Protección de Aplicaciones Móviles en iOS

Trabajo Final de Máster
Curso 2017-18

Universitat Rovira i Virgili
UOC www.uoc.edu
UAB Universitat Autònoma de Barcelona
UIB Universitat de les Illes Balears

Máster Interuniversitario de Seguridad de las Tecnologías de la Información y de las Comunicaciones (MISTIC)

Realizado por: *Félix Garcia Lainez*
Consultor: *Pau del Canto Rodrigo*
Profesor de la Asignatura: *Victor García Font*
Fecha de Entrega: *Junio de 2018*

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-SinObraDerivada 3.0 España de Creative Commons

Ficha del Trabajo Final

Título del trabajo:	*Protección de Aplicaciones Móviles en iOS*
Nombre del autor:	*Félix Garcia Lainez*
Nombre del consultor/a:	*Pau del Canto Rodrigo*
Nombre del PRA:	*Victor García Font*
Fecha de entrega (mm/aaaa):	06/2018
Titulación::	*MISTIC*
Área del Trabajo Final:	*Trabajo Final de Máster*
Idioma del trabajo:	*Castellano*
Palabras clave	*Seguridad, Aplicaciones Móviles, iOS*

Resumen

El objetivo de este trabajo ha sido la creación de una guía o manual de referencia para el desarrollo de aplicaciones móviles nativas seguras en iOS.

Para poner en contexto este trabajo, tal y como se describe en los capítulos 2.1 y 2.2 de esta memoria, la gran popularidad que han alcanzado las aplicaciones móviles en los últimos años ha propiciado la aparición de una gran cantidad de riesgos y amenazas asociadas a este tipo de tecnologías.

Con el objetivo de mitigar y prevenir estos riesgos y amenazas, tal y como se recoge en el capítulo 2.3, el OWASP trató de estandarizar el desarrollo de este tipo de aplicaciones a través de la creación del MASVS [1]. Aun a pesar de no contar con el consenso de la industria, se podría decir que a día de hoy este estándar es el principal punto de referencia en este campo.

Si bien este estándar supone un buen punto de partida, éste adolece de la falta de un conjunto de soluciones técnicas, buenas prácticas y casos de test que permitan facilitar y verificar el cumplimiento de los requisitos incluidos en el mismo. Así pues, en el capítulo 5 de este trabajo se da solución a esta problemática a través de la revisión de todos y cada uno de estos requisitos, proporcionando una línea de base para la verificación de la seguridad en las aplicaciones móviles nativas, así como la orientación necesaria durante todas las fases del desarrollo y pruebas de las mismas.

Abstract

The goal of this Master's Thesis has been the creation of a guide or reference manual for the development of secure native mobile applications for iOS.

To put this work in context, as described in chapters 2.1 and 2.2 of this thesis, the great popularity that mobile applications have achieved in recent years has encouraged the emergence of a large number of risks and threats associated with this kind of technologies.

In order to mitigate and prevent these risks and threats, as stated in chapter 2.3, the OWASP tried to standardize the development of this type of applications through the creation of the MASVS [1]. Even in spite of not having the consensus of the industry, it could be said that today this standard is the main point of reference in this field.

Although this standard is a good starting point, it suffers from the lack of a set of technical solutions, good practices and test cases to facilitate and verify compliance with the requirements included in it. In chapter 5 of this document this problem is solved through the review of each one of these requirements, providing a baseline for the verification of security in native mobile applications, as well as the necessary guidance during all phases of the development and testing of them.

Indice

1. Objetivos 7
2. Estado del Arte 7
2.1. Riesgos y Controles 8
2.2. Riesgos en Aplicaciones Móviles (Client-Side) 11
2.3. Mobile Application Security Verification Standard (MASVS) 13
3. Metodología 16
4. Planificación 17
5. Requisitos de Seguridad 20
5.1. V1: Requisitos de Arquitectura, Diseño y Modelado de Amenazas 20
5.2. V2: Requisitos de Almacenamiento de Datos y Privacidad 22
5.3. V3: Requisitos de Criptografía 33
5.4. V4: Requisitos de Autenticación y Gestión de Sesiones 36
5.5. V5: Requisitos de Comunicaciones de Red 44
5.6. V6: Requisitos de Interacción con la Plataforma 48
5.7. V7: Requisitos de Calidad del Código y Configuración de la Compilación 55
5.8. V8: Requisitos de Resistencia contra la Ingeniería Inversa 65
6. Conclusiones 79
7. Glosario 80
8. Bibliografía 83
9. Anexos 87

1. Objetivos

El objetivo de este Trabajo Final de Máster es desarrollar un manual de referencia para llevar a cabo el desarrollo de aplicaciones móviles nativas seguras para iOS. Esta guía se ha pensado desde el punto de vista del desarrollador, de tal manera que en la misma se exponen una serie de requisitos de seguridad que deberían cumplir las aplicaciones nativas desarrolladas para iOS, incluyendo para cada uno de estos requisitos diferentes procesos y técnicas para cumplir con los mismos. Al final, la idea de este manual de referencia es que un desarrollador de aplicaciones móviles nativas para iOS pueda utilizarlo tanto para verificar el nivel de seguridad de una aplicación cualquiera, como para aplicar diferentes técnicas de seguridad descritas en el mismo que le permitan cumplir con estos requisitos establecidos.

2. Estado del Arte

Según un estudio reciente del Instituto Nacional de Ciberseguridad de España (INCIBE), un usuario promedio de teléfonos inteligentes o smartphones tiene alrededor de 26 aplicaciones instaladas en su terminal, de las que el 97% tiene algún tipo de problema de privacidad. En términos de seguridad, el 86% carecía de las formas más básicas y el 75% no cifra los datos recopilados.

Las conexiones que se realizan desde estos dispositivos móviles son inalámbricas en prácticamente el 100% de los casos a través de tecnologías como pueden ser las redes de datos móviles, conexiones Wi-Fi, Bluetooth o NFC - suponiendo un punto de entrada potencial para virus, malware, etc. De la misma manera, este tipo de dispositivos almacenan todo tipo de información sobre los usuarios, de tal manera que el acceso no autorizado a dichos datos pueda comprometer la privacidad de los mismos.

Así pues, al igual que en cualquier otro tipo de tecnologías, es de vital importancia que los desarrolladores de este tipo de aplicaciones hagan hincapié en todo tipo de aspectos relacionados con la seguridad en el desarrollo las mismas, desde su concepción hasta su lanzamiento, adelantándose a los riesgos y amenazas potenciales.

Cada nueva tecnología introduce nuevos riesgos de seguridad, y mantenerse al día con esos cambios es uno de los principales desafíos a los que se enfrenta la industria de la seguridad. El lado defensivo está siempre unos pocos pasos por detrás, por ejemplo a primera vista puede parecer que los teléfonos inteligentes son como las computadoras pequeñas y las aplicaciones móviles son como el software clásico, por lo que seguramente los requisitos de seguridad deberían de ser similares en ambos casos. Pero la realidad es que esto no funciona así. Los sistemas operativos de los teléfonos inteligentes son diferentes de los de escritorio, al igual que los sistemas operativos y las aplicaciones móviles son diferentes de las aplicaciones web. Por ejemplo, el método clásico de escaneo de virus basado en firmas no tiene sentido en los entornos de SO móviles modernos: no solo es incompatible con el modelo de distribución de aplicaciones móviles, también es técnicamente imposible debido a restricciones de *sandboxing*. Además, algunas tipos de vulnerabilidades como es el caso de los desbordamientos de buffer y problemas de XSS son menos relevantes en el contexto de aplicaciones móviles corrientes que en el caso de por ejemplo las aplicaciones de escritorio y aplicaciones web.

Con el tiempo, esta industria ha logrado un mejor control sobre las amenazas existentes en las aplicaciones móviles. Como resultado, la seguridad en este tipo de dispositivos se basa principalmente en la protección de datos: las aplicaciones almacenan información personal, imágenes, grabaciones de audio, notas, datos de cuentas y tarjetas bancarias, información de negocios, ubicación y mucho más. Actúan como clientes que conectan a los usuarios con los servicios que utilizan a diario, así como centros de comunicaciones que procesan todos y cada uno de los mensajes que se intercambian con otros usuarios. Comprometer el teléfono inteligente de una persona es equivalente a obtener acceso a la vida de esa persona. Teniendo en cuenta

que los dispositivos móviles se pierden o se roban más fácilmente cada día y que el malware móvil está en auge, la necesidad en la protección de datos se hace aún más evidente.

Por lo tanto, un estándar de seguridad para el desarrollo de aplicaciones móviles debe enfocarse en cómo las aplicaciones móviles manejan, almacenan y protegen la información confidencial de los usuarios de las mismas. Aunque los sistemas operativos móviles modernos, como iOS y Android, ofrecen potentes APIs y mecanismos para el almacenamiento seguro de datos y gestión de las comunicaciones, estas APIs deben implementarse y utilizarse correctamente para que sean efectivas. El almacenamiento de datos, la comunicación entre aplicaciones, el uso adecuado de las APIs criptográficas y las comunicaciones de red seguras son solo algunos de los aspectos que requieren una cierta consideración.

Una cuestión importante que necesita consenso de la industria es a que nivel se debe llevar la protección de la confidencialidad e integridad de los datos para cada tipo de aplicación. Por ejemplo, la gran mayoría de usuarios estaría de acuerdo en que una aplicación móvil cualquiera debería verificar el certificado del servidor en un intercambio de TLS. ¿Pero sería necesario llevar a cabo un SSL certificate pinning? ¿No hacerlo sería equivalente a una vulnerabilidad? ¿Debería ser un requisito si una aplicación maneja datos confidenciales o podría llegar a ser incluso contraproducente? ¿Necesitamos encriptar los datos almacenados en las bases de datos SQLite, a pesar de que el SO sandboxes las aplicaciones? Lo que es apropiado para una aplicación puede ser poco realista para otra. El MASVS que se describe en detalle dentro del apartado 2.3 es un intento de estandarizar estos requisitos utilizando niveles de verificación que se ajustan a diferentes escenarios de amenazas para cada tipo de aplicación.

Además, la aparición del denominado root malware y las herramientas de administración remota ha creado conciencia de que los sistemas operativos móviles tienen vulnerabilidades explotables, por lo que las estrategias de contenedorización se utilizan cada vez más para proporcionar protección adicional a los datos confidenciales y evitar la manipulación en el lado del cliente. Aquí es donde las cosas se complican. Las características de seguridad respaldadas por hardware y las soluciones de contenedor a nivel del sistema operativo, como Android for Work y Samsung Knox, sí existen, pero no están disponibles para todos los dispositivos. Como forma de mitigar estas amenazas, es posible implementar medidas de protección basadas en software, pero desafortunadamente no existen estándares o procesos de prueba para verificar este tipo de protecciones.

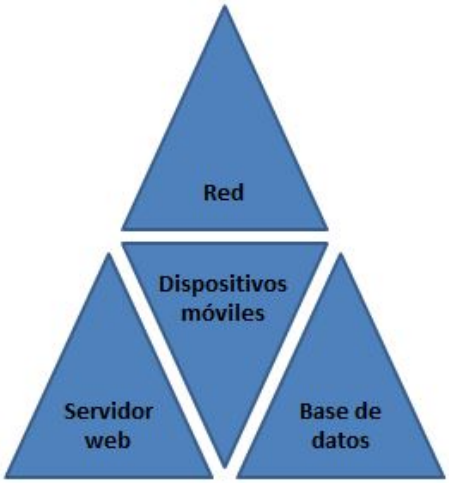
Como resultado, los informes de pruebas de seguridad de aplicaciones móviles están por todas partes: por ejemplo, algunos testers informan sobre la falta de ofuscación o detección de dispositivos rooted en una aplicación de Android como un fallo de seguridad. Por otro lado, medidas como el cifrado de cadenas, detección de depuradores u ofuscación del flujo de control no se consideran obligatorias. Sin embargo, esta forma binaria de ver las cosas no tiene sentido porque la resiliencia no es una proposición binaria: depende de las amenazas particulares del lado del cliente que se pretenden defender. Las protecciones de software no son inútiles, pero en última instancia se pueden omitir, por lo que nunca se deben usar como reemplazo de los controles de seguridad.

2.1. Riesgos y Controles

En esta sección se van describir los riesgos y amenazas existentes en las aplicaciones móviles, de tal manera que podrían clasificarse en 4 categorías principales que se describen a continuación.

Aun a pesar de que a este apartado en concreto se le ha dado un cierto enfoque al sector empresarial o institucional, comentar que estas 4 categorías junto con prácticamente todos los riesgos y controles englobados dentro de cada una de ellas se podrían aplicar a cualquier tipo de aplicación móvil nativa, independientemente de su naturaleza.

Para llevar a cabo el desarrollo de este apartado de la memoria se ha utilizada como referencia un artículo publicado en el blog del Máster Universitario en Ingeniería Informática de la Universidad de Deusto [7].



Dispositivos Móviles (Client-Side)

1. Almacenamiento de datos

   - Riesgo.- Pérdida y divulgación de datos (DLP y MCM).

   - Control.- Establecer un cifrado de los datos almacenados en el dispositivo móvil a través de algoritmos de cifrado como puede ser AES (Advanced Encryption Standard).

2. Transmisión de datos

   - Riesgo.- Pérdida y divulgación de datos (DLP y MCM).

   - Control.- Aplicar un cifrado de los datos en transmisión a través de la Capa de Puertos Seguros (Secure Sockets Layer o SSL) u otros protocolos de seguridad como pueden ser HTTPs (Acceso Web), FTPs (Transferencia de Archivos), TLS (Protocolos de Seguridad), etc.

3. Aplicación de gestión de acceso y seguridad

   - Riesgo.- Acceso no autorizado y fraude.

   - Control.- Gestión de aplicaciones móviles (MAM) se utiliza para gestionar el acceso y despliegue de aplicaciones en los terminales. Además, se mantienen unas adecuadas listas blancas y listas negras. Mediante este control la aplicación está configurada para limitar el acceso y configurada adecuadamente para uso autorizado limitado.

4. Llevar dispositivos móviles fuera del perímetro empresarial

   - Riesgo.- Pérdida o robo del dispositivo móvil, haciendo posible el acceso no autorizado a las aplicaciones móviles del dispositivo y al fraude de los datos de éstas.

   - Control.- Similares a los del punto anterior.

Red

1. Conectividad inalámbrica

   - Riesgo.- Pérdida y divulgación de datos (DLP & MCM).

   - Control.- La transmisión de datos utiliza, como mínimo, SSL o TLS. Ambos protocolos criptográficos para la transmisión segura de datos. Mediante este control el cifrado se aplica cuando se activa la conexión Wi-Fi.

2. Secuestro de sesión (Session hijacking)

   - Riesgo.- Pérdida y divulgación de datos y acceso no autorizado (DLP & MCM).

   - Control.- Los protocolos de conexión para el Localizador Uniforme de Recursos (Uniform Resource Locator: URL) a través de TLS son a través de HTTPS en lugar de HTTP para conectarse de forma segura a una URL. Mediante este control se evita el secuestro de una sesión debido a un protocolo de conexión inseguro.

Servidor Web

1. Gestión de acceso

   - Riesgo.- Pérdida y divulgación de datos y acceso no autorizado (DLP & MCM).

   - Control.- Todos los servidores web aplicables se asignan a los propietarios de sistemas técnicos y empresariales. Los roles y responsabilidades definidos son adecuados, especialmente para el personal interno y de terceros. Mediante este control los roles y responsabilidades de la propiedad son establecidos, documentados y comunicados.

2. Ataque de fuerza bruta

   - Riesgo.- Acceso no autorizado y fraude, disponibilidad de la aplicación.

   - Control.- Los protocolos de bloqueo están habilitados para cuentas con varios intentos de contraseña incorrectos. Se recomienda la utilización de CAPTCHA (programa que distingue entre seres humanos y ordenadores) para evitar DoS (Denegación de Servicio). Mediante este control la gestión de la estrategia de DoS abarca programas adecuados para bloquear los protocolos no autorizados.

Base de Datos

1. Acceso privilegiado

   - Riesgo.- Acceso no autorizado y fraude.

   - Control.- El acceso a la BD está limitado a las personas apropiadas, y las revisiones de acceso adecuadas y las cuentas del sistema documentadas se mantienen archivadas. Todas las cuentas y contraseñas predeterminadas se deshabilitan al aplicar controles de contraseña estrictos. Mediante este control el acceso elevado a las BBDD se asegura adecuadamente utilizando las mejores prácticas.

2. Inyección SQL (SQL injection)

   - Riesgo.- Acceso no autorizado y fraude.

- Control.- Se da lugar a la técnica de validación de entrada; existen reglas específicamente definidas para el tipo y la sintaxis contra las reglas clave de negocio. Mediante este control el acceso a la BD del Back-end está protegido adecuadamente de vulnerabilidades utilizando técnicas de validación de entrada apropiadas.

3. Validación de la entrada de la aplicación (cliente)

   - Riesgo.- Acceso no autorizado y fraude.

   - Control.- La limpieza de los datos de usuario de la aplicación procedentes de la aplicación móvil se protege adecuadamente mediante comprobaciones de lógica incorporada dentro de la aplicación. La correcta implementación de las comprobaciones lógicas está habilitada en el lado del servidor. Mediante este control los datos procedentes de aplicaciones móviles son examinados antes de confiar en ellos para extraerlos o enviarlos a la capa de BD.

4. Servicios de BD de aplicaciones

   - Riesgo.- Acceso no autorizado y fraude.

   - Control.- El servidor de BD se prueba adecuadamente y se protege contra ataques maliciosos. Los formularios de inicio de sesión requieren HTTPS. Las conexiones SSL son obligatorias.

2.2. Riesgos en Aplicaciones Móviles (Client-Side)

En este apartado se establece una clasificación más detallada de los principales riesgos y amenazas existentes dentro de la categoría de dispositivos móviles introducida en el apartado anterior. Puesto que este Trabajo Final de Máster se centra en el desarrollo de aplicaciones móviles nativas seguras para iOS desde el punto de vista del cliente móvil, por lo que de aquí en adelante se pondrá el foco en todo lo relacionado con esta categoría.

Comentar que para el desarrollo de este apartado de la memoria se ha tomado como referencia el artículo *Mobile Top 10 2016-Top 10* recogido dentro de la web del OWASP [4].

- **M1 - Uso Inapropiado de la plataforma.-** Esta categoría cubre el uso indebido de una característica de la plataforma o la imposibilidad de utilizar los controles de seguridad proporcionados por el sistema operativo. Dentro de esta categoría podríamos incluir permisos de la plataforma, mal uso de TouchID o FaceID, el llavero o algún otro control de seguridad que sea parte del sistema operativo móvil. Hay varias formas en que las aplicaciones móviles pueden experimentar este tipo de riesgo.

- **M2 - Almacenamiento de datos inseguro.-** Esta nueva categoría es una combinación de las categorías M2 + M4 del Mobile Top Ten 2014. Esta categoría engloba el almacenamiento de datos inseguros y la pérdida de datos no intencionada.

- **M3 - Comunicaciones inseguras.-** Esta categoría incluye handshaking inseguro, versiones incorrectas de SSL, negociación débil, envío en texto plano de datos confidenciales, etc.

- **M4 - Autenticación insegura.-** Esta categoría incluye la no autenticación o autenticación del usuario final insegura o gestión de sesión incorrecta. Esto puede incluir la no identificación del usuario cuando debería ser requerido, el no mantener la identidad del usuario cuando se requiere, debilidades en la gestión de sesiones, etc.

- **M5 - Criptografía insuficiente.-** Se aplica criptografía a cierta información sensible. Sin embargo, la criptografía es insuficiente de alguna manera. Se ha de tener en cuenta que todo

lo relacionado con TLS o SSL se engloba dentro de la categoría M3. Además, si la aplicación no puede usar la criptografía cuando debería, eso probablemente pertenece a la categoría M2. Esta categoría incluye todos los escenarios donde se intentó utilizar la criptografía, pero no se realizó correctamente.

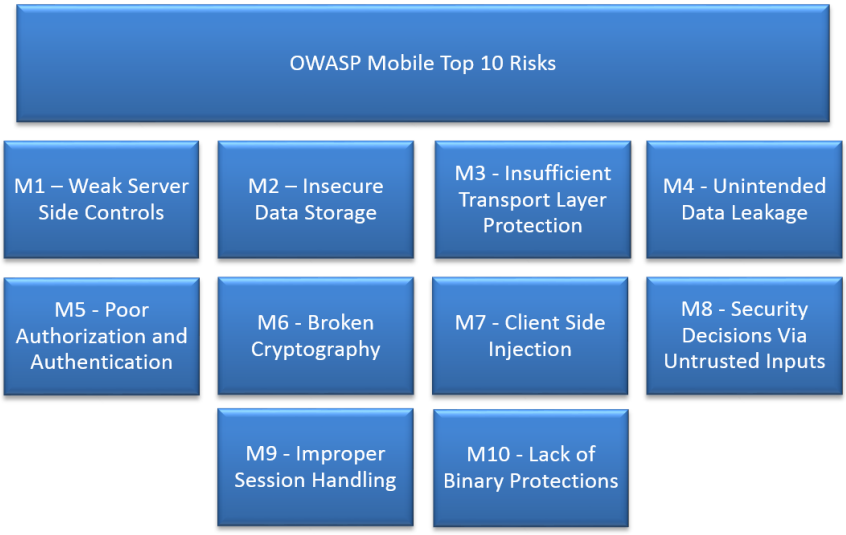
- **M6 - Autorización insegura.-** En esta categoría se recoge cualquier fallo en la autorización (e.g. decisiones de autorización en el lado del cliente, navegación forzada, etc.). Es distinto de los problemas de autenticación (e.g. registro del dispositivo, identificación del usuario, etc). Si la aplicación no autentica a los usuarios en una situación en la que debería (e.g. otorgar acceso anónimo a algún recurso o servicio cuando se autentica y se requiere acceso autorizado), entonces se trata de un fallo de autenticación, no de un fallo de autorización.

- **M7 - Calidad del código en el cliente.-** Esta categoría incluye todo tipo de problemas de implementación a nivel de código en el cliente móvil. Comentar que es distinto de los errores de codificación del lado del servidor. Esto incluiría cosas como desbordamientos de buffers, vulnerabilidades en el formato de cadenas y otros errores a nivel de código donde la solución sería reescribir alguna parte del mismo que se ejecuta en el dispositivo móvil.

- **M8 - Manipulación del código.-** Esta categoría incluye la modificación de los binarios, la modificación de recursos locales, method hooking, method swizzling y la modificación dinámica de la memoria. Una vez que la aplicación se instala en el dispositivo móvil, el código y los recursos junto con los datos residen allí. Un atacante puede modificar directamente el código, cambiar los contenidos de la memoria de forma dinámica, cambiar o reemplazar las APIs del sistema que utiliza la aplicación o modificar los datos y recursos de la aplicación. Esto puede proporcionar al atacante un método directo para subvertir el uso previsto del software de tal manera que se pueda obtener cualquier tipo de ganancia por ejemplo de tipo económico.

- **M9 - Ingeniería inversa.-** Esta categoría incluye el análisis del binario final para determinar su código fuente, librerías, algoritmos y otros activos. Software como IDA Pro, Hopper, otool y otras herramientas de inspección binaria le dan al atacante una idea del funcionamiento interno de la aplicación. Esto se puede usar para explotar otras vulnerabilidades incipientes en la aplicación, así como también para revelar información acerca de los servidores back-end, las constantes y cifrados criptográficos así como la propiedad intelectual.

- **M10 - Funcionalidades extrañas.-** A menudo, los desarrolladores incluyen funciones ocultas de puerta trasera u otros controles internos de seguridad utilizados durante la fase de desarrollo que no están destinados a ser lanzados a un entorno de producción. Por ejemplo, un desarrollador puede incluir accidentalmente una contraseña como comentario en una aplicación híbrida. Otro ejemplo incluye la desactivación de la autenticación de 2 factores durante el periodo de pruebas de la aplicación.



2.3. Mobile Application Security Verification Standard (MASVS)

Una vez definidos en el apartado anterior los riesgos y amenazas más importantes asociados al desarrollo de aplicaciones móviles nativas desde el punto de vista del cliente móvil, en esta sección se introducirá un estándar que permita mitigar estos riesgos y amenazas a través de la verificación del cumplimiento de una serie requisitos de seguridad establecidos dentro del mismo.

El Mobile Application Security Verification Standard (MASVS) es un estándar para la seguridad de aplicaciones móviles. Este estándar puede ser utilizado por arquitectos y desarrolladores de software que buscan desarrollar aplicaciones móviles seguras, así como testers de seguridad para garantizar la integridad y consistencia de los resultados de las pruebas.

El objetivo general del MASVS es ofrecer una línea de base para la seguridad de aplicaciones móviles (MASVS-L1), al tiempo que permite la inclusión de medidas defensivas en profundidad (MASVS-L2) y protecciones contra amenazas del lado del cliente (MASVS-R). El MASVS pretende lograr lo siguiente:

- Proporcionar requisitos para los arquitectos y desarrolladores de software que buscan desarrollar aplicaciones móviles seguras.
- Ofrecer un estándar para la industria contra el que se puedan poner a prueba las revisiones de seguridad de las aplicaciones móviles.
- Aclarar el papel de los mecanismos de protección de software en la seguridad móvil y proporcionar los requisitos necesarios para verificar su efectividad.
- Proporcionar recomendaciones específicas sobre qué nivel de seguridad se recomienda para diferentes casos de uso y tipos de aplicaciones.

Siendo conscientes de que es imposible de lograr un consenso del 100% dentro de la industria, se puede afirmar que el MASVS puede ser tremendamente útil para proporcionar orientación en todas las fases del desarrollo y las pruebas de aplicaciones móviles. Como estándar de código abierto, el MASVS evolucionará con el tiempo, de tal manera que cualquiera pueda contribuir y colaborar en la mejora del mismo. Este MASVS se puede usar para establecer un nivel de confianza en la seguridad de las aplicaciones móviles. Los requisitos incluidos en el mismo se desarrollaron teniendo en cuenta los siguientes objetivos:

- Usar como una métrica.- El objetivo es proporcionar un estándar de seguridad contra el que los desarrolladores y propietarios de aplicaciones puedan comparar las aplicaciones móviles existentes.
- Utilizar como orientación.- El objetivo es proporcionar orientación durante todas las fases de desarrollo y pruebas de una aplicación móvil.
- Uso durante la adquisición.- La idea es proporcionar una línea de base para la verificación de la seguridad en las aplicaciones móviles.

Es importante tener en cuenta que el MASVS solo cubre la seguridad de las aplicación móviles en el lado del cliente, así como la comunicación de red entre dichas aplicación y los servicios remotos con los que conectan, así como algunos requisitos básicos y genéricos relacionados con la autenticación del usuario y la gestión de las sesiones. Este estándar no contiene requisitos específicos para la implementación de los servicios remotos asociados con la aplicación (e.g. Web Services), exceptuando un conjunto limitado de requisitos genéricos relacionados con la autenticación y la gestión de las sesiones. Sin embargo, MASVS V1 especifica que los servicios remotos deben estar cubiertos por el modelo de amenazas general y ser verificados con estándares apropiados, como el ASVS de OWASP.

Niveles de Verificación

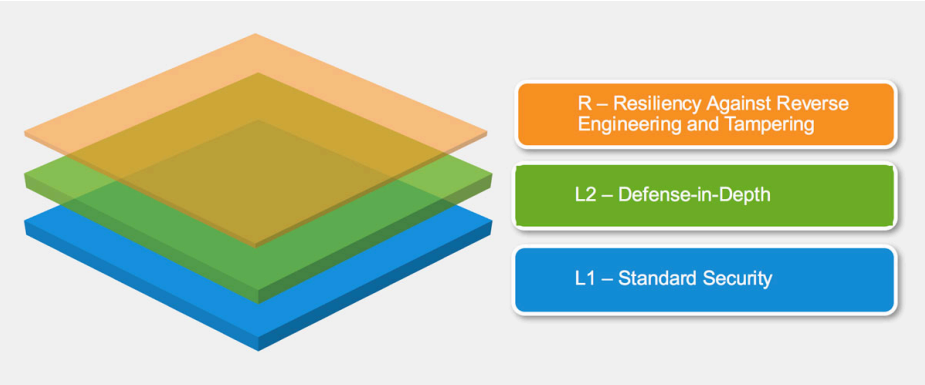
MASVS define dos estrictos niveles de verificación de seguridad (L1 y L2), así como un conjunto de requisitos de ingeniería inversa (MASVS-R) que es flexible, es decir, adaptable a un modelo de amenaza específico de la aplicación. MASVS-L1 y MASVS-L2 contienen requisitos de seguridad genéricos y se recomiendan para todas las aplicaciones móviles (L1) y aplicaciones que manejan datos altamente confidenciales (L2). MASVS-R cubre controles de protección adicionales que se pueden aplicar si la prevención de amenazas en el lado del cliente es un objetivo en el diseño de la aplicación. El cumplimiento de los requisitos en MASVS-L1 da como resultado una aplicación segura que sigue buenas prácticas de seguridad y no sufre de vulnerabilidades comunes. MASVS-L2 agrega niveles de defensa adicionales, incluyendo controles tales como pueden ser e SSL certificate pinning, lo que resulta en una aplicación que es resistente contra ataques más sofisticados, suponiendo que los controles de seguridad del sistema operativo móvil estén intactos y el usuario final no sea visto como un posible adversario. Cumplir con todos o con un subconjunto de los requisitos de protección de software en MASVS-R ayuda a impedir amenazas específicas del lado del cliente donde el usuario final es malicioso y/o el sistema operativo móvil está en peligro.

Es importante tener en cuenta que los controles de protección de software enumerados en MASVS-R y descritos en el MSTG pueden omitirse en última instancia y nunca deben utilizarse como reemplazo de los controles de seguridad estándar. Estos controles están destinados a añadir niveles de protección adicionales específicos sobre ciertas amenazas a las aplicaciones que ya cumplen con los requisitos de MASVS L1 y/o L2.

- MASVS-L1 (Seguridad Estándar).- Una aplicación móvil que cumple los requisitos establecidos en MASVS-L1 se adhiere a las buenas prácticas de seguridad en el desarrollo de aplicaciones móviles. Cumple con los requisitos básicos en términos de calidad de código, manejo de datos confidenciales e interacción con el entorno móvil. Se debe implementar un proceso de pruebas para verificar los controles de seguridad. Este nivel es apropiado para cualquier tipo de aplicaciones móviles.

