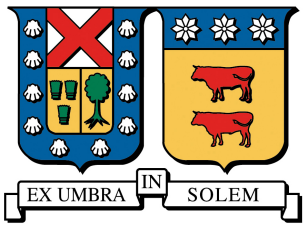
Universidad Técnica Federico Santa María
Departamento de Informática
Valparaíso, Chile



Una Arquitectura de Referencia para Web Browsers: Hacia una unificación de Conceptos de Seguridad

Paulina Andrea Silva Ghio

Memoria para optar al título de: Ingeniería Civil Informática

Profesor Guía: Raúl Monge
Profesor Correferente: Javier Cañas

24 de noviembre de 2015

Si sta come
d'autunno
sugli alberi
le foglie
***Soldati** di Guiseppe Ungaretti (1918)*

Agradecimientos

Con esta Memoria voy a dar por finalizado los largos 6 años de la carrera, que terminaron por ser casí 7 y algo más por el Magister. No me arrepiento de mis decisiones en la parte académica, pero a veces en lo personal creo que pudo haber sido de otra manera. Tuve muchos desafíos que realizar al estudiar para este trabajo, partiendo por mi nulo conocimiento en Seguridad y a veces la falta de motivación. Sin embargo, estoy acá en este momento gracias a esas decisiones y experiencias que he ganado en el intertanto. Gracias a todas las personas que me motivaron, en especial, a mi profesor guía Raúl Monge (UTFSM) y el profesor Eduardo Fernandez (Florida Atlantic University), quienes me mostraron un camino interesante a seguir.

Quiero agradecer a todos mis amigos de la Universidad, que son muchos, y aquellos que la abandonaron por un camino mejor o más interesante. Gracias por haberse presentado en mi vida y hacerla entretenida. Sin la risa y diversión en mi vida, probablemente sería una persona amargada. Gracias a todos por soportar mi malhumor y comprenderme cuando estoy angustiada. Gracias también por haberme ayudado academicamente, que si bien todos creen que soy matea nada tengo de eso; solo estudio demasiado y a veces no de la mejor manera.

Gracias Mamá y Papá por darme educación. Se que no es fácil pagar tanto dinero todos los meses, pero en verdad se los agradezco siempre. Gracias a que he tenido la oportunidad de estudiar en una Universidad, se cuán importante es la educación y que debo aprovecharla al máximo. Se que muchas veces tuvimos varios problemas mientras yo crecía, pero quiero hacerles saber que estoy agradecida de ser su hija. Gracias por cuidar y confiar en mi.

Gracias Panchomon por todo, eres una existencia muy importante en mi vida y espero que lo sepas. Gracias Micchan, por ser mi manta cuando estoy triste. Ustedes dos han sido los pilares más grandes que he tenido en estos años de universidad. Gracias Fernando, Samuel, José Miguel, Francisco, Matías y Andrés, por ser mis *dudes* y soportarme siempre. Gracias Priscilla, Scarlet y Susana, por ser mis amigas más cercanas en la Universidad. Ko-chan, Yurie, Asumi, Geso y Motoki a ustedes también gracias por apoyarme siempre. Finalmente, ¡gracias a todos!, por tenerme una gran paciencia a lo largo de estos años, por levantarme el ánimo cuando no quiero

dar más pasos hacia adelante, por inspirarme a ser mejor, y por estar en mi vida de alguna que otra forma. También muchas gracias al que lee este documento, en especial, por darse el tiempo de hacerlo.

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 permite 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, y lamentablemente tanto Stakeholders como muchos desarrolladores no están al tanto de los riesgos a los que se exponen.

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

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

Abstract

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

[1]Universidad Técnica Federico Santa María

Currently a lot of software developments create systems that are connected to the Internet, which allows to add functionality within a system and facilities to their *Stakeholders*. This leads to depend in a *web client*, as the *Web Browser*, which allows access to services, data or operations that the system delivers. Nevertheless, the Internet influences the attack surface of the new system, and unfortunately many stakeholders and developers are not aware of the risks they are exposed.

Having systems which are interconnected with the *Web Browser*, Stakeholder and Developers should be aware of the potential risks they could face. The lack of Security Education in Software developers of a project, the low and scattered documentation of each browser (and standardization), could become a great flaw in big architectural developments which depends on the browser to do their services. A Reference Architecture of the *Web Browser*, using Architectural Patterns, could be a base for understanding the security mechanisms and its architecture, which interacts with a bigger web system. This would give an unification of ideas and terminology, giving a holistic view regardless the implementation details for both the browser and the system it communicates to.

This work presented to the Departamento de Informática (DI) of the UTFSM[2] Casa Central, will seek within the scope of *Web Browser* Security, has the objective or goal to obtain semi-formal documents that can help as tools to software developers who make a strong use of a web browser within the activities of the systems they are building.

[2]Universidad Técnica Federico Santa María

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Contexto General

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

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

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

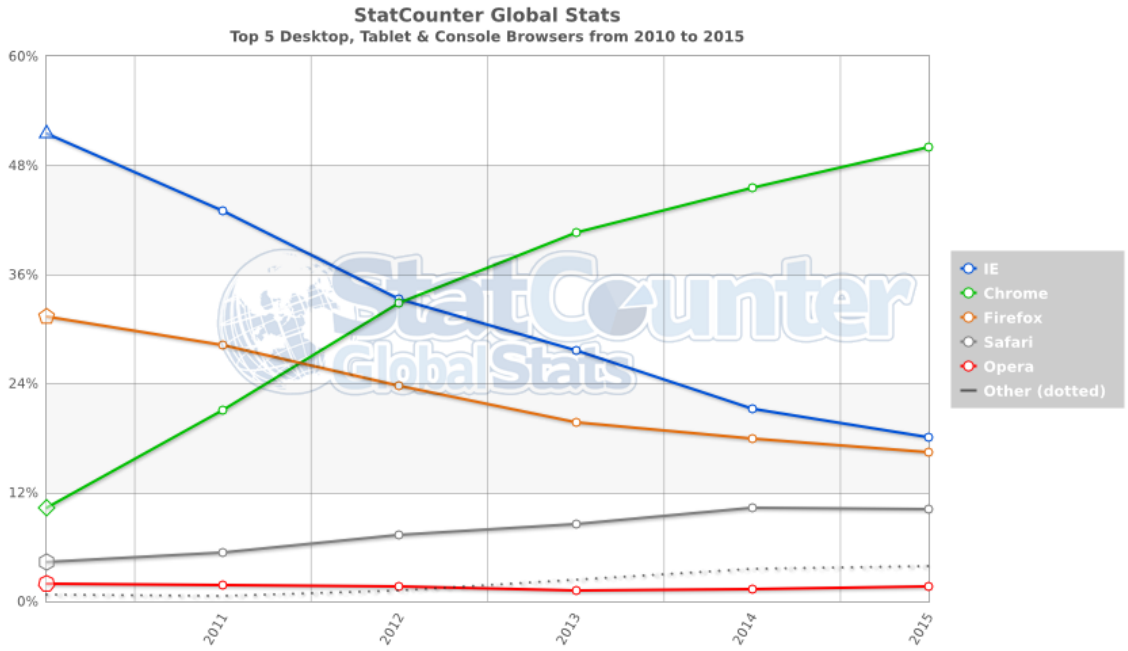


Figura 1.1: Porcentaje de uso de Navegadores. Fuente: [1]

La *Web 2.0* se inició con el uso intensivo de tecnologías como AJAX, y ésto ha permitido una nueva simbiosis entre el usuario, el *Web Browser* y el Servidor o Web Server que se comunican entre sí. El navegador Web es una herramienta indispensable para todo tipo de tareas computacionales como comunicacionales, su existencia ha 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 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 número de variables, no menos importantes a considerar. Por lo tanto, dependiendo de lo anterior, los sistemas podrían llegar a ser simples o muy complejos. En consecuencia, se hace necesario 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, generando un compromiso parcial o total del sistema [14]. Lo anterior sucede frecuentemente por que los sistemas no son desarrollados teniendo en cuenta la seguridad [15, 16, 17].

Muchas veces al desarrollar sistemas, se prefiere utilizar API's[1] de otros sistemas para poder incluir funcionalidades ya implementadas, fomentando así el reuso de piezas de Software. Si bien lo anterior es una buena práctica, si el sistema no cuenta con las medidas de seguridad necesarias, estas piezas podrían ser causa de amenazas de seguridad que terminarían por corromper el sistema y en consecuencia podría causar una pérdida monetaria a los *Stakeholders*. Lo expuesto anteriormente ejemplifica perfectamente con 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 [18], sin olvidar que podría llegar a afectar la confidencialidad, integridad y disponibilidad de los datos de los involucrados con el sistema [19]. Por esto mismo, es imperante que sean entendidos, desde el comienzo, las preocupaciones de los *Stakeholders* y los Requerimientos de seguridad asociados, y que además todos los involucrados los entiendan perfectamente. Según [17], la falta de preocupación en temas de seguridad en desarrollos de software no tiene una raíz principal, sino diversos factores como: la falta de estudios de seguridad en las mallas curriculares de las Universidades, pocos fondos para la investigación, la falta de iniciativa y preocupación desde la industria, el exceso de confianza de los desarrolladores, etc., son los causantes de las futuras vulneraciones en sistemas críticos.

La literatura que habla de la construcción de *Secure Software* o Software Seguro, indica que los practicantes de Desarrollo de Software deben entender, en gran medida, los problemas de seguridad que podrían llegar a ocurrir en sus sistemas. No basta con saber cómo 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 [20], dado que la seguridad es una Propiedad Sistémica que necesita ser vista de manera holística y al inicio del proceso.

Un sistema cualquiera conectado a Internet y accedido por un usuario a través de un navegador, estará en algún momento de su vida útil bajo amenazas que el Desarrollador debe estar al tanto. Al conocer las amenazas es posible comunicar a los *Stakeholders*, de los posibles peligros a los que se enfrentan, aún cuando en el equipo de Desarrolladores no exista un experto en seguridad. Bajo esta misma lupa, una Arquitectura de Referencia permitiría comunicar efectivamente los componentes,

[1] Application Programming Interface

interacciones y comportamientos existentes entre el *browser* y el sistema con el que se comunica, de tal manera que sería posible entender mejor los posibles problemas de seguridad que el *Browser* puede llegar a generar.

1.3. Motivación: ¿Por qué estudiar el Browser?

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

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

Sin embargo los sistemas que dependen del uso del *Browser*, deben de tener en cuenta las posibles amenazas de seguridad a las que se enfrentarán por el solo hecho de usarlo. Para un proyecto de gran envergadura, sería un error no tener en consideración los posibles peligros que trae el uso del *Browser*, y es el deber de todo integrante del equipo de desarrollo tener el conocimiento de la seguridad del Cliente Web. El entendimiento de la estructura subyacente del *Web Browser* podría asegurar que las personas que trabajen en el desarrollo, comprendan los *trade-off* al momento de diseñar un Software que necesite la colaboración del navegador Web [21, 10, 11]; poniéndolo de otra forma, un atacante es capaz de explotar un ataque en el sistema, dado que tiene el conocimiento subyacente del *Browser*, conoce sus vulnerabilidades y mecanismos de defensa.

En [14, 17] mencionan que la cantidad de tiempo dedicado en temas de seguridad, por los estudiantes en áreas de la Información/Computación/Software es casi nula. Si bien existen diversos factores [17] que pueden ser causa de este comportamientos, principalmente las universidades y las industrias son un fuerte factor en la adopción de un cambio de mentalidad. Desarrollar un sistema crítico (o no) pero seguro, representa un gran desafío. No solo para los desarrolladores, si no también para los involucrados indirectamente, como los usuarios que navegan con un *browser* para hacer uso de éste sistema. Por lo mismo, es entendible que la seguridad sea un tema complicado de

impartir, pero eso no significa que sea imposible. Nuestro trabajo tiene la intención de presentar un marco conceptual, que permita entender más este atributo de calidad y que pueda ayudar a otros desarrollos ser más seguros.

Este trabajo tiene una motivación principal. Ésta es ayudar a quien 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 Software, obtengan el conocimiento necesario al momento de trabajar junto con el navegador Web al realizar un Desarrollo de Software que dependa de éste.

De nuestro conocimiento hasta el momento, no existe ninguna propuesta de Arquitectura de Referencia del *Browser* moderno, solo existe material desactualizado que no se adecua al paisaje actual [10]. Creemos que una falta de conocimientos de seguridad con respecto al *Browser*, podría afectar de forma directa el desarrollo de aplicaciones que lo utilizan. Por esto mismo, una guía o compilado de información, semi-formal, que una los conceptos y componentes, como lo hace una Arquitectura de Referencia, podría enseñar a desarrolladores no expertos en seguridad, los peligros que existen. Nuestro trabajo considera a la seguridad como una propiedad sistémica que debe ser tomada en cuenta desde el inicio de su desarrollo [16, 22, 23, 20, 24].

1.4. Contribuciones

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

Este trabajo busca cumplir con los siguientes **Objetivos Específicos**:

- Comprender los conceptos relacionados al navegador web, sus componentes, interacciones o formas de comunicación, amenazas y ataques a los que puede estar sometido, como también los mecanismos de defensa. Esto se realizará a través del desarrollo de un Estado del Arte sobre el *Browser*.
- Identificar actores, componentes, funciones, relaciones, requerimientos y restricciones del navegador, para lograr abstraer una Arquitectura de Referencia (AR) a partir de documentación disponible en Internet, blogs de desarrolladores, pa-

pers e iniciar un pequeño catálogo de Patrones de Mal Uso. 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 un desarrollo de software y el navegador.

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

Particularmente se ha escogido como metodología base la dada por Fernandez[23, 20]. Una Arquitectura de Referencia (AR) tiene como objetivo el mismo descrito en [10, 11, 25, 26], éste es capturar la esencia de la arquitectura a través de una colección de sistemas similares, por medio del reuso arquitectónico, y además 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* usado. Una Arquitectura de Referencia permite además 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 un Patrones de Uso Indebido, que usarán la AR construida para mostrar los componentes y mensajes que una amenaza puede realizar, con tal de lograr un ataque en el *Browser*. El uso de patrones nos permitirá abstraer componentes y comunicaciones entre dichos sistemas, al mismo tiempo que vislumbrará los mecanismos de seguridad implementados. Los patrones son herramientas de gran valor, que permiten generar el entendimiento de los aspectos funcionales y pueden complementar con otros patrones relacionados para alcanzar una arquitectura más entendible. Estos patrones serán presentados usando el template POSA [27] 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. Hablar sobre las (in)seguridades en el *Browser*, a través de los posibles ataques que se pueden dar en el *Browser* y cómo éstos pueden afectar a los sistemas que lo usan.

3. Presentar un Estado del Arte sobre el *Browser*, sobre las distintas arquitecturas, lo que se ha hecho en Arquitecturas de Referencia en el *Browser*, o patrones relacionados a éste.

4. Identificar los conceptos, actores, componentes, interacciones y funciones, para poder generar una Arquitectura de Referencia del *Browser*. Con esto se construirá patrones de arquitectura que definan los componentes y responsabilidades.

5. Construir 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 2 se presenta un marco teórico de las tecnologías usadas en el navegador.
- Luego de tener un extenso conocimiento de lo que actualmente es conocido como **Web Browser**, el capítulo 3 presenta un marco teórico de la (in)seguridad del *Browser* donde se verán los posibles ataques que se puedan dar a esta pieza de software.
- En el capítulo 4 se presenta el Estado del Arte de las propuestas de Arquitecturas de Referencias o similares sobre el *Web Browser*, así como la existencia de los *Browser* más usados y Patrones existentes. Además una descripción de los componentes más importantes en cada navegador es entregada.
- La Arquitectura de Referencia, nuestra propuesta, se presenta en el capítulo 5. Esta presenta la abstracción de las propuestas vistas en el capítulo 4.
- Previo a la construcción del Patrón de Mal Uso en el capítulo 6, se analizarán las amenazas que existen en el *Browser* a través de la AR construída.
- Finalmente se presentarán las Conclusiones en el capítulo 7, sobre el trabajo realizado y se presentará posibles trabajos a futuro sobre este mismo tema.

Capítulo 2

Marco Teórico - Tecnologías Web

Nota: En la sección 8 se encuentran conceptos básicos que el lector podría encontrar necesarios antes de empezar este trabajo.

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 ha 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* esté 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 [28].

2.2. Comunicación e Información de Estado

2.2.1. HTTP: Hypertext Transfer Protocol

El Protocolo de la capa de Aplicación conocido como HTTP fue creado en los años 90 por el **World Wide Web Consortium** [2] y la **Internet Engineering Task Force**, define una sintaxis y semántica que utilizarían el software basado en una arquitectura Web para comunicarse. El protocolo sigue un esquema de solicitud-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*

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 esto se lleve a cabo, las más conocidas son:

1. postMessage [29]: Mecanismo de comunicación entre ventanas y frames, disponible en la API de HTML5, entre diversos dominios. El comando window.postMessage es usado para realizar llamadas entre diversos orígenes de forma segura. Si bien SOP normalmente denega este tipo de solicitud, si se hace de forma correcta es posible cominicarse con diversos orígenes por medio del uso de este comando.

2. XMLHttpRequest o XHR: [30] define una API que proporciona una guía de funcionalidad al cliente, para que éste pueda transferir datos del cliente al servidor. Dicho de otra manera, XHR es un objeto que permite la obtención de recursos en la Internet. Soporta peticiones en HTTP o HTTPS, en general soporta toda actividad relacionada con un *HTTP request or response*, para los métodos definidos.

3. WebSockets [31]: 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 [32]: O mejor conocido como *Web Real-Time Communication*, es una API basada en la especificcación de la W3C, que utiliza las capacidades de Javascript y HTML5 (sin la utilización de plugins externos o internos) para transmitir audio, video y compartir archivos por medio de P2P. Esta herramienta permite a los browsers comunicarse entre ellos a muy baja latencia y entrega un gran ancho de banda/*bandwidth* para poder realizar comunicaciones multimedia en tiempo real. Hasta el momento Google Chrome/Chromium y Firefox han implementado esta tecnología, con el objetivo de: mejorar la experiencia de usuario al no necesitar plugins para ser usada, y entregar seguridad dado que impone el uso de cifrado en los datos.

2.2.2. SSL/TLS Cifrado en capa de Transporte

Si bien existe una medida de seguridad en los headers que se implementa en la capa de Aplicación por medio de HTTP, esto no impide que otros puedan ver qué contienen los paquetes. La confidencialidad, autenticidad y el no repudio de lo que se envía, es un aspecto relevante cuando se está trabajando con sistemas con información crítica y confidencial. SSL (Secure Socket Layer) y TLS (Transport Layer Security) [33] tienen el objetivo de proveer un canal confiable y privado de todo lo que se envía entre dos aplicaciones que se comunican; es una seguridad *end-to-end*. TLS es el resultado de la estandarización de SSL por la Internet Engineering Task Force (IETF). SSL/TLS trabaja debajo del protocolo HTTP, usando certificados de clave pública que permiten:

- Resolver parcialmente el problema de la autenticación de un usuario, al establecer un canal seguro y cifrado mediante el uso de certificados digitales.
- Identificar que la información enviada por los dos *endpoints* sea solo de ellos dos, agregando una firma al final del paquete usando la clave privada de la entidad que envía.
- Asegurar que todo lo que se envía sea visto sólo por las entidades que crean el canal de comunicación, a través del cifrado y descifrado.

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

1. Un usuario desea conectarse por el *Browser* a un Web Server.
2. Se inicia el proceso de *Handshake* entre el *Browser* y Servidor. Éstos dos se ponen de acuerdo en cómo se cifrará la comunicación (parámetros e información de los certificados) e intercambian una llave asimétrica.
3. El navegador valida el certificado, ejemplo: revisa si está expirado, revocado o fue creado por una CA *Certificate Authority* confiable.
4. Si el servidor requiere un certificado por parte del cliente, el *Browser* le enviará el suyo. Esto permitirá tener una autenticación mutua entre las partes.
5. El *Web Browser* y el Servidor usan las llaves públicas del otro para poder acordar una clave simétrica, que es aquella que permitirá cifrar los mensajes. Sólo estas dos entidades conocerán tal clave.
6. El proceso de *handshake* termina y todo lo posterior se realiza cifrando los paquetes con la llave simétrica acordada por las partes.

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

2.2.3. Speedy o Protocolo SPDY

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

2.3. Tecnologías usadas

2.3.1. Markup Languages

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

HTML: HyperText Markup Language

HTML [34], 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*/etiqueta inicial y uno final; estos *tags* pueden ir anidados y la idea es que 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 datos entre *web Services* o interacciones **RPC**, dado que no hay forma de como malinterpretar los datos.

2.3.2. CSS: Cascading Style Sheets

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

2.3.3. DOM: Document Object Model

Es una *API* independiente del lenguage y multiplataforma para HTML válido y bien formado, que define la estructura lógica de un documento que permite ser accedido y manipulado. DOM es una especificación que permite a programas Javascript modificar la estructura del contenido de una página dinamicamente. Esto permite que una página pueda cambiar sin la necesidad de realizar nuevas peticiones al servidor y sin la interacción del usuario. Posteriormente la *W3C* [2] formó el *DOM Working Group* y con ello se creó la especificación a través de la colaboración de muchas empresas y expertos. La arquitectura de esta *API* se presenta en la Figura 2.1, donde el *Core Module* es donde están las interfaces que deben ser implementadas por todas las implementaciones conformes de DOM. Una implementación de DOM puede ser construida por uno o más módulos dependiendo del host, ejemplo de esto: la implementación de DOM en un servidor, donde no es necesaria la implementación de los módulos que manejen los triggers de eventos del mouse.

