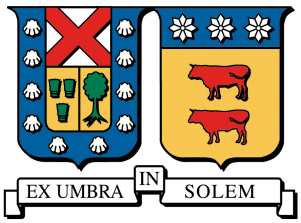
Universidad Técnica Federico Santa María
Departamento de Informática
Valparaíso, Chile



????

Paulina Andrea Silva Ghio

Memoria para obtar al título de: Ingeniera Civil Informática

Profesor Guía: Raúl Monges
Profesor Correferente: Javier Cañas

22 de abril 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.

En la actualidad, muchos Desarrollos de Software usan ésta técnología dado que grandes compañías como Google, Microsoft, Mozilla, entre la más importantes, dedican la mayoría de su tiempo en asegurar la calidad de sus Navegadores Web, lo que se traduce en un gasto menor de esfuerzo en el Desarrollo. Sin embargo, dado que la misma Internet es considerada como una red no confiable, estos sistemas que usan al navegador como Cliente Web, introducen nuevas amenazas de seguridad que deben ser tomadas en cuenta en el Desarrollo del sistema.

Esta Memoria 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

Índice general

Índice general IV

1. Introducción **1**
1.1. Contexto General . . . 1
1.2. Contexto General: Web Browser . . . 1
1.3. Problema: Desarrollo de Software y Seguridad . . . 2
1.4. Motivación: ¿Por qué estudiar el Browser? . . . 3
1.5. Contribuciones . . . 4
1.6. Metodología . . . 5
1.7. Estructura del Documento . . . 5

2. Framework Conceptual **6**
2.1. Definiciones Básicas . . . 6
2.2. Browser . . . 8
2.2.1. Arquitectura Cliente/Servidor . . . 8
2.2.2. SOP: Same Origin Policy . . . 9
2.2.3. Markup Languages . . . 10
2.2.4. CSS: Cascading Style Sheets . . . 11
2.2.5. DOM: Document Object Model . . . 11
2.2.6. Javascript, VBScript y otros . . . 12
2.2.7. CORS: Cross-Origin Resource Sharing . . . 13
2.2.8. HTTP: Hypertext Transfer Protocol . . . 14
2.3. Desarrollo de Software Seguro y Diseño de Software Seguro . . . 14
2.4. Arquitectura de Referencia . . . 15
2.5. Patrones . . . 16
2.6. Patrones de Seguridad . . . 16
2.7. Patrones de Mal Uso . . . 16

3. Estado del Arte del Browser **17**
3.1. Navegadores Existentes . . . 17
3.1.1. Google Chrome y Google Chromium . . . 17
3.1.2. Internet Explorer . . . 18
3.1.3. Firefox . . . 20

3.2. Evolución y Seguridad en el Browser 21
3.2.1. Estandarizaciones 21
3.2.2. Vulnerabilidades 21
3.2.3. Amenazas 21
3.2.4. Medidas de Mitigación o Mecanismos de Defensa 21
3.3. Arquitectura de Referencia del Browser y Patrones 22
3.4. Sumario 22

4. Arquitectura de Referencia del Browser **23**
4.1. Casos de Uso de Browser 23
4.1.1. Stakeholders (actores) y Concerns de estos 23
4.1.2. Casos de Uso 23
4.2. Patrón Browser 23

5. Patrones de Mal Uso **24**
5.1. Identificando Amenazas 24
5.2. Template de Patrones de Mal Uso 24

6. Discusión **25**

7. Conclusiones **26**
7.1. Contribuciones 26
7.2. Trabajo Futuro 26

Bibliografía **27**

A. Anexos **30**

Capítulo 1

Introducción

1.1. Contexto General

Mucho antes que aparecieran los grandes sistemas que ahora conocemos, el *Web Browser* era la herramienta usada por todos para navegar, las ya viejas páginas estáticas, y que permitía un sin número de acciones a aquellos que necesitaban conectarse al *Internet*. En la actualidad el *browser* sigue siendo 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.

