RAGnarok
Radare in Android Games

Trabajo de Fin de Grado
Curso 2020–2021

Autor
María García Raldúa

Director
Jose Luis Vázquez Poletti

Grado en Desarrollo de videojuegos
Facultad de Informática
Universidad Complutense de Madrid

RAGnarok
Radare in Android Games

Trabajo de Fin de Grado en Desarrollo de videojuegos

Autor
María García Raldúa

Director
Jose Luis Vázquez Poletti

Convocatoria: *Septiembre 2021*
Calificación: *Nota*

Grado en Desarrollo de videojuegos
Facultad de Informática
Universidad Complutense de Madrid

20 de septiembre de 2021

Dedicatoria

A Jose Vázquez Poletti por confiar en mí y despertar mi interés por la Ciberseguridad

Agradecimientos

A Jose Luis Vázquez Poletti por confiar en mí para hacer un buen trabajo y darme libertades y facilidades para desarrollarlo. También quiero agradecerle a esos amigos que siempre están ahí, el haberme animado e insitido en seguir con el proyecto cuando quería dejar todo de lado.

Resumen

RAGnarok

En la actualidad, en tecnología, los procesos manuales que requieren de mucho esfuerzo y que resultan tediosos en su realización, son sustituidos por soluciones que automatizan dichos procesos y que tratan de reducir la implicación humana. En las pruebas de seguridad en aplicaciones de móvil, tenemos una parte de análisis estático, que es inspeccionar el código y el binario de la app en busca de vulnerabilidades, una tarea que podemos automatizar en cierta medida. Con RAGnarok se pretende realizar escaneos automáticos usando distintas herramientas en un mismo framework para filtrar aquello que se considere interesante y que pueda implicar una vulnerabilidad en la app. Además facilitará la preparación del apk para su análisis.

Palabras clave

Ciberseguridad, Android, Juegos, Pentesting, Seguridad, Videojuegos, Hacking, Móvil, Python, Automatización

Abstract

RAGnarok

Nowadays, within technology, manual processes that require a lot of effort and are tedious to carry out, are replaced by solutions that automate such processes and try to reduce human implication. During mobile security assessments, we have a static analysis part, which entails application's source code and binary inspection looking for vulnerabilities, a task we can automate mostly.

RAGnarok is pretended to be used as an automatic scan using several tools within the same framework to filter interesting strings which entail a risk for the application. Besides, it eases the process of having the apk ready for analysis.

Keywords

Cybersecurity, Android, Games, Pentesting, Security, Videogames, Hacking, Mobile, Python, Automation.

Índice

1. Introducción **1**

1.1. Motivación 2

1.2. Objetivos 2

1.3. Plan de trabajo 3

2. Introduction **5**

2.1. Motivation 6

2.2. Goals 6

2.3. Working plan 7

3. Estado de Arte **9**

3.1. Ciberseguridad en videojuegos 9

3.2. OWASP Game Security Framework 10

3.2.1. Vectores de ataque según niveles 10

3.2.2. Ingeniería social 15

3.2.3. Motivaciones de los atacantes 16

3.2.4. Impacto económico y social de un ciberataque 17

3.3. Seguridad en Aplicaciones de móvil 18

3.3.1. OWASP Top Ten 19

3.3.2. Seguridad en Android 21

3.3.3. Seguridad en iOS 23

3.4. Pentesting 27

3.4.1. Fases 28

3.5. Pruebas de seguridad en móviles 29

3.5.1. Análisis estático . . . 30
3.5.2. Análisis dinámico . . . 30
3.5.3. Pruebas de seguridad en juegos Android . . . 31

4. Descripción del Trabajo **33**
4.1. RAGnarok . . . 33
4.2. Tecnologías usadas . . . 33
4.2.1. APK-Anal . . . 33
4.2.2. MobSF . . . 34
4.2.3. APKtool . . . 35
4.2.4. Radare2 . . . 35
4.2.5. Dependency-check . . . 36
4.2.6. ApkID . . . 36
4.3. Desarrollo de la herramienta . . . 36
4.3.1. Objetivos propuestos . . . 36
4.3.2. Funcionalidades propuestas . . . 37
4.3.3. Funcionalidades actuales . . . 38
4.3.4. Arquitectura de la herramienta . . . 38
4.3.5. Click . . . 41
4.3.6. Rich . . . 41
4.4. Tecnologías descartadas . . . 41
4.4.1. MARA Framework . . . 41
4.4.2. Objection . . . 42
4.4.3. StaCoAn . . . 43

5. Conclusiones y Trabajo Futuro **45**
5.1. Conclusión . . . 45
5.2. Trabajo futuro . . . 45

6. Conclusions and Future Work **47**
6.1. Conclusions . . . 47
6.2. Future Work . . . 47

A. Anexo A: Diario de desarrollo **49**

A.1. Primera fase: investigación y tecnología elegida 49
A.2. Segunda fase: Implementación 50
A.3. Tercera fase: pruebas y resultados 54

B. Anexo B: Manual de usuario **57**

B.1. Aplicaciones usadas para el caso de ejemplo 57
B.2. Descargar e instalar la herramienta 57
B.3. Usar Apktool 58
B.4. Usar unzip 58
B.5. Análisis de archivo dex con Radare2 58
B.6. Levantar servidor MobSF y subir el apk a la plataforma 58
B.7. Uso de **Dependency-check** para análisis de dependencias 59
B.8. Uso de ApkID para obtener más información del fichero apk 59

Bibliografía **67**

Índice de figuras

3.1. Boot flow 22
3.2. Boot flow completo con estados en detalle 23
3.3. Arquitectura Android 24
3.4. Ciclo de vida de aplicaciones 25
3.5. Estructura del apk 25
3.6. Estructura de iOS 26
3.7. Ciclo de vida de aplicaciones iOS 27
3.8. Estructura de un ipa 28
3.9. Captura Mobile Pentest Checklist - OWASP 31

4.1. Pip command 34
4.2. MobSF Logo 35
4.3. Estructura de las carpetas y archivos 39
4.4. Arquitectura de la herramienta 40
4.5. Función analyse de RAGnarok 40
4.6. MARA Framework 42
4.7. Listado de archivos en Objection 43
4.8. Autocomplete de comandos en Objection 43
4.9. GUI report de StaCoAn 44