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 cual posee una cierta funcionalidad e identidad. La estructura de árbol del DOM construido puede llegar a ser gigantezca, y almacena más de un árbol

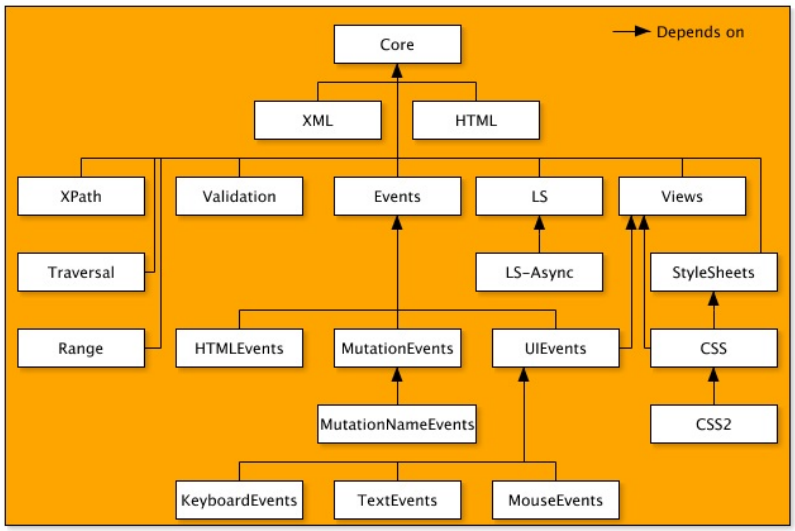


Figura 2.1: Arquitectura de DOM. Fuente: [2]

por cada documento que parsea.

2.3.4. Javascript, VBScript y otros

Ambos son lenguajes de scripting orientados a objetos. Javascript fue desarrollado por Netscape mientras que VBScript fue desarrollado por Microsft para Internet Explorer, los dos siguen el estándar 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 Explorer no usa realmente JavaScript, sino 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 [35, 36, 37, 38, 39] se muestran 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 críticas, 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*.

2.3.5. Geolocalización

Cada *Browser* posee una API que permite obtener los datos 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*.

2.3.6. WebWorkers

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

2.4. Desafíos del Navegador

- Navegación a todo tipo de contenidos/compatibilidad: sin importar el esquema de la página web, el navegador debe ser capaz de presentar todo tipo de contenido al usuario. [40] asegura que los usuarios demandan compatibilidad, por que el browser es sólo útil mientras pueda mostrar las páginas.
- Navegación personalizada: el browser debe ser capaz de entregar la información a la Web Aplicación/sistema, para que identifique al usuario por detrás, de manera que la navegación sea personalizada.

- Navegación sin inconvenientes: el navegador debe ser capaz de aislar los errores que se presenten en algunas páginas, de tal manera que no molesten a las otras páginas web que se ven al mismo tiempo [41, 11, 40]
- Seguridad: Los datos de los usuarios y el host donde se mantiene el *Browser* no deben ser expuestos a terceras partes [42, 43, 37, 38, 35, 44].

2.5. Arquitectura de Referencia o Reference Architecture (AR)

Una arquitectura de Referencia, de acuerdo a la *Open Security Architecture* o OSA [45], es considerado un elemento que describe un **estado de ser** y debe representar aceptadas buenas prácticas. En [25, 26] se explica que una AR es una arquitectura de software genérica y estandarizada, para un dominio particular e independiente de la plataforma o detalles de implementación. En ésta especifica la decomposición del sistema en subsistemas, las interacciones entre estas partes y la distribución de funcionalidad entre ellas [46].

Actualmente no hay un consenso de cómo definir una AR, lo que debería contener y cómo debería de construirse [25, 26] describe un ejemplo e indica cómo debería de ser ésta, con los siguientes elementos:

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

Las ventajas y usos que se obtienen al construir una AR en este trabajo son:

- Comprender la estructura subyacente de un *Web Browser* y las interacciones que tendrá con otros sistemas.
- Proveer una base tecnológica modular y flexible. Al tener los subsistemas compartimentalizados es posible quitar y sacar piezas, que poseen interfaces similares, y de esa manera reusar lo otro sin tener que construir un sistema nuevo.

- Entrega una base para el desarrollo de otros Navegadores Web, sin explicar detalles de implementación.

En este trabajo el enfoque estará en el primer punto, donde se quiere entender las interacciones entre un desarrollo de Software y la utilización de las funcionalidades del navegador. Dado que parte de la investigación es obtener Patrones de Mal Uso o Uso Indebido del navegador Web, es primordial concebir una Arquitectura de Referencia que permita encontrar dónde es posible aplicar Patrones de seguridad para poder mitigar los malos usos del *Browser* [47].

Una AR es una herramienta que facilita el entendimiento de sistemas complejos y la apropiada implementación a sistemas reales. Si bien una AR es usada principalmente para capturar las preocupaciones de los *Stakeholders* al comienzo de un Desarrollo de Software, también puede ser usada para educar al realizar la unión de ideas y terminologías usadas por diversos sistemas que se asemejen. Una Arquitectura de Referencia debe ser en lo posible descrita de la forma más abstracta posible, pues su función de guiar la construcción de arquitecturas concretas, sin tener en cuenta detalles de las tecnologías usadas.

2.5.1. Validación de la Arquitectura

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

Para describir la Arquitectura de Referencia nos hemos basado en los trabajos [48, 47], usando patrones para la contrucción de la AR. Así como indica [46], es posible usar patrones arquitecturales para diseñar una Arquitectura de Referencia, con tal de obtener atributos de calidad deseados.

2.6. Desarrollo de Software Seguro y Diseño de Software Seguro

La filosofía detrás de *Secure Software Development* es que detrás de cada etapa de desarrollo del software, se tengan en cuenta los prinicipios de seguridad: confidencialidad, integridad, disponibilidad y auditoría. Para cumplir este cometido es que se deben llegar a políticas y reglas que aseguren la seguridad como una propiedad sistémica.

Varias comunidades tienen diferentes enfoques y técnicas de cómo asegurar la seguridad en los sistemas, muchas pueden incluso tener similitudes y hasta trabajar juntas. En este trabajo, el enfoque tomado es aquel 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 [49, 20] sostiene que para construir un sistema seguro es necesario realizarlo de manera sistemática de tal manera que la seguridad sea parte 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. Fernandez en su libro [20] presenta una completa metodología para construir sistemas seguros a partir del Diseño Orientado a Objetos, UML y patrones, a los cuales nombra como **Security Patterns**.

Como parte de la metodología propuesta, se plantea que para diseñar primero se deben entender las posibles amenazas a las que está expuesto el sistema. La identificación de Amenazas [23, 22] es la primera tarea que presenta la metodología, que considera las actividades en cada caso de uso del sistema.

2.7. Patrones

Los Patrones encapsulan soluciones recurrentes a problemas y definen una forma de expresar los requerimientos y soluciones de una forma concisa, al mismo tiempo que proveen de un vocabulario común entre los diseñadores [27]. Un patrón encarna el conocimiento y experiencia de desarrolladores de software que puede ser reusado posteriormente en nuevas aplicaciones [16, 50, 24, 20]. Los Patrones expresan las relaciones entre un contexto, un problema y una solución. Para un contexto dado, el

patrón puede ser adaptado para encajar en diversas situaciones. La construcción de Patrones de Seguridad parte de la premisa anterior, éste permite construir sistemas seguros a través del uso de Patrones adaptados a las necesidades del sistema y preocupaciones de los *Stakeholders*. Por otra parte, una Arquitectura puede ser descrita a través de Patrones, permitiendo que haya un mejor entendimiento al momento de proveer con guías de diseño y análisis a desarrolladores.

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

La Arquitectura de Referencia a confeccionar será realizada por medio de patrones y éstos serán descritos con el template creado por [27], llamado POSA, que contiene las siguientes secciones para describir un patrón: *Intent*, Contexto, Problema, Solución, Implementación, Usos comúnes, Consecuencias y Patrones relacionados.

2.8. Patrones de Mal Uso

Para diseñar sistemas seguros, se es necesario identificar las posibles amenazas que un sistema puede sufrir. Papers como [22, 51, 23, 20] describen el desarrollo de una metodología completa para encontrar amenazas, a través del análisis de actividades 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*, pues claramente en [52, 20] 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 [51, 53, 54] se hace la unión de los conceptos de *Attack Patttern* para dar forma a la definición de *Misuse Pattern* [52, 55, 56, 57, 58, 59, 60, 61]:

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

Sin embargo, cuando un sistema ya está diseño y construido, como es el caso del *Web Browser*, lo que va a importar es saber **cómo** los componentes del sistema, pueden ser usados por el atacante para alcanzar sus objetivos. Importante es entender que éste patrón describe un contexto en dónde puede ocurrir el ataque.

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

Capítulo 3

Marco Teórico - (In) Seguridad en el Browser

En esta sección se presentan los posibles ataques que un *Browser* puede sufrir y que directamente podrían afectar al sistema con el que se comunica. Principalmente ahondaremos en los ataques en el *Browser* relacionados a las técnicas de Ingeniería Social [62]. El escenario actual de los ataques en el *browser* ha cambiado bastante, si es comparado a aquellos de la década de los noventa. Cada día los Browsers son más robustos y difíciles de explotar, y por lo mismo los ataques de tipo *drive-by downloads* o los basados en ejecución de código para vulnerar el sistema, cada vez son menores. Una nueva forma de ataque ha emergido y es bastante fácil de lograrlo, pues se basa en el engaño del usuario a realizar lo que el atacante desea. Una vez el usuario es engañado, el atacante puede lograr un control total tanto del *Browser* o del Host, sin haber tenido que vulnerar el sistema [63, 64] que aloja al *Browser*. Desarrollos de sistemas críticos que interactuán a diario con diferentes usuarios en la red, deberían de ser los más preocupados de estos ataques, pues atentan contra la confidencialidad, integridad y disponibilidad de los datos, tanto del usuario (personales) como los de los *Stakeholders* involucrados.

3.1. Ataques y Amenazas

Esta sección incluye algunos ataques posibles de realizar en un *Browser* y que podrían afectar tanto directa como indirectamente a un sistema externo. Acá no incluiremos ataques en donde el Host ya ha sido vulnerado con anticipación, o aquellos que puedan correr software con los privilegios de un usuario del sistema Host, es decir, aquellos donde el Host ya ha sido controlado directamente por medio de alguna vulnerabilidad del sistema. En el caso anterior, los Browsers ya nada pueden hacer

para detener un ataque de esa magnitud.

En el Top Ten [65] de la OWASP (Open Web Application Security Project) - los diez riesgos de seguridad más importantes en Aplicaciones Web - se puede distinguir en el año 2013 los riesgos directamente relacionados a amenazas de seguridad en el *Browser*. Algunos como: Injección (A1), Manejo de sesiones y autenticación roto (A2), XSS (A3), CSRF (A8) y uso de componentes con vulnerabilidades conocidas (A9), son los riesgos que las organizaciones podrían sufrir en sus sistemas cuando se realizan ciertos ataques en el *Browser*.

En trabajos [40, 66] se puede observar que existen ataques que pueden generar secuelas en otros sistemas, si es que el navegador es afectado en primera instancia. Algunas amenazas existentes son:

1. Compromiso de los componentes del navegador (plugins incluidos) que poseen privilegios de usuario.
2. Compromiso del Host/Sistema.
3. Robo de datos en el tráfico.
4. Compromiso de páginas web (y su data) de orígenes distintos.
5. Fijación de sesión o secuestro de ésta.
6. Compromiso de los canales de comunicación del *Browser*.

3.1.1. Social Engineering o Ingeniería Social

[62] define este tipo de acción como: El acto de manipular una persona para realizar acciones que no son parte de los mejores intereses del *blanco o víctima* (la misma persona/organización/etc u otra entidad). Un ataque de éste tipo puede darse de diversas maneras, no dejando la posibilidad de un encuentro físico o digital con el que realiza el engaño. Un ataque basado en ingeniería social, es uno que se aprovecha del comportamiento humano y la confianza de la víctima. En el contexto del *Web Browser*, el usuario engañado es la primera y última línea de defensa contra este tipo de ataques, pues un abuso en la confianza del usuario podría abrir las puertas al Host del *Browser*, logrando un daño tanto del usuario como con los sistemas externos con los que interactúa.

Si pudieramos dividir los ataques de Social Engineering, podríamos encontrarnos con un panorama parecido al de la Figura 3.1. En esta figura es posible observar que los ataques de Clickjacking se encuentran afuera del ámbito de los ataques de Ingeniería Social, esto es dado a que el usuario que está navegando, pudo haber accedido a una

página web conocida por él, sin siquiera que el atacante haya influenciado a éste a visitar el sitio malicioso. Sin embargo, Clickjacking puede terminar convirtiéndose en un ataque de Social Engineering si se combina con otros procedimientos.

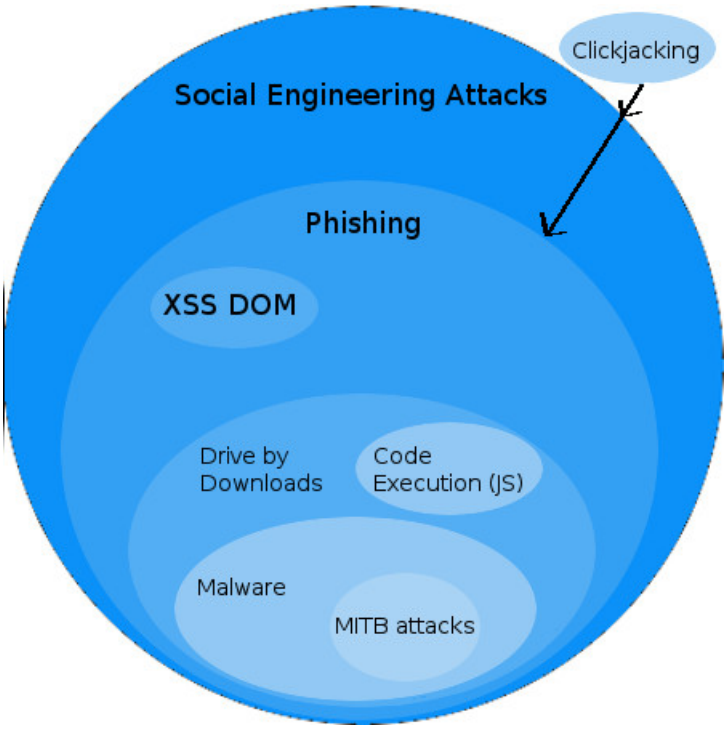


Figura 3.1: Esquematización de ataques de tipo Social Engineering.

3.1.2. Análisis de ataques

Según los estudios [67, 64, 68] indican que el *Browser* es la primera linea de defensa en contra de multiples amenazas en la Web. Sin embargo, esto se ve afectado bastante por la falta de educación de los usuarios que utilizan los navegadores y la constante evolución de las amenazas [67] Es por esto que muchos de los manufacturadores de browsers crean mecanismos de defensa [69] que actuen al momento de solicitar una página, usando black o white list, sistemas de reputación [63] con avisos de alerta al usuario, para evitar que éste al menos tome la decisión de poder ingresar al sitio malicioso.

3.1.3. Phishing

Este ataque consta principalmente del engaño al usuario, confundiéndolo a que visite una página deshonesta en vez de la que tenía pensado. En el último tiempo

muchos de los ataques que actualmente están ocurriendo se han limitado al uso de técnicas de ingeniería social. Si bien los ataques basados en drive-by downloads y clickjacking siguen siendo de alto impacto, los atacantes parecen preferir los otros por la simplicidad de éste, pues no es necesario conocer realmente vulnerabilidades del *Browser* para llevarlos a cabo. Sin embargo, el Phishing es una buena herramienta que permite generar muchos otros ataques, que son mucho más peligrosos. Un resultado obtenido dentro del experimento en [67], menciona que la vida útil en promedio de un sitio de Phishing es de aproximadamente 26 horas. Esto principalmente es debido a que los sistemas de defensa, antes de saber si es malicioso o no, debe descubrir, validar y clasificar en el Sistema de Reputación, tan pronto como sea posible. Un ataque de Phishing puede tener los siguientes ataques asociados:

Instalación de Malware o Extensiones malignas

Un ataque de éste tipo puede ser originado desde la ejecución de un ataque Phishing a una persona; en especial cuando se hace creer que lo que se va a instalar es completamente inofensivo. Sin embargo, el resultado de la aceptación de usuario a la instalación puede tener consecuencias bastante graves, tanto en el *Browser* como en el Host, donde la mayoría de las veces el ataque puede terminar como un Man-in-the-Browser [70, 71]. En la jerga de NSS Labs estos ataques son llamados SEM (Social Enginerring Malware), en el estudio [68] afirma que conocer la identidad de los atacantes que utilizan Social Engineering permite una mejor protección contra los SEM. El trabajo recomienda que las empresas y grandes organizaciones deberían revisar los reportes de otros Laboratorios de seguridad cuando seleccionan un tipo de *Web Browser*; Las empresas deben entender que el mercado de los Browsers no es estático y muchas amenazas aparecen constantemente. El estudio concluye que la Educación es un componente importante en la protección de SEM, pues aquellos usuarios que son capaces de identificar ataques de Ingeniería Social (SE), necesitan menos tecnologías para la protección de estos ataques.

Extensiones vulnerables

Muchas veces un ataque Phishing se aprovechará de las extensiones *benign-but-buggy*; extensiones que tienen un propósito benigno para el usuario que las usa, pero que un atacante puede aprovecharse de alguna vulnerabilidad [37, 38]. Además dependiendo del tipo de arquitectura que el *Browser* pueda tener, es posible que el atacante utilice una extensión que tenga permisos a ciertos recursos del Host o que pueda coludirse con otra extensión benigna para lograr un ataque en forma conjunta [44]; donde lo peor de todo, es que el tráfico no será detectado como malicioso.

Ejecución de Código Javascript

Este tipo de ataques ocurren cuando es posible ejecutar un script en el interprete del cliente, de manera que pueda generar acciones maliciosas dentro del *Browser* del Cliente. Este tipo de ataque funciona muy bien, cuando el atacante desea que el *Web Browser* de la víctima realice acciones en nombre del atacante, de tal manera que la identificación de éste se haga casi imposible. En ciertas situaciones un atacante podría aprovecharse de alguna vulnerabilidad asociada a los modelos de seguridad que Javascript y DOM poseen, pues ambos son totalmente transversales. Javascript utiliza un modelo de seguridad en sus objetos basado en **Capabilities**, mientras que DOM refuerza el **Same Origin Policy** por medio del uso de un **Reference Monitor** para controlar el acceso de código de diversos **orígenes**. Es en esta situación donde podría llevarse a cabo la ejecución de código javascript, aún cuando éste pueda ser de un **origen** distinto al receptor [36].

XSS DOM - Cross-Site Scripting DOM

Es un ataque XSS en donde su *payload* de ataque es ejecutado como resultado de la modificación del *ambiente* DOM en el navegador de la víctima usado por el código script original; en consecuencia el código de cliente se ejecuta de una manera inesperada. El ataque Cross-Site Scripting (XSS) normalmente se asocia con ataques que afectan directamente al servidor, pero existe una tercera variedad que afecta directamente al navegador [39] y que puede ser tan peligrosa como su forma **Reflejada** o **Almacenada**, ésta es llamada XSS DOM o **tipo-0 XSS** [72, 73]. Un ejemplo de un simple XSS DOM puede verse en [74]. Un UXSS o Universal XSS es un ataque más particular, y tiene la habilidad de ser gatillado al explotar una falla en el interior del *Browser* [75], a diferencia de otros XSS que buscan la vulnerabilidad dentro de la aplicación web con el que se comunican.

Man in the *Browser* (MitB)

Un Man in the *Browser* es una técnica utilizada por un atacante para poder leer o modificar datos que se envían entre el cliente Web y el Servidor que responde a las *request*. La diferencia sustancial entre un Man-in-the-Browser (MitB) y un Man in the Middle (MitM) es el dominio que ataca, el primero es al nivel de la capa de aplicación, mientras que el segundo tiene que ver más con el canal de comunicación. [71] explica las grandes diferencias entre estos dos ataques, donde principalmente un MitB se inicia con un ataque Phishing que logra convencer al usuario de instalar alguna extensión o ejecutar una pieza de código [70, 75], que permita al atacante estar entre el *Browser* y el Host, de tal manera que todas las solicitudes pueden ser escuchadas y modificadas dentro del mismo Host.

3.2. Mecanismos de Defensa del *Browser*

3.2.1. 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 como 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 [76].

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 acceder 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 [77] se 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 **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 esto.

- **SOP** para el objeto XmlHttpRequest: 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 [43], 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 éste 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.