- MASVS-L2 (Defensa en Profundidad).- Este nivel de verificación presenta controles de seguridad avanzados que van más allá de los requisitos estándar. Para cumplir con L2, debe existir un modelo de amenaza y la seguridad debe ser una parte integral de la arquitectura y el diseño de la aplicación. Este nivel es apropiado para aplicaciones que manejan datos confidenciales, como puede ser el caso de aplicaciones móviles bancarias.

- MASVS-R (Resistencia contra la ingeniería inversa y la modificación de los binarios).- La aplicación cuenta con las tecnologías de seguridad más avanzadas y también es resistente a ataques específicos, claramente definidos en el lado del cliente, como alteración, modding o ingeniería inversa para extraer código o datos confidenciales. Dicha aplicación aprovecha las características de seguridad del hardware o las técnicas de protección de software suficientemente fuertes y verificables. MASVS-R es aplicable a aplicaciones que manejan datos altamente confidenciales y puede servir como un medio para proteger la propiedad intelectual o alterar una aplicación.



Las aplicaciones se pueden verificar contra MASVS L1 o L2 en función de una evaluación de riesgos previa y del nivel general de seguridad requerido. L1 es aplicable a todas las aplicaciones móviles, mientras que L2 generalmente se recomienda para aplicaciones que manejan datos y/o funcionalidades más sensibles. MASVS-R (o algunas partes del mismo) se pueden aplicar para verificar la resistencia contra amenazas específicas, como el repackaging o la extracción de datos confidenciales, además de una verificación de seguridad adecuada. En resumen, se podría decir que están disponibles los siguientes tipos de verificación, con sus consiguientes recomendaciones de uso:

1. MASVS-L1

   - Se debería utilizar en todas las aplicaciones móviles. MASVS-L1 enumera buenas prácticas de seguridad que se pueden seguir con un impacto razonable en los costos de desarrollo y la experiencia del usuario. Se deben aplicar los requisitos de MASVS-L1 para cualquier aplicación que no califique para uno de los niveles superiores.

2. MASVS-L2

   - Industria del cuidado de la salud.- Aplicaciones móviles que almacenan información de identificación personal que puede usarse para robo de identidad, pagos fraudulentos o una variedad de esquemas de fraude. Dentro del sector de la salud de los EE.UU., las consideraciones de cumplimiento incluyen la Ley de Responsabilidad y Portabilidad del Seguro Médico (HIPAA) Privacidad, Seguridad, Reglas de Notificación de Incumplimiento y la Regla de Seguridad del Paciente.

   - Industria financiera.- Aplicaciones que permiten el acceso a información altamente confidencial como números de tarjetas de crédito, información personal, o permiten el movimiento de fondos bancarios. Estas aplicaciones garantizan controles de seguridad adicionales para evitar el fraude. Las aplicaciones financieras deben garantizar conformidad con el estándar de seguridad de datos de la industria de tarjetas de pago (PCI DSS), Gramm Leech Bliley Act y Sarbanes-Oxley Act (SOX).

3. MASVS L1+R

   - Aplicaciones móviles donde la protección de la propiedad intelectual es un objetivo comercial. Los controles de resistencia enumerados en MASVS-R se pueden utilizar para aumentar el esfuerzo necesario para obtener el código fuente original y evitar modificaciones o el cracking de las mismas.

   - Industria de juegos.- Juegos con una necesidad esencial de evitar modificaciones y trampas, así como los juegos competitivos en línea. Hacer trampas es un tema importante en los juegos en línea, ya que una gran cantidad de tramposos suele conllevar que la base de usuarios esté descontenta y en última instancia, puede hacer que el juego esta base de usuarios deje de jugar a los mismos. MASVS-R proporciona controles básicos anti-manipulación para ayudar a aumentar el esfuerzo necesario para llevar a cabo este tipo de conductas éticas.

4. MASVS L2+R

   - Industria financiera.- Aplicaciones de banca en línea que permiten al usuario mover fondos, donde las técnicas de inyección de código y la instrumentación en dispositivos comprometidos suponen un riesgo. En este caso, los controles de MASVS-R se pueden usar para impedir la manipulación, incrementando la dificultad para los autores de malware.

   - Todas las aplicaciones móviles que por diseño necesitan almacenar datos confidenciales en el dispositivo móvil, y al mismo tiempo deben soportar una amplia gama de dispositivos y versiones de sistemas operativos. En este caso, los controles de

resistencia se pueden usar como una medida de defensa en avanzada para aumentar el esfuerzo necesario por parte de los atacantes con el objetivo de extraer los datos confidenciales.

3. Metodología

Para llevar a cabo el desarrollo de este trabajo se decidió seguir la estructura del Mobile Application Security Verification Standard v1.0 (MASVS) [1] conjuntamente con el Mobile Security Testing Guide v1.0 (MSTG) [2], de tal manera que para cada uno de los requisitos de seguridad definidos en cada una de las categorías del standard se puedan establecer una serie de técnicas y procesos para verificar el correcto cumplimiento de los mismos. También se utilizó en cierta manera la Mobile App Security Checklist v1.0 [3], que al final y al cabo se podría considerar como una herramienta que permite facilitar el proceso de verificación del cumplimiento de los requisitos incluidos dentro del estándar por parte de aplicaciones móviles nativas desarrolladas para iOS y/o Android.

Siguiendo la estructura del MASVS, se puede observar que los requisitos de seguridad se pueden clasificar en 8 categorías (V1 a V8) basadas en la tipología de los mismos. A la hora de establecer una nomenclatura para cada una de las categorías y requisitos se decidió seguir el siguiente esquema recogido dentro del estándar.

- Categoría del Requisito: MASVS-Vx. - e.g. *MASVS-V3 Cryptography Requirements*.
- Requisito: MASVS-Vx.y. - e.g. *MASVS-V3.2: The app uses proven implementations of cryptographic primitives*.

Así pues, para el desarrollo de este trabajo se establecerán una serie de sub-apartados dentro del apartado 5 de esta memoria, uno por cada una de las categorías definidas en el MASVS. Este apartado 5 de la memoria se ha definido como *Requisitos de Seguridad*, y dentro de cada uno de sus sub-apartados se analizarán los requisitos de seguridad establecidos para cada una de las categorías, proporcionando casos de test que permitan verificar el cumplimiento de los mismos. En la gran mayoría de los casos, también se incluirán técnicas y procesos detallados tanto a nivel de arquitectura, código fuente, etc para facilitar y asegurar el cumplimiento de cada uno de estos requisitos.

A la hora de definir tanto los casos de test, como las técnicas y procesos detallados se hará uso del MSTG junto con otro tipo de documentación específica sobre seguridad para el desarrollo de aplicaciones móviles nativas para iOS. Este tipo de documentación específica puede ir desde manuales de desarrollo de aplicaciones seguras para iOS proporcionados por Apple [5] [6], hasta foros especializados como puede ser el caso de Stack Overflow [7].

Finalmente, comentar que estos requisitos incluidos en cada una de las categorías existentes pueden estar incluidos o no en cada uno de los niveles de verificación definidos en la sección anterior. Así pues, algunos de estos requisitos estarán incluidos dentro de los niveles de verificación MASVS-L1 y MASVS-L2, otros sólo dentro de MASVS-L2 y otros sólo dentro de MASVS-R.

4. Planificación

En esta sección se establece una planificación detallada de las tareas necesarias para llevar a cabo la ejecución del trabajo a lo largo del semestre. En primer lugar, comentar que esta planificación se basa en la definición de 5 hitos temporales que coinciden con la entrega de cada una de las PEC que es necesario entregar a lo largo del semestre. Cada uno de estos hitos temporales contendrá una serie de tareas que se deberán completar y entregar en cada una de las pruebas de evaluación continua establecidas en la asignatura. A continuación se enumeran estos hitos temporales junto con las tareas englobadas dentro de cada uno de los mismos.

- **PEC1 - Plan de Trabajo**

  Este primer hito contendrá todas las tareas relacionadas con la definición y planificación del trabajo, así como la metodología utilizada para llevar a cabo la ejecución del mismo. A su vez, se incluirá una pequeña revisión del estado del arte sobre seguridad en el desarrollo de aplicaciones móviles nativas. La fecha de inicio de este hito coincidirá con el inicio del semestre (21/2/2018) y la fecha de fin coincidirá con la fecha de entrega de la PEC1 (12/3/2018).

- **PEC2 - Requisitos de Seguridad I**

  Este segundo hito contendrá la ejecución de la primera parte de los requisitos de seguridad a analizar en el trabajo. Así pues, en este hito temporal se deberán analizar las cuatro primeras categorías definidas en el MASVS. La fecha de inicio de este hito será el 13/3/2018 y la fecha de fin coincidirá con la fecha de entrega de la PEC2 (9/4/2018).

- **PEC3 - Requisitos de Seguridad II**

  Este tercer hito contendrá la ejecución de la segunda parte de los requisitos de seguridad a analizar en el trabajo. Así pues, en este hito temporal se deberán analizar las cuatro últimas categorías definidas en el MASVS. La fecha de inicio de este hito será el 10/4/2018 y la fecha de fin coincidirá con la fecha de entrega de la PEC3 (7/5/2018).

- **PEC4 - Memoria Final**

  Dentro de este cuarto hito se deberá llevar a cabo la redacción de la memoria del trabajo, de tal manera que se deberá sintetizar el trabajo realizado durante el semestre y se deberá mostrar que se han alcanzado los objetivos propuestos. La fecha de inicio de este hito será el 8/5/2018 y la fecha de fin coincidirá con la fecha de entrega de la PEC4 (4/6/2018).

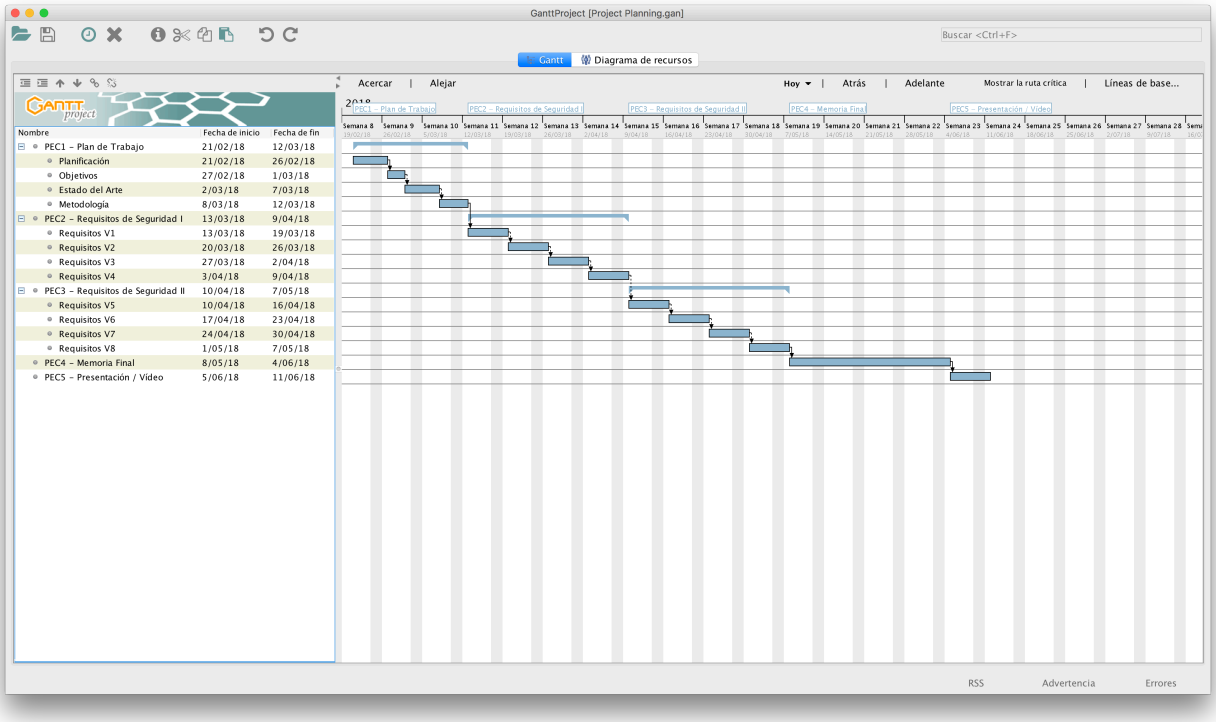
- **PEC5 - Presentación / Vídeo**

  En este último hito se deberá llevar a cabo la edición de un vídeo en el que se deberá presentar una síntesis del trabajo realizado sobre una presentación o guión (con un máximo de 20 páginas). La fecha de inicio de este hito será el 5/6/2018 y la fecha de fin coincidirá con la fecha de entrega de la PEC5 (11/6/2018).

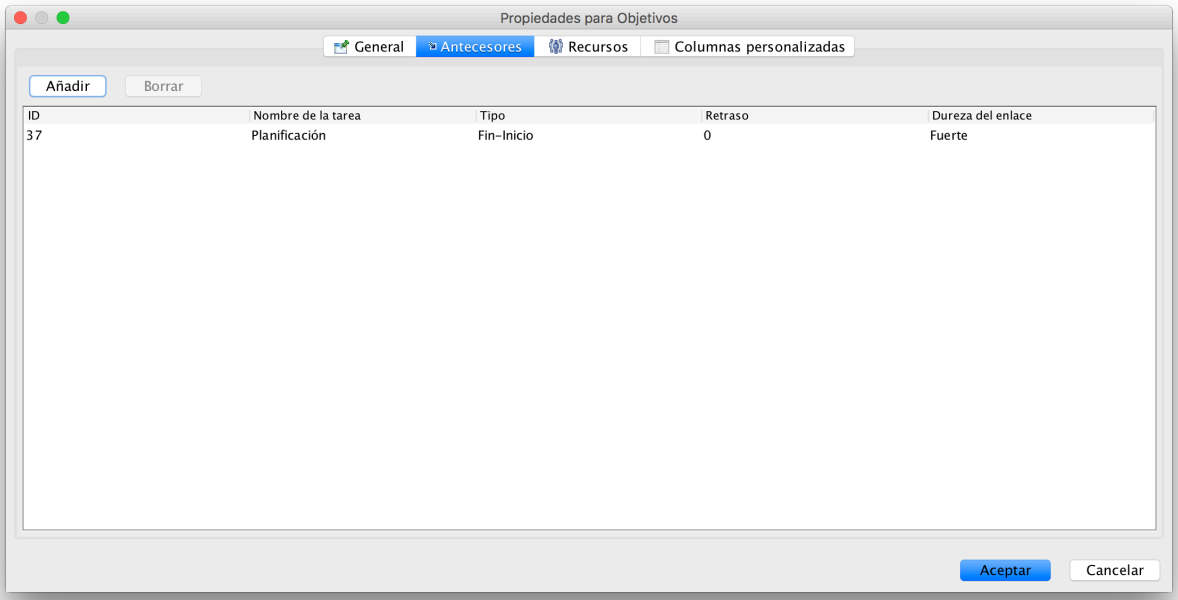
Para llevar a cabo la planificación del proyecto se ha desarrollado un diagrama de Gantt donde se describen cada uno de estos hitos temporales, junto con las tareas englobadas dentro de cada uno de los mismos. Tal y como puede verse en dicho diagrama, existen dependencias temporales entre las tareas definidas en cada uno de los hitos establecidos. Así pues, será necesario ejecutar en orden cronológico cada una de las tareas definidas en la PEC1 antes de pasar a ejecutar las tareas definidas en la PEC2. Lo mismo sucede con las tareas englobadas dentro de las PEC3, PEC4 y PEC5, será necesario ejecutar previamente las tareas de las PEC anteriores antes de pasar a ejecutar las tareas asociadas a una PEC determinada. Por ejemplo, existe una dependencia fuerte entre la finalización del análisis de los requisitos y la redacción de la memoria

o la edición del vídeo / presentación. También existe una dependencia fuerte entre las tareas definidas en la PEC1 y las tareas definidas en las PEC2 y PEC3. Otro claro ejemplo de esto es la dependencia existente entre la PEC4 y la PEC5, ya que será necesario el haber completado la memoria del trabajo antes de llevar a cabo la edición del vídeo / presentación sobre el trabajo realizado.

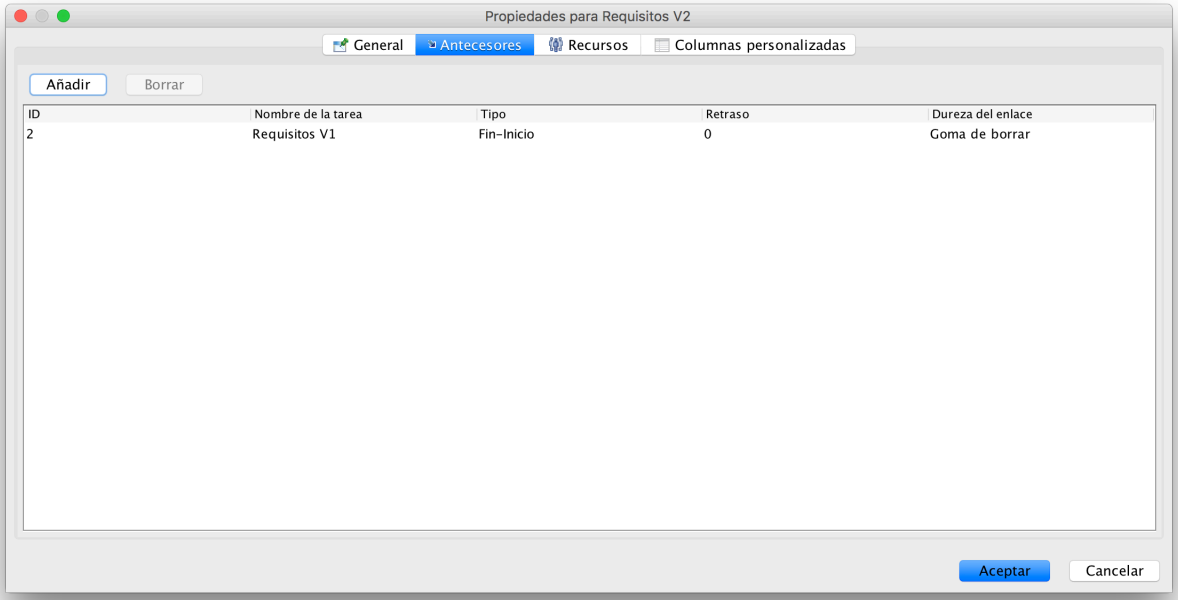
Dentro de la PEC1 también existe una dependencia cronológica entre cada una de las tareas definidas dentro de la misma. Así pues, la primera tarea a completar será la planificación del proyecto, ya que el resultado de la misma será necesario para poder continuar con la ejecución del resto de tareas del trabajo. Una excepción a esta dependencia cronológica entre las tareas es el orden en el que se deberán analizar los requisitos dentro de las PEC2 y PEC3. Si bien es cierto que se ha definido un orden temporal en base a su aparición en el MASVS, también es cierto que el análisis de estos requisitos se podría llevar a cabo sin necesidad de establecer un orden temporal, ya que en principio no existe ninguna dependencia entre los mismos. Así pues, por ejemplo se podría llevar a cabo el análisis de los requisitos englobados en la categoría V2 antes de llevar a cabo el análisis de los requisitos definidos en la categoría V1, y en principio el resultado del trabajo debería ser el mismo que siguiendo el orden cronológico establecido.



Las dependencias fuertes se han definido dentro del diagrama de Gantt a través del uso de enlaces con líneas continuas (e.g. dependencia entre las tareas *Planificación* y *Objetivos*).



Por otro lado, las dependencias débiles se han reflejado a través del uso de enlaces con líneas discontinuas (e.g. dependencia entre las tareas *Requisitos V1* y *Requisitos V2*).



5. Requisitos de Seguridad

En esta sección se llevará a cabo el análisis de los requisitos de seguridad incluidos en cada una de las 8 categorías definidas en el MASVS. A su vez, para cada una de las categorías se expondrán soluciones técnicas y procesos para cumplir con los requisitos incluidos en las mismas. En cada una de estas categorías se establece dentro de que nivel o niveles de verificación están incluidos cada uno de los requisitos analizados (MASVS-L1 y MASVS-L2, MASVS-L2 o MASVS-R).

5.1. V1: Requisitos de Arquitectura, Diseño y Modelado de Amenazas

Esta categoría incluye todos los requisitos relacionados con la arquitectura y diseño de la aplicación. La idea detrás de este conjunto de requisitos es tener en consideración la seguridad en todas las fases del desarrollo del software, no sólo en la fase de desarrollo o implementación. Así pues, el objetivo de los requisitos incluidos dentro de esta categoría es proporcionar las bases para aplicar medidas de seguridad en todas las fases del desarrollo del software, desde la concepción a la puesta en producción del mismo. Dada la naturaleza de este conjunto de requisitos, éstos no tienen asociados directamente soluciones técnicas o casos de test a nivel de código.

MASVS L1 y L2

1.1 Todos los componentes de la aplicación están identificados y son realmente necesarios.

A la hora de evaluar los riesgos y definir los requisitos de seguridad de la aplicación se deberán tener en cuenta todos los componentes de la misma, tanto externos como internos. Estos componentes deberán estar claramente identificados y deberán ser necesarios para el funcionamiento de la misma. La idea de este requisito es poder evaluar correctamente si cada uno de los componentes de la aplicación cumplen con los requisitos de seguridad establecidos para la misma, evitando en la medida de lo posible que ésta integre componentes no requeridos que puedan suponer un riesgo potencial para la seguridad global de la aplicación [2] [10].

1.2 Los controles de seguridad nunca se aplican solo en el lado del cliente, también se deben aplicar en los servicios remotos utilizados por el cliente.

Hoy en día, la gran mayoría de aplicaciones móviles utilizan servicios remotos tanto para obtener la información que se muestra a los usuarios de las mismas como para llevar a cabo operaciones de creación y actualización de datos. Así pues, dentro del ecosistema de las aplicaciones móviles, los controles de seguridad no deberían implementarse sólo a nivel de cliente, sino también en los servicios necesarios para el correcto funcionamiento de la aplicación. Una vulnerabilidad en un servicio remoto o en la base de datos utilizada por el backend podría suponer el comprometer indirectamente la seguridad en los clientes móviles que acceden a estos datos o servicios [2].

1.3 Se ha definido una arquitectura de alto nivel para la aplicación móvil y todos los servicios remotos utilizados. Se ha tenido en cuenta la seguridad en la definición de dicha arquitectura.

Muy en línea con el requisito anterior, a la hora de definir la arquitectura del sistema se debe de tener en cuenta la seguridad en cada uno de los componentes de la misma. Una vulnerabilidad

en cualquiera de estos componentes podría comprometer la seguridad del resto de componentes del mismo, incluyendo los clientes móviles [2].

1.4 Los datos considerados sensibles en el contexto de la aplicación móvil están claramente identificados.

Una parte importante dentro de la definición de los requisitos de negocio de la aplicación es definir los datos que se van a recibir, producir y procesar dentro de la aplicación. Estos datos deberán clasificarse teniendo en cuenta su nivel de confidencialidad, definiendo los riesgos asociados a que un atacante potencial pueda capturar o modificar cualquiera de estos datos. A su vez, deberán establecerse las políticas de copia de seguridad y retención de datos dentro de la aplicación [9].

MASVS-L2

1.5 Todos los componentes de la aplicación se definen en términos de los requisitos de negocio y/o funciones de seguridad que proporcionan.

La idea detrás de este requisito de seguridad es que la inclusión de todo componente dentro de una aplicación debe estar justificado o bien por un requisito de negocio o bien por una función de seguridad dentro de la misma. La principal diferencia con el proceso de desarrollo de software tradicional es que se debe tener en cuenta la seguridad en la definición de cada una de las fases de dicho proceso (i.e. Secure SDLC) [11].

1.6 Se ha definido un modelo de amenazas para la aplicación móvil y los servicios remotos asociados, identificando posibles amenazas y contramedidas.

El modelado de amenazas es un enfoque para analizar la seguridad de una aplicación. Es un enfoque estructurado que le permite identificar, cuantificar y abordar los riesgos de seguridad asociados a una aplicación. La inclusión del modelado de amenazas en el SDLC puede ayudar a garantizar que las aplicaciones se desarrollen con ciertos mecanismos de seguridad incorporados desde el principio. Este proceso se podría dividir en las siguientes fases [10]:

- Fase 1.- Descomponer la aplicación y determinar como interacciona con las entidades externas. Dentro de esta fase se deberán definir los casos de uso, puntos de entrada, niveles de confianza, etc. Para documentar esta primera fase se debería crear un documento de modelo de amenazas que incluya diagramas de flujo (DFDs).

- Fase 2.- En esta segunda fase se deberán determinar y clasificar las amenazas. Lo fundamental para la identificación de amenazas es utilizar una metodología de categorización de amenazas. Se puede usar una categorización de amenazas como STRIDE o Application Security Frame (ASF), de tal manera que se definan categorías de amenazas tales como auditoría y registro, autenticación, autorización, gestión de configuración, protección de datos en almacenamiento y tránsito, validación de datos o gestión de excepciones. El objetivo de la categorización de amenazas es ayudar a identificar las amenazas tanto del atacante (STRIDE) como de la perspectiva defensiva (ASF).

- Fase 3.- En esta tercera fase se deberán determinar las contramedidas y la mitigación a las amenazas identificadas en la fase anterior. La falta de protección contra una amenaza puede indicar una vulnerabilidad cuya exposición al riesgo podría mitigarse con la implementación de una contramedida. Tales contramedidas pueden identificarse utilizando listas de mapeo de amenazas y contramedidas. Una vez que se asigna una clasificación de riesgos a cada una de las amenazas, es posible clasificar las amenazas de mayor a menor riesgo y priorizar el

esfuerzo de mitigación, por ejemplo, respondiendo a dichas amenazas aplicando las contramedidas implementadas.

1.7 Todos los controles de seguridad tienen una implementación centralizada.

Una implementación centralizada de los controles de seguridad desplegados en el sistema permite que los diferentes clientes puedan hacer uso de los mismos, sin tener que re-implementar o duplicar estos controles en cada una de las partes del sistema. La principal ventaja de este tipo de paradigma es la eficiencia, así como una optimización evidente en el mantenimiento del sistema.

1.8 Existe una política explícita sobre la gestión de claves criptográficas (si las hay) y se impone el ciclo de vida de las claves criptográficas. Idealmente, se debería seguir un estándar de gestión de claves como puede ser NIST SP 800-57.

La implementación de políticas para la gestión de claves criptográficas es clave de cara a evitar agujeros de seguridad potenciales en lo que respecta a este aspecto. Estas políticas deberían basarse en estándares como el NIST SPF 800-57 [12].

1.9 Existe un mecanismo para forzar las actualizaciones de la aplicación móvil.

La actualización de las aplicaciones móviles en los dispositivos de los usuarios es clave de cara a asegurar que éstos dispongan de las últimas versiones de las mismas, incluyendo potenciales parches y nuevas medidas de seguridad implementadas en las mismas. El hallazgo de vulnerabilidades en versiones de aplicaciones que están en producción es muy frecuente, de tal manera que es tremendamente importante que los usuarios de la aplicación instalen cuanto antes nuevas versiones de las mismas que incluyan parches y arreglos para éstas. Un sistema de actualización de aplicaciones rápido y efectivo es clave de cara a garantizar la seguridad de los usuarios de la aplicación.

1.10 La seguridad se aborda en todas las partes del ciclo de vida de desarrollo de software (SDLC).

Este requisito se podría considerar como un sumario o resumen del resto de requisitos incluidos en esta categoría. El proceso de incorporar mecanismos y medidas de seguridad en todas las fases del desarrollo de aplicaciones móviles, y no sólo en las últimas fases del mismo, es clave de cara a garantizar la seguridad de los usuarios de las mismas (i.e. Secure SDLC).

5.2. V2: Requisitos de Almacenamiento de Datos y Privacidad

La protección de datos confidenciales, tales como las credenciales de usuario e información confidencial, es un enfoque clave en la seguridad de los dispositivos móviles. En primer lugar, los datos confidenciales pueden exponerse involuntariamente a otras aplicaciones que se ejecutan en el mismo dispositivo si los mecanismos del sistema operativo como Inter-Process Communication (IPC) se utilizan de forma incorrecta. Los datos también pueden filtrarse involuntariamente al almacenamiento en la nube, a las copias de seguridad o a la caché del teclado. Además, los dispositivos móviles se pueden perder o pueden ser robados más fácilmente en comparación con otros tipos de dispositivos, por lo que que un escenario donde un atacante obtenga acceso físico al dispositivo es más que probable. En ese caso, se pueden implementar protecciones adicionales para hacer que la recuperación de los datos confidenciales por parte de un atacante resulte más complicado.

Los datos confidenciales en el contexto de MASVS se refieren tanto a las credenciales del usuario como a cualquier otro dato que se considere confidencial en el contexto particular de la aplicación. Por ejemplo, se podrían considerar datos confidenciales los siguientes:

- Información de identificación personal (PII) que puede ser objeto de abuso por robo de identidad: números de seguridad social, números de tarjetas de crédito, números de cuentas bancarias, información sobre la salud.
- Datos altamente confidenciales que ocasionarían daños a la reputación y/o costos financieros si se comprometiera cualquier de las siguientes: información contractual, información cubierta por acuerdos de confidencialidad, información de gestión.
- Cualquier información que deba ser protegida por ley o por razones de cumplimiento.

MASVS L1 y L2

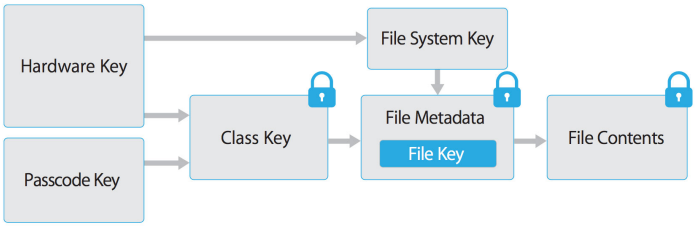
2.1 Los sistemas de almacenamiento de credenciales de los dispositivos se utilizan apropiadamente para almacenar datos confidenciales, tales como credenciales de usuario o claves criptográficas.

Una regla de oro en el desarrollo de aplicaciones móviles es que se debe guardar la menor cantidad posible de datos confidenciales en el almacenamiento local permanente. Sin embargo, en la práctica suele ser necesario almacenar algún tipo de dato confidencial relacionado con el usuario. Por suerte, iOS ofrece APIs de almacenamiento seguras que permiten a los desarrolladores hacer uso del hardware criptográfico disponible en los dispositivos iOS. Suponiendo que estas APIs se usan correctamente, los datos y archivos confidenciales pueden almacenarse de forma segura mediante un cifrado de tipo AES de 256 bits respaldado por hardware.