5.1. Comandos de r2frida 46

6.1. Comandos de r2frida 48

A.1. Pip command 50
A.2. Ejecución Apktool en el código 50

A.3. Documentación de MobSF 50
A.4. No se puede cargar la librería 51
A.5. Inicio de MobSF 51
A.6. Pantalla principal de MobSF 51
A.7. Implementación de la API 52
A.8. Comando para MobSF 52
A.9. Código para MobSF 53
A.10.Vista de los escaneos recientes 53
A.11.Contenido .profile 53
A.12.setup.sh 54
A.13.Output Dependency-check 54
A.14.Reporte de Dependency-check 55
A.15.Banner de RAGnarok 55

B.1. Repositorio RAGnarok 60
B.2. Repositorio de dependency-check 61
B.3. Salida de la ejecución de setup.sh 61
B.4. Extracción y decodificación apk 61
B.5. Contenido directorio *apk-extracted* 62
B.6. Contenido *smali* 62
B.7. Comando *zip* 62
B.8. Contenido *zip* 62
B.9. Ejecución del análisis al archivo dex 63
B.10.Clave secreta en strings 63
B.11.Ruta a fichero apk de superuser 63
B.12.Ejecución del servidor de MobSF local 63
B.13.Plataforma MobSF Web 63
B.14.Ejecución del comando *mobsf* 64
B.15.Pestaña de escaneos recientes 64
B.16.Reporte en pdf del análisis 64
B.17.Reporte en html de **Dependency-check** 65
B.18.Ejecución del comando de *apkid* 65

Capítulo 1

Introducción

"La desconfianza es la madre de la seguridad"
— Aristófanes

La tecnología ha evolucionado rápidamente en un breve lapso de tiempo en los últimos años, poniendo a prueba la capacidad del ser humano para adaptarse a las nuevas tecnologías. Siendo el objetivo principal de este desarrollo el beneficio económico a través de la satisfación de necesidades de la población, ya sean creadas o existentes, se ha dejado de lado un aspecto esencial, la seguridad de estas tecnologías.

La Ciberseguridad nace de la necesidad de proteger a las personas de los ciberdelincuentes, individuos que por distintas motivaciones, tienen como objetivo comprometer, espiar, robar, romper y un largo etcetera, la información y la identidad de empresas y personas.

Es sorprendente que a pesar de los ciberataques sufridos a lo largo de la historia, a día de hoy la seguridad en la sociedad sigue ocupando un segundo plano, llegando a recibir poco presupuesto y poca atención por parte de empresas que todavía no han sido comprometidas. En una cultura de curar antes que prevenir, casi siempre se pone la seguridad en primer plano cuando se ha sufrido un ataque, pocas veces antes.

Los videojuegos es una de las tecnologías más relevantes en la actualidad, con un incremento de ingresos del 20 %, facturando casi 180 millones de dólares en 2020 (Nesterenko, 2020). Con casi 3 billones de usuarios (Deyan, 2021), es un potencial vector de ataque para ciberdelincuentes, ya que implica a usuarios de todas las edades, además de ser piezas de software cada vez más grandes y complejas, aumentando el abanico de posibilidades para un atacante.

Antiguamente, los videojuegos eran pequeños y simples, orientados al entretenimiento y cuyo funcionamiento no era difícil de desgranar. En ese momento los llamados hackers de videojuegos utilizaban ingeniería inversa para modificar el comportamiento o ampliar la funcionalidad del juego o la consola, no siendo dañino para otros usuarios ya que se aplicaban al propio juego en modo offline. Su motivación era el aburrimiento y/o la curiosidad, modificando desde mecánicas del juego hasta el estilo visual.

El nacimiento de la modalidad online trajo consigo nuevas motivaciones y nuevos retos para estos hackers, muchos de ellos evolucionando a ciberdelincuentes. Actualmente, existen infinidad de vectores de ataque incluso aunque el juego sea offline, dado que todo juego se valida al menos una vez con un servidor dedicado para comprobar su autenticidad. Muchas

son las consecuencias de un ataque, desde la pérdida económica, hasta en ciertas ocasiones, la vida humana.

En un mundo en el que la privacidad es un lujo y los datos son de dominio público, los videojuegos se han convertido en una via fácilmente explotable hasta para los atacantes más amateur, porque nadie es capaz de poner puertas seguras a aquello que se publica en el ciberespacio.

1.1. Motivación

Actualmente todo el mundo tiene un smartphone en el bolsillo. Es una pieza de hardware cada vez más accesible e indispensable para la sociedad, dentro del cual se encuentra el pilar en el que se basará este trabajo, los juegos.

Hay muy poca documentación respecto a seguridad en el ámbito de juegos de móvil, siendo la mayoría generalista para todo tipo de aplicaciones, lo que hace que muchos desarrolladores indie no inviertan tiempo en probar si su juego además de divertido es seguro. Hacer pentest de un juego requiere muchas horas, muchas herramientas y mucho estudio que pocos se pueden permitir, ya que añadido al proceso de un pentest al uso, hay que añadirle las características propias de un juego que no tienen las aplicaciones, sean algunas de estas el cheating, nuevas formas de phishing y un mercado negro especializado.

La automatización tiene como objetivo aligerar procesos pesados y repetitivos para el usuario, lo que hace más atractivo el aplicar buenas prácticas a la hora de desarrollar aplicaciones. Por ello automatizar parte del proceso de pentesting a juegos de móvil ayudará a desarrolladores más pequeños a probar sus aplicaciones antes de pasarlas a producción y así hacerlas más seguras para los usuarios y a pentesters realizar tests de manera más rápida y eficiente.

1.2. Objetivos

Con el presente trabajo se pretende crear un framework, RAGnarok, que automatice la parte de análisis estático del proceso de pentesting de juegos de móvil, con un instalador e información de los comandos que contiene.

Este framework integra distintas herramientas en una para facilitar el pentest de un juego, cubriendo el análisis estático de una aplicación. Los objetivos de cada etapa del test de seguridad se detallan a continuación:

- Instalación y decompilación: se automatizará este proceso de manera que el usuario solo tenga que introducir por comando el nombre de la app a testear.
- Análisis estático: cubre toda aquella inspección del código de la aplicación.
    1. Análisis de código: se realizará la inspección del código en busca de fallos que puedan ser perjudiciales para la aplicación y que los desarrolladores puedan solventarlos y/o mejorar el código actual.
    2. Datos sensibles: el programa buscará aquellas vulnerabilidades más comunes dentro del código de la aplicación referentes a credenciales hardcodeadas, información sensible en texto plano, urls, etc.