La *Web 2.0*, que se inició con el uso intensivo de tecnologías como AJAX, ha permitido una nueva simbiosis entre el usuario y el Web Browser. Haciendo de éste último, una herramienta casi indispensable para todo tipo de tareas computacionales, tal que 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 nuevo tipo de contenido media para el usuario.

En esta memoria se quiere presentar al Departamento de Informática (DI) de la UTFSM[1] Casa Central, sobre la investigación que contempla un estudio del Navegador Web.

1.2. Contexto General: Web Browser

Entre 1989 y 1990, Tim Berners-Lee acuño el concepto de *World Wide Web* y con ésto realizó la construcción del primer *Web Server*, *Web Browser* y las primeras páginas *Web*. El fin de su construcción era poder

[1]Universidad Técnica Federico Santa María

1.3. Problema: Desarrollo de Software y Seguridad

Ningún Desarrollo de Software es igual al anterior. Cada vez que un nuevo Proyecto surge es necesario ver qué tipo de Proceso es el que se usará, qué personas serán parte del grupo de trabajo, qué condiciones económicas está expuesto, qué tipo de *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 es necesario tener ciertas metodologías que aseguren que se cumplan con todos los Requerimientos Funcionales como No-Funcionales del Sistema a construir. Sin embargo, por muy buen Plan de Proyecto que se tenga, Jefe de Proyecto, Analista, Arquitecto, Programadores, Testers, etc. el proyecto podría peligrar antes o después de terminado, simplemente por que tuvo **Problemas en la Seguridad**.

Un hecho recurrente para todos los sistemas distribuidos en la Internet, es la gran cantidad de vulnerabilidades y amenazas a las que se enfrentan día a día. En el área de la Ciberseguridad o *Cyber Security* (otra forma de decir *Computer Security*), se dice que tanto atacantes como defensores de la seguridad están constantemente realizando un *Juego del Gato y el Ratón*. Esto es dado a que constantemente los atacantes se ingenian nuevas formas para cometer delitos informáticos, a través de las vulnerabilidades en los sistemas que están sobre la Internet, y que las grandes compañías de seguridad, tratan de detectar - y crear una solución - lo antes posible. El 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. Sin embargo muchas veces ocurre que aunque se corrijan estos nuevos ataques, no todos los sistemas que podrían llegar a necesitar del mismo parque para protegerse del ataque, realizan la actualización y adecuada configuración de sus sistemas para que se puedan proteger de una posible amenaza que explote la vulnerabilidad recientemente encontrada.

Si bien el **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[2] 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*. Por lo mismo, lidiar con las preocupaciones de seguridad es un factor que puede ser clave para el Desarrollo de Software: Una vez que el sistema este en *Deployment*, las consecuencias de no tener en cuenta la Seguri-

[2]Application Programming Interface

dad desde el inicio del Desarrollo de Software pueden ser muy costosas [1], incluso pudiendo afectar la Confidencialidad, Integridad y Disponibilidad de los datos de quienes lo usan [2].

El problema de la mayoría del Software construído es que 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 [3]. Por esto mismo, es imperante que sean entendidos los Requerimientos de Seguridad del Sistema a construir desde el inicio de su construcción y que todos los involucrados también 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.

Un sistema seguro o **Secure Software**, no tendrá en lo posible, vulnerabilidades que puedan ser explotadas. Sin embargo, hay que tener en cuenta que dado que el Desarrollo de Software - incluso los sistemas seguros - dependen de personas, procesos, y tecnologías, será imposible tener un 100 % de confiabilidad en la Seguridad implementada, pues los "fallas humanas" o "descuidos humanos" siempre existirán. Si el sistema seguro no pudiera llegar a resistir un ataque, lo primero que debe intentar es aislarse del resto y degradar con gracia, de manera que no afecte el rendimiento del sistema. Finalmente, si un ataque tuvo a lugar lo importante es que el equipo detrás del sistema no se quede inmovil, luego de recuperarse del compromiso es necesario tener inmediatamente un parche que solucione la vulnerabilidad existente. Más aún, si esta vulnerabilidad se encontrara en un componente externo o *Third Party code*, el sistema que llegara a hacer uso de esa pieza de Software tiene que tener medidas para que no afecte la totalidad del sistema.