Tal y como se explica en el MSTG [2], los desarrolladores de aplicaciones pueden aprovechar las API de protección de datos de iOS (Data Protection API) para implementar controles de acceso detallados sobre los datos de usuario almacenados en la memoria flash. Esta API está construida por encima del Secure Enclave Processor (SEP) que se introdujo con el iPhone 5S. El SEP es un co-procesador que proporciona operaciones criptográficas para la protección de datos y la administración de claves. Cada dispositivo dispone de una clave única de hardware (i.e. UID - Unique ID) que está integrada en el secure enclave, lo que garantiza la integridad de la protección de datos incluso si el kernel del sistema operativo se pusiera en peligro.

La arquitectura de protección de datos se basa en una jerarquía de claves. El UID y la clave del código de acceso del usuario, que se deriva de la frase de contraseña del usuario utilizando un algoritmo PBKDF2, se encuentran en la parte superior de esta jerarquía. Juntos, se pueden usar para "desbloquear" las denominadas class keys que están asociadas a los diferentes estados del dispositivo (e.g. el dispositivo está bloqueado / desbloqueado).

Cada archivo almacenado en el sistema de archivos de iOS se cifra con su propia clave individual, la cual está contenida en los metadatos del archivo. Los metadatos se cifran con la clave del sistema de archivos y se envuelven con una de las class keys, dependiendo del tipo de protección seleccionado por la aplicación al crear el archivo.



A cada uno de los archivos creados se le deberá asignar uno de los siguientes tipos de protección, los cuales se explican más en detalle en la iOS Security Guide [5]. Comentar que todos los tipos de protección exceptuando NSFileProtectionNone se cifran con una clave derivada del UID del dispositivo y el código de acceso del usuario. Como resultado, el descifrado solo puede ocurrir en el propio dispositivo y requiere que se ingrese la contraseña correcta. A partir de iOS 7, la clase de protección de datos predeterminada es "Protected Until First User Authentication".

- Complete Protection (NSFileProtectionComplete).- Se utiliza una clave derivada del código de acceso del usuario y el UID del dispositivo para proteger esta class key. Se borrará de la memoria poco después de que el dispositivo esté bloqueado, haciendo que los datos sean inaccesibles hasta que el usuario desbloquee el dispositivo de nuevo.

- Protected Unless Open (NSFileProtectionCompleteUnlessOpen).- Se comporta de forma similar a Complete Protection, pero si el archivo se abre cuando está desbloqueado, la aplicación puede seguir accediendo al archivo incluso si el usuario bloquea el dispositivo. Esta clase de protección se utiliza, por ejemplo, cuando se descarga un archivo adjunto del email en segundo plano.

- Protected Until First User Authentication (NSFileProtectionCompleteUntilFirstUserAuthentication).- Se puede acceder al archivo desde el momento en que el usuario desbloquea el dispositivo por primera vez después de la inicialización del mismo. Se puede acceder incluso si el usuario bloquea el dispositivo posteriormente y la class key no se elimina de la memoria.

- No Protection (NSFileProtectionNone).- La class key para este tipo de protección solo está protegida con el UID. Se almacena en el denominado Effaceable Storage, que es una región de memoria flash en el dispositivo iOS que permite almacenar pequeñas cantidades de datos. Esta clase de protección existe para permitir el borrado remoto rápido a través del borrado de la class key haciendo que los datos sean inaccesibles inmediatamente.

A continuación se describen algunos de los mecanismos estándar que proporciona iOS para el almacenamiento de datos en local.

- The Keychain.- El iOS Keychain se puede utilizar para almacenar de forma segura bits de datos cortos y confidenciales, tales como claves de cifrado y tokens de sesión. Está implementado como una base de datos SQLite a la que solo se puede acceder a través de las APIs específicas del Keychain. En iOS sólo existe un Keychain disponible para todas las aplicaciones, pudiendo compartirse el acceso a los elementos del mismo entre diferentes aplicaciones firmadas por un mismo desarrollador (i.e. a través de la función de access groups a través del atributo kSecAttrAccessGroup). El acceso al Keychain está gestionado por el daemon securityd, el cual otorga acceso en función de los Keychain-access-groups, el application-identifier y los application-group entitlements.

  Comentar que en iOS, cuando se desinstala una aplicación, el dispositivo conserva los datos del Keychain utilizados por la aplicación, a diferencia de los datos almacenados en el sandbox, los cuales se borran. Al no existir una API específica que permita el borrado de estos datos tras la desinstalación de la aplicación, se recomienda el llevar a cabo el borrado de los datos del Keychain tras llevar a cabo la primera ejecución de la aplicación tras una nueva instalación. También se recomienda llevar a cabo el borrado de estos datos tras una operación de logout dentro de la aplicación.

  Los datos almacenados en el Keychain están protegidos a través de una estructura de clase similar a la utilizada para el cifrado de archivos, siendo las operaciones principales las siguientes.

  - SecItemAdd
  - SecItemUpdate
  - SecItemCopyMatching

- SecItemDelete

Los elementos almacenados en el Keychain se codifican como un plist binario y se encriptan utilizando una clave AES de 128 bits por elemento en Galois / Counter Mode (GCM). Se debe tener en cuenta que los blobs de datos no están destinados a ser guardados directamente en el Keychain, para este tipo de datos es más apropiado utilizar la Data Protection API descrita anteriormente. Los valores disponibles para los AccessControlFlags son los siguientes.

- kSecAccessControlDevicePasscode
- kSecAccessControlTouchIDAny
- kSecAccessControlTouchIDCurrentSet
- kSecAccessControlUserPresence.

Los valores de accesibilidad para el atributo kSecAttrAccessible son configurables y representan las clases de protección de datos del Keychain:

- kSecAttrAccessibleAfterFirstUnlock.- No se puede acceder a los datos de un elemento almacenado en el Keychain después de un reinicio hasta que el usuario haya desbloqueado el dispositivo por primera vez.

- kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly.- No se puede acceder a los datos de un elemento almacenado en el Keychain después de un reinicio hasta que el usuario haya desbloqueado el dispositivo por primera vez. Los datos no se incluirán en una copia de seguridad de iCloud o iTunes.

- kSecAttrAccessibleAlways.- Se puede acceder siempre a los datos de un elemento almacenado en el Keychain sin importar si el dispositivo está bloqueado.

- kSecAttrAccessibleAlwaysThisDeviceOnly.- Se puede acceder siempre a los datos de un elemento almacenado en el Keychain sin importar si el dispositivo está bloqueado. Los datos no se incluirán en una copia de seguridad de iCloud o iTunes.

- kSecAttrAccessibleWhenPasscodeSet.- Sólo se puede acceder a los datos de un elemento almacenado en el Keychain cuando el dispositivo está desbloqueado. Esta opción sólo estará disponible si se establece una contraseña en el dispositivo.

- kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly.- Sólo se puede acceder a los datos de un elemento almacenado en el Keychain cuando el dispositivo está desbloqueado. Esta opción sólo estará disponible si se establece una contraseña en el dispositivo. Los datos no se incluirán en una copia de seguridad de iCloud o iTunes.

- kSecAttrAccessibleWhenUnlocked.- Sólo se puede acceder a los datos de un elemento almacenado en el Keychain cuando el dispositivo está desbloqueado.

- kSecAttrAccessibleWhenUnlockedThisDeviceOnly.- Sólo se puede acceder a los datos de un elemento almacenado en el Keychain cuando el dispositivo está desbloqueado. Los datos no se incluirán en una copia de seguridad de iCloud o iTunes.

En el Anexo [1] se muestra un ejemplo de utilización de las APIs del Keychain implementando una utilidad que permite almacenar y acceder a un elemento almacenado en el mismo.

- NSUserDefaults.- La clase NSUserDefaults proporciona una interfaz programática para interactuar con el sistema predeterminado de persistencia de datos. El sistema predeterminado permite que una aplicación personalice su comportamiento para que coincida con las preferencias de un usuario. Los datos almacenados por la clase NSUserDefaults se pueden acceder desde la propia aplicación y a diferencia del Keychain se borran tras llevar la desinstalación de la aplicación.

- File System.- A través del uso de la clase NSFileManager se permite la creación de ficheros de datos dentro del sandbox de la aplicación. Esta clase se suele utilizar en conjunción con otro tipo de APIs disponibles en la clase NSData.

- Core Data.- Este framework permite gestionar la capa de modelo dentro de una aplicación. Proporciona soluciones generalizadas y automatizadas para tareas comunes asociadas con el ciclo de vida de los objetos y la gestión de relaciones entre los mismos, incluida la persistencia. Core Data puede utilizar SQLite como motor de persistencia. ya que el framework en sí no es una base de datos.

- SQLite Databases.- Es el sistema gestor de base de datos relacional más popular que suelen utilizar las aplicaciones en iOS. Está contenido dentro de una libraría en C que está incorporada a la aplicación que lo utiliza. SQLite no consta de un servicio o daemon independiente que se ejecuta en segundo plano de manera adjunta a la aplicación. Por el contrario, la aplicación lo ejecuta como parte integral de la misma. Hoy en día, SQLite dispone de su tercera versión, por lo que también se conoce comúnmente como SQLite 3.

- Realm Databases.- Las librerías Realm Objective-C y Realm Swift no existen por defecto en las APIs de iOS, pero aún así merece la pena mencionarlas aquí. Estas librerías almacenan todos los datos sin encriptar, a menos que la configuración tenga la encriptación habilitada.

- Couchbase Lite Databases.- Es un motor de base de datos embebido ligero, orientado a documentos (NoSQL) y sincronizable. Se puede compilar de forma nativa para iOS y Mac OS.

De cara a llevar un análisis estático del cumplimiento de este requisito se deberá tener acceso al código fuente de la aplicación iOS. Una vez tenga disponible dicho código fuente, se deberá tratar de detectar datos confidenciales que se guardan y procesan en toda la aplicación. Esto incluye contraseñas, claves secretas e información de identificación personal (PII), pero también podría incluir otros datos identificados como sensibles por las regulaciones de la industria, las leyes o las políticas de la compañía. Se deberán buscar instancias donde se guarden estos datos usando cualquiera de las APIs de almacenamiento local que se han enumerado anteriormente. Se deberá verificar de que los datos confidenciales nunca se almacenen sin la protección adecuada. Por ejemplo, los tokens de autenticación no se deben guardar en NSUserDefaults sin un cifrado adicional.

Para llevar a cabo un análisis dinámico se deberá analizar el directorio de datos de la aplicación. Es importante ejecutar toda la funcionalidad de la aplicación antes de llevar a cabo el análisis de los datos, ya que la aplicación podría almacenar datos confidenciales sólo cuando el usuario ejecute alguna funcionalidad específica. Luego, se puede realizar un análisis estático de los datos obtenidos basado en palabras clave genéricas y datos específicos de la aplicación. Existen herramientas como Frida (https://www.frida.re/docs/ios/) o iMazing (https://imazing.com) que pueden ser de gran utilidad para llevar a cabo este tipo de análisis.

2.2 Los datos confidenciales no se deben incluir en los logs de la aplicación.

El uso de logs dentro de las aplicaciones móviles puede ser de tremenda utilidad de cara a mantener por ejemplo un registro de crashes y errores, depuración de aplicaciones, etc. No obstante, hay que tener especial cuidado en no incluir datos confidenciales en este tipo de logs, ya que estos datos podrían quedar expuestos a potenciales atacantes o aplicaciones maliciosas. A la hora de implementar estos logs en aplicaciones para iOS, se pueden utilizar diferentes mecanismos que se enumeran a continuación:

- NSLog
- printf-like function
- NSAssert-like function
- Macros

Para llevar a cabo un análisis estático que verifique el cumplimiento de este requisito, bastará con realizar una búsqueda en el código fuente utilizando algunas palabras claves relacionadas con las funciones de logging. Para llevar a cabo un análisis dinámico bastará conectar el dispositivo iOS al laptop, abrir el Organizador (CMD + MAYÚS + 2) dentro de Xcode, y allí se podrán ver los logs de todas las aplicaciones instaladas en el dispositivo conectado.

De cara a evitar que estos logs aparezcan en aplicaciones desplegadas en producción, pero se mantengan en fases de desarrollo y pruebas, se pueden utilizar mecanismos como el siguiente. La idea es que incluir el siguiente código Objective-C en el PREFIX_HEADER (*.pch) de la aplicación.

```
#ifdef DEBUG
#   define NSLog (...) NSLog(__VA_ARGS__)
#else
#   define NSLog (...)
#endif
```

2.3 No se deben compartir datos confidenciales con terceros a menos que sea una parte necesaria de la arquitectura.

A la hora de integrar servicios proporcionados por terceros dentro de una aplicación móvil existen 2 mecanismos principales:

- Utilizando una librería independiente
- Utilizando un SDK completo

Para verificar que se cumple este requisito se deben revisar las llamadas a los métodos y funciones de las librerías para verificar que no se están enviando datos confidenciales a las mismas. En el caso de que se deban enviar este tipo de datos, primero se deberán anonimizar, de la misma manera que no se deberán enviar identificadores internos de la aplicación (e.g. cuentas de usuario, sesión, etc). A la hora de llevar a cabo un análisis dinámico se deberán analizar las requests enviadas a los servicios externos de estos terceros y comprobar que no se están enviando datos confidenciales en las mismas.

2.4 La memoria caché del teclado deberá estar desactivada en las entradas de texto que procesen datos confidenciales.

Para facilitar la entrada de datos a través del teclado, iOS implementa varios mecanismos tales como la auto-corrección, la revisión ortográfica, etc. La gran mayoría de las entradas de teclado se almacenan en caché de forma predeterminada en el fichero /private/var/mobile/Library/Keyboard/dynamic-text.dat.

El almacenamiento en caché del teclado se logra mediante la implementación del protocolo UITextInputTraits, que es adoptado por algunos controles estándar de iOS como son el UITextField, UITextView y UISearchBar. Las propiedades más importantes son las siguientes:

- autocorrectionType (UITextAutocorrectionType).- Determina si la autocorrección está habilitada o deshabilitada durante la introducción de datos. Con la autocorrección habilitada, el control busca palabras desconocidas y sugiere las palabras más adecuadas para el usuario, reemplazando el texto escrito automáticamente a menos que el usuario anule explícitamente la acción. El valor predeterminado para esta propiedad es UITextAutocorrectionTypeDefault, que para la mayoría de los métodos de entrada da como resultado la autocorrección habilitada.

- secureTextEntry (BOOL).- Identifica si el copiado y el almacenamiento en caché del texto deben estar deshabilitados y en el caso del UITextField se oculta el texto que se ingresa. Esta propiedad tiene un valor predeterminado de NO.

De cara a llevar un análisis estático del cumplimiento de este requisito se deberá buscar manualmente en el código fuente de la aplicación si la autocorrección de campos de texto está deshabilitada (UITextAutocorrectionTypeNo) y en el caso de campos de texto donde se introduzcan contraseñas y códigos PIN se deberá habilitar el campo secureTextEntry.

```
// Disable keyboard caché
textObject.autocorrectionType = UITextAutocorrectionTypeNo;

// Secure text entries like passwords, PIN, etc
textObject.secureTextEntry = YES;
```

2.5 El portapapeles deberá estar desactivado en los campos de texto que pueden contener datos confidenciales.

A la hora de ingresar datos en los campos de entrada, el portapapeles se puede usar para copiar y pegar datos en los mismos. El portapapeles es accesible por todas las aplicaciones del sistema y por lo tanto esta característica puede ser utilizada por aplicaciones maliciosas para obtener datos confidenciales almacenados en el mismo. De cara a evitar este problema, se deberá deshabilitar el portapapeles en campos de texto que puedan contener información confidencial. Por ejemplo, para deshabilitar el portapapeles en un campo de texto se puede extender la clase UITextField tal y como se hace en el siguiente snippet de código desarrollado en Swift.

```
class NoClipboardTextField: UITextField {
    override func canPerformAction(action: Selector, withSender sender:
AnyObject?) -> Bool {
        if action == #selector(NSObject.select(_:)) {
            return false
        }
        if action == #selector(NSObject.selectAll(_:)) {
            return false
        }
        if action == #selector(NSObject.cut(_:)) {
            return false
        }
        if action == #selector(NSObject.copy(_:)) {
            return false
        }
        if action == #selector(NSObject.paste(_:)) {
            return false
        }
        if action == #selector(NSObject.delete(_:)) {
            return false
        }
        return super.canPerformAction(action, withSender: sender)
    }
}
```

2.6 No se deberán exponer datos confidenciales a través de los mecanismos de IPC.

Inter-Process Communication (IPC) es un método que permite a dos procesos diferentes enviarse mensajes y datos entre sí. En caso de que dos procesos necesiten comunicarse entre sí, existen diferentes métodos para implementar IPC en iOS:

- XPC Services.- XPC es una librería de comunicación entre procesos estructurada y asíncrona que proporciona comunicación básica entre procesos y se administra mediante launchd.

- Mach Ports.- Todos los mecanismos de comunicación entre procesos depende al final de la API de Mach Kernel. Mach Ports solo permite la comunicación local (en el mismo dispositivo). Pueden implementarse de forma nativa o implementando wrappers sobre Core Foundation (CFMachPort) y Foundation (NSMachPort).

- NSFileCoordinator.- La clase NSFileCoordinator se puede usar para administrar e intercambiar datos entre aplicaciones a través de archivos que sean accesibles en el sistema de archivos del dispositivo por diferentes procesos.

Para llevar a cabo un análisis estático que permita verificar el cumplimiento de este requisito se deberá examinar el código fuente de la aplicación, de tal manera que se pueda determinar que no se están exponiendo datos confidenciales en la comunicación entre procesos. Para ello se deberán buscar palabras clave dentro del código fuente tales como NSXPCConnection, NSXPCInterface, NSXPCListener, NSXPCListenerEndpoint, mach_port_t, mach_msg_, CFMachPort, CFMessagePort, NSMachPort, NSMessagePort, NSFileCoordinator, etc.

Comentar que para evitar las restricciones propias del sandbox del sistema operativo, será necesario que ambas aplicaciones estén dentro de un mismo application group, y por lo tanto dispongan de los entitlements adecuados.

2.7 No se deberán mostrar datos confidenciales, tales como contraseñas o pins, a través del interfaz de usuario.

El enmascaramiento de datos confidenciales, mostrando asteriscos o puntos en lugar de texto plano, debe aplicarse dentro de los controles de la aplicación que permitan la introducción de datos confidenciales como pueden ser contraseñas o códigos PIN. Tanto para verificar el cumplimiento de este requisito, como para asegurar su cumplimiento se deberá hacer uso de la propiedad secureTextEntry de la clase UITextField (ver requisito 2.4).

MASVS-L2

2.8 No se deberán incluir datos confidenciales en las copias de seguridad generadas por el sistema operativo del dispositivo.

iOS ofrece funcionalidades de copia de seguridad automática que crean copias de los datos del dispositivo. En iOS, las copias de seguridad pueden realizarse a través de iTunes o la nube utilizando la función de copia de seguridad de iCloud. En ambos casos, la copia de seguridad incluye casi todos los datos almacenados en el dispositivo, exceptuando algunos tipos de datos altamente confidenciales como son la información de Apple Pay y la configuración de TouchID o FaceID.

Dado que iOS hace una copia de seguridad de las aplicaciones instaladas y sus datos, una preocupación obvia es si los datos confidenciales del usuario almacenados por la aplicación pueden filtrarse involuntariamente a través de dichas copias de seguridad. La respuesta a esta pregunta es "sí", pero solo si la aplicación almacena datos confidenciales de manera insegura.

Cuando un usuario realiza una copia de seguridad de su dispositivo iOS, también se realiza una copia de seguridad de los datos del Keychain, pero los datos almacenados en el mismo permanecen encriptados. Las class keys necesarias para descifrar los datos del Keychain no están incluidas en la copia de seguridad. Para restaurar los datos del Keychain, la copia de seguridad debe restaurarse en un dispositivo que deberá ser desbloqueado con el mismo código de acceso que el dispositivo original.

Los elementos del Keychain almacenados con restricciones para poder ser leídos sólo desde el dispositivo original (kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly) se pueden descifrar si la copia de seguridad se restaura en el mismo dispositivo. Un atacante que trate de extraer los datos del Keychain del la copia de seguridad no podría descifrarlos sin tener acceso al hardware criptográfico dentro del dispositivo de origen.

Así pues, siempre que los datos confidenciales se gestionen como se recomienda anteriormente en este capítulo (almacenados en el Keychain o encriptados con una clave almacenada dentro del Keychain), los backups no deberían suponer un problema de seguridad.

Por defecto, las copias de seguridad incluyen todos los ficheros y subdirectorios dentro del directorio privado de la aplicación (exceptuando el directorio Library/Caches). De cara a evitar que se incluyan ciertos ficheros dentro de las copias de seguridad se deberá asignar la siguiente propiedad a cada uno de los ficheros que se deseen excluir (NSURLIsExcludedFromBackupKey). A continuación se muestra un snippet de código desarrollado en Swift que permite excluir un fichero determinado de una copia de seguridad realizada en el dispositivo.

```
func addSkipBackupAttributeToItemAtURL(URL:NSURL) -> Bool {
    var success: Bool

    do {
        try URL.setResourceValue(true,
forKey:URLResourceKey.isExcludedFromBackupKey)
        success = true
    } catch let error as NSError {
        success = false
        print("Error excluding \(URL.lastPathComponent!) from backup \
(error)");
    }
    return success
}
```

De cara a verificar el cumplimiento de este requisito se deberá comprobar que no se han almacenado datos confidenciales tras llevar a cabo una copia de seguridad de la aplicación. Se deberán revisar específicamente los siguientes directorios para comprobar que no contienen datos confidenciales Documents/, Library/Caches/, Library/Application Support/ y tmp/.

2.9 La aplicación no deberá mostrar datos confidenciales de las vistas cuando esta esté en segundo plano.

Los fabricantes de dispositivos móviles proporcionan a los usuarios un efecto estéticamente agradable cuando se ingresa o sale una aplicación. Para ello introdujeron una técnica que consiste en guardar una captura de pantalla cuando la aplicación pasa al segundo plano o background, de tal manera que se muestre dicha captura cuando la aplicación vuelve al primer plano o foreground. Este mecanismo podría suponer un riesgo de seguridad para una aplicación, ya que esta captura de pantalla podría contener información sensible (e.g. una captura de

pantalla de una aplicación bancaria), y puesto que se guarda en el almacenamiento local del dispositivo, cualquier podría accederse a su contenido si obtuviese acceso físico al mismo.

Para verificar el cumplimiento de este requisito se deberá revisar que no existen datos confidenciales en las capturas de pantalla almacenadas en el siguiente directorio.

/var/mobile/Containers/Data/Application/$APP_ID/Library/Caches/Snapshots/

De cara a remediar este problema, se podría o bien mostrar una imagen por defecto cuando la aplicación se envía al background o bien añadir un efecto que distorsione el contenido de la pantalla antes de enviar la aplicación al background (e.g. blurring). A continuación se muestra un ejemplo en Swift de la primera técnica descrita.

```
func applicationDidEnterBackground(_ application: UIApplication) {
    let imageView = UIImageView(frame: self.window!.bounds)
    imageView.tag = 101
    imageView.backgroundColor = UIColor.white
    imageView.contentMode = .center
    imageView.image = UIImage(named:"overlayImage.png")
    window.addSubview(imageView)
}

func applicationWillEnterForeground(_ application: UIApplication) {
    if let imageView = window.viewWithTag(101) as? UIImageView {
        imageView.removeFromSuperview()
    }
}
```

2.10 La aplicación no deberá contener datos confidenciales en la memoria más tiempo del necesario, y la memoria se deberá borrar explícitamente después de su uso.

Analizar la memoria de una aplicación es una técnica muy común durante el proceso de depuración de aplicaciones, ya que puede ayudar a solucionar problemas tales como crashes, memory leaks, etc. De la misma manera que este análisis puede ser tremendamente útil, también puede ser peligroso ya que puede facilitar el acceso a datos confidenciales almacenados en la misma..

Para verificar que la información confidencial se exponga lo más brevemente posible en la memoria, lo primero que se deberá hacer es identificar qué información confidencial se almacena en la misma. Básicamente, si la aplicación gestiona datos confidenciales, es muy probable que en algún momento se carguen en la memoria.

Para poder analizar la memoria de una aplicación, se podrá o bien llevar a cabo un volcado de la misma o bien analizarla en tiempo real a través de un depurador o debugger. No importa el enfoque, este es un proceso muy propenso a errores desde el punto de vista de la verificación, ya que lo que se podrá obtener en un volcado concreto son los datos resultantes de las funciones que se ejecutaron, siendo muy posible que se deje de analizar la ejecución de escenarios críticos. Además, a menos que se tenga una cierta idea del rastro de los datos que se están buscando (ya sea el valor exacto o su formato), es bastante fácil no poder identificarlos durante el análisis.

Tal y como se expone en el MSTG [2], de cara a evitar una mínima exposición de datos confidenciales en la memoria de la aplicación, debe asegurarse que la aplicación esté diseñada de tal manera que la exposición de estos datos sea lo más breve posible, con la menor cantidad de copias posible. En otras palabras, se debe implementar un manejo centralizado de datos confidenciales (con el menor número posible de componentes), basado en estructuras de datos

primitivas y mutables. Ejemplos de tipos de datos recomendables incluirían char[] e int[], pero no NSString y String. Cada vez que se intenta modificar un objeto inmutable como String, en realidad se está creando una copia y aplicando el cambio sobre la misma.

Las colecciones Swift (Array, Set y Dictionary) podrían ser aceptables siempre que contengan tipos de datos primitivos como char o int y se definan como mutables (variables en lugar de constantes), en cuyo caso son más o menos equivalente a una matriz primitiva (como char []). Aún así, estas colecciones proporcionan una gestión de memoria segura, que puede dar como resultado copias no identificadas de los datos confidenciales en la memoria, en caso de que la colección necesite copiar el buffer subyacente en una ubicación diferente para poder extenderlo.

El uso de tipos de datos mutables de Objective-C, como NSMutableString también puede ser aceptable, pero adolecen del mismo problema de "memoria segura" que las colecciones de Swift. Además, se debe prestar atención cuando se usan colecciones de Objective-C ya que contienen datos por referencia y solo se permiten los tipos de datos de Objective-C. Por lo tanto, no se está buscando una colección mutable, sino una colección que haga referencia a objetos mutables.

Desafortunadamente, no muchas librerías y frameworks están diseñados para permitir la sobre-escritura de datos confidenciales. Incluso Apple no considera esto en la API oficial del iOS SDK. Por ejemplo, la mayoría de las API para la transformación de datos (parsers, serializaciones, etc.) operan con tipos de datos no primitivos. Del mismo modo, independientemente de si se define un UITextField como secureTextEntry o no, siempre devuelve los datos como String o NSString.

En resumen, al realizar un análisis estático de datos confidenciales expuestos en la memoria, se deberían seguir las siguientes recomendaciones:

- Intente identificar los componentes de la aplicación e identifique dónde se utilizan algunos de estos datos confidenciales en particular.
- Verifique que los datos confidenciales se manejen en el menor número de componentes posible.
- Verifique que las referencias a los objetos se eliminen correctamente una vez que ya no se necesite el objeto que contiene datos confidenciales.
- Para información altamente confidencial, verifique que los datos se sobreescriban tan pronto como ya no se necesiten.
    - Estos datos no se deben pasar sobre tipos de datos inmutables como String o NSString.
    - Los tipos de datos no primitivos pueden dejar rastros en memoria y por lo tanto deben evitarse en la medida de lo posible.
    - La sobreescritura debe hacerse antes de eliminar las referencias.
    - Se debe prestar atención a los componentes de terceros (librerías y frameworks). Un buen indicador de si éstos cumplen con este requisito es que su API pública maneje los datos confidenciales de acuerdo con las recomendaciones anteriores.

2.11 La aplicación aplica una política de seguridad mínima de acceso al dispositivo, como por ejemplo requerir al usuario establecer un código para acceder al dispositivo.

Cuando la aplicación gestione datos altamente confidenciales es muy recomendable que se requiera al usuario establecer un código de bloqueo en el dispositivo que en muchos casos puede ir acompañado de otros mecanismos de autenticación local como pueden ser algunos mecanismos de carácter biométrico (e.g. TouchID, FaceID, etc).

2.12 La aplicación educa al usuario sobre los tipos de información de identificación personal procesadas, así como sobre buenas prácticas de seguridad que el usuario debe seguir al utilizar la aplicación.

Para cumplir con este requisito de seguridad bastaría con incluir algún tipo de on-boarding, ayuda o manual de uso interactivo que permita a los usuarios de la aplicación conocer que tipos de datos confidenciales se están gestionando dentro de la misma, y sobre todo como podrían estos usuarios hacer un uso correcto de las funcionalidades que proporciona la aplicación de cara a gestionar estos datos de forma segura. Conocer los riesgos de seguridad de la aplicación, las amenazas más comunes y la manera en la que se propagan las mismas es el primer paso para poder evitar este tipo de amenazas. Gran parte de este tipo de amenazas se pueden englobar dentro de las denominadas como técnicas de Ingeniería Social.

5.3. V3: Requisitos de Criptografía

La criptografía es una parte fundamental cuando se trata de proteger los datos almacenados en un dispositivo móvil. El objetivo de los controles de esta categoría es garantizar que la aplicación utilice la criptografía de acuerdo con las buenas prácticas recomendadas por la industria, incluyendo las siguientes:

- Uso de librerías de criptografía contrastadas
- Elección y configuración adecuada de primitivas criptográficas
- Utilizar generadores de números aleatorios adecuados donde se requiera aleatoriedad

MASVS L1 y L2

3.1 La aplicación no utiliza criptografía simétrica con claves hardcodeadas como único método de encriptación.

La seguridad de la encriptación simétrica y los algoritmos de hash (MAC) depende de que la clave utilizada se mantenga en secreto. Si se revela la clave, la seguridad obtenida por la encriptación se pierde. Para evitar esto, nunca se deben almacenar claves secretas en el mismo lugar que los datos encriptados por dicha clave. Los desarrolladores a menudo cometen el error de encriptar los datos almacenados localmente con una clave de encriptación fija y hardcodeada en la propia aplicación. Aunque no es recomendable, si se decidiese incluir una clave fija dentro de la propia aplicación se debería ofuscar utilizando alguna de las técnicas que se enumeran a continuación:

- Estenografía.- Esta técnica consiste en almacenar la clave dentro de ficheros contenedores incluidos dentro de los recursos de la aplicación (e.g. dentro de una imagen). La idea sería el dificultar el acceso a la misma por parte de un atacante potencial que tuviese acceso a la propia aplicación.