3. Análisis de dependencias: se comprobará si la aplicación es insegura por dependencias desactualiziadas que contengan vulnerabilidades conocidas.

Esta herramienta no sustituirá en un 100 % al usuario, ya que se requiere de la visualización y juicio humano para distintos puntos del proceso.

1.3. Plan de trabajo

El desarrollo del trabajo se ha dividido en tres fases generales:

- Investigación: analizar el estado de la cuestión planteada, extraer información para determinar la viabilidad de los objetivos e investigar las distintas herramientas para contemplar su uso en el producto final.
- Implementación de la herramienta: esta parte se desarrollará en varios pasos que describan al detalle la implementación de las distintas funcionalidades.
- Pruebas y resultados: se prueban varios tipos de aplicaciones más o menos vulnerables, siendo algunas generalistas y otras más enfocadas a juegos y se extraen los reportes y los datos que se generen durante dichas pruebas.

Todo el proceso se lleva a cabo en WSL (Windows subsystem for Linux), una terminal linux dentro de Windows que sirve para ejecutar comandos de ubuntu y tener un sistema de archivos de linux.

Todas las modificaciones de código se llevan a cabo en el editor de texto Sublime3, que permite abrir carpetas y modificar código de manera rápida y visual, siendo un editor sencillo y que contiene muchas extensiones de archivo.

Para el control de cambios, se utiliza Git y se suben los cambios a un repositorio de Github [1]

El lenguaje de programación en el que está programado es python, dado que es el lenguaje del proyecto original en el que está basado esta herramienta y es el que nos da más facilidades a la hora de añadir módulos relacionado con seguridad y otras funcionalidades, aunque también habrá algunas utilidades que requieran de bash scripting.

[1] `https://github.com/Mariag39/RAGnarok`

Capítulo 2

Introduction

Technology has evolved fastly in a short period of time in recent years, testing human ability to adapt to new technologies. Since development main goal is the profit through meeting people needs, either created or that actually exist, an important aspect has been throw apart, the security within these technologies.

Cybersecurity has born due to the necessity of protect the people against cybercriminals, who following different motivations, have as goal compromising, spying, stealing, breaking and so on, the data and the identity of bussiness and people.

It is surprising that, although suffering cyberattacks through history, nowadays the security within the society is still in a second layer, receiving low budget and less atention from companies which not have been compromised yet. Within a culture of healing before preventing, usually the security is the main dish when suffering an attack, rarely before.

Videogames is one of the most relevant technologies today, with a revenue increase of 20 %, with a turnover of almost 180 million dollars in 2020(Nesterenko, 2020). With almost 3 billion users, it is a potential attack vector for cybercriminals, since it involves users of all ages, in addition to being increasingly large and complex pieces of software, increasing the range of possibilities for an attack.

In the past, videogames were small and simple, oriented to entertainment and whose functioning was not difficult to reel off. At that time the called videogame hackers used reverse engineering to modify the behavior or expand the functionality of the game or the console, not being harmful to other users since they were applied to the game itself in offline mode. Their motivation were boredom and/or curiosity, modifying from game mechanics to visual style.

The birth of the online modality brought with it new motivations and new challenges for these hackers, many of them evolving into cybercriminals. Currently, there are infinite attack vectors even if the game is offline, since every game is validated at least once with a dedicated server to verify its authenticity. Many are the consequences of an attack, from financial loss, even on certain occasions, human lives.

In a world where privacy is a luxury and data is in the public domain, videogames have become an easily exploitable way even for the most amateur attackers, because no one is capable of putting secure doors to what is published in cyberspace.

2.1. Motivation

Nowadays, everyone has a smartphone in their pockets. It is an increasingly accessible and indispensable, where it is found the pillar on which this project will be based, the games.

There is just a few documentation regarding security in the field of mobile games, with the majority being generic for all types of applications, which means that many indie developers do not invest time in testing if their game is safe as well as fun. Making pentest of a game requires many hours, many tools and so much study that few can afford, since added to the process of a pentest itself, it is necessary to add the specific characteristics of a game that applications do not have, some of these being cheating, new forms of phishing and a specialized black market.

Automation aims to lighten tedious and repetitive processes for the user, which makes it more attractive to apply good practices when developing applications. Therefore, automating part of the pentesting process to mobile games will help smaller developers to test their applications before going to production and thus make them safer for users, and pentesters to perform tests more quickly and efficiently.

2.2. Goals

With this project, it is pretended to create a framework, RAGnarok, which automates the static analysis phase of the mobile games pentesting process, with an installer and information of commands it has.

This framework integrates different tools on one to ease the pentest of a game, covering the application static analysis. The goals of each phase of the security test are described below:

- Installation and decompilation: this process will be automated so the users only has to enter the name of the app to be tested by command.
- Static Analysis: covers all application code inspection.
    1. Code analysis: the code will be inspected in search of bugs that may be harmful to the application and that the developers can solve and/or improve the current code.
    2. Sensitive data: the program will search for the most common vulnerabilities within the application code referring to hardcoded credentials, sensitive information in plain text, urls, etc.
    3. Dependency analysis: the application will be checked to see if it is insecure due to outdated dependencies that contain known vulnerabilities.

This tool will not replace the user 100 %, since visualization and human judgment is required for different steps of the process.

2.3. Working plan

Project development has been divided in three general phases:

- Research: analyse the state of art, extract information to determine the viability of the objectives and investigate the different tools to consider their use in the final product.
- Tool implementation: this part will be developed in several steps that describe in detail the implementation of the different functionalities.
- Tests and results: various types of pretty much vulnerable applications are tested, some being generalist and others more focused on games, and the reports and data generated during those tests are extracted.

The whole process is carried out in WSL (Windows subsystem for Linux), a linux terminal within Windows that is used to execute ubuntu commands and have a linux file system.

All code modifications are carried out in the Sublime3 text editor, which allows opening folders and modifying code quickly and visually, being a simple editor and containing many file extensions.

For version control, Git is used and changes are uploaded to a Github repository[1]

The programming language in which this tool is programmed is python, since it is the language of the original project this tool is based on and it is the one that gives us more facilities when adding modules related to security and other functionalities, but also there will be some utilities that require bash scripting.