3.2.2. CORS: Cross-Origin Resource Sharing

Cómo lo define su nombre, es un mecanismo (especificación) que permite al cliente realizar peticiones entre sitios de diverso **Origen u Origin**, ignorando el **SOP**. *CORS* define una forma en que el Browser y el Servidor Web puedan interactuar para

determinar si permitir o no la petición a otro **Origen**. Un *Browser* utiliza SOP para restringuir las peticiones 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 campo en el header de 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 quien lo solicitó, se podrá realizar el uso del recurso en el navegador, de lo contrario se generará un error.
- Otro campo llamado **Origin**, pero esta vez en la cabecera de la solicitud, para permitir al Servidor hacer cumplir las limitaciones en las peticiones de distinto **Origen**.
- En algunos casos un browser deberá agregar en el header el campo *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 respuesta/*response* del servidor es afirmativa luego se enviará la petición 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 la petición/*request* inmediatamente.

La gran diferencia de CORS con cualquier otro método es que permita hacer peticiones hacia un **Origen** distinto, es que el *Browser* por omisión no enviará ningún tipo de información que permita identificar al usuario. 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 Explorer [78], realizan sus solicitudes a servidores de diverso **Origen** por medio de la interfaz *XmlHttpRequest*, en el caso de Internet Explorer esta se llama *XDomainRequest*.

3.2.3. HTTP Fields/Campos HTTP

HTTP es un protocolo de la capa de aplicación del modelo OSI, que todo dispositivo debe seguir si desea conectarse a la Internet. El *header* o cabecera que utiliza este protocolo permiten configurar la comunicación entre un *Web Server* y un cliente web, en este caso con el Browser. Estos campos del *header* 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 una cabecera con campos que el cliente también debe entender. Algunos campos son necesarios y hasta obligatorios, para algunos servidores, y en otros da lo mismo como vayan.

- Content Security Policy Flag: Es un mecanismo de defensa creado 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 a la cabecera del servidor, en la petición del cliente, el campo 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 campo 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 en la cabecera/*header*, obligará al *Browser* a renderizar la página así como lo dice el campo content-type. La idea de este *flag* es poder limitar la ejecución del tipo objeto que pide el browser.
- Strict-Transport-Security [79]: 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 campo 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.

3.2.4. 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 [80]. 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*.

En el desarrollo de Chromium [40] se define un modelo de amenazas donde 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.

Mientras Google Chrome e Internet Explorer utilizan un Sandbox para sus procesos de renderizado/*rendering* [3], Firefox no ha realizado este trabajo ni siquiera en su versión monoproceso [81].

Google

El Sandbox de Google es una implementación propia y que también aprovecha las técnologías disponibles en el Sistema Operativo. La idea del Sandbox es restringuir el acceso o peticiones al sistema de archivos, de modo que la única manera es pidiendo a otro que realice el trabajo que el proceso dentro del Sandbox necesita.

Internet Explorer

Un nuevo mecanismo para aislar procesos es introducido en Windows 8, llamado AppContainer, es el principal mecanismo usado por Enhanced Protected Mode para aislar y limitar los privilegios y capacidades de un proceso con Sandboxing.

La idea principal del Sandboxing en IE es restringuir el acceso de escritura a objetos *asegurables*, como procesos, archivos o llaves de registro que sean de niveles de integridad mayor [82]. El proceso mismo de Internet Explorer posee un nivel de inte-

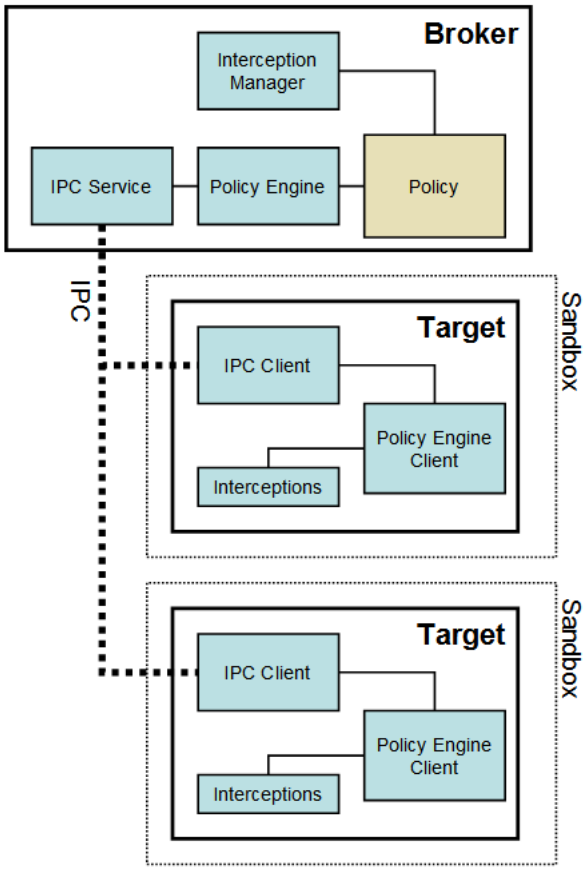


Figura 3.2: Sandbox interno de Google Chrome/Chromium. Fuente: [3]

gridad baja, por lo tanto la única manera de modificar algo es cuando se le pregunta explícitamente al usuario y éste lo permite.

3.2.5. Aislación de Contenido

Cuando hablamos de Aislación de Contenido, nos referimos a cómo los Navegadores realizan la separación del contenido que será renderizado. Esto es dado que si no se es cuidadoso, scripts de otro **Origen** podrían intervenir con una página benigna y tomar control de ésta. Google Chrome aisla el contenido de cada recurso que el usuario pide, por medio de la instanciación de *siteinstance-per-process* [42], aunque durante este último tiempo están tratando de mejorarlo a *site-per-process* [83]. Esto último significaría que cada página y *frame* tendría su propio proceso, separando completamente los espacios de memoria que cada componente de la página usaría al ser renderizada. Esto permite disminuir tanto la superficie de ataque, como incrementar la

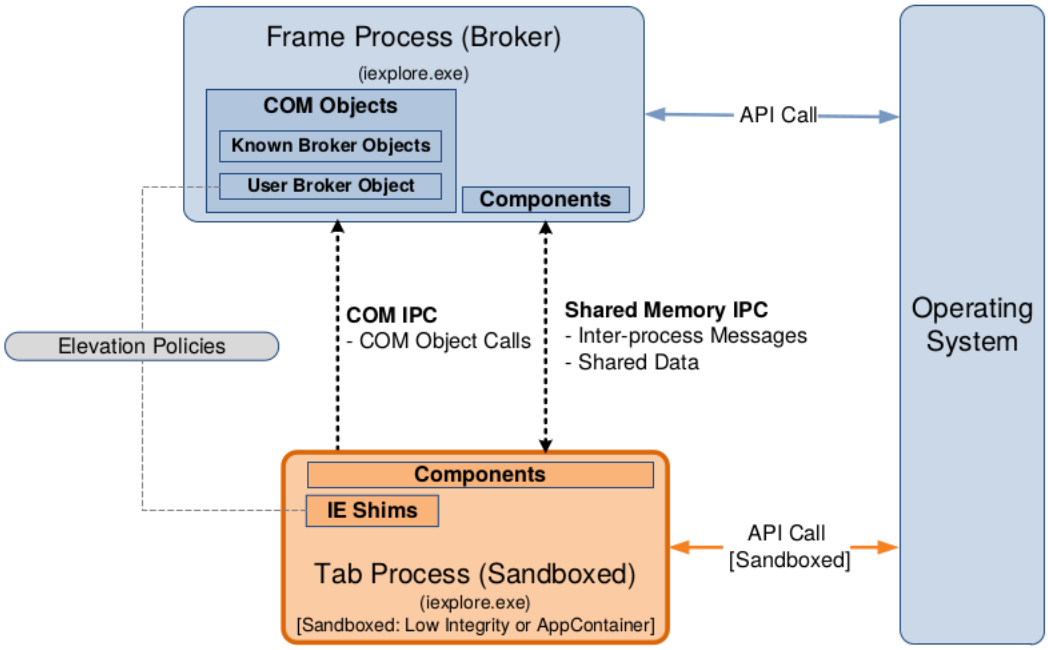


Figura 3.3: Sandbox interno de Internet Explorer. Fuente: [4]

estabilidad del *Browser*, si es que algún componente provoca algún fallo. Firefox [66], por su parte, promete que en futuras versiones del Browser multiproceso, diferentes *Tabs* podrán correr en diferentes procesos de acuerdo a su dominio, pero por ahora todas las tabs de contenidos comparten un solo process de *Content process*.

La aislación de contenido es importante pues permite proveer de seguridad, sensibilidad, estabilidad y confiabilidad. Por ejemplo, si el *Browser* utiliza un enfoque en que separa el contenido al igual que Google Chrome, cada proceso tendrá su espacio de memoria, lo que permite implementar medidas como ALSR [69], para que los espacios de memoria sean *random* cada vez que se cree un nuevo proceso. Al separar en procesos, se están crendo distintas tareas que el sistema operativo se encargará de distribuir y computar de forma paralela; por lo que si alguno de estos procesos se cae, el proceso principal y todos los asociados, no deberían de ser afectados. Con Internet Explorer no se ha podido observar qué tipo de separación de contenido posee, pero debe ser algo parecido a Google Chrome y Firefox.

Uno de los pasos que ha tomado Google Chrome/Chromium y que Firefox va para el mismo rumbo con la API **WebExtension** [84], es la aislación de las extensiones del contenido de la página, tal como lo sugiere [37]. Donde un *Content script* es el que podrá cambiar el contenido del DOM, pero no será capaz de afectar a las Tabs, Bookmarks de otro Tab y menos el sistema de archivos, a excepción que encuentre una vulnerabilidad que se lo permita.

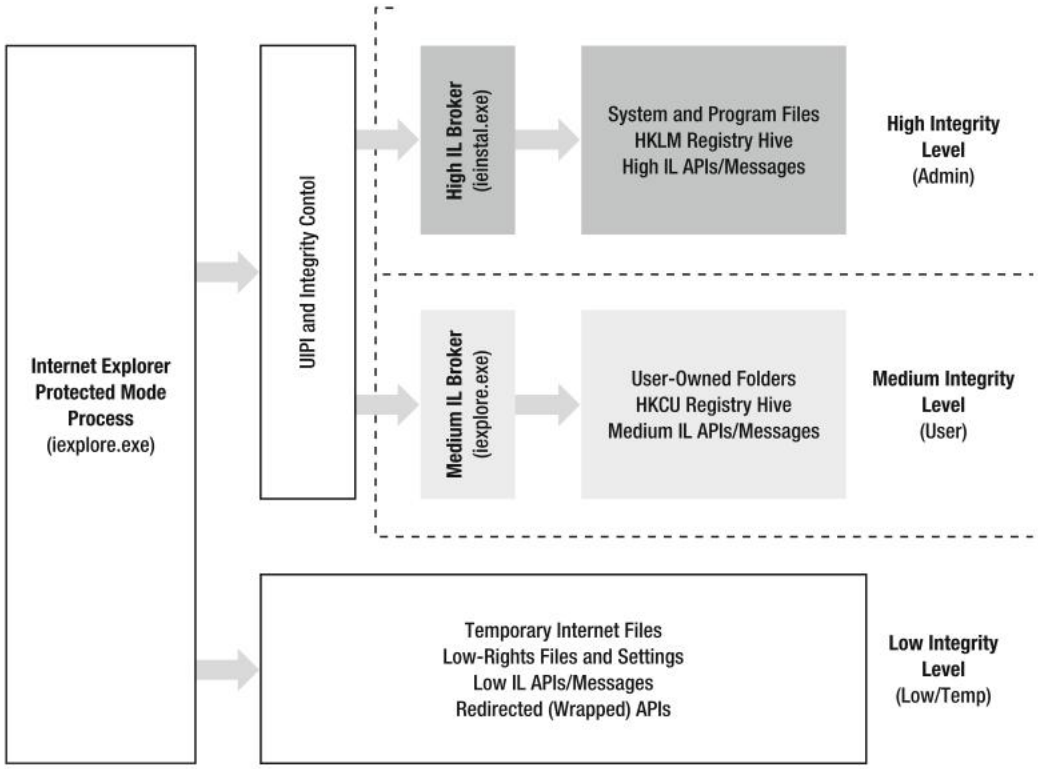


Figura 3.4: Sandbox, Protected Mode. Fuente: [5]

3.2.6. Blacklist y Whitelist de sitios web

El estudio [67] explica las más comunes y efectivas amenazas de seguridad que los usuarios hoy en día se enfrentan, entre ellas están el Software Engineering Malware y el Phishing. Un experimento es llevado a cabo en este estudio, con evidencia empíricamente validada y obtenida por NSS durante 12 días de continuo testing. Se obtuvo como resultado que Safe Browsing de Google Chrome provee una mejor protección contra ataques de tipo Phishing, si es comparado con SmartScreen de Internet Explorer. Estos mecanismos poseen habilidad para alertar a la posible víctima, de que están a punto de pedir un recurso que puede ser malicioso. Una de las conclusiones del estudio de NSS Labs [68] es que afirma que la tecnología de Google Safe Browsing no provee una protección adecuada para SEM[1], pero tecnologías basadas en CAMP[2] ayudan bastante.

Tanto Google como Firefox usan Safe Browsing para investigar si un cierto recurso, detrás de la URL pedida, es posible que se trate de un Phishing. Ambos tienen listas negras que se van actualizando muchas veces en el día, el menos 2 veces por hora.

[1]Social Engineering Malware
[2]Content-Agnostic Malware Protection

Esto permite disminuir la cantidad de ataques de este tipo, pero lamentablemente el dinamismo de este tipo de ataque a veces hace imposible tener una lista de todos los sitios manipulados.

El trabajo que explica CAMP [63], utiliza una Whitelist dentro del Cliente Web para limitar la cantidad de peticiones que recibe el Sistema de Reputación desde todos los Navegadores que necesitan información sobre un binario, que posiblemente pueda ser SEM. Internet Explorer utiliza URL Filtering, con el SmartScreen, que permite obtener una buena protección contra SEM [68] a diferencia de los otros manufacturados de navegadores.

3.2.7. Sistemas de Reputación

Cada vez que un usuario desea un recurso, por medio de su URL, el usuario podría no saber que detrás de ese *path* puede haber una amenaza. Un Sistema de Reputación trabaja en base a los distintos tipos de binarios que un usuario podría llegar a descargar, ya sea un archivo muy descargado que podría estar disfrazado como una imagen y en verdad se trata de un virus, o simplemente un pdf que es descargado por un grupo de alumnos de un curso con poca concurrencia. La idea detrás es tener un sistema centralizado que se encarga de dar una **reputación** al binario, dependiendo de la técnica usada. Tanto Internet Explorer como Google Chrome utilizan este tipo de sistema, pero cada uno lo implemente de distinta manera. Un buen sistema basado en Reputación es aquel que es preciso y rápido, de manera que sea posible de obtener altos porcentajes de detección (efectividad).

El estudio realizado por NSS Labs [68], hizo una experimentación de la capacidad de detección y bloqueo de Malware en los Browser más conocidos, entre ellos Firefox, Internet Explorer y Google Chrome. Se usaron 657 muestras de SEM, que fueron capturados por el laboratorio dentro de 14 días. Los ataques basados en SEM usan diferentes métodos para poder engañar a los usuarios. Uno de los puntos que dejan claro, es que el factor primario es el *Web Browser*, dado que es la primera línea de defensa de los usuarios en contra de la mayoría de los ataques SEM. El test realizado dentro de la experimentación demostró que Internet Explorer bloqueaba casi un $99,9\,\%$ de los SEM sacados de la muestra usadas dentro del test. Google Chrome le sigue a IE con la mejor detección de Malware y otros *Browser* con tecnologías basadas en Cloud (KingSoft Antivirus), generan una mejor detección que Firefox ($4,2\,\%$). La alta detección y bloqueo obtenido tanto por Internet Explorer y Google Chrome/-Chromium, es gracias a las tecnologías SmartScreen URL Filtering (filtro de URL, como black list) y Application Reputation, Chrome por su parte usa Safe Browsing API y Download Protection para resguardarse. Sin embargo, Internet Explorer no depende tanto en su Sistema de Reputación así como lo realiza Google Chrome, donde el $2,9\,\%$ del $99,9\,\%$ de la detección de IE es basada en la técnología CAMP y

Google Chrome detecta 66, 5 % del 70, 7 % con su Download Protection. Ésto último significa para Google Chrome/Chromium, que su sistemas de Reputación es su mejor protección contra Malware.

3.2.8. Actualizaciones Periódicas en Background

Tanto Google Chrome [80] como Firefox realizan actualizaciones automáticas periódicas del navegador, para reducir la ventana de posible vulnerabilidades que pueda tener una versión. Parte importante de esto es que este proceso no fastidia al usuario y permite que se instale apenas el browser se cierra, para que en la próxima sesión sea posible usar la nueva.

Capítulo 4

Browsers y Estado del Arte

4.1. Arquitectura de Referencia del Browser y Patrones

4.1.1. Método

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

- Google, usando *google dorks* para filtrar resultados
- Scopus y Google Scholar, usando string de búsqueda y operadores booleanos para filtrar resultados.

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 existen muchos trabajos relacionados a la construcción de una Arquitectura de Referencia para el *Browser*. Dado que la búsqueda no entregó muchos resultados, se procedió a hacer un *forward snowballing* con el único paper que creemos entrega información similar a lo que se buscaba. Sin embargo, tampoco encontramos tanta información.

4.1.2. Lo encontrado

En Larrondo-Petrie et. al [21] se realiza un análisis del web browser, con el fin de obtener un Modelo de Dominio, un Modelo de Objetos y un *Feature Tree* que describiera la estructura y funcionalidad entregada comúnmente por los *Web Browser*. El dominio, según explica el trabajo, es un set distintivo de objetos que se comportan de acuerdo a reglas y política que caracterizan el Dominio. El Análisis de Dominio es realizado para identificar dominios y cómo éstos interactuan con otros. La metodología usada para obtener los Dominios es el *Object Oriented Analyis*. Además de identificar, se clasifican estos dominios de acuerdo a su rol en el sistema terminado como: Dominio de Aplicación, Dominio de Servicio, Dominio de Arquitectura y Dominio de Implementación. El Modelo de Objetos sirve para entregar más detalles, un resumen general de las Entidades del *Web Browser* y sus relaciones. El *Feature Tree* pretende entregar detalles sobre los aspectos funcionales de la aplicación. El Modelo planteado, ségún el artículo, debería ser útil para los Desarrolladores de Software que construyen **Aplicaciones Web basadas en el uso del *Browser***. Este estudio se encuentra bastante lejos de lo que se quiere hacer en este trabajo, pero sirve para obtener un transfondo de lo que sucede en el *Web Browser*, aún cuando la información esté muy desactualizada.

En el trabajo de Grosskurth et al. [10, 11] se utiliza una herramienta de ingeniería inversa, para obtener una arquitectura de referencia de muy alto nivel en base a dos navegadores open-source: Mozilla y Konqueror. Lo obtenido captura los subsistemas fundamentales comunes a los sistemas del mismo dominio, así como las relaciones entre estos subsistemas. En esta arquitectura se identifican los siguientes subcomponentes: Interfaz Usuaria, Persistencia de Datos, Browser Engine, Rendering Engine, Networking, Interprete de Javascript, XML Parser y Display Backend. Se menciona que estos componentes están estrechamente integrados (high coupling) con el Rendering Engine, lo cual tiene sentido en la arquitectura monoproceso que poseen Mozilla y Konqueror; es una decisión de diseño muy común en los *Browser* de la época. Al identificar estos componentes, se comenta que esto podría servir tanto en el diseño y durante la mantención de un sistema, pues mejora el entendimiento de ésta al ayudar a analizar los trade-off entre diferentes opciones de diseño; o también puede servir como un *template* para obtener nuevos diseños. Una vez obtenida la arquitectura conceptual, se inició una evaluación de ésta al comparar las arquitecturas concretas de cada browser open-source, extraídas desde el código fuente, para ver qué tanto el modelo conceptual era cercano a la realidad; la constante comparación permitió ademas refinar la Arquitectura de Referencia. Los browsers usados para validar fueron: Epiphany, Safari, Lynx, Mosaic y Firefox. Si bien la arquitectura presentada entrega bastante información a alto nivel, no desarrolla más que esa capa de abstracción, además parece ser que depende también de la implementación usada en la herramienta de ingeniería inversa.

En el documento [85] realizado en el año 2000, se describe la experiencia realizada al extender el trabajo del proyecto TAXFORM. Usando PBS, una herramienta de Ingeniería Inversa, se extrajo la arquitectura de software del navegador Mozilla, con el objetivo de entender la estructuración de sus componentes; además de crear vistas arquitecturales de alto nivel del sistema. El modelo arquitectónico obtenido contiene 11 subsistemas de alto nivel, de éstos los que más se destacaron fue el *HTML Layout*, la implementación de herramientas y el código de la interfaz de usuario. En el año en que se lleva a cabo este estudio (2000), se menciona que la arquitectura ha decaído significativamente en muy poco tiempo, o su arquitectura no fue planificada cuidadosamente desde el comienzo; parte de lo anterior, el autor cree que es secuela de la *Guerra de Navegadores*. Si bien el trabajo ayuda a entender un poco la estructura detrás del navegador, este trabajo es muy antiguo y la versión más actual del navegador ha cambiado bastante. Además, lamentablemente el enfoque de este estudio no es intentar entender lo que hace cada subsistema, si no que es la implementación de la herramienta misma para obtener la arquitectura de software del browser seleccionado.