- Binary XOR.- La idea de esta técnica es generar la clave dinámicamente a partir de 2 arrays de bytes almacenados en el código fuente de la aplicación. Estos arrays se podrían dividir en varios subarrays, para posteriormente volver a concatenarse durante el proceso de generación de la clave. Para generar estos arrays de bytes se deberá definir previamente un array de bytes con la misma longitud que la clave, de tal manera que se pueda llevar a cabo la generación del array de bytes ofuscado mediante la aplicación de operaciones XOR sobre cada uno de los bytes de ambos arrays.

Otra manera de gestionar este tipo de claves para llevar a cabo este tipo de encriptación simétrica consistiría en generar dinámicamente la clave a partir de algoritmos de derivación de

claves como puede ser Diffie-Hellman o utilizar otro tipo de técnicas más complejas como puede ser la denominada White-box Cryptography.

3.2 La aplicación utiliza implementaciones probadas de primitivas criptográficas.

El inventar funciones criptográficas propietarias es complicado, costoso en términos de tiempo y además es probable que tenga un resultado fallido. En contraposición con esto, se recomienda utilizar algoritmos contrastados y probados que son ampliamente considerados como seguros. Los sistemas operativos móviles ofrecen APIs criptográficas estándar que implementan este tipo de algoritmos.

Se deberán inspeccionar cuidadosamente todos los métodos criptográficos utilizados en el código fuente, especialmente aquellos que se aplican directamente sobre datos confidenciales. Todas las operaciones criptográficas deben usar APIs criptográficas estándar para iOS, de tal manera que cualquier operación criptográfica que no invoque rutinas estándar de proveedores conocidos debe ser inspeccionada en detalle. A su vez, se deberá prestar especial atención a los algoritmos estándar que han sido modificados.

3.3 La aplicación utiliza primitivas criptográficas que son apropiadas para el caso de uso particular, configuradas con parámetros que cumplen con las buenas prácticas de la industria.

iOS proporciona librerías que contienen implementaciones de los algoritmos criptográficos más comúnmente utilizados. Un punto de referencia inicial para encontrar información sobre estas librerías es la Apple's Cryptographic Services Guide [13], la cual contiene una amplia documentación sobre cómo utilizar librerías estándar para inicializar y usar primitivas criptográficas, lo cual es tremendamente útil cuando se va a llevar a cabo un análisis del código fuente. Dentro de esta guía se puede encontrar información sobre como utilizar diferentes primitivas criptográficas siguiendo las recomendaciones y buenas prácticas del fabricante y de la industria

A continuación se muestra una lista de algoritmos recomendados para diferentes casos de uso que se pueden dar durante el desarrollo seguro de una aplicación móvil.

- Algoritmos de Confidencialidad: AES-GCM-256 or ChaCha20-Poly1305
- Algoritmos de Integridad: SHA-256, SHA-384, SHA-512, Blake2
- Algoritmos de Firma Digital: RSA (3072 bits and higher), ECDSA with NIST P-384
- Algoritmos de Generación de Claves: RSA (3072 bits and higher), DH (3072 bits or higher), ECDH with NIST P-384

En los Anexos [2] y [3] se muestran una serie de utilidades que permiten llevar a cabo la generación de hashes o resúmenes utilizando para ello algunos de los algoritmos más populares, así como llevar a cabo operaciones de encriptación y desencriptación simétrica a través de AES. Estas utilidades se han implementado utilizando la librería Common Crypto incluida dentro de las APIs estándar de iOS.

3.4 La aplicación no utiliza protocolos criptográficos o algoritmos deprecados por considerarse inseguros.

Como es lógico, es uso de protocolos criptográficos o algoritmos deprecados por considerarse inseguros puede propiciar la generación de vulnerabilidades dentro de la aplicación. A la hora de

elegir estos protocolos y/o algoritmos se deberán utilizar los recomendados por la industria para el caso de uso en cuestión (ver requisito 3.3).

A continuación se muestra una lista con algunos de estos protocolos criptográficos y algoritmos considerados débiles o inseguros.

- DES, 3DES
- RC2
- RC4
- BLOWFISH
- MD4
- MD5
- SHA1

Para verificar que no se estén utilizando este tipo de algoritmos dentro de la aplicación bastará con buscar por estas palabras clave dentro del fichero CommonCryptor.h, y buscar las constantes que hagan referencia a estos algoritmos. Posteriormente se deberán buscar estas constantes dentro del código fuente de la aplicación, de tal manera que si aparecen en llamadas a funciones dentro de las APIs criptográficas (e.g. CCCrypt, CCCryptorCreate, etc) se podrá afirmar con un alto grado de probabilidad que no se está cumpliendo con este requisito.

3.5 La aplicación no reutiliza la misma clave criptográfica para múltiples propósitos.

El principio de separación de claves para los diferentes modos de operación del cifrado de bloques establece que siempre se deben usar claves distintas para algoritmos distintos y modos de operación diferentes. Si se viola este principio, existen ataques genéricos que pueden recuperar una parte o la totalidad de los mensajes cifrados [14].

3.6 Todos los valores aleatorios se generan utilizando un generador de números aleatorios suficientemente seguro.

Apple proporciona a los desarrolladores una API denominada Randomization Services que genera números aleatorios criptográficamente seguros. Esta API utiliza la función SecRandomCopyBytes para realizar la generación de números aleatorios. Esta API se puede considerar un wrapper del archivo del dispositivo /dev/random, el cual proporciona un valor pseudoaleatorio criptográficamente seguro de 0 a 255 llevando a cabo concatenación.

Para asegurar el cumplimiento de este requisito, se deberá verificar que todos los números aleatorios utilizados en la aplicación se generen a través de esta API, ya que no hay ninguna razón por la cual los desarrolladores deban usar otra API diferente. A continuación se muestra la definición de la función SecRandomCopyBytes en Objective-C,

```
int SecRandomCopyBytes(SecRandomRef rnd, size_t count, uint8_t*bytes);
```

así como un ejemplo de uso de la misma.

```
int result = SecRandomCopyBytes(kSecRandomDefault, 16, randomBytes);
```

5.4. V4: Requisitos de Autenticación y Gestión de Sesiones

La gran mayoría de las aplicaciones móviles implementan algún tipo de autenticación de usuario. A pesar de que parte de la autenticación y la lógica de gestión de sesiones se lleva a cabo por los servicios remotos, la autenticación es una parte tan integral de la mayoría de las arquitecturas de aplicaciones móviles que la comprensión de sus implementaciones más comunes es de tremenda importancia. Esta categoría engloba todos los requisitos de autenticación tanto a nivel de cliente como a nivel de servidor.

Al revisar la arquitectura de autenticación de una aplicación, primero debe considerarse si los métodos de autenticación utilizados son apropiados en el contexto dado. La autenticación puede basarse en uno o varios de los siguientes elementos:

- Algún tipo de información que conoce el usuario (contraseña, PIN, patrón, etc.)
- Algo que tiene el usuario (tarjeta SIM, generador de contraseñas de un solo uso o token de hardware)
- Una característica biométrica del usuario (huella digital, retina, voz)

La cantidad de procedimientos de autenticación implementados por las aplicaciones móviles depende de la sensibilidad de las funciones o los recursos a los que se accede. Se deberán consultar los manuales de buenas prácticas de la industria para revisar las funciones de autenticación. La autenticación a través de nombre de usuario / contraseña (combinada con una política de contraseñas razonable) generalmente se considera suficiente para las aplicaciones que requieren un inicio de sesión de usuario y no son muy sensibles. Este tipo de autenticación es utilizada por la mayoría de las aplicaciones de social media. Para aplicaciones sensibles, agregar un segundo factor de autenticación suele ser apropiado. Esto incluye aplicaciones que proporcionan acceso a información muy confidencial (como números de tarjetas de crédito) o permiten a los usuarios transferir fondos bancarios.

MASVS L1 y L2

4.1 Si la aplicación proporciona a los usuarios acceso a un servicio remoto, se deberá llevar a cabo algún tipo de autenticación en el mismos.

Siempre que se exponga alguna funcionalidad a través de un servicio remoto se deberá implementar algún tipo de autenticación en los endpoints incluidos en dichos servicios. El mecanismo de autenticación a implementar dependerá del nivel de seguridad que sea necesario dentro del sistema. Este tipo de autenticación puede ir desde algo tan simple como el envío de las credenciales de usuario en cada una de las requests (e.g. nombre de usuario y contraseña), hasta la implementación de sistemas de autenticación de varios factores a través de SMS, llamadas telefónicas, tokens generados mediante hardware o software, etc.

4.2 Si utiliza mecanismos de gestión de sesión con estado, el servicio remoto deberá utilizar identificadores de sesión generados aleatoriamente para autenticar las requests del cliente sin necesidad de enviar las credenciales del usuario.

Cuando se implementan mecanismos de gestión de sesión con estado (i.e. stateful), durante el proceso de inicio de sesión o login se deberá generar un identificador de sesión único y aleatorio, el cual se devolverá al usuario tras llevar a cabo dicho inicio de sesión. Este identificador de sesión se deberá almacenar tanto en el cliente como en el servidor, y se deberá enviar en cada una de las requests posteriores en lugar de los credenciales de usuario. A su vez, dentro del servidor se deberá gestionar el estado de la sesión, estableciendo un mapeo entre este identificar de sesión y el usuario en cuestión, y sirviendo como referencia para obtener la información del

usuario almacenada en dicho servidor. El identificador de sesión debe de ser opaco y no debe contener ningún tipo de información sobre el usuario. Comentar que las aplicaciones web suelen utilizar este tipo de mecanismos de autenticación a través de un identificador de sesión generado aleatoriamente y que es almacenado en una cookie dentro del navegador (i.e. Session Cookie).

De cara a verificar el cumplimiento de este requisito se deberán localizar los endpoints que devuelvan información confidencial o lleven a cabo operaciones confidenciales, verificando la aplicación coherente de la autorización requerida. El servicio remoto debe verificar el identificador de sesión y asegurarse de que el usuario tenga los privilegios suficientes para acceder al recurso solicitado. Si el identificador de la sesión no se envía o no es válido, la request debe rechazarse.

4.3 Si se utiliza autenticación basada en tokens sin estado, el servidor deberá proporcionar un token que se ha firmado utilizando un algoritmo seguro.

Tal y como se explica en el MSTG [2], a la hora de implementar mecanismos de autenticación sin estado (i.e. stateless), toda la información de autenticación del usuario se almacena en un token en el lado del cliente. Este token se deberá pasar al servicio remoto en cada una de las requests, eliminando la necesidad de gestionar el estado de las sesiones en el servidor. La autenticación sin estado se suele gestionar a través de un servidor de autorización, el cual que genera, firma y opcionalmente cifra el token al producirse el inicio de sesión del usuario. Este tipo de autenticación suele ser el más utilizado por las aplicaciones móviles debido principalmente a las siguientes razones:

- Se mejora la escalabilidad y rendimiento del sistema debido a la no necesidad de gestionar el estado de las sesiones en el servidor.

- Este mecanismo basado en tokens permite a los desarrolladores desacoplar el proceso de autenticación dentro del sistema. Estos tokens pueden ser generados por un servidor de autenticación diferente al servidor en el que están desplegados los servicios remotos, de tal manera que el esquema de autenticación puede ser modificado fácilmente sin afectar al funcionamiento general de endpoints desplegados.

El formato más utilizado para este tipo de tokens de sesión es el JSON Web Token (JWT), el cual está definido en la siguiente URL https://tools.ietf.org/html/rfc7519. Este tipo de token JWT consiste en 3 partes codificadas en Base64 y separadas por puntos, y en muchos casos podría incluir el estado completo de la sesión como un objeto de tipo JSON. Así pues, el servidor no tendrá la necesidad de almacenar ningún tipo de información de la sesión o información sobre el proceso de autenticación. A continuación se muestra un ejemplo de este tipo de token codificado en Base64 extraído de la siguiente URL https://jwt.io/introduction/.


```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpv
aG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```


El encabezado generalmente consta de dos partes: el tipo de token, que es JWT, y el algoritmo de hash que se utiliza para calcular la firma. En el ejemplo anterior, el encabezado se descodificaría de la siguiente manera.


```
{"alg":"HS256","typ":"JWT"}
```


La segunda parte del token es el palyload, la cual contiene los llamados “claims”. Estos claims son declaraciones sobre una entidad (generalmente el usuario) y metadatos adicionales.

```
{"sub":"1234567890","name":"John Doe","admin":true}
```

La firma se crea aplicando el algoritmo especificado en el encabezado JWT al encabezado codificado, el payload codificado y un secreto. Por ejemplo, cuando se usa el algoritmo HMAC SHA256, la firma se podría crear de la siguiente manera.

```
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
```

Hay que tener en cuenta que el secreto se comparte entre el servidor de autenticación y el servicio remoto, de tal manera que el cliente no debería conocerlo. Esto demuestra que el token se obtuvo de un servicio de autenticación legítimo y evita que el cliente pueda alterar los claims contenidos en el token.

De cara a verificar el cumplimiento de este requisito se deberán comprobar los siguientes puntos.

- Se debería identificar la librería JWT que usan el cliente y el servidor, y asegurarse que las mismas no tienen vulnerabilidades conocidas.
- Se deberá verificar que el HMAC se comprueba en todas las requests entrantes que contengan un token.
- Se deberá verificar la localización de la clave de firma privada o la clave secreta HMAC. La clave debe permanecer en el servidor y nunca debe compartirse con el cliente. Debería estar disponible solo para el emisor y para el verificador.
- Se deberá verificar que no haya datos confidenciales incrustados en el JWT (e.g. información de identificación personal). Si por alguna razón la arquitectura requiriese la transmisión de dicha información en el token, se deberá encriptar el payload.
- Se deberán implementar mecanismos contra ataques de repetición a través de la inclusión en el claim del jti (JWT ID), de tal manera que se asigne un identificador único a JWT.
- Se deberá verificar que los tokens están almacenados de forma segura en el cliente móvil (e.g. Keychain).

También es recomendable el verificar el uso de algoritmos de encriptación simétricos en la encriptación del palyload del token cuando sea necesario, verificar que no se puede crackear la firma del token a través de ataques de fuerza bruta, así como seguir las buenas prácticas expuestas en el OWASP JWT Cheat Sheet [15].

En el Anexo [4] se muestran dos implementaciones, una de ellas en Objective-C y la otra en Swift, para codificar y decodificar este tipo de tokens dentro de una aplicación iOS.

4.4 El servicio remoto deberá finalizar la sesión existente cuando el usuario lleve a cabo una operación de logout.

Minimizar el tiempo de vida de los identificadores de sesión y tokens disminuye la probabilidad de ser víctimas de un ataque de tipo “session hijacking” (i.e. secuestro de sesión). El objetivo de este requisito es verificar la funcionalidad de cierre de sesión dentro del aplicación y determinar si la sesión se termina correctamente tanto en el cliente como en el servidor (i.e. stateful authentication) o se invalide el token de acceso (i.e. stateless authentication).

El no terminar la sesión en el lado del servidor es uno de los errores de implementación más comunes en este tipo de mecanismos de seguridad. Este tipo de error permite que se mantenga

abierta la sesión o token incluso después de que el usuario cierre la sesión en la aplicación. Un atacante que pueda obtener información de autenticación válida puede continuar usándola y llevar a cabo un ataque de tipo “session hijacking”. Esto podría resultar en un perdida de datos, comprometer la confidencialidad del usuario y propiciar que el atacante lleve a cabo acciones no legítimas.

Muchas aplicaciones móviles no cierran automáticamente la sesión de los usuarios porque es un inconveniente para los clientes el implementar sistemas de autenticación sin estado. Aun así, la aplicación debería incluir una función de cierre de sesión, la cual deberá implementarse de acuerdo con las mejores prácticas, destruyendo el token de acceso y llevando a cabo un proceso de actualización en el cliente y en el servidor. De no implementarse este tipo de mecanismo de cierre de sesión, un atacante potencial podría saltarse el proceso de autenticación si consiguiera hacerse con el token.

De cara a verificar el correcto cumplimiento de este requisito de seguridad, debería llevarse a cabo un análisis dinámico de la aplicación que incluya los siguientes pasos.

1. Iniciar sesión en la aplicación.

2. Llevar a cabo un par de operaciones que requieran autenticación dentro de la aplicación.

3. Cerrar la sesión en la aplicación.

4. Repetir alguna de las operaciones del paso 2 a través de un proxy de interceptación (e.g. Burp Repeater). De esta manera se enviará al servidor una request con el identificador de sesión o token que se invalidó en el paso 3. Si el cierre de sesión está implementado correctamente en el servidor, se debería mostrar un mensaje de error en la aplicación. Por otro lado, si se recibe la misma respuesta que en el paso 2, el token o el identificador de la sesión todavía serán válidos y por lo tanto se podrá inferir que la sesión no se han terminado correctamente en el servidor.

4.5 Existe una política de contraseñas y se aplica en el servicio remoto.

Una política de contraseñas define los requisitos de las contraseñas a los cuales deben adherirse los usuarios de la aplicación. Estas políticas de contraseñas generalmente especifican la longitud, la complejidad y las topologías de las contraseñas. Una política de contraseñas "fuerte" hace que el cracking manual o automático de una contraseña resulte muy difícil o prácticamente imposible.

A continuación se enumeran una serie de requisitos que deben de cumplir este tipo de contraseñas “fuertes” en lo que respecta a su longitud.

- Las contraseñas deben de tener una longitud mínima de 10 caracteres.

- Las contraseñas no deben tener una longitud máxima demasiado corta para permitir que los usuarios puedan utilizar passphrases. Una longitud máxima típica podría ser 128 caracteres.

Por otro lado, estas contraseñas deberán cumplir al menos 3 de las 4 reglas de complejidad enumeradas a continuación.

- Las contraseñas deben incluir al menos una letra en mayúsculas (A-Z).

- Las contraseñas deben incluir al menos una letra en minúsculas (a-z).

- Las contraseñas deben incluir al menos un número (0-9).

- Las contraseñas deben incluir al menos un carácter especial.

A la hora de realizar un análisis estático de este requisito se deberán identificar todas las funciones relacionadas con las contraseñas en el código fuente y asegurarse de que se realiza una verificación sobre el cumplimiento de las políticas en cada una de ellas. A su vez, se deberá verificar la función de verificación de contraseñas y asegurarse de que no se acepten las contraseñas que violen las políticas establecidas.

4.6 El servicio remoto implementa algún tipo de mecanismo de seguridad para protegerse contra el envío de credenciales incorrectos una cantidad excesiva de veces.

De cara a evitar ataques por fuerza bruta contra el proceso de autenticación, se deberá incluir un contador de inicios de sesión fallidos para un usuario determinado. Tras alcanzar este número máximo de intentos, se deberá bloquear la cuenta de usuario o bien de forma temporal o de forma permanente. Si se lleva a cabo un inicio de sesión correcto se deberá restablecer este contador de inicios sesión fallidos. A continuación se enumera una lista de buenas prácticas a la hora de implementar controles para evitar este tipo de ataques.

- Tras llevarse a cabo un número determinado de intentos de inicio de sesión fallidos, la cuenta de usuario deberá bloquearse de forma temporal o permanente, de tal manera que se rechacen inicios de sesión posteriores para esa misma cuenta.

- Una práctica común es establecer un bloqueo temporal de 5 minutos para las cuentas de usuario que superen el número máximo de inicios de sesión fallidos.

- Este tipo de controles deben de implementarse en el lado del servidor, ya que los controles en el lado del cliente se suelen poder omitir fácilmente.

- Los intentos de sesión fallidos se deben contar con respecto a la cuenta de usuario, no con respecto a una sesión en particular.

Finalmente, comentar que en el documento Blocking Brute Force Attacks [16] se describen otras técnicas adicionales para implementar medidas de seguridad que mitiguen o prevengan este tipo de ataques.

MASVS-L2

4.7 La autenticación biométrica, si la hay, no está unida a un evento (i.e. no usa una API que simplemente devuelve "verdadero" o "falso"). Este tipo de autenticación se debe de basar en desbloquear el Keychain / Keystore.

En el caso de implementar autenticación local biométrica de cara a permitir el acceso a servicios remotos a través de tecnologías como pueden ser TouchID o FaceID, este tipo de autenticación se deberá basar en desbloquear el Keychain para acceder a los datos de sesión o token de acceso almacenados en el mismo. Una vez obtenidos estos datos de sesión o token de acceso, la aplicación podrá acceder nuevamente al servicio remoto.

Para ello se deberá hacer uso del Security.framework, el cual incluye APIs de bajo nivel para acceder a los servicios del Keychain. Este framework proporciona una API en C que permite gestionar el control de acceso a los datos almacenados en el Keychain a nivel del sistema operativo. De esta manera, no resulta nada sencillo el saltarse este tipo de control de acceso para un atacante potencial.

El Keychain permite guardar elementos con el atributo especial SecAccessControl, el cual permitirá el acceso al elemento almacenado en el Keychain solo después de que el usuario haya pasado la autenticación mediante TouchID o FaceID (o en su defecto del código de acceso del dispositivo).

De cara a llevar un análisis del cumplimiento de este requisito, se deben verificar los siguientes aspectos:

- Se deberá verificar que los procesos confidenciales, como volver a autenticar a un usuario que va a llevar a cabo una transacción de pago, estén protegidos utilizando los métodos disponibles en los servicios del Keychain.
- Se deberá verificar que la política kSecAccessControlUserPresence y la clase de protección kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly estén establecidas cuando se llame al método SecAccessControlCreateWithFlags.

En este primer ejemplo desarrollado en Swift se implementa un método para almacenar una contraseña asociada a un nombre de usuario en el Keychain. Sólo se podrá acceder a esta contraseña almacenada cuando el passcode esté activado en el dispositivo y después de que el usuario se haya autenticado satisfactoriamente a través del TouchID.

```
func savePassword(username: String, password: String) {
    let passwordData = password.dataUsingEncoding(NSUTF8StringEncoding)
    guard let data = passwordData else { return }

    let accessControl = SecAccessControlCreateWithFlags(
        kCFAllocatorDefault,
        kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly,
        .UserPresence // this part enables TouchID support
        , nil
    )

    SecItemAdd([
        String(kSecAttrService) : serviceID,
        String(kSecClass) : kSecClassGenericPassword,
        String(kSecAttrAccessControl) : accessControl.takeUnretainedValue(),
        String(kSecAttrAccount) : username,
        String(kSecValueData) : data
    ], nil)
}
```

En este segundo también desarrollado en Swift se implementa un método para obtener una contraseña almacenada en el Keychain para un usuario dado. A la hora de acceder a la contraseña almacenada se mostrará un diálogo con un mensaje personalizado (i.e. Authenticate to Login) que permitirá al usuario autenticarse a través del TouchID.

```
func passwordForUsername(username: String) -> String? {
    let query = [
        String(kSecAttrService) : serviceID,
        String(kSecClass) : kSecClassGenericPassword,
        String(kSecReturnData) : true,
        String(kSecMatchLimit) : kSecMatchLimitOne,
        String(kSecUseOperationPrompt) : "Authenticate to Login",
        String(kSecAttrAccount) : username
    ]

    var passwordData: Unmanaged<AnyObject>? = nil
    SecItemCopyMatching(query, &passwordData)

    if let opaque = passwordData?.toOpaque() {
        return NSString(
            data: Unmanaged<NSData>.fromOpaque(opaque).takeUnretainedValue(),
            encoding: NSUTF8StringEncoding
        ) as? String
    }

    return nil
}
```

4.8 Las sesiones y los tokens de acceso se invalidan en el servicio remoto después de que se produzca un período predefinido de inactividad.

En la gran mayoría de frameworks de desarrollo de servicios remotos, se pueden definir mediante opciones de configuración los tiempos de expiración o caducidad de las sesiones. Este parámetro debe establecerse de acuerdo con las mejores prácticas especificadas en la documentación de cada uno de estos frameworks, pudiendo ir desde 10 minutos hasta dos horas, dependiendo del tipo de la aplicación.

Los riesgos asociados a una incorrecta implementación de este requisito (i.e. sesiones abiertas indefinidamente) son similares a los enumerados en el requisito 4.4 de esta misma categoría.

De cara a verificar el cumplimiento del mismo, se podría llevar a cabo un análisis dinámico de la aplicación siguiendo estos pasos.

1. Iniciar sesión en la aplicación.

2. Llevar a cabo un par de operaciones que requieran autenticación dentro de la aplicación.

3. Dejar la sesión inactiva hasta que caduque. Repetir alguna de las operaciones del paso 2 a través de un proxy de interceptación (e.g. Burp Repeater). De esta manera se enviará al servidor una request con el identificador de sesión o token que se debería haber invalidado tras el tiempo de inactividad. Si la invalidación de la sesión o token se ha producido correctamente en el servidor, se debería mostrar un mensaje de error en la aplicación. Por otro lado, si se recibe la misma respuesta que se recibió en el paso 2, el token o el identificador de la sesión todavía serán válidos y por lo tanto se podrá inferir que la sesión no se ha terminado correctamente en el servidor.

4.9 Existe un segundo factor de autenticación en el servicio remoto y el requisito de 2FA se aplica constantemente.

Para verificar el cumplimiento de este requisito, se deberá utilizar la aplicación de manera exhaustiva (siguiendo todos los flujos disponibles a nivel de interfaz de usuario). En paralelo se deberá utilizar un proxy de interceptación para capturar las requests enviadas a los endpoint remotos. A continuación, se deberán llevar a cabo las requests a los endpoints que requieran 2FA (e.g. realizar una transacción financiera) mientras se usa un token o identificador de sesión que aún no se ha elevado a través de 2FA o autenticación incremental (step-up). Si un endpoint sigue devolviendo los datos solicitados que solo deberían estar disponibles después de llevar a cabo un proceso de autenticación 2FA o incremental (step-up), se podrá afirmar que el proceso de autenticación no se ha implementado correctamente en dicho endpoint.

4.10 Las transacciones confidenciales requieren una autenticación incremental (step-up).

La autenticación de dos factores (2FA) es estándar para las aplicaciones que permiten a los usuarios acceder datos personales confidenciales o llevar a cabo operaciones confidenciales. En la gran mayoría de aplicaciones se utilizan contraseñas para el primer factor de autenticación y cualquiera de los siguientes para el segundo factor.

- Contraseña de un solo uso por SMS (SMS-OTP)
- Código de una sola vez a través de una llamada telefónica
- Token de hardware o software
- Notificaciones push en combinación con PKI y autenticación local

El segundo factor de autenticación se puede llevar a cabo durante el inicio de sesión del usuario o más tarde durante el transcurso de dicha sesión. Por ejemplo, después de iniciar sesión en una aplicación bancaria con un nombre de usuario y contraseña, el usuario estaría autorizado para realizar operaciones no confidenciales. Una vez que el usuario intenta ejecutar una transferencia bancaria o cualquier otro tipo de operación confidencial, se deberá presentar el segundo factor de autenticación (step-up authentication o autenticación incremental).

De cara a verificar el cumplimiento de este requisito, bastaría con verificar dentro de la aplicación que para llevar a cabo transacciones u operaciones confidenciales se necesita llevar a cabo el segundo factor de autenticación. A su vez, se deberá verificar que todos los servicios remotos asociados a este tipo de transacciones confidenciales sólo funcionarán correctamente en el caso de que se haya llevado a cabo previamente este segundo factor de autenticación.

4.11 La aplicación informa al usuario de todas las actividades de inicio de sesión con su cuenta. Los usuarios pueden ver una lista de los dispositivos utilizados para acceder a la cuenta y bloquear dispositivos específicos.

Este requisito de seguridad se utiliza hoy en día en la gran mayoría de servicios de Internet más populares (e.g. redes sociales). Aun a pesar de que hay un gran abanico de posibilidades a la hora de implementar este tipo de medida de seguridad, una de las formas más comunes de implementarlo es la siguiente.

- Cuando el usuario inicia sesión a través de un dispositivo no identificado previamente por el sistema, dicho usuario recibirá una alerta (e.g. a través del email, a través de una notificación de tipo push, etc).

- Una vez recibida dicha alerta, el usuario podrá o bien bloquear el acceso al servicio a ese dispositivo indicando que se ha producido un inicio de sesión fraudulento, o bien no realizar

ninguna acción, de tal manera que dicho dispositivo quedará asociado automáticamente a dicho usuario.

- El usuario deberá disponer de un dashboard donde podrá ver todos sus inicios de sesión más recientes en el sistema, así como los dispositivos utilizados en cada uno de ellos. En este dashboard se podrán bloquear / desautorizar el acceso al sistema por parte de cualquiera de los dispositivos asociados a dicho usuario.

Comentar que de cara a identificar estos dispositivos se podrán tener en cuenta factores tales como el tipo de dispositivo (e.g. smartphone, tablet, smartwatch, etc), la versión de hardware / software utilizada, la localización y dirección IP del dispositivo, etc.

5.5. V5: Requisitos de Comunicaciones de Red

El objetivo de los requisitos de seguridad incluidos dentro de esta categoría es garantizar la confidencialidad e integridad de la información intercambiada entre la aplicación móvil y los endpoints del servicio remoto. Cuanto menos, una aplicación móvil debe configurar un canal seguro y encriptado para la comunicación de red utilizando para ello el protocolo TLS con la configuración adecuada. El MASVS-L2 enumera un conjunto adicional de requisitos de seguridad, tales como el pinning de certificados SSL.

MASVS L1 y L2

5.1 Los datos se encriptan en la red usando TLS. El canal seguro se usa consistentemente a través de la aplicación.

Cualquier aplicación móvil que realice conexiones de red para obtener o enviar datos a un servicio remoto deberá hacerlo utilizando canales de comunicación seguros y encriptados.
De cara a verificar el cumplimiento de este requisito de seguridad, Apple introdujo en la el iOS SDK 9.0 un mecanismo de seguridad denominado App Transport Security (ATS) [17].

App Transport Security (ATS) es un conjunto de comprobaciones de seguridad que el sistema operativo realiza al llevar a cabo conexiones a través de NSURLConnection, NSURLSession y CFURL a nombres de dominio públicos. ATS está habilitado de manera predeterminada para las aplicaciones compiladas en el iOS SDK 9.0 y posteriores.

ATS se aplica solo cuando se realizan conexiones a nombres de dominio públicos, por lo que cualquier conexión realizada a una dirección IP, nombre de dominio no calificado o TLD de .local no estará protegida a través de ATS.

A continuación se muestra una lista extraída del MSTG [2], donde se resumen los requisitos de seguridad definidos dentro del App Transport Security.

- No se permiten conexiones HTTP.

- El certificado X.509 tiene una huella digital SHA256 y debe estar firmado con al menos una clave RSA de 2048 bits o una clave Elliptic-Curve Cryptography (ECC) de 256 bits.

- La versión de Transport Layer Security (TLS) debe ser 1.2 o superior y debe admitir Perfect Forward Secrecy (PFS) mediante el intercambio de claves Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) y cifrados simétricos AES-128 o AES-256.

Las restricciones de ATS se pueden deshabilitar configurando excepciones en el archivo Info.plist de la aplicación bajo la key NSAppTransportSecurity. Estas excepciones se pueden aplicar

globalmente o individualmente para cada uno de los dominios definidos, pudiendo ser las siguientes:

- Permitir conexiones inseguras (HTTP)
- Utilizar una versión más antigua de TLS
- Deshabilitar Perfect Forward Secrecy (PFS)
- Permitir conexiones a dominios locales

A continuación se muestra la estructura del diccionario [NSAppTransportSecurity] utilizado para configurar ATS dentro de la aplicación.