[1] `https://github.com/Mariag39/RAGnarok`

Capítulo 3

Estado de Arte

3.1. Ciberseguridad en videojuegos

La ciberseguridad en el ámbito de los videojuegos se ha convertido en una disciplina aparte que se distingue de la definición general del término. Los videojuegos son usados por gran parte de la población con un rango de edad muy amplio, son accesibles y una tecnología tan grande que es difícil de controlar al completo, lo cual hace que su seguridad se necesite estudiar y analizar como algo aparte. Según el informe de Akamai (Akamai, 2020), los videojuegos como un vector de ataque para brechas de seguridad sigue creciendo, registrándose más de 12 billones de ataques de credential stuffing fueron reportados en 17 meses entre 2018 y 2019. Un ataque puede afectar a varios aspectos: interoperabilidad, información protegida, el producto final y el modelo de negocio al completo. Dentro de los videojuegos hay vulnerabilidades que no se contemplan de distinta forma que en una aplicación al uso, como cheats que pueden llevan a la introducción de malware, nuevas formas de ingeniería social, credential stuffing, account takeover, cyberbullying, incluso hay software especializado en explotar de manera ilícita cualquier aspecto del juego (CheatEngine). Los videojuegos son un objetivo vulnerable muy atractivo en gran parte porque los jugadores confian todos sus datos a las plataformas con un fin de entretenimiento, sin sopesar las consecuencias. Al fin y al cabo, muy pocos usuarios se leen los términos y condiciones de los productos que consumen.

Algunos de los ataques más notorios dentro de la industria implican a consolas y grandes empresas (Editor, 2015), los cuales se describen a continuación:

- Play Station Network(2011): 77 millones de datos de usuarios fueron extraídos de la plataforma, lo que provocó unas pérdidas de 171 millones de dólares a Sony y un gran impacto en la confianza de los jugadores.
- Xbox and Play Station (2014/2017): durante las Navidades del 2014, el llamado grupo hacker "Lizard Squad", lleva a cabo un ataque de DDoS dejando los servidores de ambas plataformas offline durante horas. Luego en 2017, se extrayeron datos de jugadores de ambas plataformas de dos conocidos foros de jugadores. De este último no se sabe el alcance exacto del daño.
- Zynga and Facebook (2019): Zynga es conocida como la productora de juegos web

de Facebook, y durante el ataque a la base de datos del juego Words with Friends, se extrajeron los datos de al menos 200 millones de usuarios.

- Nintendo (2020): 300 mil cuentas fueron hackeadas en un periodo de varios meses en la plataforma, y no fue hasta que muchos usuarios se quejaron de transacciones digitales fraudulentas cuando Nintendo descubrió el ataque.

Aún teniendo en cuenta estos ataques que han pasado a la historia, según una firma de seguridad de EEUU (Townsend, 2019), solo un cuarto de los jugadores se preocupan de la seguridad de los juegos en el futuro, el 55 % de los usuarios reutilizan contraseñas y un gamer al uso ha experimentado aproximadamente cinco ciberataques. Esto da una visión resumida del estado de la cuestión, siendo necesaria una concienciación en los usuarios a la vez que instar a los desarrolladores a mejorar y reforzar las medidas de seguridad en los juegos.

3.2. OWASP Game Security Framework

OWASP Game Security Framework (Miessler, 2017) es un proyecto impulsado por Daniel Miessler que sigue actualmente en desarrollo. Comenzó en 2014, pero no se invirtieron muchos esfuerzos en ello. No fue hasta su segundo intento de llevarlo a cabo (2017), cuando se aportaron nuevos conocimientos que enriquecieron el ámbito de la seguridad en videojuegos, aunque actualmente sigue siendo insuficiente y en algunos casos, desfasado.

En él se analizan los distintos vectores de ataque, las vulnerabilidades comunes de cada uno de ellos, exploits típicos, las motivaciones de los atacantes, las consecuencias negativas y las defensas que existían en ese momento.

En los siguientes conceptos que analizan la ciberseguridad en los videojuegos, se usa como punto de partida este framework además de contar con los distintos elementos de la investigación.

3.2.1. Vectores de ataque según niveles

A continuación se enumeran y describen los ataques, vulnerabilidades y defensas de los distintos niveles que constituyen una aplicación y su funcionalidad.

3.2.1.1. Conexiones y tráfico de red

Se entiende por tráfico de red de un juego a los datos que se envían y reciben entre el servidor y los jugadores conectados entre sí. Dentro de un juego multiplayer online, se encuentran distintos tipos de conexión para esta transmisión de datos:

- Peer to peer (P2P): en este modelo, los jugadores se conectan directamente entre ellos y se envían paquetes de estado del juego. Este proceso se puede hacer de dos formas: uno de los jugadores de la partida se convierte en host y actúa como servidor y como jugador de dicha sesión, enviando el estado de juego al resto de integrantes de la sala y en caso de desconectarse, se migraría el host y otro jugador ocuparía

su lugar, mientras por otro lado todos los jugadores se envían la información entre ellos sin que haya un jugador actuando como servidor. En este último caso todos contienen la lógica del juego y negocian entre ellos la actualización de estado.

- Cliente-Servidor: existe un servidor dedicado al que todos los jugadores se conectan. Este servidor corre la simulación del juego y recibe paquetes de estado de las acciones de los jugadores, los valida, actualiza la simulación y manda esta información al resto de jugadores. Este es el modelo que se considera más seguro y confiable.
- Cliente-Cloud: una nueva versión del modelo anterior en la cual el juego usa computación en la nube para proveer a los jugadores de un servidor dedicado.

3.2.1.2. Ataques y vulnerabilidades

El éxito de un ataque al envío de paquetes entre los jugadores y el servidor depende de varios factores, entre ellos la complejidad del objetivo, el tipo de conexión (mencionados anteriormente) y la habilidad del ciberdelincuente. En el caso de juegos P2P, al ser el cliente quien valida los paquetes, es más sencillo hacer trampas y/o atacar al resto de jugadores, en cambio, cuando existe un servidor que valida el estado de los jugadores, el éxito de un ataque dependerá de hacer creer al servidor que la información enviada es legítima. Algunos de los ataques más comunes son:

- Packet sniffing: consiste en interceptar los paquetes enviados y sacar información de ellos. Cuando la comunicación no está encriptada o el nivel de encriptación es deficiente, es fácil para un atacante extraer datos sensibles como credenciales, información personal, datos de configuración, mensajes, etc, siempre y cuando se encuentre en la misma red. Por otro lado, el objetivo también puede ser extraer esos paquetes para modificarlos y enviarlos al servidor con los cambios aplicados, lo cual es llamado Packet Injection. Sin embargo, actualmente casi todos los juegos tienen sus comunicaciones encriptadas, usan protocolos de red complejos y requieren de habilidades en ingeniería inversa avanzada para poder realizar un ataque con éxito. Para ello existen herramientas como WireShark o TCPdump, que permiten ver el contenido de las comunicaciones en una interfaz.
- Packet Injection: si un atacante ha conseguido capturar un paquete, desencriptar la información, realizar ingeniería inversa para desgranar el protocolo de red utilizado y hacer el contenido más legible, entonces puede ser capaz de modificar dicho contenido, volver a empaquetar los datos, formar el paquete y enviarlo al servidor para que lo valide. Dicho proceso es muy complejo y requiere de mucha habilidad para llevarlo a cabo con éxito, lo cual es la razón de que gran parte de los ataques se realicen a nivel cliente de juego. Aún así, la motivación de estos ataques es que desde el cliente puede haber muchos checks de sanitización que en los paquetes no, lo que lo hace más efectivo en algunos casos.
- DoS y DDoS: los ataques de denegación de servicio consisten en enviar un gran número de peticiones a un servicio saturando su ancho de banda y haciéndolo inaccesible para el resto de usuarios. En los juegos, se usa tanto como un recurso para ganar partidas denegando el servicio al resto de jugadores, como una forma de tirar servidores globalmente por motivos diversos (venganza, activismo, etc). Es uno de los ataques más relevantes de la industria en la actualidad, ya que según Nexusguard Q3 2020

Thread Report, el 77 % de los ataques DDoS han sido dirigidos a la industria de los videojuegos.

3.2.1.3. Defensas actuales

Actualmente muchos juegos implementan medidas para hacer más difícil a un atacante el usar la vía de las comunicaciones entre servidor y cliente como vector de ataque, pero no existe un método que lo haga imposible de hackear. Lo más común es encriptar los paquetes que contengan información sensible o que otros usuarios no deban saber, haciendo que un atacante tenga que averiguar los pasos a seguir para desencriptar los paquetes, desmontarlos, y según sus objetivos, volver a montarlos o extraer la información que contienen.

3.2.1.4. Servidor de videojuegos

Los videojuegos con funcionalidad online tienen un servidor que se encarga de proveer de la última versión del juego, conectar a los usuarios entre sí y enviar y recibir el estado de juego de todos los participantes. Según el planteamiento del juego, se pueden encontrar distintos tipos de servidores:

- Servidor dedicado: todos los jugadores se conectan al servidor desde un cliente. Corre una simulación del juego y va actualizando su estado enviando esa información a todos los jugadores con el input que recibe de cada uno de ellos. Es uno de los modelos más fiables ya que el servidor es el encargado de validar las acciones de los usuarios, además de reducir la latencia de las conexiones.
- Servidor escucha: el servidor opera en el mismo proceso que los clientes, consumiendo mucho ancho de banda. Es común en partidas LAN donde no hay un gran número de jugadores y no se requiere de velocidad de conexión. No tiene coste y no requiere de una infraestructura especial.
- Peer to peer: en una de las implementaciones de este tipo, un jugador hace de cliente y servidor al mismo tiempo, corriendo el juego a la vez que envía y recibe el estado de la partida del resto de jugadores. Por otro lado, todos los participantes pueden ser cliente y servidor a la vez, mandándose directamente el estado entre ellos.

3.2.1.5. Ataques y vulnerabilidades

Según el modelo de red que tiene un determinado juego, las vulnerabilidades varían:

- Servidor dedicado: Los vectores de ataque en servidores de videojuegos profesionales son limitados, siendo común los ataques DDoS (descritos anteriormente en el apartado de Conexiones y tráfico de red). Para poder cambiar variables en el servidor, se tendría que encontrar un fallo en la lógica, lo cual es muy complejo. Si el servidor pertenece a un particular, puede contener vulnerabilidades derivadas de la falta de hardening del servidor por desconocimiento de buenas prácticas, además de poder ser el propio host el agente malicioso que altere el estado del juego, introduzca malware o use técnicas de phishing para obtener información. Algunas de estas vulnerabilidades incluyen no tener los puertos bien filtrados, usar contraseñas fácilmente crackeables,

revelar demasiada información en códigos de error, logs o posibles páginas web, usar frameworks no actualizados con vulnerabilidades conocidas, DDoS, etc.

- P2P: En el caso de un modelo de conexión P2P en el que el servidor es un cliente a la vez, es fácilmente alterable el estado de juego, pudiendo el atacante enviar el estado que quiera al resto de jugadores, a la vez que bloquear el input de estos.

3.2.1.6. Defensas actuales

Dejando de lado el hardening de servidores, lo más común es el filtrado y tratamiento del input y output del servidor, esto es, validar por un lado el input que se recibe de los jugadores y por otro controlar la información que se debe enviar a los usuarios en cada momento. Por ejemplo, si un jugador está escondido, el servidor no enviará la posición de dicho usuario al resto de jugadores. Esa es la definición de server autoritario.

En el caso del modelo P2P, se implementan algunas defensas para legitimar la información manejada:

- Consenso: los jugadores se convierten en mediadores y deciden cuando una información es válida entre dos usuarios cuya comunicación no les incumbe, pero conocen el estado del juego.
- Guardianes: dentro del software se pueden implementar métodos que comprueben la legitimidad de la información haciendo comprobaciones de si el estado ha sido alterado.
- Content delivery networks: la información que se quiere validar se envía a unos nodos que contienen el estado del juego y del jugador en concreto. Es un método que requiere de mucho capital.
- Host de confianza: se elije como host de una partida a un jugador considerado como fiable según unos parámetros definidos por los desarrolladores del juego.
- Host híbrido: los desarrolladores pueden actuar como mediadores/coordiandores cuando no hay un host fiable en determinadas sesiones.

3.2.1.7. Cliente

Los jugadores pueden comunicarse con un servidor y/o entre ellos a través de un software, el cliente de juego. Es el encargado de renderizar el juego, enviando y recibiendo información del servidor y otros usuarios. Cada cliente presenta una versión diferente del estado de juego según sus circunstancias aunque se esté jugando al mismo juego y decodifica la información recibida.