1.4. Motivación: ¿Por qué estudiar el Browser?

En el pasado era bien visto que cada Desarrollo de Software, pudiera levantar todo su sistema sin tener que depender de externos para funcionar. Este pensamiento, sin embargo, ha cambiado mucho en el último tiempo tanto debido a factores sociales, económicos, técnicos y otros.

Actualmente las nuevas formas para Desarrollar Software, tanto métodos como procesos, tratan de ajustarse lo más posible a lo que su Cliente le pida, para poder cumplir con las metas del Proyecto a Desarrollar en el tiempo estipulado. Muchas veces esto conlleva a reducir gastos que terminan por afectar: la calidad del *Software*, sus atributos sistémicos, y en consecuencia la Seguridad tanto del sistema, como la del cliente que lo utiliza.

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 para el usuario del sistema.
- Actualmente la Seguridad implementada en los *Web Browser* es bastante buena, dado que existen en la actualidad grandes compañias se aseguran de eso (Google, Microsft, Mozilla entre las más conocidas). Esto permite que usar el *Browser* sea una opción bastante económica para el desarrollo de otros sistemas (y concentrarse solo en eso).
- 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, compras, y muchos más.

Sin embargo, los sistemas que dependen del uso del *Browser* deben de tener en cuenta de las posibles anemazas de seguridad a las que se enfrentan, por el solo hecho de usar el Navegador. 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 del nuevo Sistema, tener conocimientos de la seguridad de Cliente.

En [3] hace referencia a un hecho que personalmente la autora considera es una realidad incluso en la actualidad de Chile, ésta es la falta de cursos o educación en temas de Seguridad a los estudiantes de Ingeniería de Software (Nota: preguntar en otras universidades). De tal manera que éstos estudiantes no aprenden acerca de 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 [5, 4].

Este trabajo desea 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.

1.5. 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 fácil de entender, tanto para Profesionales como Estudiantes del área Informática.

Particularmente, este trabajo busca cumplir con los siguientes Objetivos Específicos:

- Comprender los conceptos relacionados al navegador web, sus componentes, interacciones o formas de comunicación, seguridad y ataques a los que puede estar sometido, mecanismos de defensa y otros, a través del Estado del Arte a construir.
- Construir una Arquitectura de Referencia del Web Browser e iniciar un pequeño catálogo de Patrones de Mal Uso, *Misuse Patterns*, que indican los posibles mal usos que se puede dar con la información que posee un Navegador. Esto permitirá condensar el conocimiento obtenido en el punto anterior a través de documentos semi-formales.
- Clasificar los ataques y mecanismos de defensa (mitigación) de los navegadores Web. (REVISAR)
- Profundizar el conocimiento en ataques relacionados con métodos de Ingeniería Social.

1.6. Metodología

XXX

1.7. 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* [6], 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. Las vulnerabilidades más críticas son documentadas en la *Common Vulnerabilities and Exposures* (CVE) [7].

- 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" [8], 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 [9].

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 *Script* 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.

SOP es la base de la mayoría de los principios de seguridad, dado que sin él 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 [10] menciona que no existe una sola forma de SOP, si no que es una serie de mecanismos que superficialmente se parecen, pero al mismo tiempo poseen diferencias:

- SOP para acceso al DOM: se dará permiso de modificar el DOM solamente aquellos scripts que ponean el mismo dominio, puerto (para todos los browsers excepto Internet Explorer) y protocolo.

- SOP para el objeto XMLHTTPRequest
- SOP para *cookies*
- SOP para Flash, donde usa políticas para realizar peticiones fuera del dominio através de un archivo **crossdomain.xml**
- SOP para Java
- SOP para Silverlight, parecido al de Flash solo qe utiliza otros elementos.
-

Para algunos SOP puede ser un tanto molesto, tanto para los atacantes como los desarrolladores de aplicaciones. 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 [11], 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 (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 [12] 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 aninados 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 aprovechada por los atacantes para insertar'cósigo scripting en las páginas.