```
NSAppTransportSecurity : Dictionary {
    NSAllowsArbitraryLoads : Boolean
    NSAllowsArbitraryLoadsForMedia : Boolean
    NSAllowsArbitraryLoadsInWebContent : Boolean
    NSAllowsLocalNetworking : Boolean
    NSExceptionDomains : Dictionary {
        <domain-name-string> : Dictionary {
            NSIncludesSubdomains : Boolean
            NSExceptionAllowsInsecureHTTPLoads : Boolean
            NSExceptionMinimumTLSVersion : String
            NSExceptionRequiresForwardSecrecy : Boolean
NSRequiresCertificateTransparency : Boolean
        }
    }
}
```

Aunque en la gran mayoría de los casos no es una práctica recomendada, para desactivar ATS dentro de una aplicación se deberá configurar una excepción global de la siguiente manera.

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSAllowsArbitraryLoads</key>
    <true/>
</dict>
```

Para verificar el cumplimiento de este requisito bastará con verificar que se cumplen los requisitos por defecto de ATS y que no se han incluido excepciones relacionadas con permitir conexiones inseguras dentro de dicha configuración. Tampoco se deberá haber desactivado ATS dentro de la aplicación.

5.2 La configuración de TLS está en línea con las buenas prácticas actuales o lo más cerca posible de las mismas si el sistema operativo móvil no es compatible con los estándares recomendados.

Tal y como se especifica en el punto anterior, App Transport Security (ATS) establece unos requisitos de seguridad para llevar a cabo la configuración de TLS dentro de la aplicación. Entre estos requisitos podríamos enumerar los siguientes (ver requisito 5.1).

- El certificado X.509 tiene una huella digital SHA256 y debe estar firmado con al menos una clave RSA de 2048 bits o una clave Elliptic-Curve Cryptography (ECC) de 256 bits.

- La versión de Transport Layer Security (TLS) debe ser 1.2 o superior y debe admitir Perfect Forward Secrecy (PFS) mediante el intercambio de claves Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) y cifrados simétricos AES-128 o AES-256.

Por otro lado, el cipher suite deberá ser uno de los siguientes.

- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA

De cara a verificar el cumplimiento de este requisito bastará con verificar que se cumplen los puntos anteriores en la configuración de ATS, y verificar que no se han configurado excepciones relacionadas con la configuración de TLS (e.g. no se deben permitir versiones de TLS anteriores a la 1.2). Tampoco se deberá haber desactivado ATS dentro de la aplicación.

5.3 La aplicación verifica el certificado X.509 del endpoint remoto cuando el canal seguro está establecido. Solo se deberán aceptar certificados firmados por una CA confiable.

El cumplimiento de este requisito es relativamente sencillo, ya que por defecto las APIs de conectividad de iOS ya establecen que no se puedan llevar a cabo conexiones remotas con servidores que no dispongan de certificados firmados por una CA confiable.

Así pues, para verificar el cumplimiento de este requisito bastará con verificar a nivel de código que no se ha implementado a nivel de código ningún tipo de excepción relacionada con esto. Por ejemplo, de cara a permitir este tipo de certificados no firmados por una CA confiable (e.g. self-signed certificates) se debería incluir el código de ejemplo mostrado dentro del Anexo [5]. Este ejemplo se ha implementado en Objective-C y Swift utilizando la API del NSURLSession junto con el NSURLSessionTaskDelegate y el NSURLSessionDelegate respectivamente.

Comentar que a partir de iOS 9.0, con la introducción de ATS, por defecto no se permiten llevar a cabo conexiones a servidores que no tengan instalados certificados firmados por una CA confiable. Para saltarse esta medida de seguridad, se deberá desactivar ATS dentro de la aplicación (ver requisito 5.1).

MASVS-L2

5.4 La aplicación utiliza su propio almacén de certificados o lleva a cabo un pinning del certificado o clave pública del servicio remoto. La aplicación no deberá establecer conexiones con servicios remotos que tengan un certificado o clave diferente, incluso si está firmado por una CA de confianza.

El pinning de un certificado es el proceso de asociación de la aplicación móvil con un certificado X509 particular instalado en un servidor, en lugar de aceptar cualquier certificado firmado por una CA confiable. Una aplicación móvil que almacena ("pins") el certificado del servidor o la clave pública posteriormente solo establecerá conexiones con dicho servidor conocido. Al eliminar la confianza en las autoridades de certificación externas, las posibilidades de un ataque potencial

se reducen (después de todo, hay muchos casos conocidos en los que las autoridades de certificación se han visto comprometidas o engañadas al emitir certificados a impostores).

Se puede llevar a cabo el pinning del certificado o bien durante el desarrollo o bien en el momento en que la aplicación se conecta por primera vez al servicio remoto. En este segundo caso, el certificado es asociado o “anclado" al host en el momento en que se lleva a cabo la primera conexión con el servicio remoto. Esta segunda variante es menos segura, ya que un atacante que intercepte la conexión inicial podría inyectar su propio certificado.

Para verificar el cumplimiento de este requisito se deberá comprobar que se está llevando a cabo un pinning del certificado instalado en el servidor. Este pinning se puede implementar de las siguientes maneras.

1. Incluyendo el certificado del servidor dentro de la propia aplicación y realizando la verificación en cada una de las conexiones establecidas. Esto requiere un mecanismo de actualización cada vez que se actualiza el certificado en el servidor.

2. Limitar el emisor del certificado a por ejemplo una entidad, agrupando la clave pública de una CA intermedia en la aplicación. De esta forma, se limitarían las posibilidades de un ataque potencial y se aseguraría disponer de un certificado válido.

3. Poseer y administrar una PKI propia. La aplicación contendría la clave pública de la CA intermedia. Esto evita tener que actualizar la aplicación cada vez que cambie el certificado en el servidor, debido por ejemplo a la expiración del mismo. Se debe tener en cuenta que el uso de una CA propia provocaría que el certificado fuera self-signed.

En el Anexo [5] se muestra un snippet de código que muestra cómo es posible verificar si el certificado proporcionado por el servidor coincide con el certificado almacenado en la aplicación a través del NSURLSession junto con el NSURLSessionDelegate.

5.5 La aplicación no depende de un único canal de comunicación inseguro (e.g. correo electrónico o SMS) para operaciones críticas, como por ejemplo la creación y recuperación de cuentas de usuario.

De cara a implementar mecanismos que permitan llevar a cabo operaciones críticas tales como la creación y recuperación de cuentas de usuario, recuperación de contraseñas, etc se deberán implementar mecanismos de seguridad que incluyan la posibilidad de utilizar diferentes canales de comunicación dentro de los mismos. El objetivo de incluir diferentes canales de comunicación (aunque sean inseguros) es evitar la posibilidad de poder llevar a cabo la operación crítica debido a que un único canal de comunicación inseguro este inaccesible por diferentes motivos (e.g. pérdida de credenciales de usuario, desactivación o pérdida de acceso al canal por parte del usuario, etc).

5.6 La aplicación solo utilizará librerías de conectividad y seguridad actualizadas.

En el campo de la seguridad una de las máximas es utilizar versiones actualizadas de cualquier librería o framework utilizado por la aplicación para llevar a cabo este tipo de operaciones. La idea detrás de este requisito de seguridad es que la aplicación siempre incluya versiones de las librerías que no tengan vulnerabilidades de seguridad conocidas, ya que los desarrolladores y vendors de este tipo de librerías suelen llevar a cabo actualizaciones regulares que incluyan arreglos ante potenciales agujeros de seguridad encontrados en las mismas. Este requisito de seguridad es muy similar al requisito 1.9 descrito en la categoría V1 del MASVS, ya que éste trata sobre la necesidad de que los usuarios de la aplicación dispongan de versiones actualizadas de la misma para propiciar que dispongan de versiones sin vulnerabilidades de seguridad identificadas durante el mayor tiempo posible.

5.6. V6: Requisitos de Interacción con la Plataforma

Los controles de seguridad englobados dentro de esta categoría aseguran que la aplicación use las API de la plataforma y los componentes estándar de forma segura. Además, dentro de esta categoría se incluyen los controles que cubren la comunicación entre procesos (IPC).

MASVS L1 y L2

6.1 La aplicación solo solicita el conjunto mínimo de permisos necesarios.

Los usuarios deben conceder permisos para que una aplicación acceda a su información personal como la ubicación actual, el calendario, la información de contacto, los recordatorios y las fotos. Si bien es cierto que existen muchos aspectos positivos en permitir a las aplicaciones tener acceso a esta información, también es cierto que los usuarios deben tener el control sobre sus datos privados. Por ejemplo, a los usuarios les gusta poder etiquetar fotografías automáticamente con su ubicación actual o encontrar amigos cercanos, pero también deberán disponer de la opción de desactivar dichas funcionalidades. A continuación se muestran una serie de recomendaciones de cara a gestionar los permisos dentro de las aplicaciones.

- Sólo se deberán solicitar datos personales cuando la aplicación claramente lo necesite. Es natural desconfiar de una solicitud de información personal, especialmente si no hay una necesidad obvia de hacerlo. Se deberá asegurar que las solicitudes de permiso solo se realicen cuando los usuarios utilicen funciones que claramente necesitan datos personales. Por ejemplo, una aplicación solo debería solicitar acceso a la ubicación actual cuando se activa una función de seguimiento de ubicación.

- Se deberá explicar por qué la aplicación necesita la información si no es obvio. Se deberán agregar textos personalizado en las alertas de solicitud de permisos proporcionadas por el sistema. El texto de estas alertas deberá ser claro y específico, para que los usuarios no se sientan intimidados. No es necesario incluir el nombre de la aplicación ya que el sistema ya identifica la aplicación que realiza la solicitud.

- Se deberá solicitar permiso en el lanzamiento de la aplicación solo cuando sea necesario para que dicha aplicación funcione correctamente. A los usuarios no les molestará esta solicitud si es obvio que la aplicación depende de su información personal para funcionar.

- No se deberá solicitar información de ubicación innecesariamente. Antes de acceder a la información de ubicación, se deberá verificar si el servicio de ubicación está habilitado. Sabiendo esto, se puede retrasar la alerta hasta que una funcionalidad realmente lo requiera, pudiendo evitar la alerta si no se llega a ejecutar dicha funcionalidad.

6.2 Todas las entradas procedentes de fuentes externas o de los usuarios son validadas y saneadas si fuese necesario. Esto incluye datos recibidos a través de la interfaz de usuario, mecanismos de IPC, URL personalizadas y comunicaciones de red.

En el campo de la seguridad nunca se debe confiar en los datos procedentes de una fuente externa o de los usuarios, ya que un atacante podría manipularlos arbitrariamente. Tal y como afirma Michael Howard en su famoso libro Writing Secure Code [18], ”All Input is Evil”, lo cual se podría considerar como una regla de oro en el campo de la seguridad. Desafortunadamente, las aplicaciones complejas a menudo tienen una gran cantidad de puntos de entrada, lo cual dificulta que los desarrolladores apliquen esta regla.

Si bien es cierto que algunos tipos de ataques como pueden ser las Inyecciones SQL o el Cross-Site Scripting (XSS) son más propios de aplicaciones web, también es cierto que este tipo de ataques podrían ocurrir en cualquier tipo de aplicación. En muchos casos estos datos

introducidos en aplicaciones nativas se almacenan en bases de datos que pueden ser utilizadas por otro tipo de aplicaciones como pueden ser las aplicaciones web. A continuación se muestran algunos de los ataques más populares que se pueden llevar a cabo sino se validan correctamente las entradas de datos [6].

- Desbordamiento de Buffer.- Si la aplicación recibe información de un usuario u otra fuente no confiable, nunca debe copiar datos en un buffer de longitud fija sin verificar la longitud y truncarlo si es necesario. De lo contrario, un atacante potencial podría utilizar el campo de entrada para causar un desbordamiento del buffer.

  Dentro de la sección Avoiding Buffer Overflows and Underflows del documento Security Coding Guide [6], se puede encontrar más información sobre este tipo de vulnerabilidades así como la forma de prevenirlas.

- Ataques de Cadenas de Formato.- Si se recibe información de un usuario u otra fuente que no es de confianza y se muestra la misma, se debe tener cuidado de que las rutinas de visualización no procesen las cadenas de formato recibidas de la fuente que no es de confianza. Por ejemplo, en el siguiente código, la función de la librería estándar C syslog se utiliza para escribir una request HTTP recibida en el registro del sistema. Como la función syslog procesa cadenas de formato, procesará las cadenas de formato incluidas en el paquete de entrada.

```
/* Receiving HTTP Packet */
int size = recv(fd, pktBuf, sizeof(pktBuf), 0);
if (size) {
    syslog(LOG_INFO, "Received new HTTP request!");
    syslog(LOG_INFO, pktBuf);
}
```

  En este ejemplo, algunos tipos de cadenas de formato podrían causar problemas. Por ejemplo, supongamos que un atacante pasa la siguiente cadena en el paquete de entrada.

```
AAAA%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%n
```

  Esta cadena recupera ocho elementos de la pila. Suponiendo que la secuencia de formato en sí misma se almacena en la pila, dependiendo de la estructura de la pila, esto podría mover efectivamente el puntero de la pila al principio de la cadena de formato. Entonces, el token %n hará que la función de impresión tome el número de bytes escritos hasta el momento y escriba ese valor en la dirección de memoria almacenada en el próximo parámetro, que pasa a ser la cadena de formato. Por lo tanto, suponiendo una arquitectura de 32 bits, el AAAA en la cadena de formato en sí se trataría como el valor del puntero 0x41414141, y el valor en esa dirección se sobrescribiría con el número 76.

  Hacer esto normalmente causará un crash la próxima vez que el sistema tenga que acceder a esa ubicación de memoria, pero si se utiliza una cadena cuidadosamente diseñada para un dispositivo y sistema operativo específico, el atacante podría escribir datos arbitrarios en cualquier ubicación.

  Para evitar ataques de cadena de formato, se deberá verificar de que nunca se pasa información de entrada como parte de una cadena de formato. Para solucionarlo, se deberá incluir sólo una cadena de formato en cada llamada a esa función. Por ejemplo, la siguiente llamada podría considerarse vulnerable.

```
printf(buffer);
```

En cambio, esta otra no lo sería ya que todos los caracteres del parámetro del buffer, incluidos los signos de porcentaje (%), se imprimirían en lugar de interpretarse como tokens de formateo.

```
printf("%s", buffer);
```

- Manejo de URLs y Ficheros.- Si la aplicación tiene registrado algún tipo URL scheme personalizado, se debe tener cuidado con la forma en que se procesan los comandos enviados a dicha aplicación a través de la cadena de URL. Independientemente de que estos comandos sean públicos o no, un atacante potencial intentará enviar comandos a la aplicación. Si por ejemplo se proporciona un enlace o enlaces para iniciar su aplicación desde su sitio web, un atacante podrá ver qué comandos se están enviando e intentará cada variación de los mismos que se pueda imaginar. Se debe estar preparado para manejar, o filtrar, cualquier comando que se pueda enviar a la aplicación, no solo aquellos comandos que se está esperando recibir. Si por ejemplo se acepta un comando que hace que la aplicación envíe los credenciales a un servidor web, nunca se deberá hacer que el handler sea lo suficientemente genérico para que un atacante pueda sustituir la URL por la de su propio servidor web. En el requisito 6.3 se muestran más detalles sobre este tipo de mecanismos.

- Ataques de Inyección de Código.- Los comandos URL no validados y las cadenas de texto a veces permiten a un atacante insertar código en un programa que posteriormente ejecutará. Se corre el riesgo de recibir ataques de inyección cada vez que el código fuente funcione con datos que estén estructurados de manera flexible y que contengan una combinación de dos o más tipos diferentes de datos.

  Por ejemplo, si la aplicación pasa consultas a una base de datos SQL, esas consultas contendrán dos tipos de datos: el comando en sí (le dice a la base de datos qué hacer) y los datos que utiliza el comando. Los datos generalmente se separan del comando entre comillas. Sin embargo, si los datos que están almacenando contienen comillas, se deberán identificar adecuadamente estas marcas para que no se interpreten como el final de los datos. De lo contrario, un atacante potencial podría pasar a la aplicación una cadena que contenga comillas seguidas de un punto y coma para finalizar el comando, seguido de un segundo comando para ejecutar, de tal manera que en algún momento la base de datos SQL ejecutará el código inyectado por el atacante.

  Dentro de la sección Avoiding Injection Attacks and XSS del documento Security Coding Guide [6], se puede encontrar más información sobre este tipo de ataques así como la forma de prevenirlos.

- Ingeniería Social.- La ingeniería social puede utilizarse a través de vulnerabilidades de entrada no validadas para convertirse en un problema importante. Por ejemplo, si una aplicación acepta un comando URL para eliminar un archivo, pero primero se muestra un diálogo que solicita permiso al usuario, es posible que pueda enviar una cadena suficientemente larga para desplazarse por el nombre del archivo que se eliminará después del final del diálogo. De esta manera se podría engañar al usuario haciéndole creer que estaba borrando algo inofensivo, como datos innecesarios almacenados en la caché del sistema. Un ejemplo de este tipo de ataque sería el siguiente.

```
myapp://cmd/delete?file=los datos en caché que ralentizan el sistema,realfile
```

En este caso, el usuario podría ver un cuadro de diálogo con el texto "¿Está seguro de que desea eliminar los datos en caché que ralentizan el sistema?" El nombre del archivo real estaría fuera de la vista del usuario debajo de la ventana de diálogo. Sin embargo, si el usuario hace clic en el botón "Aceptar", se borrarían los datos reales del mismo.

6.3 La aplicación no exporta funcionalidad sensible a través de custom URL schemes, a menos que estos mecanismos estén protegidos adecuadamente.

Los custom URL schemes permiten que las aplicaciones se comuniquen a través de un protocolo personalizado. Una aplicación debe declarar que soporta un esquema en particular y manejar las URL entrantes que usan dicho esquema. Una vez que se haya registrado el URL scheme, otras aplicaciones podrán abrir la aplicación que registró el esquema y pasar una serie de parámetros a través de la creación de la URL con el formato apropiado y ejecutándolas a través del método openURL.

Los problemas de seguridad surgen cuando una aplicación procesa este tipo de llamadas sin validar adecuadamente la URL y sus parámetros, así como cuando no se solicita confirmación a los usuarios antes de desencadenar una acción importante.

Un claro ejemplo de este tipo de vulnerabilidades fue la vulnerabilidad descubierta en 2010 en la aplicación Skype Mobile. La aplicación Skype registró el URL scheme skype://, lo cual permitía que otras aplicaciones activaran llamadas a otros usuarios y números telefónicos de Skype. Desafortunadamente, la implementación incorrecta que hizo Skype no solicitaba permiso a los usuarios antes de realizar las llamadas, por lo que cualquier aplicación podría llamar a números arbitrarios sin que el usuario lo supiera. Para explotar dicha vulnerabilidad bastaba con poner un <iframe src = "skype://xxx? Call"></ iframe> invisible (donde xxx se reemplazaba por un número premium), por lo que cualquier usuario de Skype que inadvertidamente visitara un sitio web malicioso realizaba una llamada a un número premium. En el siguiente enlace se pueden encontrar más detalles sobre dicha vulnerabilidad.

https://software-security.sans.org/blog/2010/11/08/insecure-handling-url-schemes-apples-ios/

De cara a verificar el correcto cumplimiento de este requisito, bastará con averiguar si una aplicación registra algún URL scheme personalizado. Esta información se encuentra en el archivo info.plist en el directorio de la aplicación. Para ver los URL schemes registrados, se deberá abrir el proyecto en Xcode, ir a la pestaña Info y abrir la sección URL Types. El siguiente paso será determinar cómo se genera y valida el URL path. El método openURL del UIApplicationDelegate es responsable de manejar las URL de los usuarios. En la implementación de este método se deberán buscar y verificar los controles implementados, es decir cómo se validan las URL (la entrada que acepta) y si necesita permisos del usuario cuando se usa el custom URL scheme.

6.4 La aplicación no exporta funcionalidad sensible a través de mecanismos de IPC, a menos que estos mecanismos estén debidamente protegidos.

A diferencia de Android, plataforma que dispone de un gran abanico de posibilidades para llevar a cabo Inter-Process Communication (IPC), iOS ofrece pocas opciones para la comunicación entre aplicaciones (de hecho, no hay forma de que las aplicaciones se comuniquen directamente entre sí). En cambio, Apple ofrece dos tipos de comunicación indirecta: la transferencia de archivos a través de AirDrop y los custom URL schemes (ver requisito 6.3).

Así pues, se podría decir que este requisito de seguridad se cumple automáticamente debido a la implementación del sandbox del propio sistema operativo, siempre y cuando el dispositivo no esté jailbroken [19].

Una excepción a esto sería que varias aplicaciones pertenecieran a un mismo application group, ya que en este caso si que se podrían llevar a cabo diferentes tipos de comunicación entre procesos. En este escenario, la validación del cumplimiento de este requisito sería muy similar al establecido en el requisito 2.6, salvo que se debería verificar que no se exporta funcionalidad sensible en vez de datos confidenciales.

6.5 JavaScript está deshabilitado en las WebViews a menos que se requiera explícitamente.

Las WebViews son componentes de tipo navegador que permiten mostrar contenido web interactivo en una aplicación. Se pueden usar para incrustar contenido web directamente en la interfaz de usuario de una aplicación nativa. Las WebViews disponibles dentro de iOS permiten la ejecución de código JavaScript por defecto, por lo que serían vulnerables a ataques propios de aplicaciones web (e.g. XSS, script injection, etc).

De cara a verificar el cumplimiento de este requisito bastaría con buscar en el código fuente alguno de los siguientes componentes.

- UIWebView (iOS 7.1.2 y anteriores)
- WKWebView (iOS 8.0 y posteriores)
- SFSafariViewController

El componente UIWebView está deprecado y por lo tanto no debería utilizarse. En su lugar debería utilizarse WKWebView, el cual proporciona las siguientes mejoras de seguridad con respecto al UIWebView.

- La propiedad javaScriptEnabled de la clase WKPreferences se puede utilizar para deshabilitar completamente JavaScript en un WKWebView. Esto evita todos los posibles defectos de la inyección de scripts.

- La propiedad javaScriptCanOpenWindowsAutomatically de la clase WKPreferences se puede utilizar para evitar que JavaScript abra nuevas ventanas, tales como ventanas emergentes.

- La propiedad hasOnlySecureContent de la clase WKWebView se puede usar para verificar que los recursos cargados por WKWebView se recuperen a través de conexiones seguras.

- WKWebView implementa el renderizado del contenido web fuera del proceso, por lo que potenciales errores de corrupción de la memoria no afectarán el proceso principal de la aplicación.

Comentar que en el componente SFSafariViewController no se pueden deshabilitar el JavaScript, por lo cual es preferible utilizar WKWebView para incluir contenido web dentro de aplicaciones nativas.

En el Anexo [6] se muestra un snippet de código en Objective-C y Swift que muestra como crear un WKWebView y deshabilitar el código JavaScript en el mismo.

6.6 Las WebViews están configuradas para permitir solo un conjunto mínimo de controladores de protocolos necesarios (idealmente solo se debería soportar HTTPS). Los controladores potencialmente peligrosos, tales como file, tel y app-id, deberían estar deshabilitados.

En primer lugar, comentar que no existe una manera directa de deshabilitar los controladores de protocolos no necesarios dentro de un WKWebView. Así pues, de cara a cumplir con este requisito se debería implementar el método decidePolicyForNavigationAction del interface WKNavigationDelegate tal y como se muestra a continuación. En este ejemplo desarrollado en

Swift se bloquean las llamadas a links del tipo “tel” utilizado para iniciar llamadas telefónicas (e.g. tel:34-976-111-111).

```
func webView(webView: WKWebView, decidePolicyForNavigationAction
navigationAction: WKNavigationAction, decisionHandler:
((WKNavigationActionPolicy) -> Void)) {
    let url = navigationAction.request.URL
    let schemeToBlock: NSString = "tel"
    if (url!.scheme == schemeToBlock) {
        // Intercepting redirect and block it
       decisionHandler(.Cancel)
       return
    }
    decisionHandler(.Allow)
}
```

6.7 Si los métodos nativos de la aplicación están expuestos a una WebView, se deberá verificar que esta WebView solo presente JavaScript dentro del paquete de la aplicación.

Tanto UIWebView como WKWebView proporcionan métodos de comunicación entre la WebView y la aplicación nativa. Así pues, cualquier información confidencial o funcionalidad nativa expuesta al motor JavaScript de la WebView también sería accesible para cualquier código malicioso de JavaScript que se ejecute potencialmente dentro de la WebView. A continuación se muestra una descripción de estos dos componentes extraída del MSTG [2].

- UIWebView.- A partir de iOS 7, el framework JavaScriptCore proporciona un contenedor Objective-C al motor WebKit JavaScript. Esto permite ejecutar código JavaScript desde Swift y Objective-C, así como hacer que los objetos Objective-C y Swift sean accesibles en tiempo de ejecución desde JavaScript. Un entorno de ejecución de JavaScript está representado por un objeto JSContext, debiéndose tener especial cuidado con el código que asigna objetos nativos al JSContext asociado con un WebView. En Objective-C, el JSContext asociado con un UIWebView se podría obtener de la siguiente manera.

```
[webView valueForKeyPath:@"documentView.webView.mainFrame.javaScriptContext"]
```

  - Objective-C blocks.- Cuando se asigna un bloque Objective-C a un identificador en un JSContext, JavaScriptCore adapta automáticamente dicho bloque a una función de JavaScript.

  - JSExport protocol.- Este protocolo se utilizar para exportar las clases Objective-C y Swift y sus métodos de instancia, métodos de clase y propiedades al código JavaScript.

- WKWebView.- A diferencia de UIWebView, no es posible hacer referencia directamente al JSContext de un WKWebView ya que el layout y el código JavaScript se renderizan en otro proceso (ver requisito 6.6). Así pues, en este control la comunicación entre el código JavaScript y el código nativo de la aplicación se implementa utilizando un sistema de mensajería. El código JavaScript puede enviar mensajes a la aplicación nativa utilizando el método postMessage de la siguiente manera.

```
window.webkit.messageHandlers.myHandler.postMessage(“Send from JavaScript!”)
```

La aplicación nativa recibirá los mensajes enviados desde el código JavaScript a través el siguiente método disponible en el protocolo WKScriptMessageHandler.

```
func userContentController(userContentController:
WKUserContentController!,didReceiveScriptMessage message: WKScriptMessage!) {
    if(message.name == “myHandler") {
        println("JavaScript is sending a message \(message.body)")
    }
}
```

El método postMessage serializa automáticamente objetos de JavaScript en objetos nativos Objective-C o Swift. El message handler asociado al WebKit se configura utilizando el método addScriptMessageHandler.

Comentar que los métodos nativos expuestos dentro de la aplicación sólo estarán accesibles al código JavaScript de la WebView dentro de la propia aplicación debido a las propias restricciones del sistema operativo (i.e. sandboxing).

6.8 La serialización y deserialización de objetos, si la hay, se implementa utilizando APIs seguras de serialización.

De cara a implementar la serialización y deserialización de objetos en una clase cualquiera, dicha clase deberá adoptar el protocolo NSCoding e implementar sus métodos. Este protocolo declara dos métodos que se envían a los objetos que deben ser serializados o deserializados.

De acuerdo con los principios del diseño orientados a objetos, un objeto cualquiera es responsable de codificar y decodificar su estado. Un coder ordena al objeto que lo haga invocando los métodos encodeWithCoder o initWithCoder. El método encodeWithCoder instruye al objeto para que codifique su estado con el codificador provisto, de tal manera que un objeto cualquiera puede recibir este mensaje cualquier cantidad de veces. El mensaje initWithCoder instruye al objeto para que se inicialize a partir de los datos del coder proporcionado, de tal manera que reemplace cualquier otro método de inicialización y se envíe solo una vez por objeto.

Cuando un objeto recibe el mensaje encodeWithCoder debe codificar todo su estado (a menudo representado por sus propiedades o variables de instancia), después de reenviar el mensaje a su superclass si ésta también implementa el protocolo NSCoding. Un objeto no tiene que codificar todo su estado ya que algunos valores pueden no ser importantes para restablecer su estado y otros pueden derivarse del estado relacionado con la decodificación. A continuación se muestra un ejemplo de la implementación de este método en Objective-C (se está suponiendo que su superclass también implementa el protocolo NSCoding).

```
- (void)encodeWithCoder:(NSCoder *)coder {
    // Call superclass
    [super encodeWithCoder:coder];

    // Encode properties
    [coder encodeObject:self.firstName forKey:KPersonFirstName];
    [coder encodeObject:self.lastName forKey:KPersonLastName];
    [coder encodeFloat:self.height forKey:KPersonHeight];
}
```

En la implementación del método initWithCoder, el objeto debe invocar primero el inicializador designado de su superclass para inicializar el estado heredado, y luego debe decodificar e inicializar su propio estado. Los diferentes valores serializados (i.e. keys) se pueden decodificar

en cualquier orden. Una implementación del método initWithCoder para el ejemplo anterior podría quedar de la siguiente manera en Objective-C.

```
- (id)initWithCoder:(NSCoder *)coder {
    // Call superclass
    self = [super initWithCoder:coder];
    if (self) {
        // Decode properties
        _firstName = [coder decodeObjectForKey:KPersonFirstName];
        _lastName = [coder decodeObjectForKey:KPersonLastName];
        _height = [coder decodeFloatForKey:KPersonHeight];
    }
    return self;
}
```

5.7. V7: Requisitos de Calidad del Código y Configuración de la Compilación

El objetivo de los requisitos definidos dentro de esta categoría es garantizar que se sigan un conjunto de buenas prácticas de seguridad en el desarrollo del código de la aplicación, así como un correcto uso de las medidas de seguridad que ofrecen tanto el entorno de desarrollo como el compilador.