En [86] se propone un *Browser* llamado Anfel SOFT, donde gracias al uso de Inteligencia Artificial, crea agentes que permiten mejorar la experiencia del usuario. El trabajo asegura que el browser será capaz de aprender el comportamiento de navegación del usuario, y guiará al usuario en su navegación para que ésta sea lo más efectiva posible. El paper obtiene los subsistemas que se pueden encontrar en un browser de la misma manera que lo realiza [10]. Si bien la arquitectura que muestra refleja parte de lo visto en los 3 browsers escogidos en este estudio, no da detalles acerca de cada subsistema identificado. Además la Arquitectura de Referencia que entrega es la misma vista en [10, 11], y a pesar que identifica otros posibles componentes, no agrega nada nuevo.

Podemos ver en los trabajos que algunos construyen una Arquitectura de Referencia basada en técnicas de Ingeniería Inversa. En cada uno de ellos el trabajo ha sido a muy alto nivel y la descripción de los subcomponetes del sistema es mínima. Si bien explican las relaciones entre éstos, no dan un mayor entendimiento en cómo se comportan en ciertas situaciones. En este trabajo se espera profundizar un poco más en la abstracción obtenida, incluyendo información de tanto los casos de uso del *Browser* como las actividades que se realizan con otros usuarios. Desafortunadamente para esta memoria, no existe mucha literatura sobre el desarrollo de una Arquitectura de Referencia del *Browser*, y de lo que hay, el trabajo más actual es el realizado por [86] en el año 2009.

4.2. 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 buscador y rápida-

mente llegó a ser dueño de la mayor parte de búsquedas del mercado. Tiene servicios de almacenamiento en la nube, correo electrónico, *e-wallet* y otros más. Google ha sido responsable por la construcción del Navegador Web Google Chrome y Google Chromium. En el 2008, Google liberó gran parte del código de Chrome bajo el proyecto de nombre Google Chromium, el cual es open-source, que permitiría a desarrolladores *third-party* estudiar el código fuente y ayudar en la implementación para las plataformas Linux y OS X. La diferencia entre Chrome y Chromium son: actualizaciones automáticas y Adobe Flash integrado (aunque este último ya se está cambiando el uso por HTML5). En [87] se menciona que este navegador ha sido el último que ha salido y se llevado una gran parte del mercado [1].

La arquitectura de Chrome o Chromium se basa principalmente en dos módulos: el Browser Kernel y el Rendering Engine, como se puede ver en la Figura 4.1.

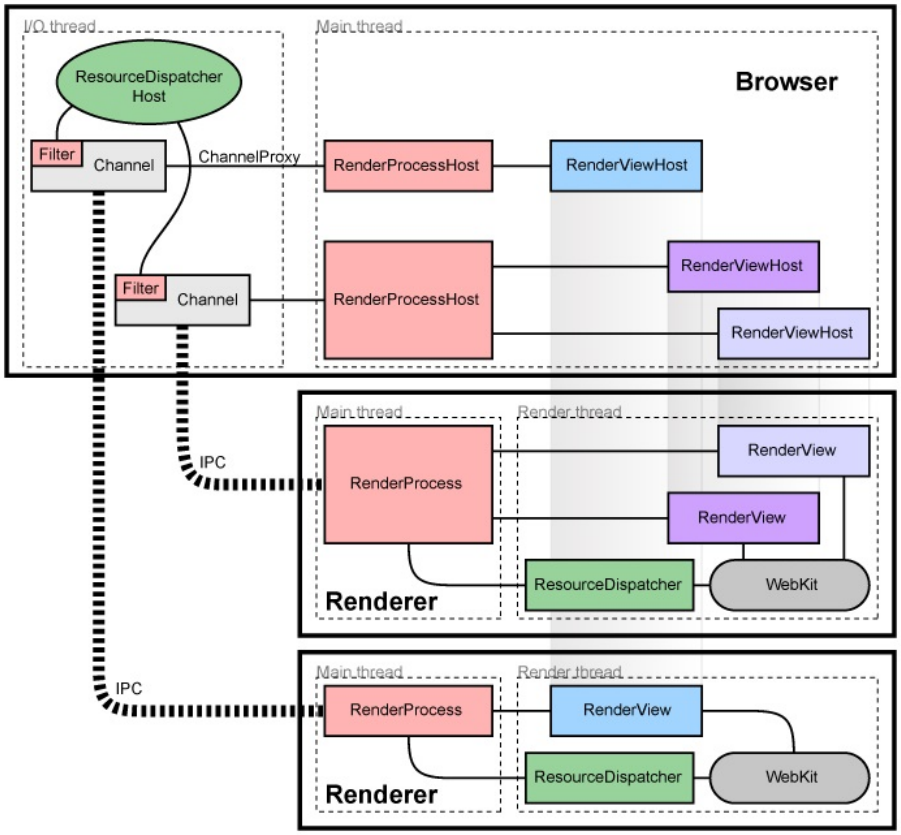


Figura 4.1: Architectura Multiprocesos de Google Chrome. Fuente: [6]

En la documentación de Google Chromium [6], que es base para Google Chrome, se 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; por lo que es una arquitectura Modular o Multiprocesos. El proceso principal es llamado *Browser Process/Kernel/Engine* y se encarga de la Interfaz de

Usuario, manejo de las tabs y los procesos de los *plug-in*. Cada tab es asociado un Rendering Engine, éstos tienen restricciones de acceso (*Sandboxing*) a los demas y al sistema, lo que permite que exista una protección de la memoria y un control de acceso. Google Chrome expone en [80] que existen ciertas lecciones que han ido utilizando para mejorar la calidad de su browser, éstas 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 los datos 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.

En [40] 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 [80], 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 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.

4.2.1. Browser Kernel/Process y Renderering Engine/Process

El Browser Process provee al usuario con lo necesario para el manejo de sesiones (cookies, tokens, etc), historial, almacenamiento de passwords, manejo de interfaz de usuario, barra de localización, sistema local de blacklist, API para realizar llamadas al sistema tanto para almacenar datos como para realizar peticiones del usuario, agente de descarga, entro otros. El Browser Kernel tiene la tarea de ser un interceptor de las llamadas de los procesos Rendering que están en un Sandbox, revisando las políticas y qué acciones están permitidas.

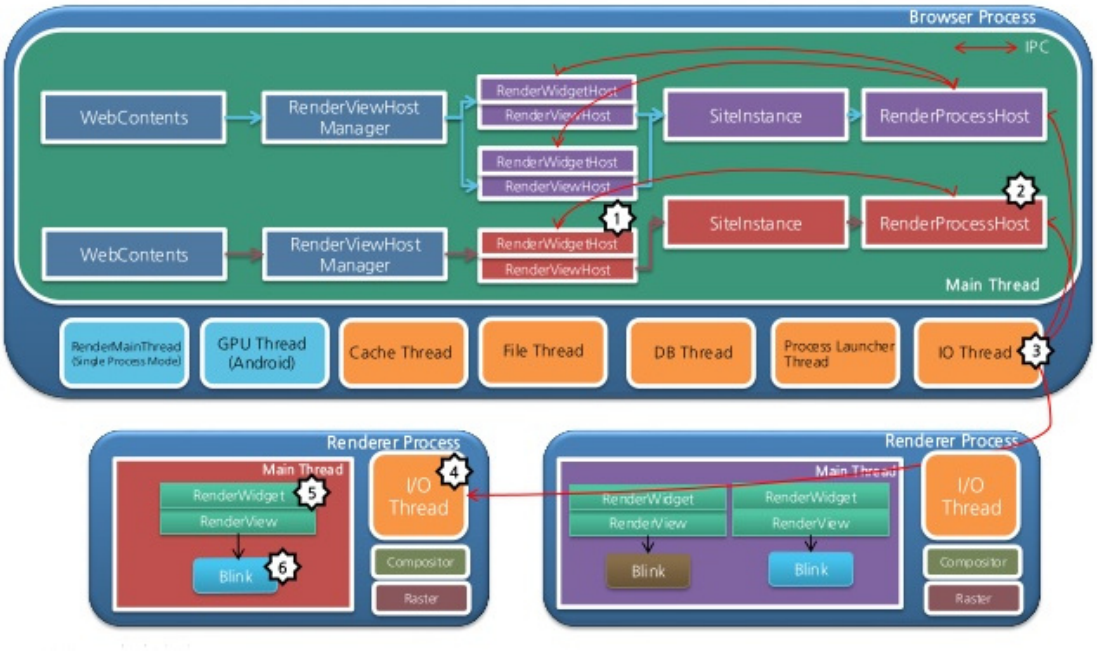


Figura 4.2: Architectura de Chromium en detalle. Fuente: [7]

El *Rendering Engine* usado por Google Chrome/Chromium, llamado Blink, es un *forking* del trabajo original llamado Webkit. Su objetivo principal es soportar la arquitectura de multiprocesos que posee el navegador, y al mismo tiempo poder reducir el nivel de complejidad. Cada nueva ventana o tab abre un nuevo proceso, y el *Browser* instruye a ese proceso a crear un componente que permitirá la visualización del recurso en el navegador. Cada proceso abierto para instanciar un Renderer estará en un Sandbox, donde puede pasar que 2 *tabs* de diferentes dominios estén en el mismo proceso, dado que puede haber una relación en la navegación de estas páginas.

4.3. Internet Explorer

Internet Explorer es el navegador gráfico 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 obtenido 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 utilizar los recursos para crear diferentes funcionalidades; ejemplo de esto son: toolbars, Microsoft Active X controls, etc. En la Figura 4.3 [88] se puede ver los principales componentes

de la arquitectura del browser mencionado. IE utiliza *COM* o *Component Object Model*, una interfaz binaria estándar 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 como CORBA y Java Beans. El uso de *COM* gobierna la forma y la interacción de los componentes que se comunican, y permite que haya un reuso y extensibilidad de éstos.

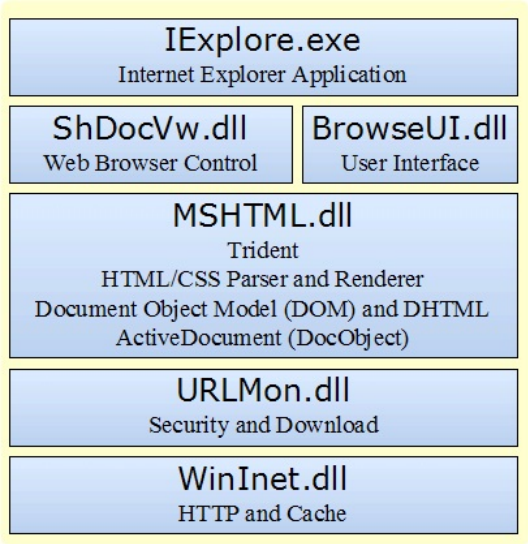


Figura 4.3: Arquitectura de Internet Explorer. Fuente: [8]

4.3.1. Frame Process y Rendering Engine

MSHTML es un Rendering Engine privativo, sin embargo es posible usarlo como librería de Windows **mshtml.dll**. Según [5] es un objeto OLE (Object Linking and Embedding) Active Document que representa el *layout* de Internet Explorer y permite mostrar graficamente las páginas por medio del *display* del host. Dentro de éste se manejan las Extensiones, el *engine* de Javascript y la librería que contiene la API para tareas de *networking*, además de proveer una capa de seguridad y manejar las descargas de archivos.

4.4. Firefox

Firefox fue creado en 1998 a partir del navegador *Netscape*. 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:

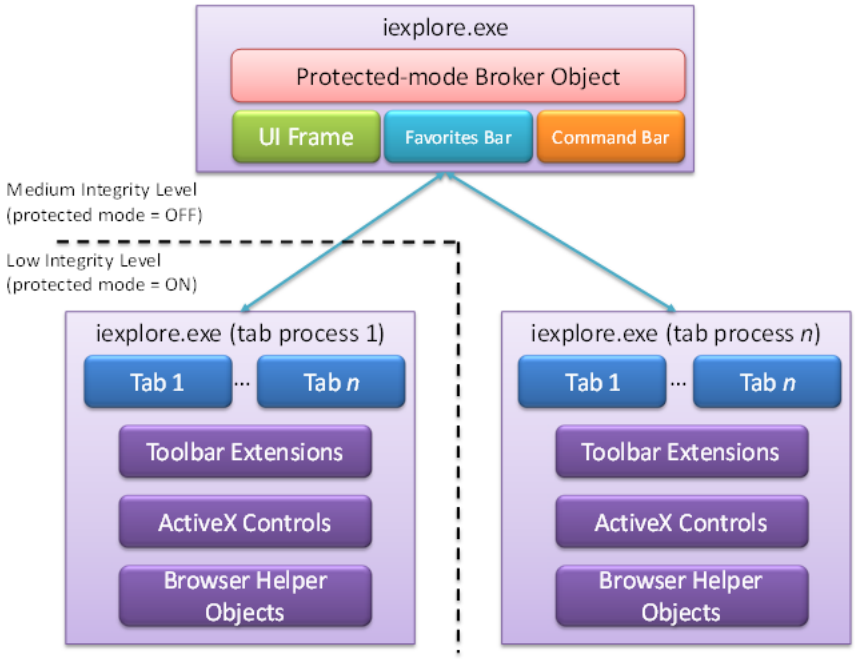


Figura 4.4: Arquitectura de Internet Explorer más detallada. Fuente: [9]

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

4.4.1. Firefox Monoproceso y Gecko

La arquitectura de este browser puede ser vista en la Figura 4.5, donde se pueden observar los siguientes componentes principales:

- La interfaz de usuario, la cual puede ser reutilizada para otras aplicaciones por su bajo acoplamiento con el sistema entero.
- La persistencia de los datos, tanto de bookmarks como de datos de bajo nivel como el *cache*.
- EL *Rendering Engine*, que permite el renderizado de documentos HTML/XML por un tipo de parser basado en web estándars. Este Engine es capaz de renderizar la interfaz de la aplicación multiplataforma.

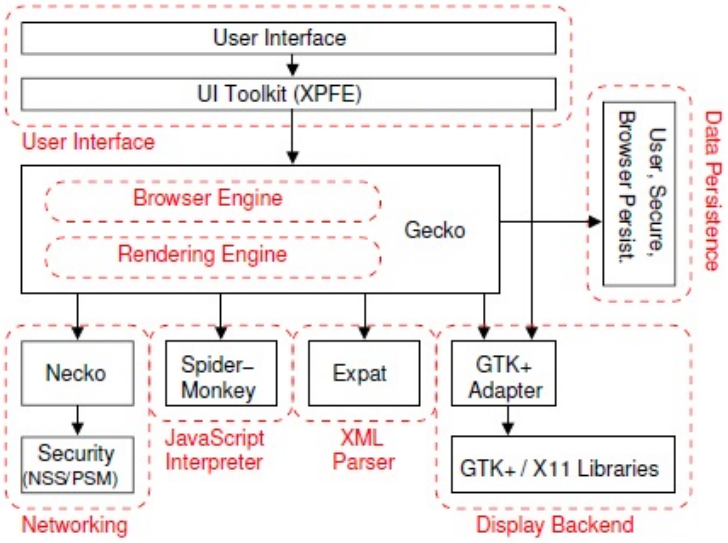


Figura 4.5: Arquitectura Monoproceso obtenida en Fuente [10, 11]

- XPCOM o Cross Platform Component Object Model, similar a Microsoft COM, proporciona un conjunto básico de componentes que pueden ser implementados en diferentes lenguajes.

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. Si bien esta figura muestra lo más importante, algunos detalles relevantes son olvidados, como las extensiones y el componente XPCOM (cross, platform component object model) que es similiar al componente COM de Microsoft,

Gecko es un motor de renderizado *Open Source* que utiliza Firefox, escrito en C++, creado en un comienzo por Netscape, predecesor de Mozilla Foundation/Corporation. La función de este componente en Firefox (y otros browsers que lo integran) es leer el *web content* de tipo HTML, CSS, XUL (para renderizar Interfaz de Usuario) y Javascript, y mostrarlo al usuario en un formato gráfico. Tiene un gran rendiemiento al transformar a formato gráfico una página con lenguaje de marcado, ya que soporta multithreading en el parser de HTML. Gecko fue diseñado para soportar *Open Internet Standards*, y por ende sigue al pie de la letra todas las especificaciones de HTML 4, CCS 1 y 2, DOM, XML y Javascript. Los componentes de Gecko incluyen:

- Parser de Documentos (HTML y XML).
- *Layout Engine* con un modelo de contenido; ésta es la información que el display del host mostrará al usuario.

- Sistema de estilos.
- Motor de Javascript. En el caso de Gecko éste se llama **SpiderMonkey** que está escrito en C/C++.
- Librería de imágenes.
- Librería de *Networking* o Necko.
- Renderizado gráfico específico a la plataforma y widget de acuerdo al sistema operativo.
- Librerías de seguridad, NSS
- Mozilla Plug-in API (NPAPI).
- Lbrería de preferencias de usuario, entre otros más.

En la figura 4.6 se explica como el Rendering Engine es capaz de crear una página a través de un HTML y su correspondientes CSS. Éste parte como 2 pipelines que leen tanto HTML como CSS, y cuando llegan al Frame Constructor, se vuelven una. Ya en esa fase, se espera que solo una hebra se encargue del contenido visual que se mostrará, pues si se tuviera más habría problemas con las prioridades de los elementos a mostrar en el DOM y el Layout final.

Finalmente, en el día de hoy los Plugins ya son considerados tecnología *legacy*, en especial el uso de Adoble Flash, que estaba bien arraigado. Ahora se ha empezado a utilizar HTML5 como una alternativa que se ajusta a los estándares de la W3C.

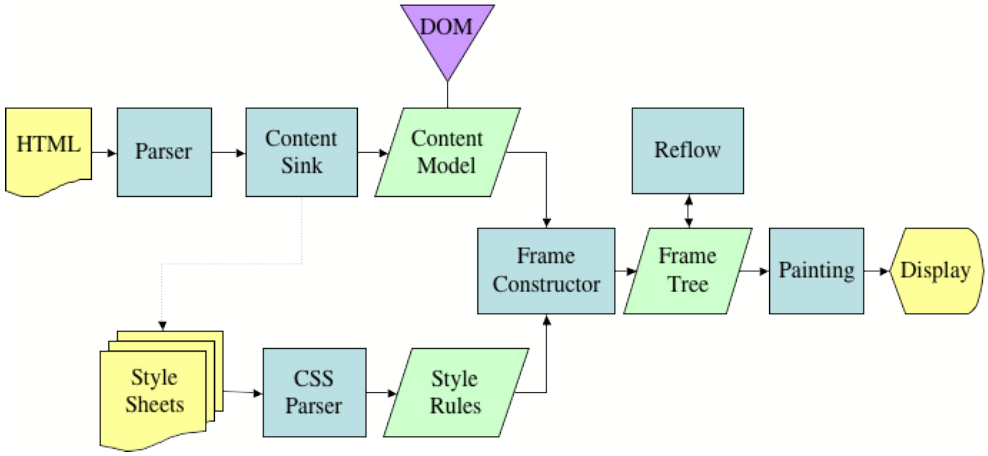


Figura 4.6: Gecko Rendering Engine. Fuente: [12]

Estabilidad y Recuperación a fallos

Este tipo de Firefox utiliza un proceso que se encarga de realizar tanto el Renderizado como la comunicación con las páginas web, y tiene la desventaja de que si sufre un fallo pequeño, todas las Tabs asociadas al proceso pueden ser corrompidas por el fallo. Así como también, el cierre inesperado puede causar una pérdida total de los datos. Aunque Firefox sea uno de los *Browser* con mejor compatibilidad y sigue bastante bien los estandares Web de la W3C, le ha sido bastante difícil moverse a una arquitectura Modular como Google Chrome y Explorer [89]

4.4.2. Firefox Multiproceso (Electrolysis, e10s)

El proyecto Electrolysis o e10s, comenzó el 2009 y que por motivos desconocidos [89] se tuvo que poner en espera. El 2012 la Arquitectura del Sistema Operativo Firefox empezó a usar la idea de multiprocesos que se había estado trabajando, por lo que el 2013 se recontinuó el proyecto, sin embargo aún no hay una versión estable y solo se puede ocupar como testing en la versión Nightly Build [90]. La nueva Arquitectura se basa en usar la aislación que provee el sistema operativo a los procesos, de la misma manera que lo hace Google Chrome, pero por supuesto el diseño no es el mismo.

Chrome Process y Content Process

Respecto a este componente de la nueva arquitectura que Electrolysis, no mucha información de que es lo que hace está en línea. Parte de los componentes son el sistema de comunicación que existe entre los procesos o Message Manager, que se encarga de enviar mensajes de control desde el Chrome Process a los procesos hijos o recibir los mensajes desde sus procesos hijos para realizar las acciones necesarias para mostrar el contenido (Figura 4.7, 4.8).