3.2.1.8. Ataques y vulnerabilidades

En lo referente a cheating, es común que se modifique el cliente para conseguir ventaja o molestar a otros jugadores en el juego. Actualmente existen muchas herramientas y documentación cuyo objetivo principal es este, siendo muy conocido Cheat Engine, además de la existencia de un amplio mercado de cheats en internet, los cuales muchas veces acaban

en la introducción de malware. Pero no todo se reduce a hacer trampas, dado que muchos clientes incluyen funcionalidades de inicio de sesión y compras dentro de la aplicación que incluyen otras vulnerabilidades:

- Fuzzing credentials: ataque de fuerza bruta para averiguar las credenciales de las cuentas de otros jugadores de un juego. Este ataque puede ser motivado por el filtrado de cuentas de usuario en plataformas como pastebin o por el conocimiento de posibles nombres de usuario (nicks) a partir de foros o partidas o por implementaciones de login deficientes que pueden ser bypasseables.

- Cheating: con programas como Cheat Engine, se pueden modificar algunas variables del juego hookeando el proceso y descifrando cómo es tratada la información del estado de juego para que se pueda modificar. También existen programas que modifican/añaden nuevas capabilities al juego dando una ventaja concreta como aimbots o triggerbots, así como métodos que permiten ralentizar el envío de datos para poder anticiparse a otros jugadores.

- Bots: dentro del cheating se pueden encontrar los bots, software que automatiza el comportamiento de un jugador para alcanzar objetivos más rápido en tareas repetitivas o para fines más dañinos como el spam y el acoso.

Más adelante se describen las vulnerabilidades en las compras internas de un juego y los ataques a partir de ingeniería social.

3.2.1.9. Defensas comunes

A partir de la máxima "nunca confíes en el cliente", muchos desarrolladores tratan de que solo la mínima información indispensable llegue al cliente y que sea un servidor autoritario el que valide todas las acciones y filtre la información que se envía de vuelta a cada jugador. En algunos casos, como pueden ser los juegos de móvil, también se obfusca el código para hacer más difícil a un atacante averiguar el comportamiento del juego y poder modificarlo.

Otra medida poco popular entre los jugadores y cada vez más común en los juegos, es el software anticheating, un escáner que comprueba que no haya otros programas abiertos aparte del juego y que se usen para hacer trampas, tales como Warden en Blizzard y Vanguard en Riot. Muchos jugadores reportan problemas con estos programas a la hora de correr los juegos y en muchos casos pueden ser bypasseables.

Respecto a los bots, todavía se trata de encontrar la mejor manera de detectar comportamientos automáticos en los jugadores, ya sea con estadísticas o supervisando las acciones manualmente.

En cuanto a minimizar el impacto del ataque de fuzzing, actualmente muchas desarrolladoras cuentan con una política de contraseñas para evitar que los jugadores usen credenciales fáciles de adivinar por programas de cracking, así como formas de encriptar y proteger los datos de sus usuarios.

3.2.1.10. Economía del juego

Dentro de algunos juegos se incluye la funcionalidad de realizar compras dentro de la aplicación y muchos poseen su propia moneda para intercambiar y comprar objetos. Es una de las funcionalidades más críticas en muchos casos, ya que se requiere información sensible de los usuarios para realizar las transaciones, entre ellas la tarjeta de crédito, y un ataque exitoso supone una gran pérdida monetaria y la desconfianza de los usuarios.

3.2.1.11. Ataques y vulnerabilidades

Dentro del apartado de cheating, también podemos encontrar la modificación de la cantidad de dinero desde el cliente, así como aprovechar fallos poco comunes como transaciones no cifradas y bugs en la plataforma de pago. La vulnerabilidad más importante y más difícil de controlar es la obtención de datos de pago mediante ingeniería social y phishing.

3.2.1.12. Defensas actuales

Actualmente la mayoría de juegos cuentan con plataformas de pago seguras que cifran el intercambio de datos y obligan a hacer varias verificaciones al usuario para comprobar la legitimidad del pago, así como que dicha información sea controlada por el servidor y no por el cliente

Por otro lado, algunas empresas cuentan ya con departamentos de Threat Intelligence que tratan de introducirse en la mente de los atacantes y descubrir los intentos de phishing así como la venta de cheats y otros elementos ilegítimos que puedan afectar negativamente al usuario.

3.2.2. Ingeniería social

Siendo el mundo de los videojuegos tan atractivo y adictivo para muchas personas, ha surgido una forma de ingeniería social enfocada directamente a este mundo que busca obtener cuentas de usuario, dinero, objetos y/o datos de los jugadores mediante el engaño. Es el vector de entrada más común en la mayoría de ataques, ya que es muy eficiente para introducir malware dentro del sistema de un usuario a través de falsos enlaces, páginas web clonadas sobre el juego, cheats infectados, etc. Algunos de estos ataques son:

- Ransomware: el atacante cifra todo el contenido del juego, entre ellos el progreso del jugador y pide un rescate al usuario en moneda digital.
- Keyloggers: para obtener contraseñas, un atacante puede instalar este software en el ordenador de un jugador a partir de enlaces o software infectados.
- Scareware: el usuario instala un software que aparentemente legítimo en el que luego el atacante hace creer a la víctima que su dispositivo está infectado y tiene que pagar una cantidad de dinero para mantenerse a salvo.

- Account Takeover: mediante phishing, un atacante consigue que el usuario introduzca sus credenciales en una página que cree legítima para luego apoderarse de su cuenta y usarla como parte de una botnet o realizar fraudes.
- Fraude: el atacante hace creer al usuario que vende material o progreso de un juego y obtiene así dinero de manera ilegítima.
- Spam: consiste en el envío de mensajes no deseados de manera repetitiva que en muchos casos son intentos de phishing.

La principal defensa para estos ataques (Cihodariu, 2019), desde un punto de vista social, es la educación de los jugadores en seguridad a la hora de crearse una cuenta y conectarse a una partida, así como participar en la comunidad que rodea al producto. Algunas de las directrices que se tratan de transmitir a los jugadores se describen a continuación:

- Usar contraseñas seguras: la desarrolladora debe tener una política severa de contraseñas y mientras los jugadores no deben usar credenciales usadas anteriormente, deben evitar palabras conocidas o relacionadas con su entorno, no compartir esa información con nadie y usar todos los medios recomendados de autenticación como 2FA.
- Identificar intentos de fraude: por un lado la desarrolladora debe tener un departamento que detecte estos intentos de fraude dentro y fuera de un juego, mientras que los jugadores tienen que ser conscientes de que todas las ofertas que prometen ventajas en el juego y que no se transmiten a través de canales oficiales, son intentos de frause.
- No hacer trampas: la venta de cheats en el canal principal para infectar los ordenadores con malware. Los jugadores deben de ser castigados si usan cheats y avisados del peligro de instalar ese software que es en pocos casos fiable.
- Obtener los juegos por canales oficiales: descargar el software de fuentes fiables, nunca de páginas que lo ofrecen sin coste o a menor precio, así como versiones crackeadas.
- Actualizar el software y los dispositivos: los jugadores deben siempre mantener los juegos y los dispositivos que los alojan actualizados, ya que muchas de esas actualizaciones incluyen parches para vulnerabilidades encontradas.