MASVS L1 y L2

7.1 La aplicación está firmada y provisionada con un certificado válido.

Tal y como se estable en el MSTG [2], la firma de código de una aplicación asegura a los usuarios de la misma que esta procede de una fuente o desarrollador conocido, así como que la aplicación no se ha modificado desde que se firmó por última vez. Antes de que la aplicación pueda integrar los servicios del sistema operativo, instalarse en un dispositivo o enviarse al App Store, debe estar firmada con un certificado emitido por Apple. Se puede encontrar más información sobre el proceso de firmado y distribución de aplicaciones en la App Distribution Guide [20].

De cara a verificar el cumplimiento de este requisito se puede utilizar la herramienta codesign disponible en la consola de OS X. Esta herramienta permite crear, verificar y mostrar firmas de código, así como verificar dinámicamente el código firmado dentro del sistema.

Para verificar que una aplicación está firmada por un certificado válido se deberá obtener el fichero .ipa de la aplicación, renombrarlo a .zip y descomprimir el fichero. Una vez descomprimido, se deberá navegar al directorio Payload y allí se podrá encontrar el fichero .app. Una vez encontrado dicho fichero, se deberá ejecutar el siguiente comando.

```
codesign -dvvv <yourapp.app>
```

Este comando mostrará cierto tipo de información sobre la aplicación, como por ejemplo el identificador, las arquitecturas soportadas, el algoritmo de hash utilizado, el certificado utilizado para llevar a cabo la firma, etc. Más información sobre el proceso de firmado de puede encontrar en el documento Code Signing Guide [21].

7.2 La aplicación se ha construido en modo release, con las configuraciones apropiadas para una compilación de release (e.g. non-debuggable).

De cara a construir una aplicación en modo release bastará con seguir los siguientes pasos dentro de un proyecto de Xcode.

1. Verificar que se ha configurado correctamente el firmado de la aplicación a través de los ajustes del proyecto en Xcode (ver Code Signing Guide [21]).

2. Seleccionar Product > Scheme > YOUR_APP

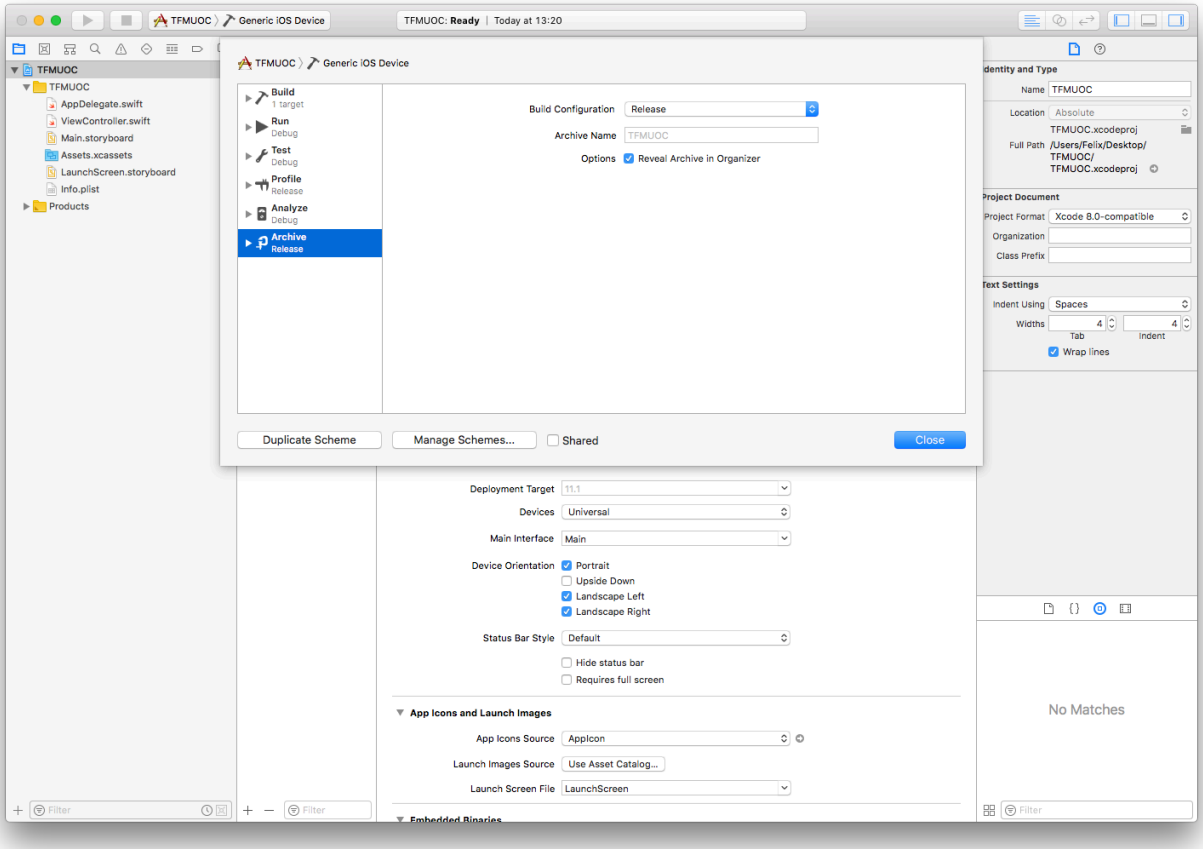
3. Seleccionar Product > Destination > Generic iOS Device

4. Seleccionar Product > Scheme > Edit Scheme

5. Dentro de este diálogo se deberá seleccionar la pestaña Archive del menu situado en la izquierda y verificar que se ha seleccionado Release dentro del Build Configuration.

6. Seleccionar la opción Product > Archive

Si la configuración del firmado de la aplicación es correcta, se debería generar un .xcarchive que se mostrará automáticamente en el Xcode Organizer. Este organizer es accesible a través de la opción del menú Window > Organizer (ver Anexo [11]).



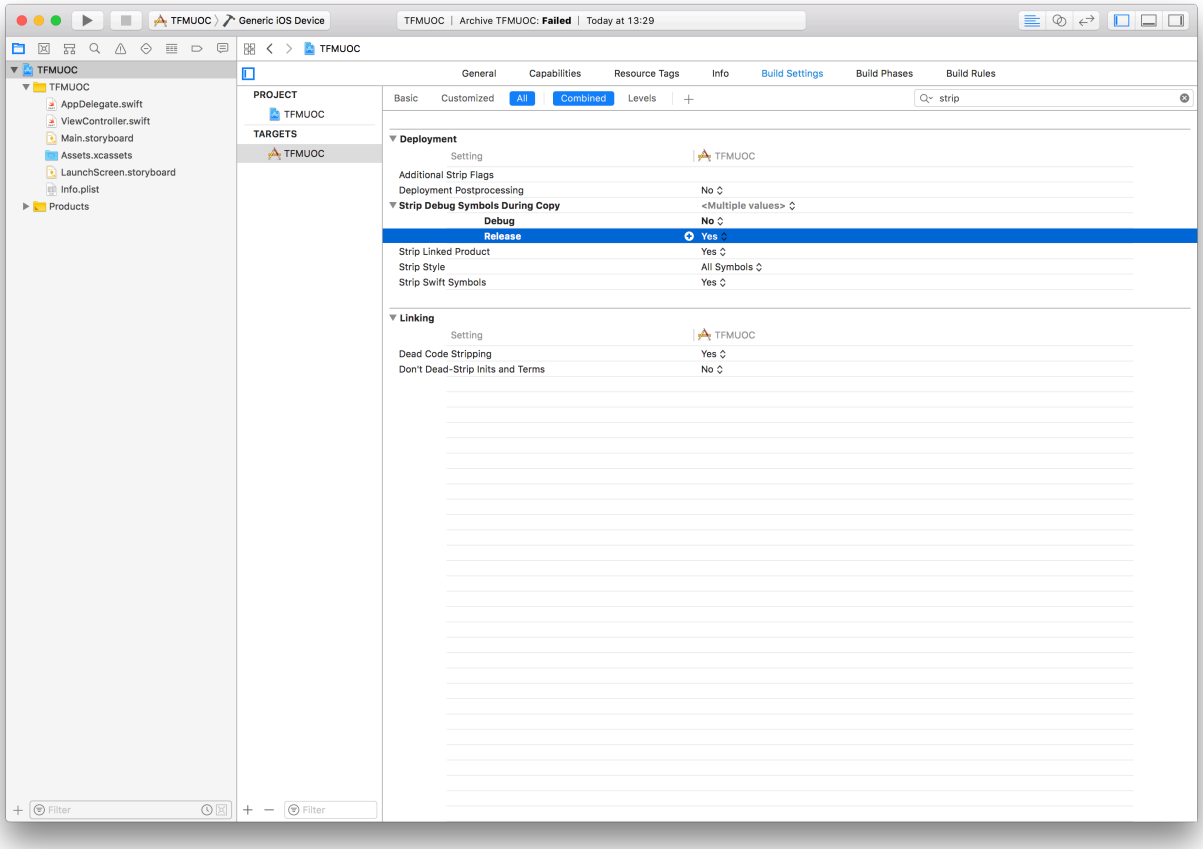
7.3 Los símbolos de depuración se han eliminado de los binarios nativos.

Como regla general, se deberá incluir la menor cantidad de información posible en los binarios de las aplicaciones distribuidas a los usuarios en modo release. Ciertos tipos de información como la información de depuración o debugging symbols, los números de línea, así como los nombres de métodos y funciones que sean muy descriptivos, facilitan el llevar a cabo procesos de ingeniería inversa. Así pues, este tipo de información no debería ser incluida en los binarios construidos en modo release sin que esto llegue a afectar al correcto funcionamiento de la aplicación.

En las aplicaciones para iOS estos símbolos se pueden guardar o bien en formato Stabs o bien en formato DWARF. Al utilizar el formato Stabs, tanto los símbolos de depuración como otros tipos de símbolos se almacenan en la tabla de símbolos normal. Con el formato DWARF, los símbolos de depuración se almacenan en un segmento especial __DWARF dentro del binario. Los símbolos de depuración de DWARF también se pueden guardar como un archivo separado de información de depuración. En este caso se debe verificar que no haya símbolos de depuración en el binario publicado (ya sea en la tabla de símbolos o en el segmento __DWARF).

De cara a verificar el correcto cumplimiento de este requisito se deberá verificar que no se han incluido símbolos del tipo Stabs o DWARF en el binario principal. Para llevar a cabo estar verificación se puede utilizar la herramienta godjdump incluida dentro del paquete binutils que se puede descargar a través de Homebrew dentro de OS X.

Para no incluir estos símbolos dentro de un binario generado en modo release de una aplicación iOS, se deberá configurar el parámetro Strip Debug Symbols During Copy como YES dentro de los ajustes del target en modo release en Xcode (ver Anexo [11]).



7.4 El código de depuración se ha eliminado y la aplicación no registra los errores detallados ni los mensajes de depuración.

Este requisito es muy similar al requisito 2.2 incluido dentro de la categoría de V2: Requisitos de Almacenamiento de Datos y Privacidad. La idea de ambos requisitos es que no se debe incluir código de depuración dentro de binarios construidos en modo release. Este código de depuración puede incluir el logging de información de los datos y estado de la aplicación en diferentes fases de la ejecución a través de métodos como pueden ser el NSLog, print, println, dump, etc. Este tipo de información puede facilitar enormemente el proceso de ingeniería inversa de la aplicación, por lo cual no debería incluirse dentro de binarios generados en modo release.

Tal y como se explica en el requisito 2.2, para llevar a cabo un análisis estático que verifique el cumplimiento de este requisito, bastará con realizar una búsqueda en el código fuente utilizando algunas palabras claves relacionadas con las funciones de logging. Para llevar a cabo un análisis dinámico bastará conectar el dispositivo iOS al laptop, abrir el Organizador (CMD + MAYÚS + 2) dentro de Xcode, y allí se podrán ver los logs de todas las aplicaciones instaladas en el dispositivo conectado.

De cara a evitar que estos logs aparezcan en aplicaciones desplegadas en producción se podrían utilizar macros para llevar a cabo implementaciones condicionales de los métodos de logging utilizados en la aplicación. Un ejemplo en Swift similar al expuesto en el requisito 2.2 para Objective-C sería el siguiente.

```
func printDebugMode(_ object: Any) {
    #if DEBUG
        print(object)
    #endif
}
```

En general se podría utilizar el siguiente mecanismo de compilación condicional en Objective-C.

```
#ifdef DEBUG
    // Debug-only code
#endif
```

Su equivalente en Swift se muestra en el siguiente snippet de código.

```
#if DEBUG
    // Debug-only code
#endif
```

7.5 Todos los componentes de terceros utilizados por la aplicación móvil, tales como librerías y frameworks, están identificados y se ha verificado que no sufren ningún tipo de vulnerabilidad conocida.

Este requisito, muy en línea con los requisitos 1.1 y 5.6, tiene como idea que la aplicación sólo incluya versiones de las librerías que no tengan vulnerabilidades de seguridad conocidas.

En el requisito 1.1 se establece que cualquier componente utilizado dentro de la aplicación con una vulnerabilidad de seguridad conocida podría suponer un riesgo para la seguridad de los usuarios de la propia aplicación.

En el requisito 5.6 se hace referencia a que una aplicación siempre debe utilizar las versiones más actualizadas de cualquier librería o framework utilizado en la misma, ya que los desarrolladores y vendors de este tipo de librerías suelen llevar a cabo actualizaciones regulares que incluyan arreglos ante potenciales agujeros de seguridad encontrados en las mismas.

Así pues, para cumplir con este requisito bastará con verificar que no existen vulnerabilidades conocidas en las versiones de las librerías y frameworks utilizados dentro de la aplicación. Si se encontrase algún tipo de vulnerabilidad en alguno de los componentes externos utilizados por la aplicación, este componente se debería actualizar tan pronto como el vendor o desarrollador proporcione una actualización que incluya un parche o solución para dicha vulnerabilidad.

7.6 La aplicación detecta y maneja todas las posibles excepciones.

El objetivo de este requisito es verificar que la aplicación gestiona las excepciones correctamente, de tal manera que durante su ejecución siempre se encuentre en un estado seguro sin exponer ningún tipo de información confidencial en el interfaz de usuario y en los mecanismos de logging utilizados por la aplicación.

El manejo de excepciones es bastante diferente en Objective-C que en Swift, ya que ambos lenguajes proporcionan sus mecanismos propios para gestionar este tipo de errores. Así pues, en Objective-C podríamos decir que existen 2 tipos de errores principales.

- NSException.- Este tipo de error se utiliza principalmente para gestionar errores en el desarrollo del código (e.g. divisiones entre 0, arrays fuera de rango, etc).

```
@try {
     // do work here
 }
@catch (NSException *e) {
    // recover from exception
}
@finally {
    // cleanup
}
```

- NSError.- Se utiliza para cualquier otro tipo de error que no se maneje a través de excepciones. En muchas de las APIs de Cocoa se debe pasar una variable de tipo NSError por referencia para gestionar los errores en el caso de que algo vaya mal durante la ejecución del método en cuestión. En este tipo de métodos se suele devolver un BOOL que indica si la función ha terminado con éxito o no, de tal manera que si ha fallado en su ejecución se devuelvan los detalles del error en el parámetro NSError pasado como referencia. En el caso de que el método tenga que devolver cualquier otro tipo de objeto, se deberá devolver un nil en el caso de que se produzca un error de ejecución en dicho método, y al igual que en el caso anterior se incluirán más detalles sobre dicho error en el parámetro NSError pasado como referencia.

A continuación se muestran las 4 maneras principales de gestionar los errores en Swift.

- Se puede propagar el error declarando un método o función cualquiera como throws.

```
func dosomething(argumentx:TypeX) throws {
    try functionThatThrows(argumentx: argumentx)
}
```

- Manejar el error con un do-catch statement.

```
do {
    try functionThatThrows()
    defer {
      //use this as your finally block as with Objective-c
    }
    statements
} catch pattern 1 {
    statements
} catch pattern 2 where condition {
    statements
}
```

- Manejar el error como un opcional.

```
// In this case the value of x is nil in case of an error.
let x = try? functionThatThrows()
```

- Asegurar que el error no ocurrirá a través de la instrucción try!.

Tal y como se expone en el MSTG [2], de cara verificar el correcto cumplimiento de este requisito se deberán verificar los siguientes puntos en Objective-C.

- La aplicación utiliza un esquema bien diseñado y unificado para gestionar los errores y excepciones.
- Las excepciones producidas en los frameworks de Cocoa se manejan correctamente.
- La memoria reservada en los bloques @try se librera en los bloques @finally.
- Por cada @throw el método que lo ejecuta dispone de un bloque @catch.
- La aplicación no expone información confidencial en el proceso de gestión de errores, pero explica correctamente estos errores al usuario.
- Cualquier tipo de información confidencial siempre se elimina en los bloques @finally en el caso de aplicaciones que manejan información confidencial.
- La instrucción raise() sólo se usa en casos excepcionales cuando se necesita terminar el programa sin la necesidad de proporcionar ningún tipo de información al usuario.
- Los objetos de tipo NSError no contienen ningún tipo de información confidencial.

En Swift, los puntos a verificar serían los siguientes.

- La aplicación utiliza un esquema bien diseñado y unificado para gestionar los errores y excepciones.
- La aplicación no expone información confidencial en el proceso de gestión de errores, pero explica correctamente estos errores al usuario.
- Cualquier tipo de información confidencial siempre se elimina en los bloques defer en el caso de aplicaciones que manejan información confidencial.

- La instrucción try! sólo se debe utilizar cuando se ha verificado previamente que no se va a producir ningún tipo de error de ejecución en el método asociado con la misma.

De cara a facilitar el cumplimiento de los puntos anteriores, se deberán seguir una serie de buenas prácticas y recomendaciones que se enumeran a continuación.

- Se deberá asegurar que la aplicación utiliza un esquema bien diseñado y unificado para gestionar los errores y excepciones.

- Se deberá asegurar que toda información de logging asociada al manejo de errores no se mostrará en binarios generados en modo release.

- En aplicaciones Objective-C que manejan datos confidenciales se recomienda crear un exception handler personalizado que borre todo tipo de información confidencial en el caso de que se produzca una excepción durante la ejecución de la aplicación. Este excepción handle se utilizará a través de la instrucción NSSetUncaughtExceptionHandler.

- Cuando se utilice Swift se deberá asegurar que no se usa la instrucción try! a no ser que se haya verificado previamente que no se producirá ningún error en el método asociado a la misma.

- Cuando se utilice Swift se deberá asegurar que los errores no se propagan demasiado a través de método intermedios.

7.7 La lógica de gestión de errores en los controles de seguridad deniega el acceso por defecto.

Este requisito trata de asegurar que siempre que se requiera que el usuario proporcione acceso para utilizar alguna característica o funcionalidad del sistema operativo o del dispositivo (e.g. librería de photos, cámara, información de localización, etc), se deniegue el acceso a la misma por defecto. De esta manera, se asegura que ante cualquier fallo o error en el proceso de gestión de permisos en tiempo de ejecución la aplicación no tenga acceso a dicha característica o funcionalidad a menos que el usuario haya concedido permisos explícitamente para acceder a la misma.

7.8 En el código no administrado, la memoria se asigna, libera y usa de forma segura.

En primer lugar, comentar que en las aplicaciones desarrolladas para iOS tanto en Objective-C como en Swift el código generado se ejecuta directamente a nivel del sistema operativo (i.e. código máquina). Esto significa que a diferencia de otros lenguajes de programación como Java o .NET que se ejecutan sobre una máquina virtual, el código generado para las aplicaciones iOS se ejecuta directamente sobre el sistema operativo (i.e. código nativo). Así pues, se podría decir que Objective-C y Swift pertenecen al conjunto de los unmanaged programming languages (i.e. lenguajes compilados) y Java o .NET pertenecen al conjunto de los managed programming languages (i.e. lenguajes interpretados).

Dicho esto, en lo que respecta a la gestión de la memoria, tanto Swift como Objective-C proporcionan un mecanismo denominado Automatic Reference Counting (ARC), el cual permite gestionar la memoria de forma transparente en tiempo de compilación. Este mecanismo de gestión de memoria es mucho más óptimo que por ejemplo el recolector de basura o garbage collector utilizado en lenguajes como Java, ya que en el primero se lleva a cabo la gestión de la memoria en tiempo de compilación y en el segundo se requiere que la máquina virtual gestione la memoria en tiempo de ejecución.

La principal diferencia entre Swift y Objective-C es que mientras en el primero se lleva a cabo toda la gestión de la memoria a través de ARC, en el segundo hay algunas APIs tales como Core Graphics en las que todavía hay que llevar a cabo un proceso manual de gestión de la memoria. Otra diferencia importante entre Swift y Objective-C es que en el segundo todavía es posible llevar a cabo una gestión manual de la memoria a través de un proceso denominado MRR (manual retain-release). Si bien es cierto que esta gestión manual de la memoria no es recomendada, también es cierto que esta posibilidad está disponible para los desarrolladores, y por lo tanto debería tenerse en cuenta sobre todo de cara al mantenimiento de bases de código antiguas que todavía no han sido migradas a ARC.

Así pues, para verificar el correcto cumplimiento de este requisito se deberá verificar que se lleva a cabo una gestión de la memoria eficiente y segura dentro de la aplicación, utilizando para ello herramientas como pueden ser el NSZombie o el Xcode Instruments.

Más información sobre la gestión de la memoria en aplicaciones desarrolladas en Objective-C se puede encontrar en el documento Advanced Memory Management Programming Guide [22].

7.9 Los mecanismos de seguridad ofrecidos por el entorno de desarrollo, tales como el byte-code minification, la protección de la pila, el PIE support y el conteo automático de referencias, están activados.

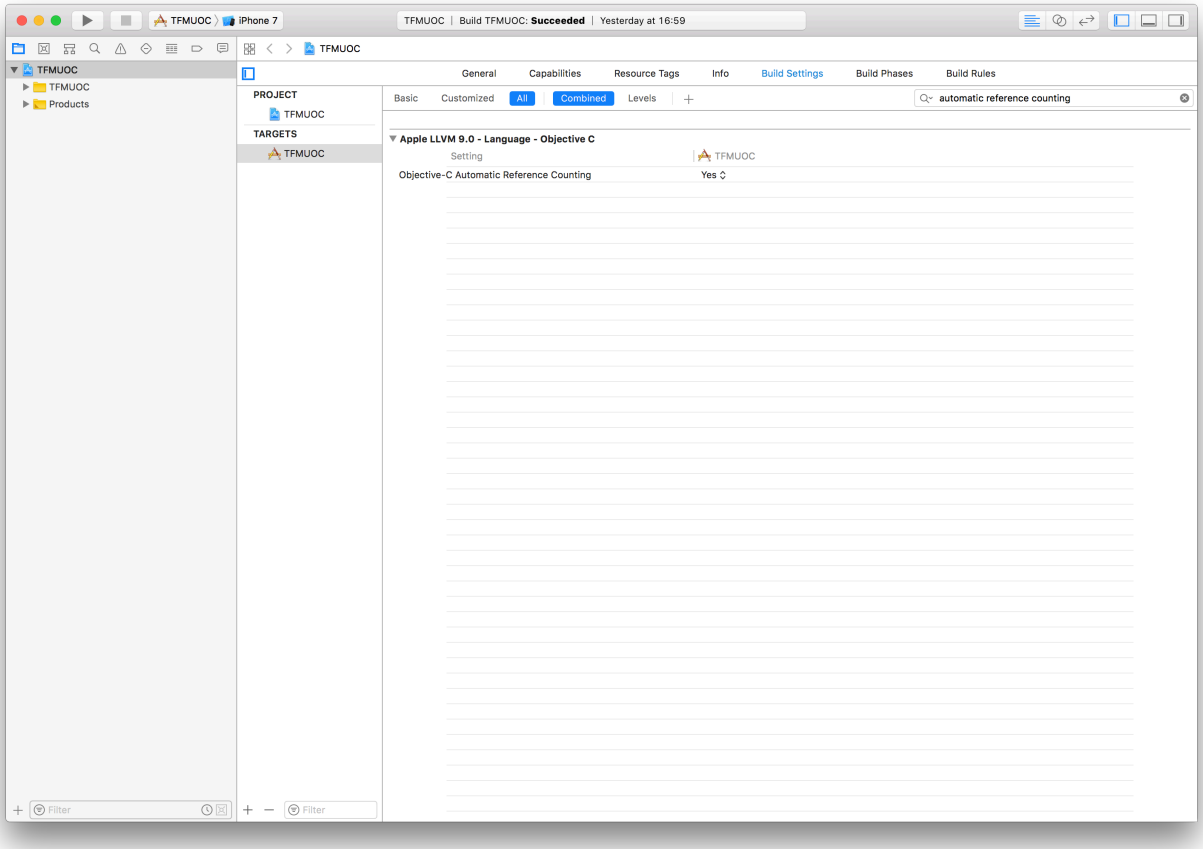
Este requisito trata de verificar que los mecanismos de seguridad ofrecidos por Xcode estén activados dentro de nuestra aplicación. Si bien es cierto que un un principio este requisito debería cumplirse automáticamente para todas las aplicaciones relativamente nuevas ya que Xcode tiene activado todos estos mecanismos por defecto, también es cierto que en algunas aplicaciones antiguas todavía podrían estar desactivadas algunas de estas características. A su vez, también podrían darse casos en los que el desarrollador ha desactivado manualmente alguna de estas características por error. Entre los mecanismos de seguridad proporcionados por Xcode podríamos destacar los siguientes.

- ARC.- Automatic Reference Counting es un mecanismo de gestión de memoria que en tiempo de compilación añade llamadas a los métodos retain y release cuando sea necesario.

  De cara verificar si se está utilizando ARC en un .app, se deberán buscar referencias a los símbolos _objc_retainAutoreleaseReturnValue, _objc_autoreleaseReturnValue, _objc_retain, _objc_release, y _objc_storeStrong en el output del siguiente comando de otool.

```
$ otool -Iv MyApp.app | grep release
0x0045b7dc 83156 ___cxa_guard_release
0x0045fd5c 83414 _objc_autorelease
0x0045fd6c 83415 _objc_autoreleasePoolPop
0x0045fd7c 83416 _objc_autoreleasePoolPush
0x0045fd8c 83417 _objc_autoreleaseReturnValue
0x0045ff0c 83441 _objc_release
[SNIP]
```

  En la siguiente captura de pantalla se muestra como debería configurarse el target de una aplicación para activar ARC dentro de un proyecto de Xcode. Básicamente bastaría con configurar el parámetro Objective-C Automatic Reference Counting como YES dentro de los ajustes del target de la aplicación (ver Anexo [11]).

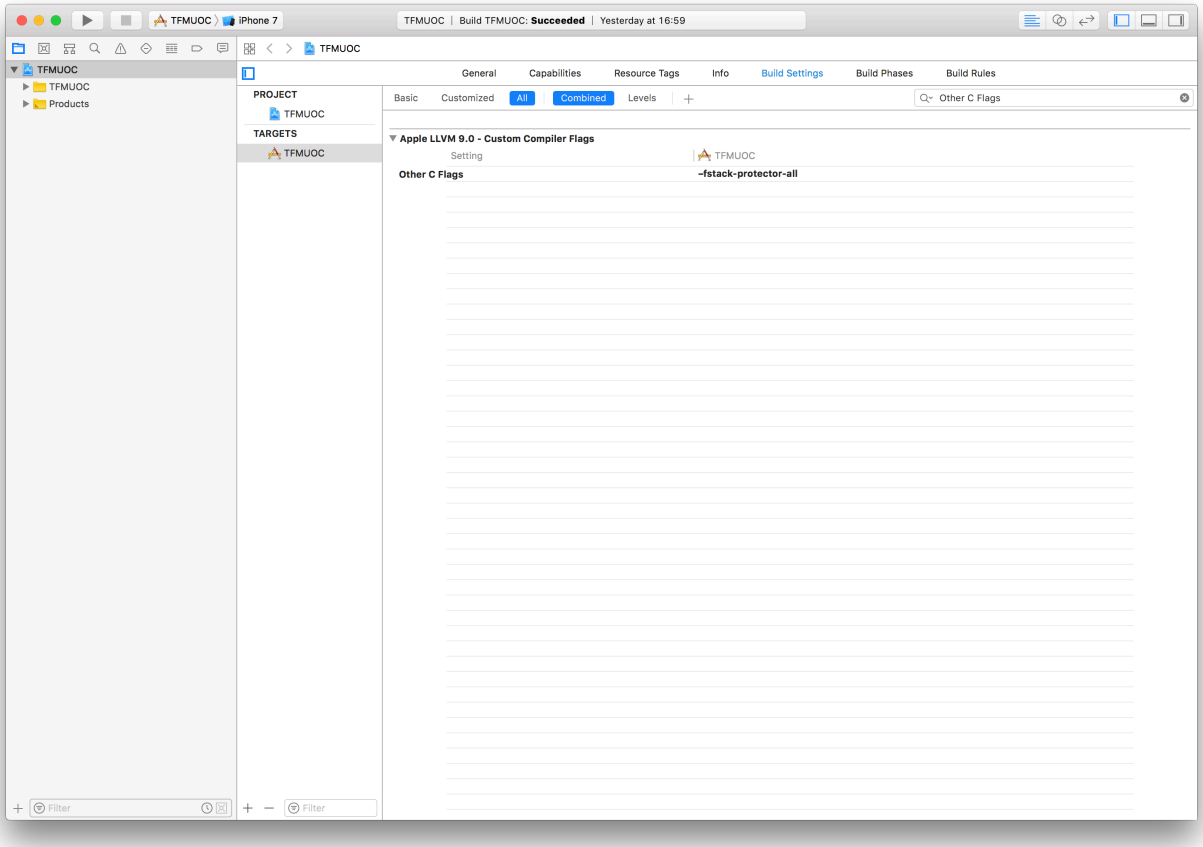


- Stack Canary.- Previene ataques de tipo buffer overflow o desbordamiento de buffer.

