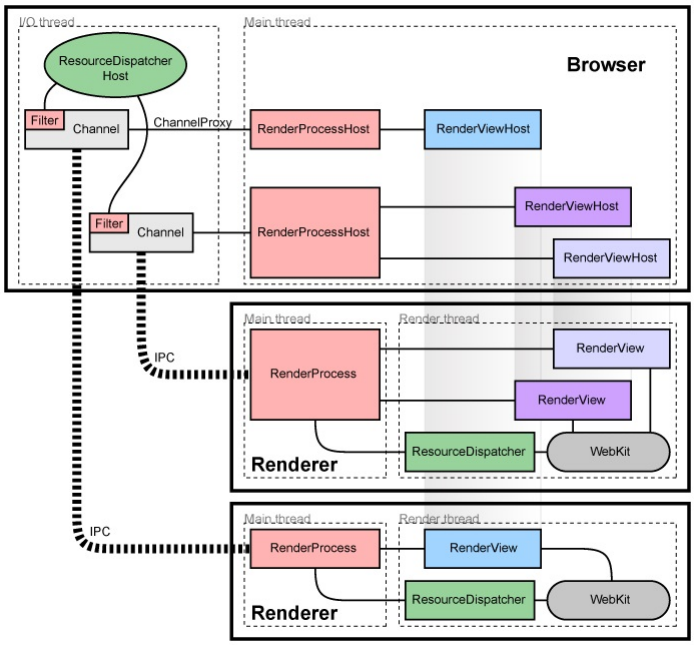
3.1. Navegadores Existentes

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*, siendo la segunda la versión open source. Así como se puede ver en [34] éste Navegador ha sido el último que ha salido y se llevado una gran parte del mercado [35].

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 [36], 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 [37] 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 [38], 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.

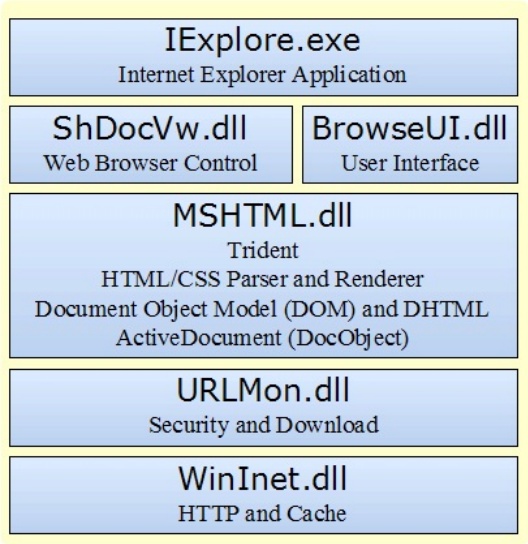
Google Chrome expone en [38] que existen ciertas lecciones que han ido utilizando para mejorar la calidad de su browser. Estas son:

- 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 [39] 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.



El ejecutable IExplore.exe es la base para el corazón del navegador *Mshtml.dll* que se encarga del *parsing* tanto de *CSS* como del *HTML* así como también de la función de renderizado de la página en el navegador; tiene también la tarea de alojar otros componentes dependiendo del contenido del HTML parseado, cómo JScript, XML Data, etc. Con codename *Trident*, expone su interfaz para poder ser alojado por el componente *Shdocvw.dll*, llamado también WebBrowser Control, que se encarga de

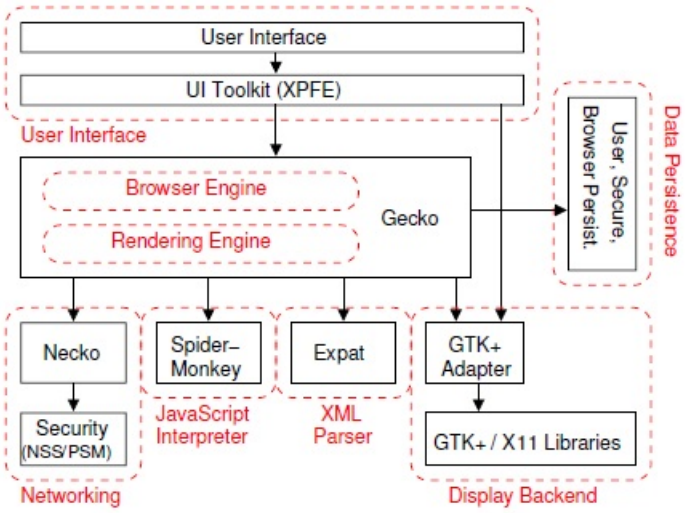
poder dar funcionalidad, navigabilidad y un historial al browser, permitiendo que sea alojada facilmente en una *Aplicación Windows*. La interfaz de usuario es proveída por *Browsui.dll*. Cuando se realiza una descarga de un recurso, *Urlmon.dll* pasa por una serie de pasos para asegurar que el tipo de archivo calce con el tipo *MIME* declarado por el servidor. Finalmente, *WinInet.dll* se encarga de implementar los protocolos de aplicación HTTP y FTP, agregando también el manejo del cache del browser.

La arquitectura de Internet Explorer, basada en la tecnología **COM**, permite extender sus capacidade, por ejemplo: **Mediante Extensiones del browser**, **Extensiones de Contenido** y **Alojo y Reuso**.

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

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. Éstas relaciones entre las dos entidades pueden llegar a ser bien importantes en los desarrollos de Software, pues puede explicar los posibles problemas que indirectamente sufre; considerar al Web browser como un *concern* en el desarrollo de SW 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.

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. Si bien existe un trabajo en concreto donde se plantea una Arquitectura de Referencia como el de [5], no se ha podido encontrar otros similiares.

En el trabajo de [5] se entrega la arquitectura de Referencia basada en el descubrimiento de las Arquitecturas concretas de los browser usados a través de herramientas de Ingeniería Inversa *Reverse Engineering*. Sin embargo lo que relatan en el trabajo es algo que ya está muy desactualizado para los browsers modernos.

En [6] se realiza una actualización del trabajo (un año después) pero enfocada a descubrir más arquitecturas concretas de otros web browser conocidos, en especial aquellos Open Source como: Konqueror, Lynx, Firefox, entre otros.

3.3. Evolución y Seguridad en el Browser

En [5] y [6] 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 [37] 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

Capítulo 4

Arquitectura de Referencia del Browser

4.1. Casos de Uso de Browser

4.1.1. Stakeholders (actores) y Concerns de estos

4.1.2. Casos de Uso

Actor 1: XXX

Actor 2: XXX

4.2. Patrón Browser

Capítulo 5

Patrones de Mal Uso

5.1. Identificando Amenazas

5.2. Template de Patrones de Mal Uso

Capítulo 6

Discusión

Capítulo 7

Conclusiones

7.1. Contribuciones

7.2. Trabajo Futuro

Bibliografía

[1] 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.

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

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

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

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

[6] 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.

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

[8] 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.

[9] 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.

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

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

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

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

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

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

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

[17] W3C Working Group. Html5 specification.

[18] World Wide Web Consortium W3C. About.

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

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

[21] Open Security Architecture. Definitions by osa.

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

[23] 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.

[24] 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.

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

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

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

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

[34] Google Team. Evolution of the web.

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

[36] Google Chromium. Multi-process architecture.

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

[38] Charles Reis, Adam Barth, and Carlos Pizano. Browser security: lessons from google chrome. *Queue*, 7(5):3, 2009.

[39] Internet Explorer Architecture.

Apéndice A

Anexos