Así como Google e Internet Explorer, se tiene en Electrolysis al Chrome Process como intermediario para cualquier tipo de acción dentro del *Browser* lo que permite controlar las peticiones del usuario o scripts que se ejecuten en el Content Process (hijo). Finalmente, la propuesta que han desarrollado con Electrolysis, si bien ha capturado buenas prácticas, aún no ha podido realizar una implementación del Sandbox y está lejos aún.

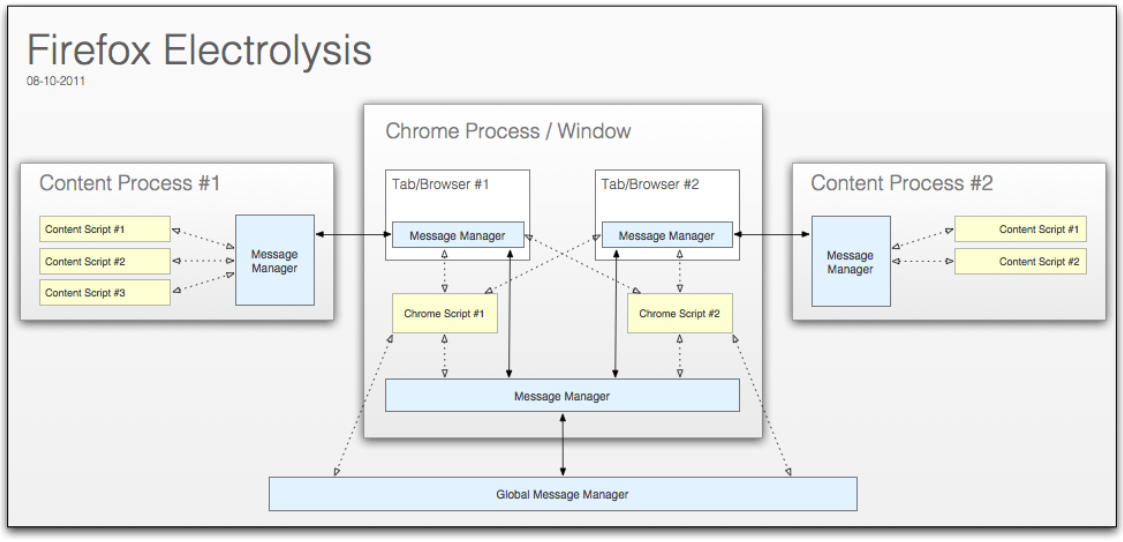


Figura 4.7: Firefox Electrolysis, Comunicación de procesos 1. Fuente: [13]

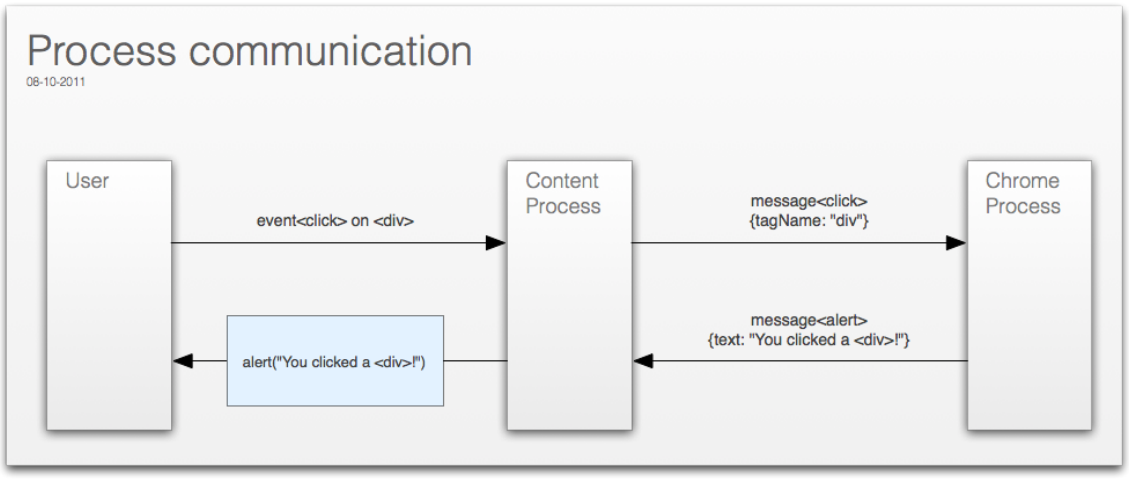


Figura 4.8: Firefox Electrolysis, Comunicación de procesos 2. Fuente: [13]

Capítulo 5

Definición de una Arquitectura de Referencia para el Web Browser

Para poder entender algún *Browser* se buscó en específico trabajos, papers, documentación gris a través de buscadores web como: IEEE Xplore, ACM digital Libray, Scopus y otros más, con tal de encontrar documentos formales sobre sus arquitecturas. Lo que más se encontró fueron blogs de desarrollo y White papers sobre Google Chrome, sin embargo llama la atención la escases en literatura sobre el tema.

En vista a la poca, casi nula, documentación formal sobre el navegador Web, la necesidad de hacer una Arquitectura de Referencia (AR) para entender cómo la arquitectura de este sistema puede relacionarse con el futuro desarrollo de otros sistemas, puede llegar a ser de gran utilidad para explicar los conceptos de seguridad detrás del navegador. 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 esta herramienta ya existen convenciones de qué elementos funcionan mejor. Por consiguiente, no es de extrañar que diferentes browsers estén construidos de formas muy similares, y puedan ser conceptualizados en una AR que manifieste los componentes, mecanismos de comunicación y funciones de esta pieza de Software.

La AR a construir en este trabajo, fue realizada principalmente a través de la abstracción de las propuestas actuales de los Web Browsers más usados: Google Chrome, Internet Explorer y Firefox (propuesta Electrolysis). Primero se identificarán y analizarán los stakeholders, se identificarán los casos de uso relacionados a cada uno de estos stakeholders y se dará una descripción breve. Se han identificado 3 patrones: Browser Infrastructure, Web Content Renderer y Browser Kernel. El primero corresponde a la infraestructura completa del *Browser*, mientras los otros dos son subsistemas internos que en conjunto forman el navegador. Para este trabajo solamente se desarrollará el patrón que representa el sistema completo, es decir, el patrón Browser Infrastructure; en éste están los componentes principales, la comunicación con el host

y la interacción entre éstos. El patrón Browser Insfrastructure será descrito utilizando un template POSA [27] y notación UML para precisar cómo en una AR se relacionan los componentes.

5.1. Casos de Uso del Browser

5.1.1. Stakeholders (actores) y Concerns de estos

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

Browser User (BU)

En principio, representa a un usuario del Host que utiliza el navegador. De este stakeholder depende que se realice el inicio de una petición para buscar una página web, recurso o servicio (Sin éste la utilidad del *Web Browser* es nula). Puede tratarse de una entidad no humana, como un plugin, extensión o instancia de una página web, pueden requerir hacer peticiones por medio de las interfaces del navegador, con la intención de cumplir con el deseo del usuario del host de mostrar el contenido en el *Browser*.

Host (H)

Para este caso nos referiremos solamente a la máquina cliente dónde se aloja el *Web Browser*. Ésta crea los procesos necesarios para iniciar el navegador, además de entregar un ambiente al *Browser* para que éste pueda funcionar adecuadamente. Esta identidad se encarga de aplicar políticas de seguridad sobre el *Browser* cuando se necesiten realizar operaciones o el navegador desee crear nuevos procesos.

Provider (P)

Es toda la infraestructura relacionada con el Servicio Web que pudiera entregar una máquina remota a un navegador. Este puede ser un: Web Server, Web Aplicación, Servicio de actualización del *Browser*, etc. Su interacción con el *Browser* se limita a entregar contenido a éste.

5.1.2. Casos de Uso

Los casos de usos y actores están listados a continuación. Notar que se presentarán aquellos casos de uso relacionados exclusivamente con el *Browser*.

Casos de Uso para Browser User (BU)

El BU representa todas las interacciones posibles del usuario con el *Browser* y sin él no habría razón para la existencia del navegador; un servidor tampoco existiría dado el mismo principio. Al encontrar los casos de uso de este stakeholder veremos las preocupaciones de éste, lo que nos permitirá entender las necesidades de seguridad para proteger al navegador. Los casos de uso principales de éste son: Realizar Request, Cancelar Request y Guardar Recurso.

1. Realizar Petición/Request: El caso de uso más importante de este sistema, se basa en la arquitectura cliente/servidor. Esta acción considera la petición de recursos al Provider (P) y la búsqueda del recurso bajo la URI dada y la consecuente respuesta de éste. La descarga no es siempre necesaria, pues existe lo que se conoce como una petición *preflight* donde solo se envía una petición para saber si es seguro realizar una petición/*request*. Tanto petición como respuesta se han condensado en un solo caso de uso. Este caso de uso considera la visualización del recurso en el Host a través del periférico predeterminado.
2. Cancelar request: Un BU pide al *Browser* detener la búsqueda y todo proceso en tránsito. Si el recurso es obtenido por el *Browser*, la detención de éste puede considerar la detención del componente Renderer para mostrar por pantalla la proyección de la página web obtenida.
3. Guardar recurso: Un BU puede necesitar guardar el recurso recientemente adquirido en el sistema de archivos del host (H). El caso de uso inicia una vez que el BU haya manifestado dónde desea guardar el recurso.

Casos de Uso para el Host (H)

Los casos de uso del Host representan todas las interacciones que relacionan servicios provistos al BU al hacer uso de *system calls*, por medio del *Browser*. El Host debe para esto contar con la respuesta explícita del usuario detrás de BU. Los casos de uso principales de H son: Operación CRUD en el sistema de archivos del host, Monitorear proceso y Pedir recursos.

1. Operación CRUD en el sistema de archivos del host: Acción realizada cuando el *Browser* obtiene una solicitud explícita o no (automática), del usuario para

realizar acciones que involucren permitir el acceso al sistema de archivos para: leer, escribir o modificar/eliminar algún recurso del host.

2. Monitorear proceso: En un browser es normal la comunicación entre los procesos que lo componen y el Sistema Operativo por medio de un canal de comunicación, en especial cuando se necesita ver la cantidad de recursos usados.

3. Pedir recursos: Bajo el pedido explícito del usuario, la entidad BU, se requieren recursos para iniciar las operaciones deseadas. Estos recursos son obtenidos a partir de llamadas al sistema al host.

Casos de Uso para el Proveedor (P)

La meta del Proveedor es escuchar las peticiones provenientes de diversos tipos de BU que necesiten los servicios o recursos del Proveedor. En este escenario tan heterogéneo y distribuido que es la Internet, se enfocará solamente en las peticiones de BU. Además, no para toda *request* del BU habrá un *response* de P, pues puede pasar que BU esté solamente intentando averiguar si P existe.

1. Escuchar requests: P siempre está escuchando en sus puertos habilitados a posibles peticiones de sus clientes. Una petición es recibida en un formato que P pueda interpretar y en consecuencia realizar las acciones correspondientes que BU le pide.

2. Enviar Response: Por cada solicitud del cliente, en este caso el *Web Browser*, habrá una respuesta/response por parte del P con la data Resource asociada. Este caso de uso no se realiza, si dentro del header de uno de los paquetes está indicado que la petición fue hecha en modo *preflight*.

3. Crear recurso: Por cada recurso que P crea, una URL única será usada para ser identificada en la Internet. Un BU hace uso de tal URL para adquirir el recurso, mientras el host del Proveedor lo deje visible.

4. Configurar opciones: definir distintas medidas en el Proveedor como: acceso a recursos, acceso de Browser Users, etc.

Los casos de Uso descritos anteriormente se pueden divisar en la Figura 5.1

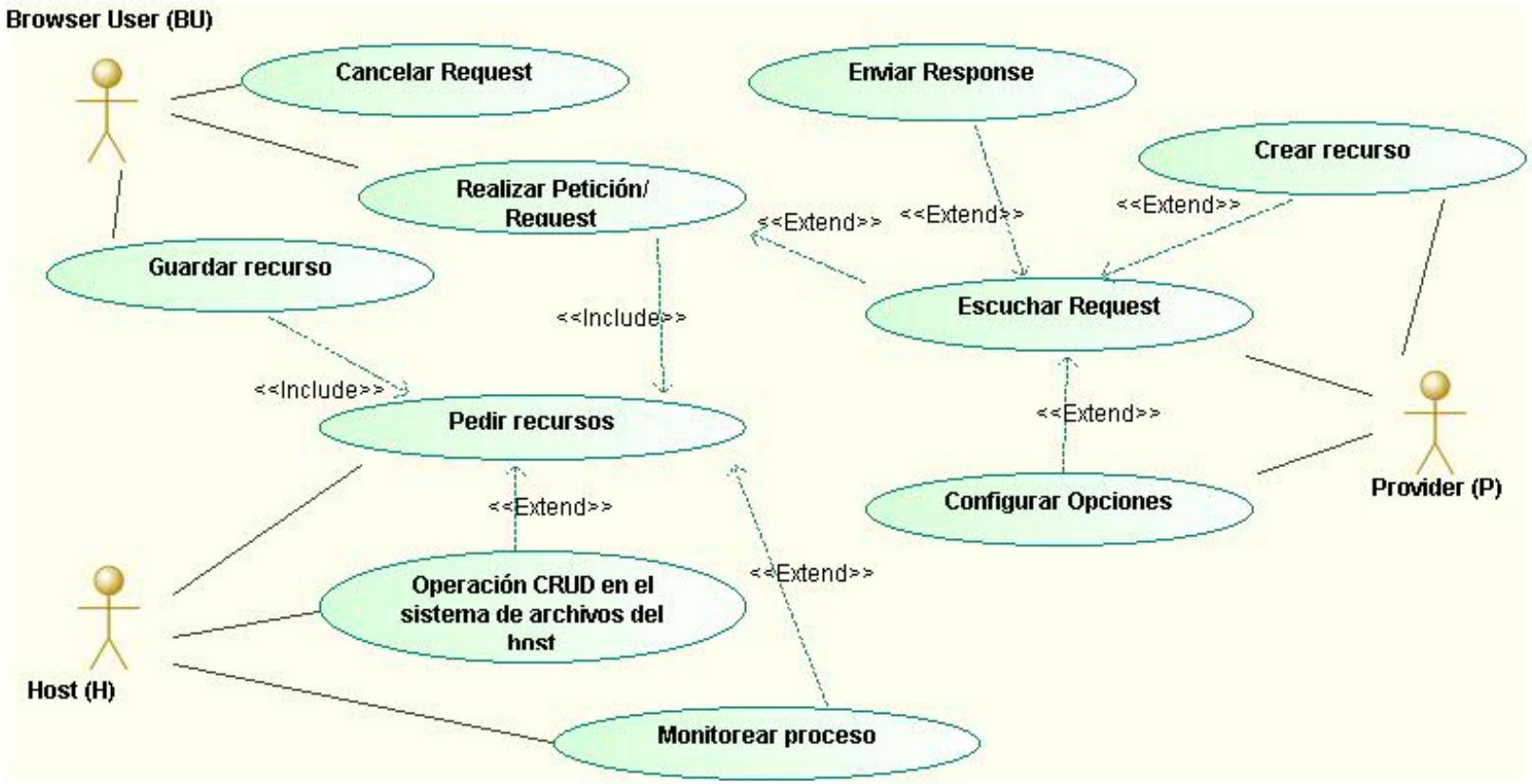


Figura 5.1: Diagrama de Caso de Uso del *Web Browser*.

5.2. Patrón Browser Infrastructure

5.2.1. Intent

El patron Browser Infrastructure pattern permite la realización de la solicitud o *request* de un recurso web en la Internet por parte de un BU, que es un usuario del Host que utiliza un navegador. El patrón permite visualizar la comunicación entre los componentes que conforman el browser así como la comunicación con el proveedor, al que se le realiza el *request*.

5.2.2. Ejemplo

Dentro de un host es posible que exista una falta de recursos que necesite el usuario que lo opera. La solicitud de servicios o recursos externos es una de las razones de existir de la Internet. Esta operación es posible de realizarla de diferentes maneras, todo depende de lo que el Provider desea entregar.

5.2.3. Contexto

Browser User se encuentra al interior del Host y el Provider es accedido a través de Internet. Es común que el contacto se da a través de aplicaciones Web, servers que se comunican a través del protocolo HTTP. Para poder visualizar los recursos que el BU puede necesitar y que el proveedor pueda quizás entregar, un navegador web permite completar tal labor de forma transparente al usuario.

5.2.4. Problema

Algunos BU puede que necesiten obtener recursos desde un Provider, pero el usuario puede que necesite utilizarlos en algún formato en especial o que sean presentados por pantalla para ser mejor visualizados. En este caso, si no se utiliza la herramienta adecuada puede que no sirva de mucho haber conseguido el recurso si no se puede ocupar correctamente. ¿Cómo pueden el Host y Provider estar preparados para esta situación? La solución a este problema debe resolver las siguientes problemáticas:

- Transparencia: el usuario detrás del Host no se debe preocupar de la maquinaria que existe mientras se realiza una petición a un Provider.
- Estabilidad: el *Browser* debe ser capaz de seguir funcionando, aún cuando una página haya tenido problemas para ser mostrada o exista un error.

- Aislación: cada *request* realizada no debe interferir con las demás peticiones que se estén realizando.
- Heterogeneidad: Sin importar el tipo de Provider con el que se comunique, el Browser User debe poder comunicarse con cualquier tipo de Provider y también debe poder mostrar adecuadamente al usuario el recurso obtenido.
- Disponibilidad: El usuario del Host debe ser capaz de poder realizar peticiones en todo momento, sin perder fluidez.

5.2.5. Solución

Un *Web Browser* puede satisfacer las *request* del user del Host a través de un Browser User, ya sea por una o varias instancias de Browser User, lo que permite tener una diversidad de opciones para navegar por sitios de Internet. Un *Browser* debe ser capaz de poder entregar una navegación rápida y estable, sin que cada sitio accedido afecte a los otros recursos adquiridos.

5.2.6. Estructura

El Browser Client (BC) es la entidad que representa el proceso principal de un navegador Web y comprende la cantidad mínima de componentes que constituye un *Web Browser*. Un Host (H) que aloja e interactua con el BC, está compuesto por Hardware (HW) y un Sistema Operativo (SO). Al mismo tiempo, el Provider (P) posee también HW y SO, pero adicionalmente posee un Web Server (WS) que se encarga de recibir las peticiones externas. Browser Client, Sandbox, GPU Instance (GPUI) y Plugin son del tipo Process (Pr), que residen dentro de un Host (H) con un cierto tipo de sistema Operativo (OS). La mayoría de los navegadores existentes usan un componente central para realizar las operaciones que necesitan afectar al Host del *Browser*. La figura 5.2 muestra el diagrama del clases para el patrón Browser Infrastructure. Por cada recurso que un BC solicite, se crearán Sandbox (S) que alojarán una instancia del patrón Web Content Renderer el que permiten realizar la navegación y posterior visualización del recurso. El componente BC actúa como un broker de las solicitudes de las Sandbox, lo que permite tener un control fino de los mensajes enviados, usando IPC/IPDL/COM, entre los procesos que se comunican. El GPU Instance y Plugins son elementos que pueden comunicarse directamente con el Sandbox, de esta manera cualquier necesidad de recursos del Host pasarán también por el Browser Client. Un usuario que desee hacer un *request* de un recurso en Internet por medio de un navegador es lo que llamaremos Browser User (BU). Éste usa el Browser Client para realizar peticiones al Provider, donde éste último posiblemente utilice un Web Server para ese cometido (Figura 5.2)

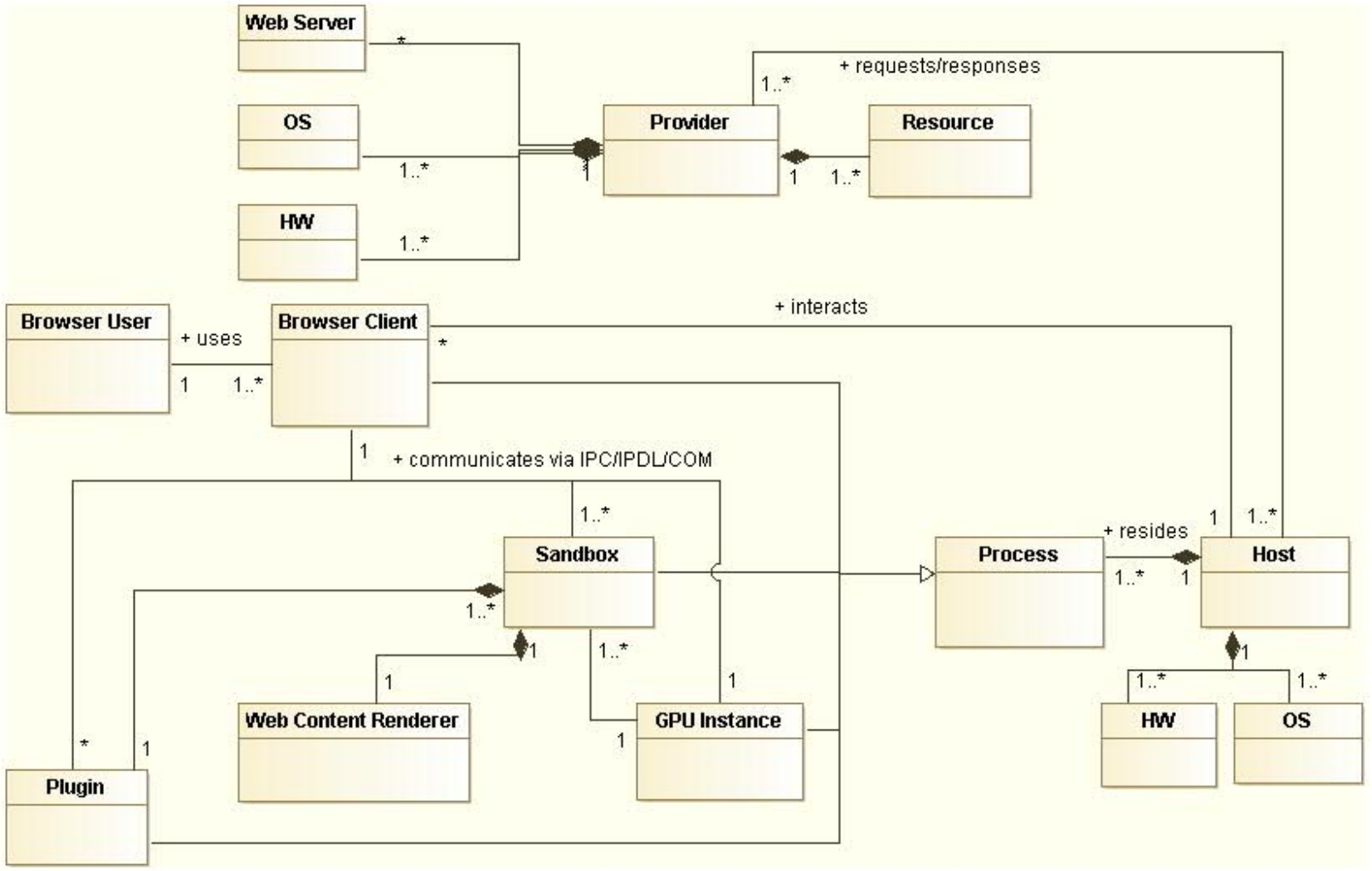


Figura 5.2: Componentes de alto nivel del *Browser*.