3.2.3. Motivaciones de los atacantes

Todos los ciberataques son motivados por una razón, ya sea tan simple como el aburrimiento o tan compleja como una conspiración gubernamental. Todos los casos tienen en común la existencia de una o varias víctimas que sufren el ataque, por lo que en esta sección se pretende enumerar la mayor parte de motivaciones posibles de estos ataques:

- Aburrimiento: dado que uno de los mayores "defectos"de los hackers en la curiosidad, hay casos en los que un atacante quiere probar hasta donde es capaz de llegar dentro de un juego. A veces no se busca perjudicar directamente a un usuario y es involuntario el alcance de las consecuencias. En otros casos, dado que internet nos proporciona cierto grado de anonimato y nos desconecta emocionalmente de ciertas acciones, un atacante puede sentirse tentado de comprobar cuánto daño puede hacer.

- Beneficio económico: el atacante espera obtener mucho dinero a través del fraude, venta de cheats, phishing, hacer trampas, etc. En este caso no se busca "destruir.[el] servicio, si no aprovecharse de él.
- Obtener ventaja in-game: en algunos casos el atacante tiene una motivación competitiva de ser el mejor en el juego y para ello usa cheats o neutraliza las conexiones del resto de jugadores.
- Venganza: el atacante quiere hacer daño a un jugador o entidad por alguna afrenta anterior que no siempre le ha afectado directamente, como por ejemplo que una compañía tome una decisión que le puede perjudicar.
- Intereses empresariales: en el ámbito de la competitividad entre entidades empresariales por un mercado como el de los videojuegos, algunos ataques son movidos por el interés de hacer daño a una compañía en beneficio de otra, ya sea tirando su servicio como robando información sensible.
- Ego: un atacante busca la satisfacción del reconocimiento por un ataque, el destacar por encima del resto y demostrar de lo que es capaz por su propio ego.

3.2.4. Impacto económico y social de un ciberataque

A continuación se exponen los dos sectores más afectados por un ciberataque y sus consecuencias.

3.2.4.1. Jugadores

Los jugadores son siempre los afectados en todo tipo de ataques, incluso aunque no vayan dirigidos expresamente a ellos, siguen sufriendo las consecuencias. A continuación se enumeran algunos de los impactos que acarrea un ataque:

- Pérdida de confianza: cuando un usuario sufre un ataque que no podía prevenir, como el robo de sus datos a la compañía a la que se los proporcionó o que otros usuarios usen cheats libremente, este deja de confiar en la seguridad del servicio y cambia de proveedor o deja de utilizar el servicio. Además de poder producirse el efecto dominó, en el cual un jugador expresa a otros su desconfianza y se unen a la opinión de dicho individuo.
- Salud mental: en algunos casos, un ataque puede conllevar consecuencias en la salud mental debido a la pérdida de datos sensibles, dinero o por sufrir acoso.
- Descontento: en casos menos críticos como puede ser el cheating, la consecuencia más común es el descontento de los jugadores, que demandan un control para partidas justas y limpias a los proveedores. No necesariamente conlleva al abandono del juego por parte de los usuarios afectados.

3.2.4.2. Compañías

Los ataques que tienen como objetivo hacer daño a una compañía, suelen ser los más críticos,ya que pueden implicar robo de datos a gran escala o tirar los servicios que proveen.

Incluso los ataques individuales que sufren los usuarios por medio de ingeniería social acaban afectando a una compañía por el reclamo de responsables y el posible abandono de la plataforma. A continuación se describen algunos de los impactos más relevantes que sufren las desarrolladoras como consecuencia de un ataque:

- Pérdida económica: el impacto más común y más dañino es la pérdida de dinero, ya sea como consecuencia del abandono masivo del juego, que dicho juego esté inoperativo o que se produzca una brecha de seguridad que implique el robo de dinero a la compañía.
- Despidos: una de las consecuencias menos visibles despúes de un ataque, es aquella que afecta a los empleados de la compañía. Ya sea por la pérdida económica o por señalar culpables, una empresa puede decidir prescindir de personas que formaban parte del proyecto.
- Pérdida de popularidad: dado que actualmente todo está en la red y todo el mundo está comunicado como nunca antes, uno de los factores más importantes para el éxito dentro de la industria del videojuego es la popularidad, que conozcan tus productos y tus servicios y que haya una buena opinión de ellos. Despúes de sufrir un ataque, la opinión cambia y se expande llegando a todo el mundo, lo que hace que decaiga la popularidad y como resultado se ignore o se rechace esa compañía.

3.3. Seguridad en Aplicaciones de móvil

Las aplicaciones de móvil en su mayoría tienen una arquitectura cliente-servidor. El cliente corre en el dispositivo iOS o Android y es con lo que interacciona el usuario, mientras por detrás hay un servidor a donde se mandan interacciones del usuario para validar o guardar datos en concreto, como transferencias, inicios de sesión, progreso, etc. Este servidor suele ser una aplicación web que se comunica con el cliente a través de una API, lo cual hace de la seguridad un aspecto muy amplio en el que tener en cuenta la aplicación persé, la web y la API. Actualmente, a pesar de que muchas aplicaciones ya tienen por defecto que solo puedan acceder a los directorios del móvil donde están instaladas y que no puedan modificar archivos del sistema, los desarrolladores pueden cometer errores a la hora de elaborar el producto y pueden encontrarse fallas de seguridad que son aprovechadas posteriormente por atacantes.

Las razones por las cuales hablamos de seguridad en aplicaciones móvil son, entre otras, el hecho de que el desarrollo se centre más en la funcionalidad que en la seguridad, que en muchas ocasiones los desarrolladores no piensen en las características de la plataforma para la que desarrollan y que los usuarios no sean conscientes de la seguridad, además de confiar en la nueva tecnología con facilidad y ser vulnerables a la ingeniería social.