  Para verificar si se está utilizando Stack Canary, se deberán buscar referencias a los símbolos ___stack_chk_fail y ___stack_chk_guard en el output del siguiente comando de otool.

```
$ otool -Iv MyApp.app | grep stack
0x0046040c 83177 ___stack_chk_fail
0x0046100c 83521 _sigaltstack
0x004fc010 83178 ___stack_chk_guard
0x004fe5c8 83177 ___stack_chk_fail
0x004fe8c8 83521 _sigaltstack
0x00000001004b3fd8 83077 ___stack_chk_fail
0x00000001004b4890 83414 _sigaltstack
0x0000000100590cf0 83078 ___stack_chk_guard
0x00000001005937f8 83077 ___stack_chk_fail
0x0000000100593dc8 83414 _sigaltstack
```

  En la siguiente captura de pantalla se muestra como debería configurarse el target de una aplicación para activar la protección de pila dentro de un proyecto de Xcode. Tal y como se puede ver, bastaría con incluir -fstack-protector-all en el parámetro Other C Flags dentro de los ajustes del target de la aplicación (ver Anexo [11]).

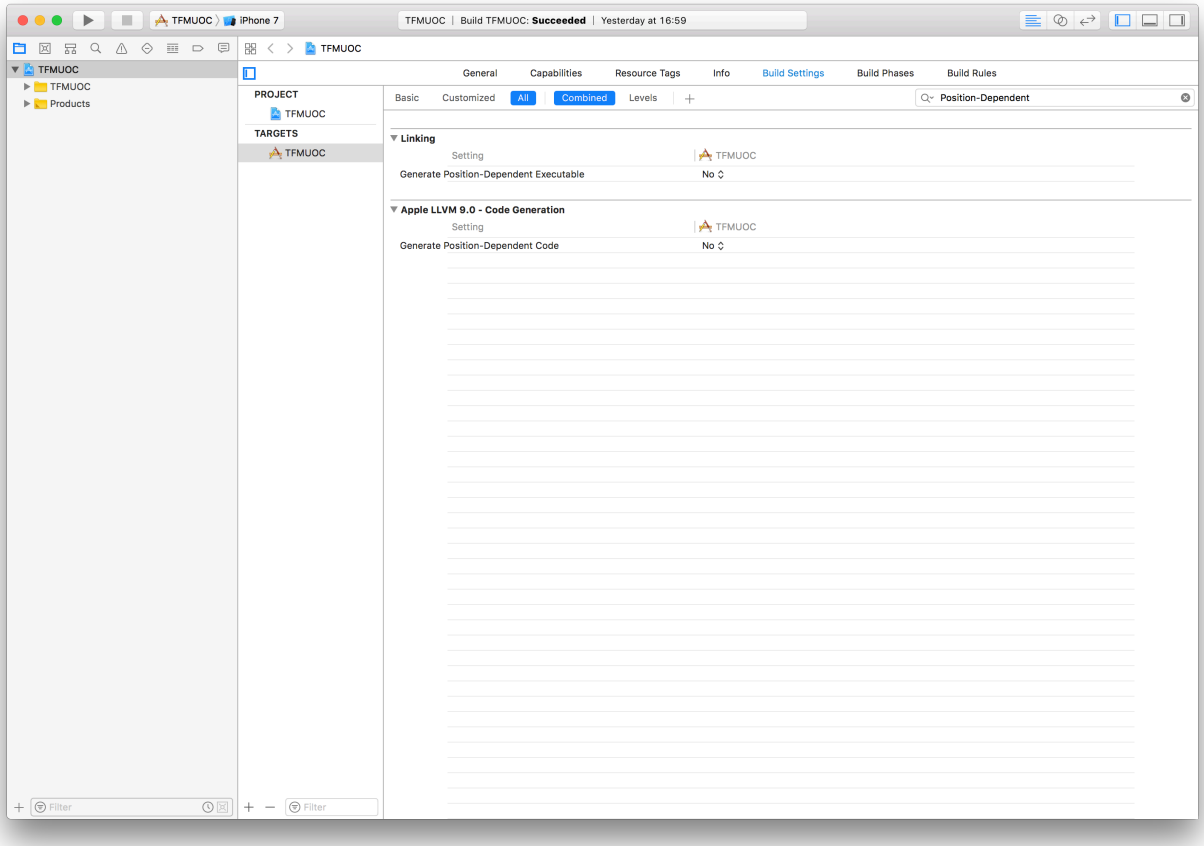


- PIE.- Position Independent Executable es un mecanismo que permite habilitar ASLR (Address Space Layout Randomization) dentro de la aplicación.

  En el caso de verificar si se está utilizando PIE correctamente, se deberá comprobar que la aplicación no se carga en una dirección de memoria fija en el output del siguiente comando de otool.

```
$ otool -hv MyApp.app
MyApp (architecture armv7):
Mach header
magic cputype cpusubtype caps filetype ncmds sizeofcmds flags
MH_MAGIC ARM V7 0x00 EXECUTE 38 4292 NOUNDEFS DYLDLINK TWOLEVEL
WEAK_DEFINES BINDS_TO_WEAK PIE
MyApp (architecture arm64):
Mach header
magic cputype cpusubtype caps filetype ncmds sizeofcmds flags
MH_MAGIC_64 ARM64 ALL 0x00 EXECUTE 38 4856 NOUNDEFS DYLDLINK TWOLEVEL
WEAK_DEFINES BINDS_TO_WEAK PIE
```

  En la siguiente captura de pantalla se muestra como debería configurarse el target de una aplicación para activar el PIE dentro de un proyecto de Xcode. En principio sería tan sencillo como configurar los parámetros Generate Position-Dependent Executable y Generate Position-Dependent Code como NO dentro de los ajustes del target de la aplicación (ver Anexo [11]).



5.8. V8: Requisitos de Resistencia contra la Ingeniería Inversa

En esta categoría se engloban una serie de requisitos de seguridad que permiten mejorar la resistencia contra la ingeniería inversa y cierto tipo de ataques específicos en el lado del cliente. Estos requisitos o controles se deberán aplicar cuando se considere necesario, basando esta decisión en una evaluación de los riesgos causados por la modificación no autorizada del binario de la aplicación (i.e. unauthorized tampering) y/o la ingeniería inversa en el código de la aplicación. De cara a verificar los posibles riesgos y amenazas asociados a este tipo de ataques se recomienda consultar el documento Technical Risks of Reverse Engineering and Unauthorized Code Modification Reverse Engineering and Code Modification Prevention [23].

Para que los controles y requisitos incluidos dentro de esta categoría sean efectivos, la aplicación deberá cumplir al menos todos los requisitos incluidos en el nivel de verificación básico (i.e. MASVS-L1), de la misma manera que todos los controles con menor numeración dentro de esta categoría. Así pues, por ejemplo cualquiera de los controles incluidos dentro de la sub-categoría Impede Comprehension se deberán combinar con los controles incluidos en las sub-categorías Impede Dynamic Analysis and Tampering y Device Binding.

Las medidas de seguridad descritas en cada uno de estos requisitos se recomiendan para los tipos de aplicaciones que procesen o proporcionen acceso a información o funcionalidades confidenciales. La no implementación de alguna de las protecciones incluidas en los requisitos de esta categoría no conlleva que la aplicación tenga automáticamente algún tipo de vulnerabilidad asociada, ya que estas protecciones no están pensadas para sustituir a los controles incluidos en los niveles de seguridad MASVS-L1 y MASVS-L2. Los controles incluidos dentro de MASV-R están pensados para añadir medidas de seguridad y controles adicionales en aplicaciones que ya cumplen con los requisitos de seguridad básicos del MASVS.

MASVS-R

Impede Dynamic Analysis and Tampering (Impedir el Análisis Dinámico y Modificación de la Aplicación)

8.1 La aplicación detecta y responde a la presencia de un dispositivo rooted o jailbroken alertando al usuario o finalizando la aplicación.

De cara a mitigar el uso de algunos tipos de herramientas utilizadas en los procesos de ingeniería inversa, se deberán añadir mecanismos de seguridad que permitan detectar cuando un dispositivo está jailbroken. Si bien este tipo de mecanismos de seguridad denominados como "jailbreak detection" no son muy efectivos por si solos, añadir algunos de estos mecanismos dentro de la aplicación si que pueden ayudar a mejorar el proceso de anti-tampering implementando en la misma. A continuación se enumeran una serie de técnicas extraídas de la MSTG [2], las cuales se pueden utilizar para llevar a cabo la detección de dispositivos jailbroken.

- Comprobaciones Basadas en Ficheros.- Esta técnica consistirá en comprobar si existe alguno de los ficheros y directorios asociados normalmente a las técnicas de jailbreak.

```
/Applications/Cydia.app
/Applications/FakeCarrier.app
/Applications/Icy.app
/Applications/IntelliScreen.app
/Applications/MxTube.app
/Applications/RockApp.app
/Applications/SBSettings.app
/Applications/WinterBoard.app
/Applications/blackra1n.app
/Library/MobileSubstrate/DynamicLibraries/LiveClock.plist
/Library/MobileSubstrate/DynamicLibraries/Veency.plist
/Library/MobileSubstrate/MobileSubstrate.dylib
/System/Library/LaunchDaemons/com.ikey.bbot.plist
/System/Library/LaunchDaemons/com.saurik.Cydia.Startup.plist
/bin/bash
/bin/sh
/etc/apt
/etc/ssh/sshd_config
/private/var/lib/apt
/private/var/lib/cydia
/private/var/mobile/Library/SBSettings/Themes
/private/var/stash
/private/var/tmp/cydia.log
/usr/bin/sshd
/usr/libexec/sftp-server
/usr/libexec/ssh-keysign
/usr/sbin/sshd
/var/cache/apt
/var/lib/apt
/var/lib/cydia
```

- Comprobaciones Basadas en Permisos de Ficheros.- Otro tipo de comprobación puede ser el intentar crear un fichero fuera del sandbox de la aplicación. Si el fichero se crea correctamente entonces se podrá inferir que el dispositivo está jailbroken. A continuación se muestra un snippet de código en Swift que muestra como llevar a cabo esta comprobación.

```
let stringToWrite = "Jailbreak Test"

do {
    try stringToWrite.write(toFile:"/private/JailbreakTest.txt",
atomically:true,  encoding:String.Encoding.utf8)

    // Device is jailbroken
    return true
} catch {
    // Device is not jailbroken
    return false
}
```

- Comprobaciones Basadas en Controladores de Protocolo.- Esta técnica consiste en intentar abrir la App Store de Cydia, aplicación que se instala por defecto en casi todas las versiones de jailbreak. Esta aplicación registra URL scheme cydia://, de tal manera que si está registrado el mismo, es muy probable que el dispositivo esté jailbroken. En el siguiente snippet de código Swift se muestra como se puede llevar a cabo este tipo de verificación.

```
if UIApplication.shared.canOpenURL(URL(string:"cydia://package/
com.example.package")!) {
    // Device is jailbroken
    return true
} else {
    // Device is not jailbroken
    return false
}
```

- Llamada a las APIs del Sistema.- Esta técnica consiste en llamar a una función del sistema que no se pueda ejecutar debido a las restricciones del sandbox.

  Por ejemplo, si se llama a la función system() con un argumento NULL, en un dispositivo sin jailbreak devolverá un 0. Si se realiza la misma llamada en un dispositivo jailbroken se devolverá un 1. Esto es así porque la función comprobará si se puede acceder a /bin/sh, el cual sólo es accesible en los dispositivos jailbroken.

  Otro ejemplo podría ser el llamar a la función fork(), de tal manera que si el identificador del proceso (PID) devuelto por la misma no es válido, se podría deducir que la aplicación está funcionando en un dispositivo sin jailbreak. Si se devuelve un PID válido, esto significará que se ha creado un nuevo proceso satisfactoriamente, por lo cual se podría inferir que la aplicación está funcionando fuera de su sandbox y por lo tanto el dispositivo está jailbroken.

En el Anexo [7] se muestra una utilidad que incluye dos funciones desarrolladas en C que permiten detectar y terminar una aplicación si el dispositivo está jailbroken. Estas funciones se han implementado basándose en la técnica de llevar a cabo llamadas a las APIs del sistema descrita anteriormente.

8.2 La aplicación evita la depuración y/o detecta, y responde, a un depurador adjunto. Todos los protocolos de depuración disponibles deben estar cubiertos.

Una de las técnicas más útiles dentro de los procesos de ingeniería inversa es el uso de depuradores o debuggers. El uso de este tipo de herramientas permite a los atacantes no sólo tracear el valor de las variables críticas de la aplicación, sino también leer y modificar la memoria de la misma en tiempo de ejecución.

Puesto que este tipo de técnicas son tremendamente útiles en los procesos de ingeniería inversa, los desarrolladores de aplicaciones deberán implementar medidas de seguridad que permitan verificar primero la presencia de un depurador adjunto a la ejecución de la aplicación y luego tomar las medidas oportunas si se verificara este hecho (e.g. terminar la aplicación).

A continuación se muestran algunas de las técnicas de anti-debugging más utilizadas dentro de las aplicaciones móviles para iOS. Este conjunto de técnicas se han extraído de la MSTG [2].

- Utilizar ptrace.- El sistema operativo iOS se ejecuta en un XNU kernel, el cual implementa una llamada al sistema de tipo ptrace, siendo esta implementación menos potente que las implementaciones de Unix o Linux. El XNU kernel expone otra interfaz a través de Mach IPC para permitir la depuración. La implementación de iOS de ptrace proporciona una característica importante: evitar que se depure un proceso. Esta característica se implementa como la opción PT_DENY_ATTACH de la llamada al sistema ptrace. El uso de PT_DENY_ATTACH es una de las técnicas de anti-debugging más conocida, por lo que es muy posible se encuentre durante los pentests en iOS.

  El uso de ptrace con PT_DENY_ATTACH garantiza que ningún otro depurador pueda conectarse al proceso de llamada, de tal manera que si un depurador intenta adjuntarse, el proceso terminará automáticamente. Puesto que ptrace no forma parte de las APIs públicas de iOS, y este tipo de APIs no están permitidas según las guidelines del App Store, para llevar a cabo esta llamada se deberá de hacer indirectamente a través del dlsym.

  A continuación se muestra un snippet de código que incluye una implementación de esta técnica en C.

```
#import <dlfcn.h>
#import <sys/types.h>
#import <stdio.h>

typedef int (*ptrace_ptr_t)(int _request, pid_t _pid, caddr_t _addr, int
_data);

void anti_debug() {
    ptrace_ptr_t ptrace_ptr = (ptrace_ptr_t)dlsym(RTLD_SELF, "ptrace");
    ptrace_ptr(31, 0, 0, 0); // PTRACE_DENY_ATTACH = 31
}
```

- Utilizar systcl.- Otro tipo de técnica anti-debugging se basa en utilizar la llamada al sistema systcl, la cual permite obtener información del sistema y permite a los procesos con privilegios suficientes el modificar cierta información del sistema.

  Esta función también se puede utilizar para obtener la información sobre el proceso actual, de tal manera que se puede determinar si el proceso está siendo depurado. Básicamente, la idea de esta técnica es llevar a cabo una llamada al sysctl para obtener la información del proceso en cuestión y tras realizar la llamada determinar si el flag P_TRACED es distinto de 0.

  En el Anexo [8] se muestra una utilidad que incluye una función desarrollada en C que permite detectar y terminar una aplicación si se está utilizando un depurador o debugger utilizando para ello la técnica descrita en este apartado.

8.3 La aplicación detecta y responde a la manipulación de los archivos ejecutables y datos críticos dentro de su propio sandbox.

El objetivo de este requisito de seguridad es implementar una serie de controles que permitan verificar en tiempo de ejecución que no se ha llevado a cabo ningún tipo de modificación del código de la aplicación por parte de un atacante (i.e. binary patching). A su vez, también se deberá verificar la integridad de los datos almacenados en cualquiera de los mecanismos de almacenamiento local proporcionados por el sistema operativo (e.g. Keychain, NSUserDefaults, SQLite Databases, etc).

A continuación se muestran los dos tipos de comprobaciones que se deberían implementar para verificar el correcto cumplimiento de este requisito.

- Integridad del Código Fuente de la Aplicación.- En lo que respecta a la integridad del código fuente de la aplicación, comentar que Apple ya implementa un sistema de seguridad utilizando su propio DRM. En un principio, si los usuarios sólo instalaran aplicaciones a través del App Store no debería existir ningún tipo de problema o vulnerabilidad debido a una potencial manipulación o modificación de una aplicación oficial. El problema surge cuando los usuarios disponen de dispositivos jailbroken y por lo tanto pueden instalar aplicaciones procedentes de otras fuentes. En este caso si que existe un riesgo real de instalar aplicaciones cuyo código fuente haya sido manipulado y por lo tanto exista un riesgo potencial para la seguridad de los mismos.

  En el Anexo [9] se muestra una utilidad que permite verificar en tiempo de ejecución la firma del código fuente de la aplicación, comparándola con otra firma generada previamente, de tal manera que se pueda verificar si se ha producido algún tipo de modificación en el binario de la aplicación.

- Integridad de los Datos de la Aplicación.- En lo que respecta a la integridad de los datos almacenados por la aplicación, se deberá generar un HMAC o firma de cualquier dato almacenado en la aplicación. En el caso de tener la necesidad de encriptar los datos almacenados por la aplicación, se deberá generar el HMAC o firma de dichos datos una vez llevado a cabo el proceso de encriptación.

  De cara a llevar el proceso de generación de un HMAC con CC, se deberán seguir los siguientes pasos.

  1. Obtener los datos a través de un objeto de tipo NSMutableData.
  2. Obtener la clave probablemente del Keychain.
  3. Calcular el valor del hash.
  4. Añadir el valor del hash a los datos actuales.
  5. Almacenar los datos resultantes del punto anterior.

  De cara a verificar el HMAC con CC, se deberán seguir los siguientes pasos.

  1. Obtener el mensaje y los bytes del HMAC como un objeto NSData separado.
  2. Se deberán repetir los pasos del 1 al 3 del proceso de generación del HMAC.
  3. Se deberá comparar el resultado del HMAC obtenido a el resultado del paso 1.

  A continuación se muestra un primer ejemplo extraído del MSTG [2], en el cual se muestra el proceso de generación de un HMAC en Objective-C.

```
// Get the actual data
NSMutableData* actualData = [getData];

// Get the key from the keychain
NSData* key = [getKey];

// Allocate a buffer to hold the digest
NSMutableData* digestBuffer = [NSMutableData
dataWithLength:CC_SHA256_DIGEST_LENGTH];

// Generate the digest
CCHmac(kCCHmacAlgSHA256, [actualData bytes], (CC_LONG)[key length],
[actualData
     bytes], (CC_LONG)[actualData length], [digestBuffer mutableBytes]);

// Append the HMAC to the actual data
[actualData appendData: digestBuffer];
```

En este segundo ejemplo desarrollado también en Objective-C, se muestra el proceso de verificación de un HMAC generado a través del ejemplo anterior.

```
// Ge the HMAC bytes
NSData* hmac = [data subdataWithRange:NSMakeRange(data.length -
CC_SHA256_DIGEST_LENGTH, CC_SHA256_DIGEST_LENGTH)];

// Get the actual data bytes
NSData* actualData = [data subdataWithRange:NSMakeRange(0, (data.length -
hmac.length))];

// Allocate a buffer to hold the digest
NSMutableData* digestBuffer = [NSMutableData
dataWithLength:CC_SHA256_DIGEST_LENGTH];

// Generate the digest
CCHmac(kCCHmacAlgSHA256, [actualData bytes], (CC_LONG)[key length],
[actualData bytes], (CC_LONG)[actualData length], [digestBuffer
mutableBytes]);

// Compare the digest with the retrieved HMAC
return [hmac isEqual: digestBuffer];
```

8.4 La aplicación detecta y responde a la presencia de los frameworks y herramientas de ingeniería inversa más comunes en el dispositivo.

Este requisito tiene como objetivo detectar algunas de las herramientas y frameworks más utilizados en los procesos de ingeniería inversa, de tal manera que si se detecta su presencia se puedan tomar las medidas adecuadas (e.g. terminar la aplicación). Algunas de estas herramientas más populares son Cycript, Hopper, Frida, Cydia Substrate, etc.

Una de las técnicas más populares para detectar la presencia de este tipo de herramientas consiste en buscar nombres de imágenes o firmas conocidas dentro de los módulos cargados en la memoria de la aplicación. En el siguiente ejemplo se muestra una implementación en C de este mecanismo, el cual itera sobre cada una de las imágenes cargadas en memoria y comprueba si el nombre de alguna de éstas contiene el nombre de algunas de las librerías utilizadas por estas

herramientas incluidas en una blacklist. En este caso se está verificando que no se estén utilizando las herramientas Cycript y Cydia Substrate.

```
inline void checkDylibInjected() attribute((always_inline));

void checkDylibInjected() {
    uint32_t count = _dyld_image_count();

    // Examples of malicious dylibs
    char* blacklist[] = { "Substrate", "cycript" };

    for(uint32_t i=0; i<count; i++) {
        const char *dyld = _dyld_get_image_name(i);
        int length = strlen(dyld);
        int j;

        for(j=length-1; j>= 0; --j)
            if(dyld[j] == '/')
                break;

        char *name = strndup(dyld + ++j, length - j);

        for(int x=0; x<sizeof(blacklist)/sizeof(char*); x++)
            if(strstr(name, blacklist[x]) || strstr(dyld, blacklist[x]))
                // malicious dylib loaded!

        free(name);
    }
}
```

8.5 La aplicación detecta y responde cuando se ejecuta en un emulador.

El uso de emuladores o simuladores puede ser de gran utilidad de cara a llevar a cabo procesos de ingeniería inversa, ya que éstos permiten obtener una gran cantidad de información sobre los diferentes estados de la aplicación en tiempo de ejecución (e.g. variables, pila de llamadas, contenido de la memoria, etc). Como es evidente, esta información puede ser de gran ayuda de cara a entender el funcionamiento interno de la aplicación.

Así pues, el objetivo de este requisito es evitar que se pueda llevar a cabo la ejecución de una aplicación dentro de un emulador, de tal manera que un atacante potencial no pueda utilizar este tipo de herramienta en un proceso de ingeniería inversa.

A continuación se muestra un snippet de código en Objective-C que implementa un método que comprueba si una aplicación está funcionando dentro de un simulador.

```
- (BOOL)isAppRunningOnSimulator {
    #if TARGET_IPHONE_SIMULATOR
        return YES;
    #else
        return NO;
    #endif
}
```

La versión Swift del snippet anterior quedaría de la siguiente manera.

```
func isAppRunningOnSimulator() -> Bool {
    // Use this line in Xcode 7 or newer
    return TARGET_OS_SIMULATOR != 0

    // Use this line in Xcode 6
    // return TARGET_IPHONE_SIMULATOR != 0
}
```

8.6 La aplicación detecta y responde a la alteración del código y los datos en su propio espacio de memoria.

El objetivo de este requisito de seguridad es implementar una serie de controles que permitan verificar en tiempo de ejecución que no se ha llevado a cabo ningún tipo de modificación del código y/o datos de la aplicación en el espacio de memoria del proceso por parte de un atacante.

Un tipo de técnica muy común es la denominada como code injection o inyección de código, la cual permite explorar y modificar el espacio de memoria de un proceso en tiempo de ejecución. La inyección se puede implementar de varias maneras, pero normalmente no es necesario conocer todos los detalles gracias a la gran cantidad de herramientas que automatizan este tipo de procesos. Estas herramientas proporcionan acceso directo a la memoria del proceso y a las estructuras de datos más importantes, tales como los objetos instanciados por la aplicación. Estas herramientas proporcionan una gran variedad de funcionalidades que son realmente útiles de cara a determinar las librerías cargadas, hooking methods y funciones nativas. La manipulación de la memoria del proceso es más difícil de detectar que el file patching (ver requisito 8.3), por lo que es el método preferido en la gran mayoría de los casos.

Substrate, Frida y XPosed son algunos de los frameworks más populares para llevar a cabo técnicas tales como el hooking y la inyección de código. Estas tres herramientas o frameworks difieren en su filosofía de diseño y detalles de implementación. Substrate y Xposed se centran en la inyección de código y/o hooking, mientras que Frida se diseñó con el objetivo de ser un framework de instrumentación dinámica completo, incorporando funcionalidades como son la inyección de código, language bindings, una VM de JavaScript inyectable y una consola.

Entre las técnicas más populares de inyección de código en Objective-C y Swift podríamos destacar las siguientes:

- Swizzling methods en tiempo de ejecución
- Swizzling virtual Swift methods analizando el ejecutable y modificando los bits adecuados
- Modificando los call targets
- Swizzling imported symbols cambiando los symbol stub targets
- Utilizar dyld para forzar la carga de librerías o cambiar las librerías cargadas por la aplicación
- Reemplazar las librerías compartidas que linkan con la aplicación

En principio se podría afirmar que inyectar código en una aplicación Swift es más difícil de lo que es en una aplicación Objective-C, pero aún así es posible. La mayoría de las técnicas expuestas anteriormente son independientes del lenguaje, lo que significa que ningún lenguaje de programación hará más segura una aplicación. Comentar que no hay una forma 100% efectiva de evitar cualquiera de estas técnicas en un entorno controlado completamente por un atacante potencial.

Entre las medidas de seguridad que se podrían implementar podríamos destacar las encaminadas a detectar el uso de herramientas de ingeniería inversa dentro de la aplicación (ver

requisito 8.4), técnicas de detección de debuggers o depurados (ver requisito 8.2) y sobre todo técnicas que permitan la detección de dispositivos jailbroken (ver requisito 8.1).

8.7 La aplicación implementa múltiples mecanismos en cada una de las categorías de defensas expuestas en los requisitos expuestos en esta categoría (8.1 a 8.6). Se debe tener en cuenta que la resistencia se escala con la cantidad, diversidad y originalidad de los mecanismos utilizados.

Este requisito tiene como objetivo resumir un poco lo que sería la implementación de mecanismos de defensa ante técnicas de ingeniería inversa y tampering. De cara a llevar la implementación de este tipo de mecanismos, la idea es utilizar una combinación de técnicas que permitan mitigar cada uno de los tipos amenazas existentes dependiendo de la naturaleza y nivel de protección requerido por la aplicación.

Así pues, si por ejemplo se está desarrollando una aplicación que no guarda ningún tipo de información en el dispositivo no tendrá sentido implementar ningún tipo de medida de seguridad que verifique que los datos de la aplicación no han sido modificados por un atacante potencial. De la misma manera, si se está desarrollando una aplicación que muestra información pública y no maneja datos confidenciales (e.g. datos de usuarios, datos corporativos, etc), probablemente no tendrá sentido implementar mecanismos para detectar si el dispositivo está jailbroken (e.g. un periódico).

Dicho esto, el verificar el correcto cumplimiento de este requisito sería tan sencillo como comprobar que se han implementado los mecanismos de seguridad adecuados, teniendo en cuenta el modelo de amenazas y riesgos de la aplicación (ver requisitos del 8.1 al 8.6).

8.8 Los mecanismos de detección desencadenan respuestas de diferentes tipos, incluidas las respuestas diferidas y silenciosas.

Este requisito tiene como objetivo el verificar que los mecanismos de repuesta ante la detección de procesos de ingeniería inversa sean los adecuados. Hasta ahora, en cada uno de los requisitos de esta categoría se han incluido diferentes mecanismos de seguridad para llevar a cabo la detección de amenazas relacionadas con procesos de ingeniería inversa y/o tampering. Cada uno de estos mecanismos de detección debería tener asociada una respuesta ante la detección de un ataque sobre la aplicación, de tal manera que ésta sea lo más adecuada de cara a evitar posibles contramedidas o bypasses por parte de un atacante potencial.

Entre las respuestas más comunes ante este tipo de ataques se podrían destacar las siguientes.

- La terminación de la aplicación de manera silenciosa
- La terminación de la aplicación con previa notificación al usuario
- Borrado de los datos de la aplicación
- Notificar al desarrollador de manera silenciosa
- Forzar un fallo o error dentro de la aplicación

Tal y como se ha comentado anteriormente, si estos mecanismos de respuesta no se implementan adecuadamente, un atacante puede saltárselos de manera sencilla tras realizar un proceso de ingeniería inversa sobre los mismos. Una de las reglas de oro en la implementación de estos mecanismos de respuesta es que deben revelar la menor información posible sobre la implementación de los mismos así como sobre los mecanismos de detección. Si por ejemplo se implementara un mecanismo de respuesta tan simple como terminar la aplicación justo después de llevar a cabo la detección de la amenaza, un atacante potencial podría localizar fácilmente la posición dentro del código donde se está llevando a cabo la respuesta, y por lo tanto podría intentar llevar a cabo algún tipo de modificación o tampering en el código de la aplicación que inutilizará al mismo.

A continuación se enumeran algunas de las características más importantes que deben de tener estos mecanismos de respuesta.

- Debe existir una separación espacial entre el mecanismo de respuesta y sus efectos. Así pues, se debe llevar a cabo la respuesta en una parte de la aplicación pero sus efectos deberán aparecer en otras partes de la misma.

- Debe existir una separación temporal entre el mecanismo de respuesta y sus efectos. Por ejemplo se podrían insertar delays entre el inicio de la respuesta y sus efectos. De esta manera se conseguirá desbaratar los procesos de ingeniería inversa detectados y los atacantes tendrán más complicado el localizar estos mecanismos de respuesta.

Así pues, para verificar el correcto cumplimiento de este requisito se debería comprobar que los mecanismos de respuesta son difícilmente detectables por un atacante potencial introduciendo retrasos dentro de la ejecución de los mismos, introduciendo técnicas de probabilidad para que sus efectos sólo se produzcan en algunas ocasiones, utilizando mecanismos de fallo silenciosos (e.g. en vez de terminar la aplicación se podría producir una excepción tal como una división entre cero), etc.

8.9 La ofuscación se aplica a las defensas programáticas, que a su vez impiden la de-ofuscación a través del análisis dinámico.

El objetivo de este requisito es verificar que se aplican técnicas de ofuscación en el código fuente desarrollado en cada uno de los mecanismos de seguridad implementados dentro de la aplicación. Este proceso de ofuscación tiene como objetivo el dificultar que se lleven a cabo mecanismos de ingeniería inversa sobre las propias medidas de seguridad implementadas. Durante estos procesos de ingeniería inversa, un atacante podría llegar a identificar y modificar en tiempo de ejecución el código fuente de estos mecanismos de seguridad con el objetivo de desactivarlos y poder llevar a cabo el proceso de ingeniería inversa satisfactoriamente.