5.2.7. Dinámica

Los casos de uso relacionados al patrón son los siguientes:

- Realizar request (actor: Browser User)
- Cancelar request (actor: Browser User)
- Guardar recurso (actor: Browser User)
- Recibir Request (actor: Provider)
- Pedir recursos (actor: Host)

Detallaremos el caso de uso Realizar Request (Figura 5.3):

Sumario

Un Browser User requiere de abrir un recurso URL que puede ser obtenido mediante el uso del protocolo HTTP, según los requerimientos del Provider. El Browser Client será usado por un Browser User para poder realizar la visualización del recurso URL.

Actor

Browser User

Precondiciones

El Host debe tener uno o más Browser Client para el usuario del Host. Además de estar conectado a una red o Internet. El Provider al que se desea contactar también debe estar disponible.

Descripción

Nota: Los mensajes entre el Browser Client y el Sandbox pueden ser tanto síncronos como asíncronos [91, 92]. No especificaremos en gran detalle , pues lo que importa en este trabajo será el origen y destino de los mensajes (no está dentro del ámbito ver la sincronización).

1. Un Browser User requiere acceder a una URL para obtener cierto recurso de un Provider, para esto usa un Browser Client ya instanciado por el Host. En el interior del Sandbox existe una instancia del patrón Web Content Renderer.

2. El Sandbox requiere los recursos del Host para obtener lo que hay detrás de la URL. Una petición es realizada desde el Sandbox al Browser Client a través de un canal de comunicación como: IPC, IPDL o COM (dependiendo el tipo de *Browser* usado), usando la API limitada que posee para comunicarse a un proceso de mayor privilegio.

3. El Browser Client recibe la solicitud, verifica a través de su motor de política-s/normas si la acción del Sandbox puede ser permitida.

4. Si es permitada la acción del Sandbox, se utilizará la Network API que contiene el Browser Client para obtener recursos del Host (a través de llamadas al sistema). El Browser Client se comunica internamente con el Host, y este último debe revisar sus políticas y asegurarse que el Browser Client posee el privilegio de hacer la petición del recurso del Host.

5. Si se permite el acceso el recurso, el Browser Client podrá realizar un *request* a través de la Network API. Si el *request* no es del tipo *pre-flight*, el Provider recibirá el *request* y trabajará sobre la petición.

6. El Provider enviará un *response* al *request* enviado. Dependiendo como esté implementado el Browser Client, éste podría o no tener que esperar a la respuesta (síncrono o asíncrono) del Provider.

7. Una vez obtenida la respuesta ésta es almacenada en cache, salvo que indique de no hacerlo.

8. La respuesta del *request* es enviada por el canal de comunicación al Sandbox del que se originó y posteriormente al Web Content Renderer. Si fue recibida una respuesta por parte de la *request*, el Web Content Renderer está listo para preparar el parsing de la página web o utiliza un plugin o GPU para apoyar la visualización del recurso obtenido por la URL. En caso contrario el Web Content Renderer dentro del Sandbox creará una página de error.

9. El *Renderer* obtiene un bitmap que debe enviar al Browser Client, para que el Host pueda mostrarlo visualmente. Antes de realizarlo, debe revisar que el Sandbox que contiene el Web Content Renderer posea los permisos.

10. Si los permisos son suficientes, el Browser Client envía el bitmap como parámetro en la llamada al sistema realizada al Host. Finalmente, el Host debe revisar que la llamada al sistema que realizó el Browser Client, tenga los permisos necesarios; de poseerlos, el bitmap se mostrará al Browser User por pantalla.

Curso Alternativo

- El Provider no esté disponible.
- El recurso al que apunta la URL no exista.
- Se cancela el *request* realizado.

Condiciones póstumas

El *Browser* recibe el recurso indicado por la URL y se muestra por el perísférico la salida del recurso al usuario del Host.

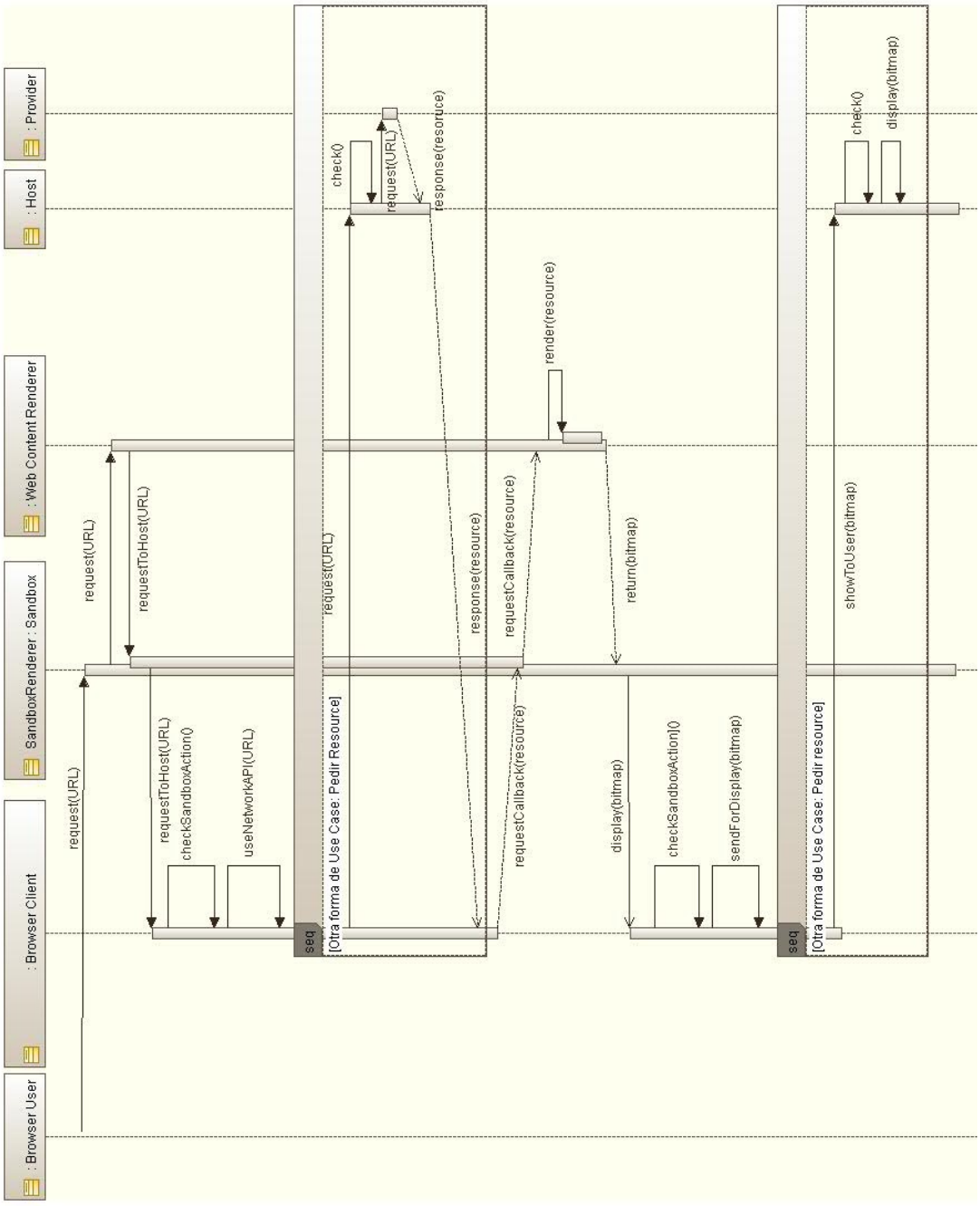


Figura 5.3: Diagrama de Secuencia: Realizar Request.

5.2.8. Implementación

- El Sandbox puede ser implementado de diversas maneras. Google Chrome [3] se basa en no reinventar la rueda y utiliza los mecanismos de protección que el Sistema Operativo (por ejemplo: Windows o Linux) del Host ya tiene incorporados para proteger al usuario, evitando que el proceso no tenga acceso al sistema de archivos y teniendo una API de llamadas al sistema muy restrictivas en el Web Content Renderer. Google Chrome, Firefox e Internet Explorer asumen que los Sandoboxs son procesos que deben regirse bajo el principio de menor privilegio (least privilege). La mínima configuración del Sandbox se compone de 2 procesos: El proceso privilegiado o Broker que es representado por el Browser Client, y el o los procesos que están bajo el Sandbox o targets.
- Para hacer cumplir con el Same Origin Policy, Google Chrome, Firefox e Internet Explorer utilizan diferentes esquemas; por ejemplo: Google Chrome deja el trabajo a su Renderer (Web Content Renderer en este caso) para dejar aisladas las páginas/recursos de diversos sitios.

5.2.9. Consecuencias

El patrón Browser Infrastructure provee los siguientes beneficios:

- Transparencia: La navegación del usuario se realizará casi de manera automática, solo en casos muy puntuales el usuario tendrá que tomar una decisión sobre el recurso que desea pedir.
- Estabilidad: Gracias a que el Browser Client, Sandbox, Plugin y GPU Instance son procesos del Host independientes, el fallo de uno no generará problemas en el otro (crash, corrupción de memoria, etc).
- Aislación: Dependiendo del tipo de aislación es posible separar los distintas peticiones, de manera que no interfieran entre sí, a menos que se desee.
- Heterogeneidad: Dado que cada Browser Client trata de seguir los estándares de la W3C, toda página que también siga éstas guías podrá ser visualizada, así como también otro tipo de recursos.
- Disponibilidad: Cada proceso es independiente y posee sus propias hebras de ejecución, y fueron creadas específicamente para que la interfaz de usuario pueda mantenerse fluida.

Al mismo tiempo, el patrón posee las siguientes debilidades:

- Dado que se inician procesos independientes para navegar a un recurso (dependiendo el esquema que utilice el browser), es posible que una gran cantidad de recursos se vayan a usar para mantener todo abierto.
- Aquellos Provider que no hayan cumplido con las especificaciones de la W3C, mostrarán su resource de forma incorrecta por el *Web Browser*.

5.2.10. Usos Comúnes

- Actualmente, la separación de los componentes del *Browser* en varios procesos, con diferentes niveles de acceso, se conoce como una Arquitectura Modular [93]. Esto permite la separación de preocupaciones del navegador, lo que se traduce en una mayor estabilidad, aislación, seguridad y rapidez.
- Google Chrome se basa en la arquitectura modular, donde cada proceso Renderer se comunica con el Browser Kernel [6]. Internet Explorer, por ser privativo no da mucha información sobre su estructura o detalles de su implementación; [5] sobre la arquitectura Loosely-Coupled [41] y sus componentes, pero sin entrar tanto en detalle. Firefox, por su parte posee las dos implementaciones: monoprocesos y multiproceso. Electrolysis es el nombre de la arquitectura modular que se está implementado, pero que aún no ha sido finalizada completamente.

5.2.11. Ejemplo Resuelto

Con el patrón entregado ahora es posible poder navegar de forma fluida a todos los recursos en la Internet que deseamos. Es posible proveer a través de la aislación de los componentes: rapidez, seguridad y estabilidad. El Browser User solamente se molestará en la navegación, cuando se requiera de su autorización para ingresar a ciertos recursos del Host que sean privilegiados, como el sistema de archivos. Cada usuario del Host, puede utilizar el Browser Client que desee, dado que cada uno de ellos es aislado por procesos independientes, así también los otros Browser Clients entre si.

5.2.12. Patrones Asociados

- El patrón Web Content Renderer, el cuál está en proceso de desarrollo, representa el subsistema dentro del Sandbox que permite realizar el parsing del *resource* o recurso obtenido por medio de una petición.

- El patrón Browser Kernel, también en desarrollo, representa el subsistema que representa el navegador Web. Este componente actua como un Reference Monitor [20] para todas las solicitudes que el Renderer llegara a necesitar.
- El Sandbox se comunica con el Browser Client siguiendo un patrón como el que indica el Policy Authorization [20]. Por cada petición del Renderer, el Sandbox tendrá el acceso si las políticas que el Browser Client permiten tal acción.

Capítulo 6

Patrones de Mal Uso

6.1. Identificando Amenazas y Patrones de Mal Uso

En este capítulo presentaremos el análisis e identificación de las amenazas dentro del patrón presentado en el capítulo anterior, y 2 patrones de mal uso. Estos artefactos servirán de ejemplo de cómo usar la Arquitectura de Referencia obtenida. El objetivo de éste capítulo es conocer y visualizar los mal usos del sistema, en este caso del *Browser*, para poder educar a los desarrolladores de proyectos que basan sus sistemas en el uso de éste. Este capítulo realiza los siguientes ítems: Un análisis de amenazas realizado sobre las actividades relacionadas en el caso de uso **Realizar Request** y **Recibir Request**, luego a partir del resultado del análisis se aplica la metodología explicada en el trabajo de [Bra08] para obtener una lista de amenazas. A través del listado de amenazas es posible detectar o inferir actividades de mal uso que pueden aparecer en uno o más casos de uso, que podrían resultar en una vulneración del sistema. Se presentarán los mal usos de actividades a través de patrones. Se han identificado 2 patrones y serán presentados a continuación.

6.2. Identificando amenazas

La identificación de amenazas es base para poder tomar medidas adecuadas para poder evitarlas. A través de la metodología de [23] es posible detectar y enumerar las amenazas de manera sistemática, al considerar cada actividad en cada caso de uso. Para enumerar las amenazas los diagramas de actividad ayudan en esta tarea. Usando los casos de uso listado en el capítulo anterior, es posible revelar una lista de amenazas extensas. Solo para sintetizar el proceso, se usará un caso de uso y de éste se extraerá un patrón de mal uso. La figura 6.1 describe el diagrama de actividad compuesto, que

involucra los casos de uso **Realizar Request** y **Recibir Request**. Varias amenazas pueden ser reveladas a través de las interacciones visualizadas en el diagrama, por ejemplo: Cuando se realiza la petición al servidor del recurso especificado por la URL, y se entrega otro recurso que no es el pedido.

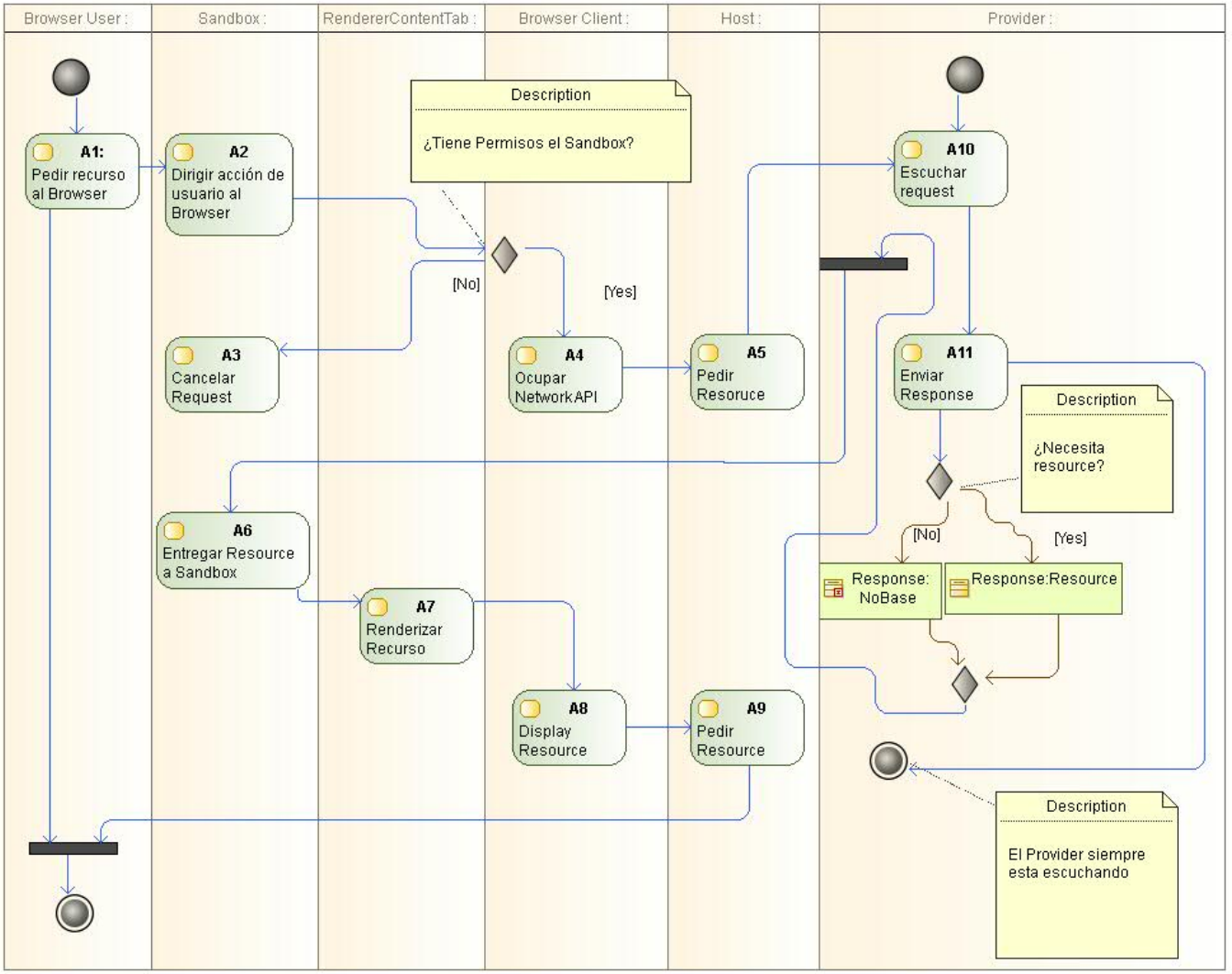


Figura 6.1: Diagrama de Actividad Compuesto para los casos de uso **Realizar Request** y **Recibir Request**.

Cuadro 6.1: Tabla con amenazas.

Actor	Acción	ID	Sec. Attrib (CO/IN/AV/ACC)	Source (AIn/UIn/Out)	Descripción	Attacker	Asset
Browser User	A1	1.1	CO/IN	Out	Modificación de Tráfico/paquetes	Externo	Browser Client, Host
		1.2	IN/CO	UIn/Out/In	Pedir recurso dentro del Host	Externo	Browser Client, Host
		1.3	CO/IN/AV	UIn/Out	Contactar un Provider Malicioso	Externo	Browser Client, Host
		1.4	CO	Out/UIn	Predecir comportamiento o análisis de tráfico	Externo, Process malicioso	Browser Client
		1.5	CO/IN/ACC	Out	Conseguir contraseñas o información personal	Externo	Browser Client, Host
Host	A5 y A9	5.1	CO	Ain/UIn/Out	Divulgar información	Externo	Browser Client, Host
		5.2	IN/CO	AIn/Out	Imitar a un Host honesto	Administrador Malicioso de Host	Browser Client, Provider
		5.3	CO/IN/AV	Ain/Out/UIn	Contactar Provider Malicioso	Externo	Browser Client, Host
		5.4	AV	Out	Impedir navegación	Externo	Browser Client
Provider	A10	10.1	IN/AV/ACC	Out	Aceptar request de cliente comprometido	Administrador Malicioso, Externo	Provider
		10.2	CO	Ain/Out/UIn	Divulgar información	Externo	Provider

Hemos identificado algunas amenazas al analizar el curso de los eventos de los casos de uso: **Realizar Request** y **Recibir Request** (Figura 6.1). La tabla 6.1 hace un sumario del análisis de cada acción en el diagrama de actividad, teniendo en cuenta los atributos de seguridad que podrían ser comprometidos, el origen de la amenaza y los recursos que pueden peligrar. Los atributos de seguridad principales son: Confidencialidad (CO), Integridad (IN), Disponibilidad (AV) y Auditoría (AC). El origen de la amenaza puede ser: actor autorizado (Ain), un actor no autorizado (UIn) o un externo (Out).