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 es usado junto a HTML o XML para definir la capa de presentación de las páginas web que el navegar renderiza al usuario. La W3C o World Wide Web Consortium 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 permitiría a programas JavaScript modificar la estructura del contenido de una página y además de ser portable para los Browser, en el tiempo en que se usaba Dynamic HTML para esa tarea. Posteriormente la *W3C* [13] 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.

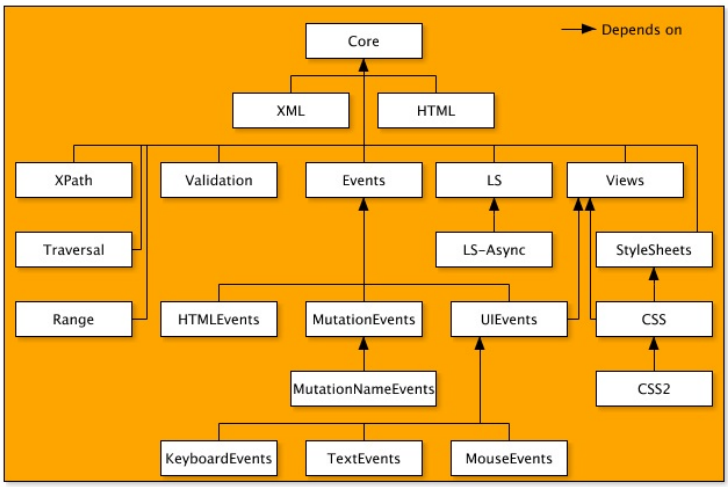


Figura 2.1: Arquitectura de DOM [13]

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, puede llegar a almacenar 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, Microsft 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 es 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 [14] 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. CORS: Cross-Origin Resource Sharing

Cómo lo define su nombre es un mecanismo que permite al cliente realizar request entre sitios de diverso *Origen*. *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.

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

2.2.8. HTTP: Hypertext Transfer Protocol

El Protocolo de la capa de Aplicación conocido como HTTP fue creado en los años 90 por el W3C o **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.

Canales de comunicación en HTTP

postMessage

XHR: XMLHttpRequest

WebSockets

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

Una arquitectura de Referencia, de acuerdo a la *Open Security Architecture* o OSA[16], es considerado un elemento que describe un **estado de ser** y debe representar aceptadas buenas practicas.

En este trabajo lo que se pretende hacer con la Arquitectura de Referencia, es dar a entender los componentes y elementos que la mayoría de los Web Browser tienen. Se sabe que el Browser es un pieza de Software que ha sufrido varios cambios desde 1990, 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, incluso existan ciertos **patrones** que pueden explayarse de buena manera mediante una Arquitectura de Referencia, que manifeste los componentes, mecanismos de comunicación, funcionamientos, etc.

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, 17, 18, 5] 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 [19, 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 [18, 20, 21] se hace la unión de los conceptos de *Attack Patttern* para dar forma a la definición de *Misuse Pattern* [19, 22, 23, 24, 25, 26, 27, 28]. 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