Un claro ejemplo de este tipo de ofuscación podría ser el no utilizar nombres de funciones y métodos que sean descriptivos. Por ejemplo, si un atacante encontrara una función con el nombre isJailbrokenDevice, podría reemplazar su implementación en tiempo de ejecución a través de algunas de las técnicas descritas en el requisito 8.6, de tal manera que se propiciará un fallo en el mecanismo de detección de un dispositivo jailbroken. Así pues, de cara a dificultar este proceso se podrían utilizar nombres de métodos y funciones menos descriptivos. De esta manera se dificultará que un atacante potencial pueda encontrar e identificar fácilmente los métodos y funciones que implementan los mecanismos de seguridad dentro de la aplicación. Este mismo ejemplo se puede extrapolar a los nombres de las clases, variables, constantes, etc utilizados en la implementación de estos mecanismos de seguridad.

Otro ejemplo podría ser la ofuscación de las cadenas de caracteres hardcodeadas en la aplicación. Por ejemplo en el mecanismo de defensa descrito en el requisito 8.4 se puede ver que se han definido dos cadenas de texto dentro de un array de cara a poder identificar el uso de las herramientas de ingeniería inversa Cydia Substrate y Cycript. De cara a saltarse el mecanismo de seguridad implementado en dicho requisito, sería tan sencillo como que un atacante reemplazara estas cadenas en el espacio de memoria de la aplicación por otras cualquiera. Para ofuscar este tipo de cadenas se podrían utilizar técnicas como por ejemplo el Binary XOR (ver requisito 3.1).

Device Binding (Enlace con el Dispositivo)

8.10 La aplicación implementa una funcionalidad de 'device binding' utilizando una huella dactilar registrada en el dispositivo derivada de múltiples propiedades exclusivas del mismo.

Tal y como se recoge en el MSTG [2], el objetivo del device binding es impedir que un atacante potencial pueda copiar una aplicación desde un dispositivo A a otro dispositivo B, y pueda continuar la ejecución de la misma en el dispositivo B. La idea de este mecanismo de seguridad es evitar que se pueda transferir el estado de una aplicación junto con la misma, entre dos dispositivos diferentes. Por ejemplo, si un usuario se ha autenticado en una aplicación instalada en un dispositivo A, al transferir esta aplicación a un dispositivo B no debería poder continuar la ejecución de la misma en este estado (i.e. debería autenticarse de nuevo en el dispositivo B).

De cara a verificar el correcto cumplimiento de este requisito, se deberá verificar que una aplicación copiada desde un dispositivo A un dispositivo B no continua en el mismo estado de ejecución en el que estaba en el dispositivo A. Para llevar a cabo un análisis estático del código fuente de la aplicación que pueda identificar malas prácticas en la implementación de este mecanismo de seguridad, se podrían seguir las siguientes pautas.

- MAC Address.- A partir de iOS 7, el sistema siempre devuelve el valor fijo 02:00:00:00:00:00 cuando se intenta obtener la dirección MAC del dispositivo. Así pues, este parámetro no debería utilizarse para llevar a cabo el device binding. Para verificar que no se esté utilizando esta dirección se podría buscar dentro del código las funciones CTL_NET y NET_RT_IFLIST, o buscar la cadena de formato %x:%x:%x:%x:%x:%x.

- UDID.- A partir de iOS 6, el UDID no está disponible por motivos de seguridad y privacidad de los usuarios. Así pues, la función [[UIDevice currentDevice]uniqueIdentifier] quedó deprecada, de tal manera que siempre devuelve el valor fijo FFFFFFFF. Debido a esto, este parámetro tampoco debería utilizarse para llevar a cabo el device binding.

- Keychain o Filesystem.- Para llevar a cabo el device binding no debería utilizarse ningún dato almacenado en el Keychain o el sistema de ficheros sino está protegido por algún tipo de SecAccessControlCreateFlags o se utilizan las protection classes kSecAttrAccessibleAlways o kSecAttrAccessibleAlwaysThisDeviceOnly.

Una vez expuestas algunas de las malas prácticas más comunes en la implementación de este tipo de mecanismo de seguridad, a continuación se expondrán algunos de los mecanismos que podrían considerarse como válidos para llevar a cabo el device binding en iOS.

- Vendor ID (IDFV).- Este identificador es único para cada una de las instalaciones de una aplicación o aplicaciones de un mismo desarrollador en un dispositivo dado. Así pues, se podría utilizar el mismo para llevar a cabo el device binding en un dispositivo con iOS 6 y posteriores. A continuación se muestra un snippet de código en Objective-C donde se muestra como obtener este identificador.

```
UIDevice *currentDevice = [UIDevice currentDevice];
NSString *vendorId = [[currentDevice identifierForVendor] UUIDString];
```

El mismo ejemplo en Swift quedaría de la siguiente manera.

```
let vendorId = UIDevice.currentDevice().identifierForVendor?.UUIDString
```

- Otra manera de llevar a cabo el device binding consistiría en almacenar un elemento en el Keychain para identificar una misma instancia de la aplicación. Este elemento podría ser un

identificador único universal o UUID generado dentro de la aplicación, o cualquier otro elemento que identifique a la aplicación unívocamente. Para evitar que este elemento sea accesible sino se ha configurado un passcode o el TouchID y además no se incluya dentro de las copias de seguridad del dispositivo se debería utilizar la clase de protección kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly. También se podría utilizar la clase de protección kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly para evitar el acceso sino se ha desbloqueado el dispositivo por primera vez y además no se incluya en las copias de seguridad del dispositivo.

A continuación se muestra un ejemplo en Objective-C que muestra como generar un UUID dentro de una aplicación.

```
NSString *uuid = [[NSUUID UUID] UUIDString];
```

El mismo ejemplo en Swift quedaría de la siguiente manera.

```
let uuid = UUID().uuidString
```

En el Anexo [10] se muestra una utilidad desarrollada en Objective-C que permite generar y almacenar un UUID en el Keychain, así como recuperarlo del mismo.

- Google Instance ID.- Otra opción sería el utilizar esta herramienta desarrollada por Google. Se puede encontrar más información sobre la misma en la siguiente URL.

  https://developers.google.com/instance-id/guides/ios-implementation

Impede Comprehension (Impedir la Comprensión del Código y Datos de la Aplicación)

8.11 Todos los archivos ejecutables y las librerías que pertenecen a la aplicación están encriptados en el nivel de archivo y/o código importante y los segmentos de datos dentro de los archivos ejecutables están encriptados o empaquetados. El análisis estático trivial no revela código o datos importantes.

Toda aplicación enviada a la App Store de Apple está preparada para su descarga de una manera confidencial conocida solo por Apple. Cuando un usuario descarga una aplicación, se encripta una versión única de la aplicación para cada dispositivo en el que el usuario decide instalar la aplicación. La idea es garantizar que esta aplicación no se ejecute en un dispositivo donde no deba estar. Apple aplica un enfoque de seguridad similar a las descargas de video y música desde iTunes, donde el nombre del sistema de encriptación se llama Apple Fairplay DRM.

Cuando un usuario descarga una aplicación del App Store en un Mac, ésta se almacena en el sistema de archivos de la computadora. Estas aplicaciones nativas se almacenan localmente con una extensión de archivo .ipa (lo que indica que son archivos de archivos de aplicaciones de iOS). Este archivo no es más que un archivo comprimido que contiene meta-archivos y una carpeta Payload donde se almacena la aplicación de forma muy similar a las aplicaciones macOS (indicadas con la extensión .app).

Dentro del directorio .app, se almacenan todos los recursos de la aplicación, tales como las listas de propiedades (.plist), storyboards, imágenes, sonidos, scripts, etc. Además de los archivos de recursos, hay un archivo adicional en cada .app denominado el archivo binario ejecutable. Este archivo es un archivo binario sin extensión explícita, el cual se ajusta al formato de archivo ejecutable de Mach-O.

En términos generales, los binarios de Mach-O se pueden dividir en dos grupos: universales y para arquitectura específica, siendo ambos utilizados en iOS. Las arquitecturas de CPU de los dispositivos iOS son armv6, armv7, armv7s y arm64. Si se desea construir una aplicación que funcione tanto en el iPhone más nuevo como en uno antiguo, entonces la aplicación debe compilarse para diferentes arquitecturas, lo que da como resultado un archivo universalmente ejecutable. Esos binarios multiarquitectura también se conocen como “fat binary" porque contienen su código compilado varias veces para diferentes arquitecturas (“fattened"). Los binarios universales se pueden ver como colecciones de binarios de arquitectura única. Cada binario de arquitectura única tiene tres partes principales.

- Mach-O Header
- Load Commands
- Datos

La primera parte contiene meta-información, información de cifrado y varios offsets. La segunda parte contiene las definiciones de todas las funciones dentro de la aplicación y la tercera parte contiene las implementaciones de esas funciones, entre otros recursos.

Los archivos de Mach-O pueden ser encriptados, siendo este el caso de los archivos ejecutables de las aplicaciones distribuidas a través de la App Store. Este cifrado solo se puede descifrar en el dispositivo donde se descargó la aplicación a nivel de hardware. Este descifrado se realiza solo cuando el archivo ejecutable es cargado por el procesador en la RAM, de tal manera que todo el proceso de descifrado se lleva a cabo a nivel de hardware. Por esta razón es muy difícil crear herramientas de descifrado offline. La única forma de descifrar los datos binarios encriptados es en un dispositivo jailbroken a través de algunas herramientas especiales.

Un atacante potencial encontrará dificultades adicionales incluso después de conseguir llevar a cabo un descifrado exitoso del binario. La razón de esto, es que este binario contiene la lógica de la aplicación en código máquina, el cual no es fácilmente legible para el ser humano. Decompiladores como IDA y Hopper pueden ayudar presentando la lógica de la aplicación en ensamblador y mostrar información básica sobre los nombres de las clases y métodos. Después de un análisis en profundidad de este código, se podría llegar a dibujar un diagrama que ilustre la lógica de dicha aplicación. Comentar que esta es una tarea realmente complicada, ya que leer el código ensamblador de una aplicación compleja es extremadamente difícil.

En resumen, podríamos concluir que este requisito se cumplirá automáticamente siempre y cuando la aplicación se distribuya a través del App Store, debido a las medidas de seguridad implementadas por Apple en el proceso de distribución de aplicaciones a través de la misma.

8.12 Si el objetivo de la ofuscación es proteger las funcionalidades confidenciales, se utiliza un esquema de ofuscación que es apropiado para la tarea en particular y robusto contra los métodos de de-ofuscación automáticos y manuales, teniendo en cuenta las investigaciones publicadas actualmente. La eficacia del esquema de ofuscación debe verificarse mediante pruebas manuales. Se debe tener en cuenta que las características de aislamiento basadas en hardware son preferibles a la ofuscación siempre que sea posible.

Tal y como se explica en el requisito 7.8, tanto Objective-C como Swift son lenguaje compilados, y el compilador Clang está altamente optimizado. En las aplicaciones construidas en modo release se eliminan los símbolos de depuración y se lleva a cabo un proceso de optimización del código, mecanismos que ya por si mismos suponen un algo grado de ofuscación. Si que es cierto que existen decompiladores para lenguajes compilados, pero los resultados son realmente horribles y difíciles de interpretar. La protección del código fuente de la aplicación a través de procesos de encriptación se lleva a cabo a través del propio ecosistema de distribución de aplicaciones proporcionados por Apple (ver requisito 8.11)

Así pues, se puede afirmar que no existe una herramienta de ofuscación oficial para los lenguajes de programación utilizados en el desarrollo de aplicaciones para iOS. Si bien Apple no proporciona este tipo de herramientas de manera oficial, si que existen herramientas Open Source (e.g. PPiOS-Rename) y comerciales (e.g. iXGuard) que permiten llevar a cabo este tipo de procesos.

https://github.com/preemptive/PPiOS-Rename

https://www.guardsquare.com/en/ixguard

Este tipo de herramientas basan su funcionamiento en llevar a cabo el renombrado de los nombres de las clases, protocolos, propiedades y métodos. De esta manera se dificulta la legibilidad e interpretación del código fuente de la aplicación al llevar a cabo procesos de ingeniería inversa.

De cara a implementar este tipo de ofuscación de forma manual en algunas partes del código de la aplicación, se podría utilizar un mecanismo como el que se muestra en el siguiente ejemplo implementado en Objective-C.

```
#ifndef DEBUG
#define MyClass frew
#define myMethod hytr
#endif

@interface MyClass : NSObject {
}

- (void)myMethod;
```

Este tipo de mecanismo de ofuscación manual podría ser de gran utilidad por ejemplo de cara a llevar a cabo del proceso de ofuscación en las defensas programáticas implementadas en la aplicación (ver requisito 8.9).

6. Conclusiones

Una vez finalizado este Trabajo Final de Máster, tras realizar una retrospectiva sobre el trabajo desarrollado, se puede concluir que se han alcanzado los objetivos establecidos.

Tal y como se describe en los objetivos del trabajo, la idea era crear un manual de referencia o guía para llevar a cabo el desarrollo de aplicaciones móviles seguras en iOS. El contenido de este manual de referencia se debería crear desde el punto de vista del desarrollador, de tal manera que cualquiera que desee desarrollar una aplicación segura en iOS pueda utilizarlo como guía de referencia.

Otro de los objetivos era que este manual fuera auto-contenido, es decir que aun a pesar de incluir bastantes referencias y enlaces, incluya todo el contenido necesario para dar solución por si mismo a cada uno de los requisitos de seguridad analizados. El cumplimiento de este objetivo ha tenido como resultado que quizás la memoria del trabajo tenga una extensión un poco mayor de la requerida en este tipo de trabajos.

Una de las peculiaridades de este Trabajo Final de Máster es que la memoria no sintetiza el trabajo desarrollado, sino que el propio trabajo en sí se recoge en la misma. Esto significa que el manual de referencia o guía de desarrollo de aplicaciones seguras para iOS se puede considerar la sección 5 de esta memoria junto con los anexos de código fuente incluidos en el entregable del trabajo y referenciados a lo largo de dicha sección. Este punto es bastante diferente a la gran mayoría de trabajos similares, ya que en los mismos se utiliza la memoria para explicar y sintetizar algún tipo de desarrollo, mostrar los resultados de una investigación, etc.

De cara a desarrollar el trabajo se ha utilizado como base el MASVS [1], de tal manera que se han analizado todos y cada uno de los requisitos de seguridad definidos en cada una de las categorías de dicho estándar. Para verificar el correcto cumplimiento de estos requisitos de seguridad, así como para proporcionar soluciones técnicas a los mismos, se ha seguido principalmente el contenido del MSTG [2], junto con otra serie de documentos, artículos y guías de desarrollo de aplicaciones seguras. Tal y como puede observarse a lo largo de la memoria, se han incluido una gran cantidad y variedad de snippets de código en Objective-C y Swift, que pueden ser de gran utilidad para cualquier desarrollador de aplicaciones para iOS. También se han creado una serie de anexos referenciados a lo largo de la memoria, los cuales incluyen otra serie de ejemplos de código fuente más extensos.

En lo que respecta a la originalidad y utilidad real del trabajo, comentar que como desarrollador de aplicaciones móviles con más de 14 años de experiencia en el sector, hasta la fecha no he encontrado un sólo manual de referencia similar al desarrollado en este trabajo. Si que es cierto que por un lado existen ciertos estándares como el MASVS para tratar de dar solución al problema del desarrollo de aplicaciones móviles seguras, y por otro lado existen una gran cantidad de documentos y guías de desarrollo de aplicaciones seguras, pero lo que no existía hasta este momento es un documento donde se relacionen ambos aspectos. Es decir, por un lado se definían los requisitos de seguridad a través de estos estándares, y por otro lado se proporcionaban soluciones y técnicas relacionadas con diferentes aspectos de la seguridad en el desarrollo de aplicaciones móviles. Lo que no existía hasta la fecha es un documento auto-contenido donde se relacionaran cada uno de los requisitos de seguridad definidos dentro de un estándar con las técnicas y mecanismos necesarios para verificar el correcto cumplimiento así como dar solución a los mismos.

Finalmente, comentar que ha sido un trabajo realmente interesante y didáctico. Al final, debido a mi background profesional ya había trabajado en este campo de la seguridad, pero nunca desde un punto de vista académico. Al final, uno de mis objetivos personales a la hora de cursar este Máster en general y este Trabajo Final de Máster en particular, era el profundizar en el campo de la seguridad en el desarrollo de aplicaciones móviles, y creo que he alcanzado este objetivo más que satisfactoriamente.

7. Glosario

- iOS.- Sistema operativo móvil creado y desarrollado por Apple Inc. exclusivamente para su hardware.

  https://www.apple.com/lae/ios/ios-11/

- Android.- Android es un sistema operativo móvil desarrollado por Google, basado en una versión modificada del kernel de Linux y otro software de código abierto y diseñado principalmente para dispositivos móviles con pantalla táctil como teléfonos inteligentes y tabletas.

  https://www.android.com

- OWASP (Open Web Application Security Project).- Es un proyecto de código abierto dedicado a determinar y combatir las causas que hacen que el software sea inseguro.

  https://www.owasp.org

- Snippet.- Los snippets son fragmentos de código que solucionan un problema concreto. Estos fragmentos suelen estar formados por un conjunto reducido de líneas de código.

- DLP.- Prevención en la pérdida de datos (Data Loss Prevention).

- MCM.- Software capaz de enviar y almacenar contenido a los dispositivos móviles (Mobile Content Management).

- MAM.- Software y servicios responsables de aprovisionar y controlar el acceso a aplicaciones móviles desarrolladas internamente y disponibles en el mercado que se utilizan en entornos empresariales tanto en teléfonos inteligentes como en tabletas, proporcionados por la compañía (Mobile Application Management).

- SDLC.- El proceso para el desarrollo de software, también denominado como ciclo de vida del desarrollo de software es una estructura aplicada al desarrollo de un producto de software (Software Development Lifecycle).

- Secure SDLC.- El proceso de incorporar mecanismos y medidas de seguridad en todas las fases del desarrollo de software (Secure Software Development Lifecycle).

- IPC.- La comunicación entre procesos es una función básica de los sistemas operativos. Los procesos pueden comunicarse entre sí a través de la compartición de espacios de memoria, ya sean variables compartidas o buffers, o a través de las herramientas provistas por las rutinas de IPC. La IPC provee un mecanismo que permite a los procesos comunicarse y sincronizarse entre sí, normalmente a través de un sistema de bajo nivel de paso de mensajes que ofrece la red subyacente (Inter-Process Communication).

- Diffie-Hellman.- Es un protocolo criptográfico de establecimiento de claves entre partes que no han tenido contacto previo, utilizando un canal inseguro y de manera anónima (no autentificada). Se emplea generalmente como medio para acordar claves simétricas que serán empleadas para el cifrado de una sesión (establecer clave de sesión). Siendo no autenticado, sin embargo, provee las bases para varios protocolos autenticados.

  https://es.wikipedia.org/wiki/Diffie-Hellman

- White-box Cryptography.- Es un conjunto de técnicas de criptografía destinadas a proteger las implementaciones de software de algoritmos criptográficos contra la recuperación de claves.

Un algoritmo de criptografía común recibe como parámetros de entrada la clave y el texto plano (modo de cifrado) / texto cifrado (modo de descifrado) y devuelve el texto cifrado (modo de cifrado) / texto plano (modo de descifrado). Una implementación White-box del mismo algoritmo de criptografía recibiría como parámetros de entrada un texto plano (modo de cifrado) / texto cifrado (modo de descifrado) y devolvería el texto cifrado (modo de cifrado) / texto plano (modo de descifrado). Se podría decir que en cierto modo la clave estaría "incrustada" dentro del propio algoritmo.

http://www.whiteboxcrypto.com

- Session Cookie.- Una cookie de sesión es un tipo de cookie que contiene información que se almacena en una ubicación de memoria temporal y luego se elimina una vez que se termina la sesión o se cierra el navegador web. Esta cookie almacena información que el usuario ha ingresado y rastrea los movimientos del usuario dentro del sitio web. Una cookie de sesión también se conoce como cookie transitoria.

- PKI.- Una PKI es una disposición que vincula claves públicas con respectivas identidades de entidades. La vinculación se establece a través de un proceso de registro y emisión de certificados en y por una autoridad certificadora (CA).

- CA.- En criptografía, una autoridad de certificación es una entidad que emite certificados digitales. Un certificado digital certifica la propiedad de una clave pública por el sujeto designado del certificado. Esto permite que otros (partes que confían) se basen en firmas o en afirmaciones hechas sobre la clave privada que corresponde a la clave pública certificada. Una CA actúa como un tercero confiable, tanto por el sujeto (propietario) del certificado como por la parte que confía en el certificado. El formato de estos certificados está especificado por el estándar X.509 (Certificate Authority).

- TLD.- Un dominio de nivel superior o dominio de primer nivel (del inglés top-level domain) es la más alta categoría del FQDN (fully qualified domain name) que es traducida a direcciones IP por los DNS oficiales de Internet. Los nombres de los servidores oficiales del DNS son administrados por la Corporación de Internet para Nombres y Números Asignados (ICANN). Alternativamente a los DNS oficiales, hay una serie de servicios de DNS alternativos, como es OpenNIC. Por ejemplo, en el FQDN www.uoc.edu el TLD será .org.

- App Store.- Es un servicio para iPhone, iPod Touch, el iPad y Mac OS X Snow Leopard o posterior, creado por Apple Inc, que permite a los usuarios buscar y descargar aplicaciones de iTunes Store o Mac App Store en caso de Mac OS X, desarrolladas con el iPhone SDK y publicadas por Apple.

  https://www.apple.com/ios/app-store/

- ASLR.- Address Space Layout Randomization es una técnica de seguridad implicada en la protección de los ataques de desbordamiento de la pila o stack overflow. El objetivo de ASLR es introducir aleatoriedad en las direcciones de memoria utilizadas por un proceso dado. Se trata de organizar de forma aleatoria las posiciones de las áreas de datos clave de una aplicación, tales como la base del ejecutable y las posiciones de la pila, datos, y bibliotecas, en el espacio de direcciones del proceso.

- Jailbreak.- Es el proceso de suprimir algunas de las limitaciones impuestas por Apple en dispositivos que utilicen el sistema operativo iOS mediante el uso de núcleos modificados. Tales dispositivos incluyen el iPhone, iPod Touch, iPad, Apple Watch y la Apple TV de segunda generación y watchOS 4. El jailbreak permite a los usuarios acceder por completo al sistema operativo, permitiendo al usuario descargar aplicaciones, extensiones y temas que no están disponibles a través de la App Store oficial.

- DRM.- La gestión de derechos digitales (DRM, del inglés Digital Rights Management), también llamado programas anticopia, es un término que se refiere a las tecnologías de control de

acceso por editoriales y titulares de derechos de autor para limitar el uso de medios o dispositivos digitales a personas o equipo no autorizadas. También se usa para referirse a las restricciones asociadas a las instancias específicas de las obras digitales o dispositivos. Los DRM se solapan, hasta cierto punto, con la protección de la copia de software, aunque el control del DRM es generalmente aplicado a medios creativos -música, películas, etc.

8. Bibliografía

[1] OWASP Mobile Application Security Verification Standard (MASVS) v1.0
Bernhard Mueller and Sven Schleier, Febrero de 2018

https://github.com/OWASP/owasp-masvs/releases/download/1.0/OWASP_Mobile_AppSec_Verification_Standard_v1.0.pdf

---

[2] OWASP Mobile Security Testing Guide (MSTG) v1.0
Bernhard Mueller and Sven Schleier, Febrero de 2018

https://github.com/OWASP/owasp-mstg

---

[3] OWASP Mobile App Security Checklist v1.0
Bernhard Mueller and Sven Schleier, Febrero de 2018

https://github.com/OWASP/owasp-mstg/blob/master/Checklists/Mobile_App_Security_Checklist.xlsx

---

[4] OWASP Mobile Top 10 2016-Top 10
OWASP, Febrero de 2017

https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10

---

[5] iOS Security Guide (iOS 11)
Apple Inc, Enero de 2018

https://www.apple.com/business/docs/iOS_Security_Guide.pdf

---

[6] iOS Secure Coding Guide
Apple Inc, Septiembre de 2016

https://developer.apple.com/library/content/documentation/Security/Conceptual/SecureCodingGuide/Introduction.html#//apple_ref/doc/uid/TP40002477-SW1

---

[7] Stack Overflow - Where Developers Learn, Share & Build Careers

https://stackoverflow.com

---

[8] Factores de riesgo en las aplicaciones móviles y algunos controles básicos con los que abordarlos.
Jorge Sarabia González, Facultad de Ingeniería de Deusto, Noviembre de 2016

https://blogs.deusto.es/master-informatica/factores-de-riesgo-en-las-aplicaciones-moviles-y-algunos-controles-basicos-con-los-que-abordarlos/

---

[9] Application Security Architecture Cheat Sheet
Lenny Zeltser (1st Draft in 2012), Tony Turner (2015 Format Change and Revisions) and Vin Miller (2015 Revisions)

https://www.owasp.org/index.php/Application_Security_Architecture_Cheat_Sheet

---

[10] Application Threat Modeling
OWASP, Mayo de 2017

https://www.owasp.org/index.php/Application_Threat_Modeling

---

[11] Secure SDLC Cheat Sheet
OWASP, Octubre de 2016

https://www.owasp.org/index.php/Secure_SDLC_Cheat_Sheet

---

[12] Recommendation for Key Management, Part 1: General
Elaine Barker (NIST), William Barker (NIST), William Burr (NIST), Tim Polk (NIST) y Miles Smid (Orion Security Solutions), Marzo de 2017

https://csrc.nist.gov/publications/detail/sp/800-57-part-1/revised/archive/2007-03-01

---

[13] Cryptographic Services Guide
Apple Inc, Julio de 2014

https://developer.apple.com/library/content/documentation/Security/Conceptual/cryptoservices/GeneralPurposeCrypto/GeneralPurposeCrypto.html

---

[14] On the Importance of the Key Separation Principle for Different Modes of Operation
Danilo Gligoroski, Suzana Andova y Svein Johan Knapskog, Abril de 2008

https://link.springer.com/chapter/10.1007/978-3-540-79104-1_29

---

[15] OWASP JWT Cheat Sheet
OWASP, Diciembre de 2017

https://www.owasp.org/index.php/JSON_Web_Token_(JWT)_Cheat_Sheet_for_Java

---

[16] Blocking Brute Force Attacks
OWASP, Agosto de 2017

https://www.owasp.org/index.php/Blocking_Brute_Force_Attacks

---

[17] App Transport Security (ATS)
Apple Inc, Enero de 2018

https://developer.apple.com/library/content/documentation/General/Reference/InfoPlistKeyReference/Articles/CocoaKeys.html

---

[18] Writing Secure Code
Michael Howard and David LeBlanc, Microsoft Press 2002

https://books.google.es/books/about/Writing_Secure_Code.html?id=s-lQAAAAMAAJ

---

[19] Inter-App Communication
Apple Inc, Marzo de 2017

https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Inter-AppCommunication/Inter-AppCommunication.html

---

[20] Xcode Help - Distribute your App
Apple Inc, 2018

https://help.apple.com/xcode/mac/current/#/dev8b4250b57

---

[21] Code Signing Guide
Apple Inc, Septiembre de 2016

https://developer.apple.com/library/content/documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html

---

[22] Advanced Memory Management Programming Guide
Apple Inc, Julio de 2012

https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html#//apple_ref/doc/uid/10000011-SW1

---

[23] Technical Risks of Reverse Engineering and Unauthorized Code Modification
OWASP, Abril de 2014

https://www.owasp.org/index.php/Technical_Risks_of_Reverse_Engineering_and_Unauthorized_Code_Modification

9. Anexos

En esta sección se incluyen una serie de ejemplos de código fuente que demuestran la utilización e implementación de algunas de las APIs y librerías mencionadas, así como configuraciones de seguridad en proyectos de Xcode. Estos anexos se entregan dentro de un fichero *Anexos.zip*, haciéndose referencia a los mismos a lo largo de las diferentes secciones de la memoria.

---

[1] La clase KeychainItemWrapper es un wrapper implementado por Apple que permite abstraer el acceso a un elemento del Keychain.

KeychainItemWrapper.h, KeychainItemWrapper.m

---

[2] La clase CryptoHash es una utilidad que permite llevar a cabo operaciones de Hash utilizando algunos de los algoritmos más populares (MD5, SHA1, SHA256 y HMAC).

CryptoHash.h, CryptoHash.m, CryptoBase64.h, CryptoBase64.m

---

[3] La clase CryptoAES es una utilidad que permite llevar a cabo la encriptación y desencriptación de datos a través del algoritmo de cifrado simétrico AES.

CryptoAES.h, CryptoAES.m

---

[4] En este anexo se incluyen dos librerías implementadas en Objective-C y Swift para codificar y decodificar JSON Web Token (JWT) dentro de una aplicación iOS.

https://github.com/stannie/swift-jwt

https://github.com/ducreyna/objective-c-jwt-simple

Jwt.h, Jwt.m, Algorithm.h, Algorithm.m, SwiftJWT.swift

---

[5] En este anexo se incluyen una par de snippets de código que muestran como aceptar un certificado no firmado por una CA de confianza y como llevar a cabo un SSL certificate pinning.

CertificatesHandling.m, CertificatesHandling.swift

---

[6] En este anexo se incluyen un snippet de código que muestran como crear una vista que incluya un WKWebView y deshabilitar el código JavaScript en la misma.

WebViewController.m, WebViewController.swift

---

[7] En este anexo se incluye una utilidad que incluye dos funciones desarrolladas en C, las cuales permiten detectar y terminar una aplicación si el dispositivo está jailbroken.

JailbreakDetection.h, JailbreakDetection.c

---

[8] En este anexo se incluye una utilidad que incluye una función desarrollada en C que permite detectar y terminar una aplicación si se está utilizando un depurador o debugger.

AntiDebug.h, AntiDebug.c

---

[9] En este anexo se incluye una utilidad que incluye una función desarrollada en C que permite verificar en tiempo de ejecución la firma del código fuente de la aplicación, comparándola con otra firma generada previamente, de tal manera que se pueda verificar si se ha producido algún tipo de modificación en el binario de la aplicación.

IntegrityCheck.h, IntegrityCheck.c

---

[10] En este anexo se incluye una utilidad desarrollada en Objective-C que permite generar y almacenar un UUID en el Keychain, así como recuperarlo.

https://github.com/taka0125/TAKUUID

TAKUUID.h, TAKUUIDStorage.h, TAKUUIDStorage.m

---

[11] En este anexo se incluye un proyecto de Xcode utilizado en algunas secciones de la memoria para explicar algunas de las configuraciones de seguridad explicadas a lo largo de la misma.

Proyecto de Xcode TFMUOC