6.3. Template de Patrones de Mal Uso

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

Nombre

El nombre del patrón debe corresponder al nombre genérico dado al tipo específico de ataque en los repositorios estándares de ataques, como por el usado por el CERT [94].

Intent o descripción básica

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

Contexto

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

Problema

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

Solución

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

Componentes del sistema afectados (Dónde buscar evidencia)

Esta sección adicional al *template* original de un Patrón de Seguridad [20] es nueva, dado que tiene relación con el mal uso realizado. La solución debe representar todos los componentes que están involucrados en el ataque, pero no debe ser una extensa lista, solo los más importantes para prevenirlo o lo escencial para una examinación forense. Esto puede ser representado por un diagrama de clases, que puede ser un subset o un superset del diagrama del contexto.

Usos comúnes

Es prefeerido indicar los incidentes específicos en donde el ataque ha ocurrido. Pero en el caso de vulnerabilidades nuevas, donde el ataque aún puede que aún no se haya realizado, un contexto específico dónde el ataque pueda llevarse acabo es suficiente.

Consecuencias

Discute los beneficios y desventajas del patrón de mal uso o uso indebido desde el punto de vista del atacante. ¿Es acaso el esfuerzo y costo gastado comparable a los resultados obtenidos? Esta es una evaluación que debe ser realizada por el atacante al decidir realizar el ataque; Diseñadores deben evaluar el riesgo de sus activos usando

alpún enfoque de análisis de riesgos. La enumeración incluye buenos y malos aspectos, que deben ser emparejados a las fuerzas o *forces*

Contramedidas y Evidencia Forense

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

Patrones Similares

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

6.4. Patrón de Mal Uso: Modificación de tráfico en el *Web Browser*

En esta sección presentaremos un patrones de mal uso que describe la amenaza encontrada en el patrón Browser Infrastructure que hemos obtenido en este trabajo. Una de las amenazas es que un atacante haya sido capaz de comprometer la respuesta obtenida desde el Provider contactado. El atacante podría tratar de reemplazar algunos parámetros de la respuesta del provider, entregando al Browser User un contenido distinto al original.

6.4.1. Intent

Un atacante podría cambiar contenido o dar uno diferente al esperado cuando el *Web Browser* (Browser User) recibe la response del Servidor o Provider. Realizando esta acción, el navegador podría interpretar la información de una manera distinta a que si hubiera recibido el tráfico original.

6.4.2. Contexto

Un navegador web obtiene resources desde un Provider para poder satisfacer un Browser User que lo necesita. Un Provider posee resources en forma de páginas web, servicios web u otro tipo. El Provider generalmente consiste en una Web App o Web server, que puede permitir entrada y salida de datos a otras aplicaciones, normalmente éstos están construidos usando HTML, Javascript y CCS. Sea que éste desea entregar servicios, como también otros deseen conectarse a él, todo tipo de comunicación estará encima del protocolo HTTP. Un Provider, dependiendo del tipo, puede llegar a recibir muchas peticiones de diversos Host para obtener recursos de éste. Dependiendo del tipo de peticiones, éstas pueden o no ser permitidas. Aquellas que son permitidas, el Provider generará una response a la *request* del Host, la cuál terminará llegando de vuelta (o no) al Browser Client que generó el *request*.

6.4.3. Problema

¿Cómo podría un atacante engañar al Browser User y Host, por medio de la modificación de tráfico entre las entidades participantes en la comunicación? Es posible que un atacante esté escondido entre medio de la comunicación que hay entre el Host y el Provider, lo que resulta en que el contenido modificado podría afectar dentro del Web Content Renderer o el Browser Client. Esto en consecuencia podría traer diferentes resultados, desde el robo de información privada del Browser User que utiliza el Browser Client para personalizar la navegación como también los token de autenticación hacia los Providers que el Browser User ha utilizado previamente. Dependiendo del tipo de atacante, es posible que éste pueda incluso afectar al Host del *Browser*, dejando que el atacante pueda realizar actos maliciosos con los recursos del Host. El ataque puede sacar ventaja de las siguientes vulnerabilidades:

- El **origen** u **origin** que define el Same Origin Policy que hace cumplir el *Browser* difiere en cada tipo de *Web Browser* [95, 42, 96, 5, 75].
- El **origin** no es suficiente como método de aislación entre resources (páginas web, scripts, css y otros) [97, 36, 4, 38].
- Cualquiera puede crear una extensión o plugin, etc. para algún tipo de *Browser* y hacerlo pasar como algo inofensivo, pero el usuario no se daría cuenta que un ataque Man-in-the-Browser podría estar ocurriendo [71, 70, 38, 37]. Éste programa, posiblemente malicioso, podría afectar el tráfico sin que queden rastros, pues tiene permisos del usuario al ser éste mismo quien autorizó su instalación.
- Es posible afectar al Browser Client, y en consecuencia al Host, sin tener que buscar vulnerabilidades en el *Web Browser*. Solamente utilizando métodos de

ingeniería social, sobre el usuario del Host basta para lograr un ataque, pues es **el eslabón más débil**.

- La arquitectura para extender la funcionalidad del navegador, a través de extensiones, plugins y otros, depende demasiado del manufacturador. Y probablemente posee una gran superficie de ataque.

El ataque puede ser facilitado por:

- Existen muchas herramientas para realizar ataques de ingeniería social, lo que permite hacer que el Browser User acepte realizar la instalación de extensiones o plugin maliciosos más fácilmente.
- Cualquier script basado en texto puede ser usado para explotar el intérprete del navegador web. Muchas veces también es posible utilizar los mismos elementos del lenguaje de scripting para poder pasar ciertas barreras de seguridad entregadas por el SOP, pues el lenguaje está basado en prototipos.
- Los manufacturadores de Navegadores no poseen aún mecanismos de defensa que permitan identificar efectivamente cuando un recurso puede ser malicioso.
- Los métodos de cifrado no pueden hacer nada contra un ataque que puede modificar el tráfico antes de enviar o después de recibir el mensaje.
- No existen mecanismos de defensa estandarizados, pues cada manufacturador realiza lo necesario para la marca a la que se acopla. Si el Host posee muchos browser, la superficie de ataque es bastante mayor.

6.4.4. Solución - Estructura

La solución estructural usada es la misma creada por el patrón Browser Infrastructure que se revisó en el capítulo 4. La clase Attacker es cualquier entidad que podría llevar a cabo una acción arriesgada contra la integridad del *Browser*, usuario, Host y Provider (Figura 6.2). El atacante es capaz tanto de interceptar las response que el Provider envía al Browser Client usando al Host como receptor, o haciendo que el *Browser* realice modificaciones en el input que va hacia el Provider, utilizando scripts u otro resources.

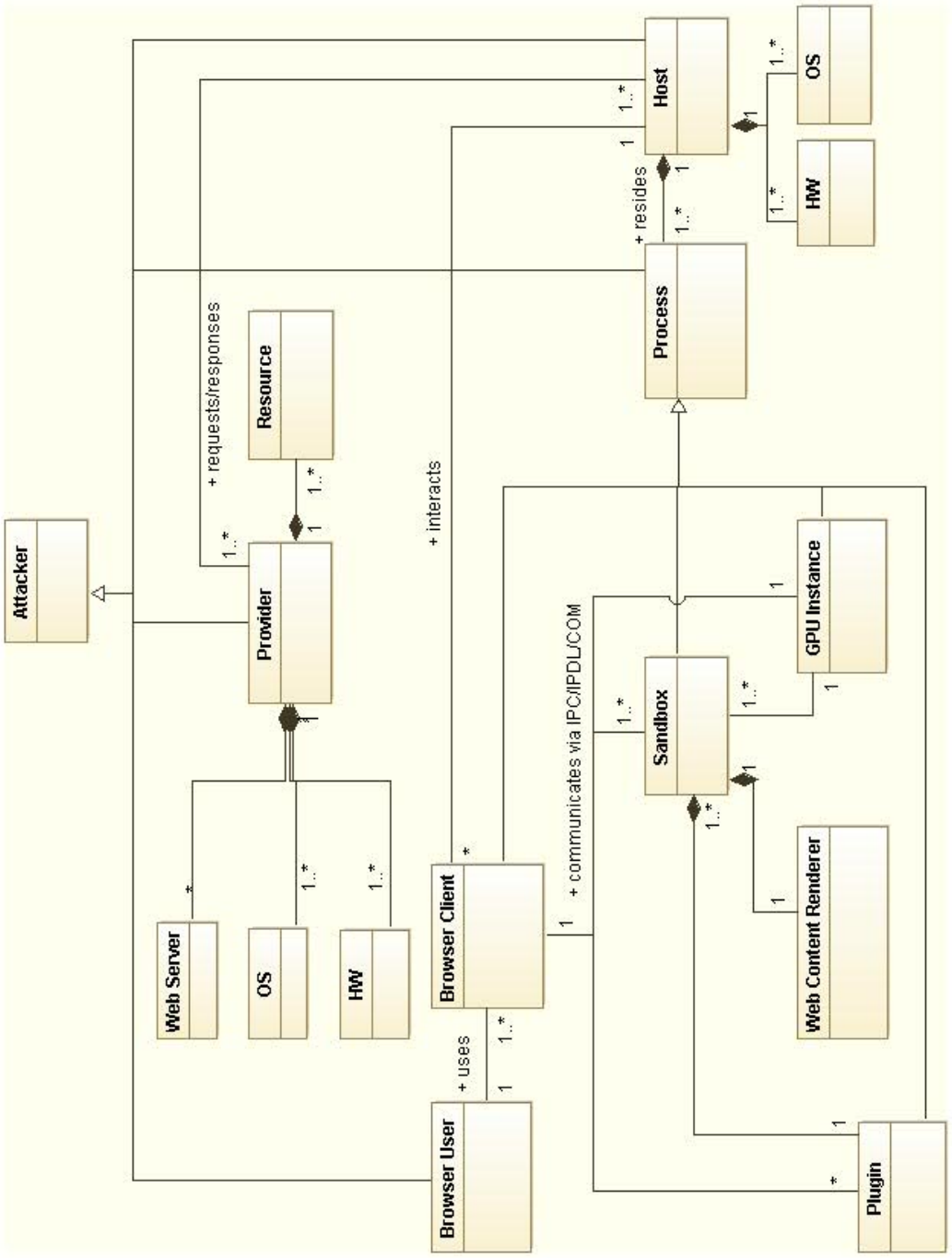


Figura 6.2: Diagrama de Clases para el patrón de Misuse.

6.4.5. Dinámica

En la figura 6.3 se muestra la serie de pasos necesarios, para realizar uno de los tantos mal usos que se pueden realizar durante el caso de uso Realizar *request*. El atacante queda entre el Browser Client y el Host, interceptando la realización del *request* original y modificando el tráfico a su gusto; usualmente un ataque basado en este mal uso se le llama Man-in-the-Browser (MITB) [38, 37, 70, 71]. Esto podría perfectamente suceder cuando el Browser User ha permitido la instalación de plugins, extensiones o programas externos en el Host y Browser Client.

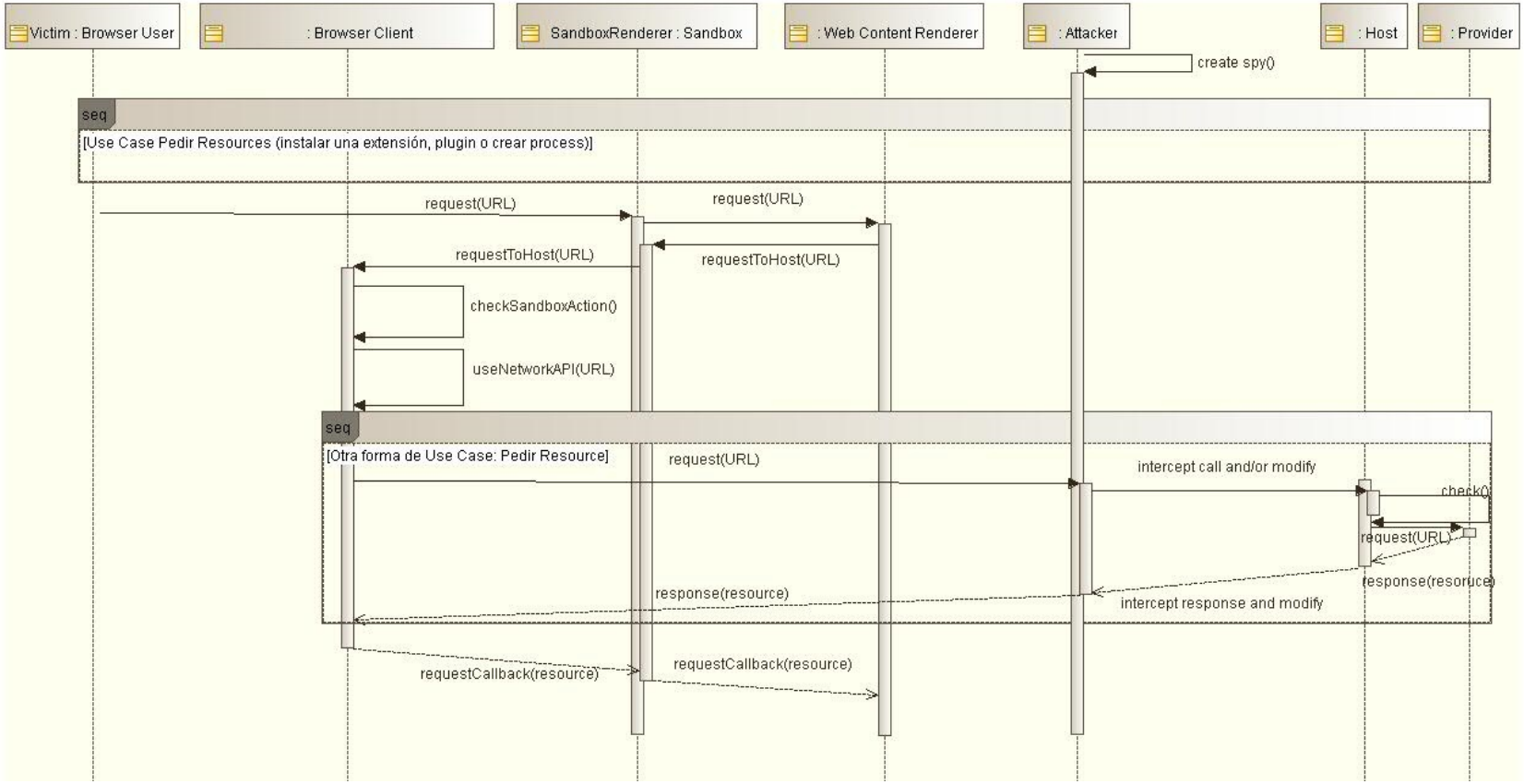


Figura 6.3: Diagrama de Secuencia para el Mal uso: Modificación de tráfico en el *Web Browser*.

Sumario

El atacante intercepta el tráfico entre Host y Browser Client.

Actor

Atacante

Precondiciones

Para que el ataque pueda pasar desapercibido, es necesario que el Browser User o el usuario detrás del browser haya caído primero en un ataque de ingeniería social o el atacante haya podido instalar directamente un proceso o componente malicioso en el Host directamente.

Descripción

1. Un atacante utiliza alguna técnica de ingeniería social o vulnerabilidad en el sistema, para crear una entidad que se encargará de estar entre medio del Browser Client y el Provider. Para esto realiza el caso de uso **Pedir Resources** para que el proceso, plugin o extensión se aloje en el Host.
2. Un Browser User desea hacer un *request* a cierta URL, por lo que los primeros pasos de **Realizar Request** son similares.
3. En el momento en que el Browser Client va a realizar la llamada al sistema para enviar el mensaje del Host al Provider, el Browser Client llamará al plugin, extensión o process malicioso, pues el Browser Client ha sido intervenido para que realice esa acción de este modo.
4. El atacante entonces recibirá todo el tráfico del Browser Client, el cuál podría modificar a su antojo.
5. Finalmente la víctima está totalmente comprometida.

Post condiciones

La víctima quedará completamente comprometida y probablemente no sea posible detectar la alteración del mensaje, pues también es posible que se modifiquen los log del Host.

Usos Conocidos

El *Web Browser* es un software que posee diversas implementaciones, por lo tanto la cantidad de vectores de ataque es significativa. Algunos de éstos son:

- Una extensión basada en la arquitectura de Chrome o la API WebExtension Firefox, podría interceptar los datos antes que llegue al Browser Client [75] o dado a una vulnerabilidad del mismo elemento un atacante se está aprovechando de su funcionalidad para realizar ataques [38, 37]. Dado que el Plugin, la extensión o el process son elementos que el Host confía, es posible que el ataque sea indetectable y los métodos de cifrado no sirven como medida de mitigación.
- Éste tipo de ataque puede ser usado como base para otros ataques más avanzados. Un ejemplo es cuando el *Browser* posee vulnerabilidades *cross-origin javascript capability leaks*, donde los diferentes modelos de seguridad usados por Javascript y el DOM pueden interferir entre sí, causando que una petición *cross-origin* se pueda realizar aún cuando SOP debería ser capaz de detener tal ataque [36].

6.4.6. Consecuencias

El mal uso tiene las siguientes consecuencias para el Attacker:

- Objetivos: pueden ser diversos, destacándose el vandalismo, personificar a otra persona u obtener una ganancia monetaria. Mientras el atacante se pueda interponer entre el Host y el tráfico que se envía al Provider, la confidencialidad e integridad de los datos está completamente perdida. La privacidad del usuario ya no se puede asegurar tampoco.
- Silencioso: Dado que el atacante ha logrado interponerse entre las llamadas de sistema que se realizan al Host para enviar los datos al Provider, el Host no reconocerá ni logueará ninguna anomalía. Las llamadas hechas al Host son totalmente legales y nada fuera de lo normal para éste.
- El atacante podría realizar acciones que afecten la integridad del Host.

Posibles fuentes de fallo:

- Si el Browser User es capaz de evitar o ignorar el ataque de Ingeniería Social realizado al comienzo, no existiría este mal uso. Esto debe considerar que el usuario también no se encuentre con páginas o contenido malicioso, que podrían afectar otro componente de *Browser*, pero que causarían en el mismo efecto del Mal Uso señalado.

6.4.7. Contramedidas

Para prevenir este tipo de mal uso se recomienda tomar las siguientes medidas preventivas:

- Servicios de Reputación como Smart Screen de Internet Explorer y Download Application de Google Chrome, ayudan a identificar páginas y contenido/resources que podrían contener malware que se instale como plugins, entensiones o process en el Host del Browser User.
- Entregando educación sobre los peligros de navegar por Internet y aclarar al usuario que él es la última línea de defensa contra éste tipo de ataques.
- White y Black list instaladas en los browser son una medida preventiva para evitar la navegación en páginas o recursos maliciosos ya conocidos.
- Navegadores como Google Chrome e Internet Explorer ofrecen el Sandboxing. Éste mecanismo de defensa limita las acciones del atacante, que pudieran afectar la integridad del sistema.

6.4.8. Evidencia Forense

¿Dónde es posible encontrar evidencia? Dependiendo de lo deseado por el atacante, las acciones que cometerá pueden diferir. Sin embargo el log interno del browser debería de poder servir para auditar el sistema, esto gracias a que mientras no se haya encontrado una vulnerabilidad en el Sandbox del *Browser*, el atacante no puede borrar completamente sus huellas.

6.4.9. Patrones relacionados

- En el patrón Browser Infrastructure, el Browser Client actúa como el Reference Monitor explicado en [98].

Capítulo 7

Conclusiones

Un navegador Web pareciera ser un Software de mediana complejidad para tanto usuarios como desarrolladores sin experiencia en seguridad, pero lamentablemente esta pieza de Software permite realizar una variadad de vectores de ataque, tanto en un usuario usándolo como en el sistema con el que interactúa. Por lo tanto es importante comprender su estructura y como éste interactúa con Stakeholders internos como externos.

7.1. Contribuciones

Durante el curso de este trabajo se realizaron las siguientes contribuciones:

- Una base conceptual para aquellos desarrolladores que no comprendan sobre términos de seguridad relacionados al navegador Web.
- Construcción del primer Patrón Arquitectural sobre la infraestructura del *Web Browser*, para poder entender de manera holística los componentes, interacciones y relaciones.
- Una parte de la Arquitectura de Referencia ha sido construída, a través de la abstracción del Patrón Browser Infrastructure. Además se ha conseguido caracterizar los Stakeholders y Casos de Uso más importantes. De lo que tenemos por conocido, esta es la segunda Arquitectura de Referencia construída del *Browser*. Sin embargo, nuestra propuesta es la más actual y se ajusta más al tipo de arquitectura que ahora los Browsers utilizan, usando una arquitectura Modular.
- Construcción de un Patrón de Mal Uso, como primera instancia para entender los conceptos de seguridad, como amenazas y ataques posibles de realizar dentro del navegador Web.