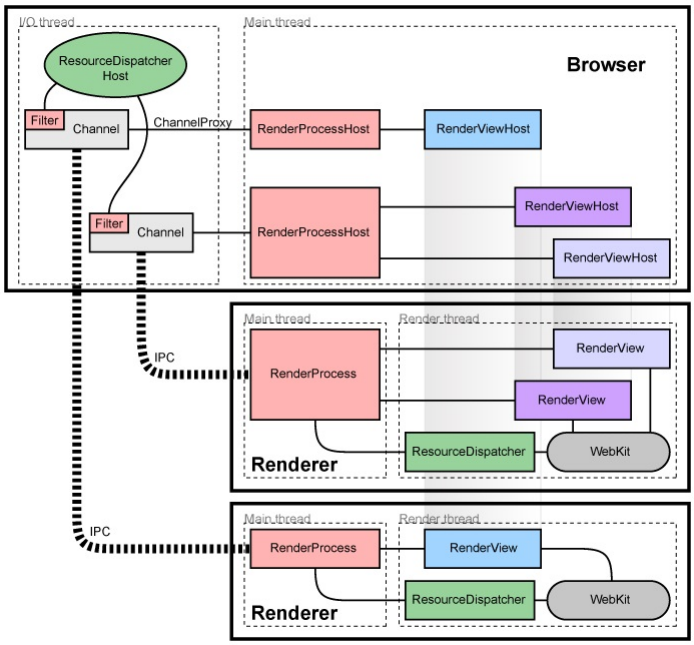
3.1. Navegadores Existentes

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 en el mundo. 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.

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 [29], 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 [30] 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 [31], 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 [31] 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.

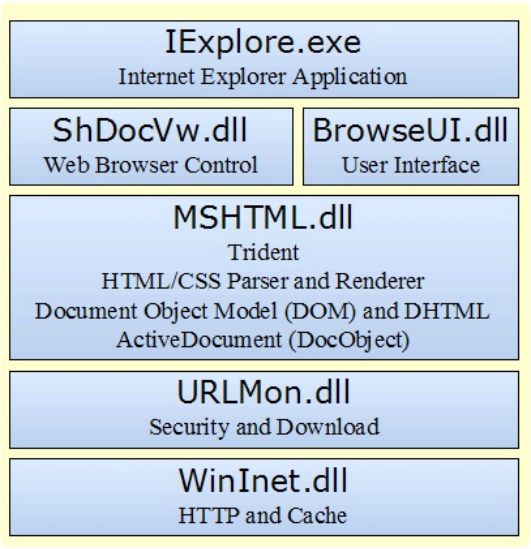
- Reducción de la frecuencia de exposición, Google trabaja con StopBadware.org para entregar una mayor seguridad al descubrir nuevos tipos de ataques y vulnerabilidades relacionadas con el browser.

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 [32] 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

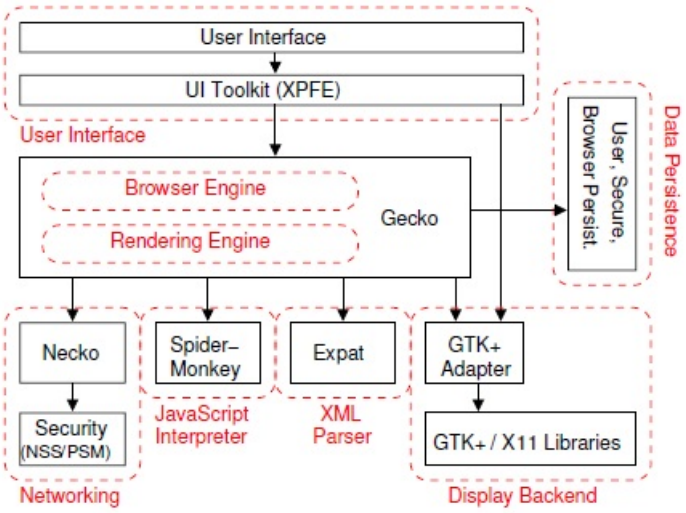
protocolos de aplicación HTTP y FTP, agregando también el manejo del cache del browser.

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. Evolución y Seguridad en el Browser

3.2.1. Estandarizaciones

3.2.2. Vulnerabilidades

3.2.3. Amenazas

3.2.4. Medidas de Mitigación o Mecanismos de Defensa

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

2. Sandoxing
   En el desarrollo de [30] 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

3.3. Arquitectura de Referencia del Browser y Patrones

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

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

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

[5] 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.

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

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

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

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

[10] Technical report.

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

[12] W3C Working Group. Html5 specification.

[13] World Wide Web Consortium W3C. About.

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

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

[16] Open Security Architecture. Definitions by osa.

[17] 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.

[18] 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.

[19] 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.

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

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

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

[23] 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.

[24] 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.

[25] 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.

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

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

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

[29] Google Chromium. Multi-process architecture.

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

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

[32] Internet Explorer Architecture.

Apéndice A

Anexos