7.2. Resumen

El estudio de la seguridad del *Web Browser* y la construcción de una Arquitectura de Referencia mediante patrones, conduce a las siguientes conclusiones:

- Una śintesis y abstracción de la información correspondiente a los Web Browsers, para generar un lenguaje de comunicación de los conceptos. El uso de patrones permite ayudar a un conjunto de personas a unificar conceptos, así como también puede ser guía en futuros desarrollos.
- El trabajo propuesto permite comprender mejor, por medio de la Arquitectura de Referencia parcialmente construída, tanto componentes como amenazas existentes. Además como no está sujeto a implementaciones específicas, es posible generalizar ciertos resultados en otros Browsers.
- Comunicar en primera instancia, los conceptos de seguridad básicos para un mejor entendimiento de los subsistemas que interactuan en la Internet. Los patrones de Mal Uso permiten condensar la información y transmitirla, de forma ordenada y descriptiva.
- Es de esperarse que en el futuro la mayoría de los *Web Browser* tomen forma de una Arquitectura Modular. Por lo tanto, es importante que los desarrolladores conozcan los procesos internos del *Web Browser* al momento de desarrollar un sistema que se comunicará con éste. Tanto la Arquitectura de Referencia como el patrón de Mal Uso presentados, apuntan a entregar el conocimiento básico de los componentes e interacciones entre el *Web Browser* y un proveedor de recursos externo; así como también de las amenazas que existen.

7.3. Trabajo Futuro

El trabajo futuro que se realizará para obtener el grado de Magister, irá relacionado a la creación de una Arquitectura de Referencia de seguridad del *Web Browser*, utilizando la misma metodología presentada acá. Otros patrones relacionados al Patrón Browser Infrastructure serán obtenidos para así completar la AR ya iniciada, como por ejemplo el patrón Web Content Renderer y Browser Kernel. Un ejemplo del tipo de trabajo que se pretende realizar puede ser vista en [99], donde este estudio realiza actividades para poder construir software seguro y evaluar los niveles de seguridad de un sistema ya construído.

Se planea construir más Patrones de Mal Uso, para el Patrón Browser Infrastructure para continuar con el estudio de amenazas posibles dentro del *Browser*, como

una manera de educar a los Desarrolladores y Stakeholders de los peligros existentes. Al mismo tiempo estos patrones permitirán la construcción de esta AR de seguridad. En esta misma dirección, además de encontrar las amenazas posibles de existir en el sistema, se necesita encontrar las contramedidas o defensas de seguridad que permitan evitar o preveer esas amenazas a través de Patrones de Seguridad sobre la Arquitectura de Referencia construída. Lo anterior es posible de realizar bajo el mismo ejercicio ya realizado en este trabajo, buscando amenazas sobre cada acción realizada en cada Caso de Uso del navegador.

En cuanto a los *Web Browser*, los ataques basados en Ingeniería Social parece que no disminuirán en un buen tiempo, pues no existen tecnologías actuales que puedan detectar en un 100 % y sin falsos positivos los posibles peligros que pueden traer. Tecnologías como CAMP (Content-Agnostic Malware Protection) parecen ser parte de la solución, pero aún están lejos de ser perfectos.

Capítulo 8

Glosario

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* [100], 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/Flaw*:
  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) [94].

- 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 realizada por un sujeto 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 término 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" [62], 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 modifica-

da/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 datos en un cierto instante de tiempo, conlleva a la pérdida 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 víctima visite un sitio o aprete un link de manera que se de la **autorización explícita** 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 los datos o sistema de un usuario. La mayoría de las veces puede ser instalado con el permiso del usuario realizando una técnica de Phishing, y otras veces engañando al usuario a que acepte sin darse cuenta.

- *Man-in-the-Middle*
  Ataque que causa una pérdida en la Confidencialidad o la Integridad de la información que es revelada. La causa de este ataque puede ser por ataques de tipo Phishing, como a través de vulnerabilidades del sistema que debieron ser explotadas antes para causar el ataque MiTM.

- *Fingerprinting*
  Es la acción de recolectar información de un dispositivo o sistema remoto para poder identificar a quién esté detrás de él.

- *Timing attack*
  Es un tipo de *side channel attack*, en donde el atacante intenta comprometer la implementación criptográfica de un sistema al analizar el tiempo que le toma ejecutar el algoritmo criptográfico.

Bibliografía

[1] G. S. StatCounter, "Top 5 desktop, tablet and console browsers," 2015. [Online]. Available: http://gs.statcounter.com/

[2] W. W. W. C. W3C, "About." [Online]. Available: http://www.w3.org/Consortium/

[3] "Sandbox - The Chromium Projects." [Online]. Available: http://www.chromium.org/developers/design-documents/sandbox

[4] M. V. Yason, "Diving into IE 10's Enhanced Protected Mode Sandbox."

[5] M. Crowley, *Pro Internet Explorer 8 & 9 Development: Developing Powerful Applications for The Next Generation of IE*, 1st ed. Berkely, CA, USA: Apress, 2010.

[6] "Multi-process Architecture - The Chromium Projects." [Online]. Available: https://www.chromium.org/developers/design-documents/multi-process-architecture

[7] "Chromium Rendering Pipeline." [Online]. Available: http://www.slideshare.net/HyungwookLee/android-chromium-rendering-pipeline

[8] "Internet Explorer Architecture (Internet Explorer)." [Online]. Available: https://msdn.microsoft.com/en-us/library/aa741312(v=vs.85).aspx

[9] "IE8 and Loosely-Coupled IE (LCIE) - IEBlog - Site Home." [Online]. Available: http://blogs.msdn.com/b/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx

[10] A. Grosskurth and M. W. Godfrey, "A reference architecture for web browsers," 2005, pp. 661–664, uRL: `http://grosskurth.ca/papers.html#browser-refarch`.

[11] ——, "Architecture and evolution of the modern web browser," uRL: `http://grosskurth.ca/papers.html#browser-archevol`. Note: submitted for publication.

[12] "How browsers work." [Online]. Available: http://taligarsiel.com/Projects/howbrowserswork1.htm

[13] "Firefox Electrolysis 101." [Online]. Available: https://timtaubert.de/blog/2011/08/firefox-electrolysis-101/

[14] K. M. Goertzel, T. Winograd, H. L. McKinley, L. J. Oh, M. Colon, T. McGibbon, E. Fedchak, and R. Vienneau, "Software security assurance: A state-of-art report (sar)," DTIC Document, Tech. Rep., 2007.

[15] J. Yoder, J. Yoder, J. Barcalow, and J. Barcalow, "Architectural patterns for enabling application security," *Proceedings of PLoP 1997*, vol. 51, p. 31, 1998. [Online]. Available: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Architectural+patterns+for+enabling+application+security#0

[16] E. B. Fernandez, "A methodology for secure software design." in *Software Engineering Research and Practice*, 2004, pp. 130–136.

[17] B. Whyte and J. Harrison, "State of Practice in Secure Software: Experts' Views on Best Ways Ahead." IGI Global. [Online]. Available: http://www.igi-global.com/chapter/state-practice-secure-software/48404

[18] C. M. U. Computer Emergency Response Team, "Early identification reduces total cost (segment from cert's podcasts for bussiness leaders)." [Online]. Available: http://www.cert.org/podcasts/podcast_episode.cfm?episodeid=34820

[19] M. Hicks, "Interview to **Kevin Haley** (from **Symantec**)," 2014, mike Hicks (Profesor of Software Security course in Coursera.org).

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

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

[22] E. B. Fernandez, M. VanHilst, M. M. L. Petrie, and S. Huang, "Defining security requirements through misuse actions," in *Advanced Software Engineering: Expanding the Frontiers of Software Technology*. Springer US, 2006, pp. 123–137.

[23] F. A. Braz, E. B. Fernandez, and M. VanHilst, "Eliciting security requirements through misuse activities," in *Database and Expert Systems Application, 2008. DEXA'08. 19th International Workshop on*. IEEE, 2008, pp. 328–333.

[24] E. B. Fernandez, N. Yoshioka, H. Washizaki, J. Jurjens, M. VanHilst, and G. Pernu, *Using Security Patterns to Develop Secure Systems*, H. Mouratidis,

Ed. IGI Global, 2011. [Online]. Available: http://www.igi-global.com/chapter/using-security-patterns-develop-secure/48405

[25] P. Avgeriou, "Describing, Instantiating and Evaluating a Reference Architecture: A Case Study," *Enterprise Architect Journal*, vol. 342, no. 1, p. 347, 2003. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/21213183

[26] M. Galster and P. Avgeriou, "Empirically-grounded Reference Architectures: A Proposal," pp. 153–157, 2011.

[27] F. Buschman, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, "A system of patterns: pattern-oriented software architecture," 1996.

[28] W. Alcorn, C. Frichot, and M. Orrù, *The Browser Hacker's Handbook*. John Wiley & Sons, 2014.

[29] "HTML5 Web Messaging." [Online]. Available: http://www.w3.org/TR/webmessaging/

[30] . D. XMLHttpRequest Level 1, "Xmlhttprequest specification." [Online]. Available: http://www.w3.org/TR/2014/WD-XMLHttpRequest-20140130/

[31] T. W. API, "Websocket specification." [Online]. Available: http://www.w3.org/TR/2012/CR-websockets-20120920/

[32] W. . R. time Communication Between Browsers, "Webrtc specification." [Online]. Available: http://www.w3.org/TR/2015/WD-webrtc-20150210/

[33] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2." [Online]. Available: https://tools.ietf.org/html/rfc5246

[34] W. W. Group, "Html5 specification." [Online]. Available: http://www.w3.org/TR/html5/

[35] A. Barth, C. Jackson, and W. Li, "Attacks on javascript mashup communication," in *Proceedings of the Web*, vol. 2. Citeseer, 2009.

[36] A. Barth, J. Weinberger, and D. Song, "Cross-Origin JavaScript Capability Leaks : Detection , Exploitation , and Defense," *Opera*, vol. 147, pp. 187–198, 2009.

[37] A. Barth, A. P. Felt, P. Saxena, and A. Boodman, "Protecting Browsers from Extension Vulnerabilities," *Ndss*, vol. 147, pp. 1315–1329, 2010. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.154.5579&rep=rep1&type=pdf

[38] L. Liu, X. Zhang, G. Yan, and S. Chen, "Chrome extensions: Threat analysis and countermeasures," *. . . of the Network and Distributed Systems . . .*, 2012. [Online]. Available: https://www.cs.gmu.edu/~sqchen/publications/NDSS-2012.pdf

[39] A. Singh and S. Sathappan, "A Survey on XSS web-attack and Defense Mechanisms," vol. 4, no. 3, pp. 1160–1164, 2014.

[40] A. Barth, C. Jackson, C. Reis, T. Team *et al.*, "The security architecture of the chromium browser," 2008.

[41] "IE8 and Loosely-Coupled IE (LCIE) - IEBlog - Site Home." [Online]. Available: http://blogs.msdn.com/b/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx

[42] C. Reis and S. D. Gribble, "Isolating web programs in modern browser architectures," *Proceedings of the fourth ACM european conference on Computer systems EuroSys 09*, vol. 25, no. 1, p. 219, 2009. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1519065.1519090

[43] A. Barth, C. Jackson, and J. C. Mitchell, "Securing frame communication in browsers," *Communications of the ACM*, vol. 52, no. 6, pp. 83–91, 2009.

[44] A. Saini, M. S. Gaur, and V. Laxmi, "Privacy Leakage Attacks in Browsers," pp. 257–276, 2014.

[45] O. S. Architecture, "Definitions by osa." [Online]. Available: http://www.opensecurityarchitecture.org/cms/definitions

[46] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012.

[47] O. Encina, "Towards a Security Reference Architecture for Federated Inter-Cloud Systems," 2014.

[48] K. Hashizume, E. B. Fernandez, and M. M. Larrondo-petrie, "A reference architecturefor cloud computing. Submitted for publication." 2014.

[49] E. B. Fernandez, H. Washizaki, N. Yoshioka, and M. VanHilst, "An approach to model-based development of secure and reliable systems," *Proceedings of the 2011 6th International Conference on Availability, Reliability and Security, ARES 2011*, pp. 260–265, 2011.

[50] E. Fernández and M. Larrondo, "Security Patterns and Secure Systems Design," no. June, pp. 1–12, 2006.

[51] E. Fernandez, J. Pelaez, and M. Larrondo-Petrie, "Attack patterns: A new forensic and design tool," in *Advances in digital forensics III*. Springer New York, 2007, pp. 345–357.

[52] E. Fernandez, N. Yoshioka, and H. Washizaki, "Modeling misuse patterns," in *Availability, Reliability and Security, 2009. ARES '09. International Conference on*, March 2009, pp. 566–571.

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

[54] ——, "Integration of attack patterns and protective patterns," in *1st International Workshop on Software Patterns and Quality (SPAQu'07)*, 2007, p. 45.

[55] J. C. Pelaez, E. B. Fernandez, and M. M. Larrondo-Petrie, "Misuse patterns in voip," *Security and Communication Networks*, vol. 2, no. 6, pp. 635–653, 2009.

[56] E. B. Fernandez, N. Yoshioka, and H. Washizaki, "A worm misuse pattern," in *Proceedings of the 1st Asian Conference on Pattern Languages of Programs*. ACM, 2010, p. 2.

[57] K. Hashizume, N. Yoshioka, and E. B. Fernandez, "Misuse patterns for cloud computing," in *Proceedings of the 2nd Asian Conference on Pattern Languages of Programs*. ACM, 2011, p. 12.

[58] J. Muñoz-Arteaga, E. B. Fernandez, and H. Caudel-García, "Misuse pattern: spoofing web services," in *Proceedings of the 2nd Asian Conference on Pattern Languages of Programs*. ACM, 2011, p. 11.

[59] E. B. Fernandez, E. Alder, R. Bagley, and S. Paghdar, "A misuse pattern for retrieving data from a database using sql injection," in *BioMedical Computing (BioMedCom), 2012 ASE/IEEE International Conference on*. IEEE, 2012, pp. 127–131.

[60] A. Alkazami and E. B. Fernandez, "Cipher suite rollback: A misuse pattern for the ssl/tls client/server authentication handshake protocol," 2014.

[61] O. Encina, E. B. Fernandez, and R. Monge, "A misuse pattern for denial-of-service in federated inter-clouds," 2014.

[62] J. Talamantes, "The social engineer's playbook." [Online]. Available: http://www.thesocialengineersplaybook.com/

[63] M. Rajab, L. Ballard, and N. Lutz, "CAMP: Content-agnostic malware protection," *Proceedings of Annual . . .*, 2013. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.295.6192&rep=rep1&type=pdf

[64] N. S. S. Labs and A. R. Abrams, "Evolutions In Browser Security," no. October, pp. 1–20, 2013.

[65] "Top 10 2013 - OWASP." [Online]. Available: https://www.owasp.org/index.php/Top_10_2013

[66] "Security/ProcessIsolation/ThreatModel." [Online]. Available: https://wiki.mozilla.org/Security/ProcessIsolation/ThreatModel

[67] R. Abrams, J. Pathak, and O. Barrera, "Browser security comparative analysis: Phishing protection," 2013.

[68] ——, "Browser Security Comparative Analysis: Socially Engineered Malware Blocking," 2014.

[69] J. Drake, P. Mehta, C. Miller, S. Moyer, R. Smith, C. Valasek, and A. Q. Approach, "Browser Security Comparison," *Accuvant Labs*, 2011.

[70] N. Utakrit, "Review of Browser Extensions, a Man-in-the-Browser Phishing Techniques Targeting Bank Customers," *Proceedings of the 7th Australian Information Security Management Conference*, pp. 110–119, 2009. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84864552184&partnerID=40&md5=3d08a9c7c4ba9dbe5e04fb831ad5257b$\delimiter"026E30F$nhttp://ro.ecu.edu.au/ism/19/

[71] T. Dougan and K. Curran, "Man in the Browser Attacks," *International Journal of Ambient Computing and Intelligence*, vol. 4, no. 1, pp. 29–39, 2012.

[72] "DOM Based XSS - OWASP." [Online]. Available: https://www.owasp.org/index.php/DOM_Based_XSS

[73] "[DOM Based Cross Site Scripting or XSS of the Third Kind] Web Security Articles - Web Application Security Consortium." [Online]. Available: http://www.webappsec.org/projects/articles/071105.shtml

[74] "272620 – XSS vulnerability in internal error messages." [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=272620

[75] S. D. Paola and G. Fedon, "Subverting Ajax," *23rd Chaos Communication Congress*, no. December, 2006. [Online]. Available: http://events.ccc.de/congress/2006/Fahrplan/attachments/1158-Subverting_Ajax.pdf

[76] A. B. <ietf@adambarth.com>, "The Web Origin Concept." [Online]. Available: http://tools.ietf.org/html/draft-abarth-origin-09

[77] M. Zalewsk, "Browser security handbook, part 2," Google, Web page, 2008. [Online]. Available: https://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy

[78] B. Sullivan and V. Liu, *Web application security.* McGraw-Hill, 2012.

[79] J. Hodges, C. Jackson, and A. Barth, "HTTP Strict Transport Security {(HSTS)}," Internet Engineering Task Force (IETF), RFC 6797, 2012.

[80] C. Reis, A. Barth, and C. Pizano, "Browser Security: Lessons from Google Chrome," *Commun. ACM*, vol. 52, no. 8, pp. 45–49, 2009. [Online]. Available: http://dl.acm.org/ft_gateway.cfm?id=1556050&type=html$\delimiter"026E30F$npapers3://publication/doi/10.1145/1538947.1556050

[81] "Necko: Electrolysis design and subprojects." [Online]. Available: https://wiki.mozilla.org/Necko:_Electrolysis_design_and_subprojects

[82] R. Colvin, "SmartScreen," 2010. [Online]. Available: http://blogs.msdn.com/b/ie/archive/2010/10/13/stranger-danger-introducing-smartscreen-application-reputation.aspx

[83] "Site Isolation - The Chromium Projects." [Online]. Available: https://www.chromium.org/developers/design-documents/site-isolation

[84] "The Future of Developing Firefox Add-ons | Mozilla Add-ons Blog." [Online]. Available: https://blog.mozilla.org/addons/2015/08/21/the-future-of-developing-firefox-add-ons/

[85] M. W. Godfrey and E. H. S. Lee, "Secrets from the Monster: Extracting Mozilla's Software Architecture," *In Proc. of 2000 Intl. Symposium on Constructing software engineering tools (CoSET 2000*, pp. 15–23, 2000. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.9211

[86] A. Systems and N. Lwin, "Agent Based Web Browser," *2009 Fifth International Conference on Autonomic and Autonomous Systems*, 2009.

[87] G. Team, "Evolution of the web." [Online]. Available: http://www.evolutionoftheweb.com/

[88] "Internet Explorer Architecture (Internet Explorer)." [Online]. Available: https://msdn.microsoft.com/en-us/library/aa741312(v=vs.85).aspx

[89] "Multiprocess Desktop Firefox | Air Mozilla | Mozilla, in Video." [Online]. Available: https://air.mozilla.org/inter-presentation-schuster/

[90] "Firefox Nightly Builds." [Online]. Available: https://nightly.mozilla.org/

[91] "Multiprocess Firefox | Bill McCloskey's Blog on WordPress.com." [Online]. Available: https://billmccloskey.wordpress.com/2013/12/05/multiprocess-firefox/#ipc

[92] "Inter-process Communication (IPC) - The Chromium Projects." [Online]. Available: https://www.chromium.org/developers/design-documents/inter-process-communication

[93] T. Vrbanec, "The evolution of web browser architecture," pp. 472–480, 2013.

[94] T. M. Coporation, "Common vulnerabilities and exposures." [Online]. Available: https://cve.mitre.org/about/terminology.html

[95] W3C, "Same Origin Policy," W3C, Web page, 2010. [Online]. Available: https://www.w3.org/Security/wiki/Same_Origin_Policy

[96] C. Jackson and A. Barth, "Beware of finer-grained origins," *Web 2.0 Security and Privacy*, 2008. [Online]. Available: http://seclab.stanford.edu/websec/origins/fgo.pdf

[97] M. Silic, J. Krolo, and G. Delac, "Security vulnerabilities in modern web browser architecture," *MIPRO, 2010 Proceedings of the 33rd International Convention*, 2010.

[98] E. B. Fernandez and R. Pan, "A pattern language for security models," *proceedings of PLOP*, vol. 1, pp. 1–13, 2001. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.90.5898

[99] E. B. Fernandez, R. Monge, and K. Hashizume, "Building a security reference architecture for cloud systems," in *Proceedings of the WICSA 2014 Companion Volume*. ACM, 2014, p. 3.

[100] I. C. Security, "Avoiding the top 10 security flaws." [Online]. Available: http://cybersecurity.ieee.org/center-for-secure-design/avoiding-the-top-10-security-flaws.html