Los ataques a aplicaciones tienen como objetivos conseguir credenciales (del móvil o del servicio), datos personales, datos bancarios y acceso al dispositivo. Podemos distinguir entre client-side y server-side, siendo ligeramente más común el primer tipo, un 60 % (Technologies, 2019).

La mejor fuente de conocimiento respecto a este tema, además de ser gratuita y abierta, es OWASP, que ha elaborado un estándar para la seguridad en aplicaciones de móvil con tres guias:

- Mobile Security Testing Guide (MSTG): manual para testear la seguridad de aplicaciones y hacer ingeniería inversa de iOS y Android, muy útil para testers.
- Mobile Application Security Verification Standard (MASVS): estándar para la seguridad en el desarrollo de aplicaciones móviles, dirigido a desarrolladores para hacer apps más seguras y a testers de seguridad para asegurar que los tests son completos y coherentes.
- Mobile Application Security Checklist: lista de vulnerabilidades que comprobar durante tests de seguridad, basándose en las dos guias anteriores.

3.3.1. OWASP Top Ten

OWASP también elaboró un top ten de vulnerabilidades en aplicaciones móviles con ayuda de toda la comunidad, actualmente con dos versiones: 2014 y 2016 (Haddix, 2016), esta última más centrada en la parte de aplicación móvil en vez del servidor. A continuación se listan y describen cada una de las vulnerabilidades:

- M1 - Improper Platform Usage: todo lo que concierne al uso inadecuado de las características de seguridad que tiene el sistema operativo de una plataforma, incluyendo guias y mejores prácticas, como los permisos de las apps, la keyChain de iOS, Android intents, etc. Esta vulnerabilidad supone una amplia variedad de escenarios de ataque, en su mayor parte del lado del servidor.

  Para remediar este riesgo, desde el lado del servidor tiene que haber un código seguro y seguir las mejores prácticas en seguridad, así como limitar la comunicación entre aplicaciones, implementar permisos de archivo, usar keychain de iOS, establecer controles, etc.

- M2 - Insecure Data Storage: en el caso de que el dispositivo sea extraído por un atacante o sea víctima de un malware que permita a el acceso a los datos, si la aplicación no almacena los datos en una localización que no sea accesible fácilmente por otra app o un individuo, datos sensibles pueden ser filtrados. Un desarrollador nunca debe dar por hecho que nadie va a tener acceso a los archivos del sistema, dado que un dispositivo puede usar Jailbreak o Rooting, que permite el control total del móvil.

  Algunas formas de mitigar el impacto de esta vulnerabilidad incluyen evaluar cuando es necesario encriptar los datos, implementar obfuscación y protección contra tampering, establecer comprobaciones de autenticación frecuentemente y evitar tratar datos si no es necesario.

- M3 - Insecure Communication: esta vulnerabilidad afecta a los datos transmitidos entre cliente y servidor. A veces la aplicación manda información sensible innecesariamente o sin estar encriptada, lo que hace que esté disponible para un atacante que intercepta las comunicaciones. Esto puede hacerse a través de una misma red, por malware o con un proxy.

  Por ello es recomendable implantar certificados SSL de CA y avisar si se detecta un fallo en dichos certificados, además de seguir los protocolos recomendados de encriptación de datos así como seguir las mejores prácticas de seguridad en ese aspecto.

- M4 - Insecure Authentication: en muchas aplicaciones, un usuario tiene que identificarse para acceder a los datos de dicha app. Esta vulnerabilidad implica la capacidad de un atacante de aprovechar fallos en la aplicación para bypassear la autenticación. Se debe comprobar la identidad del usuario durante las requests que se hacen al backend de la app y que implican la transmisión de datos sensibles o acciones críticas.

  Se recomienda que los métodos de autenticación se ejecuten en el lado del servidor, además de añadir capas de seguridad como usar MFA e incluir una identificación del dispositivo que se está usando como parte de la autenticación.

- M5 - Insufficient Cryptography: en el caso de que el dispositivo sea extraído o se haya infectado con un malware con acceso a los datos de la aplicación, si dichos datos no están debidamente encriptados, serán fácilmente accesibles por un atacante. Esto puede ser el resultado de usar de manera incorrecta las claves digitales, implantar algoritmos de encriptación pobres o usar métodos fácilmente bypasseables.

  Es recomendable usar algoritmos fuertes que cumplan con los estándares de criptografía y evitar guardar datos sensibles en el dispositivo siempre que sea posible.

- M6 - Insecure Authorization: esta vulnerabilidad afecta a los permisos que tiene cada usuario dentro de la aplicación, cuando no se comprueban o definen debidamente los privilegios de un usuario cuando interacciona con la app. Esto puede suponer el robo de datos sensibles, entre otros.

  Para mitigar el impacto es importante verificar en cada requests los permisos que tiene el usuario que las hace, así como comprobar con la información del backend y no la local la identidad de un usuario.

- M7 - Client Code Quality: esta vulnerabilidad está relacionada con errores en el código cuando se implementan métodos propios y/o customizados, lo cual tiene como consecuencia que sea vulnerable a ataques de buffer overflow, remote code execution, etc. También supone el uso de librerías de terceros que incluyen fallos en el código, lo que las hace vulnerables.

  En la actualidad hay muchas herramientas automaticas que analizan el código de una aplicación en busca de bugs y malas implementaciones que mitigan la posibilidad de que el código sea vulnerable. Además se recomienda que se revise el código durante el desarrollo y se realicen test de seguridad de manera continua, no solo al final, integrándolo en el SDLC.

- M8 - Code Tampering: un atacante modifica el código de una aplicación que publica posteriormente en tiendas de terceros o hace que un usuario la instale mediante phishing. Estas modificaciones pueden hacerse en los binarios, en los recursos, etc y el atacante puede hacer que dicha app parezca legítima.

  Para evitar esta vulnerabilidad, se recomienda que la app detecte cuando su código ha sido alterado y que se implementen métodos para validar la legitimidad de la app (con certificados propios, checksums, etc), así como detectar si el dispositivo está rooteado o usa Jailbreak.

- M9 - Reverse Engineering: un atacante decompila y hace ingeniería inversa sobre la aplicación, analizando su funcionamiento para posteriores ataques, así como inspeccionar los binarios en busca de datos sensibles con herramientas como IDA Pro, Radare2 o Hopper.

Para mitigar el impacto de esta vulnerabilidad, se recomienda obfuscar el código de manera que no pueda ser de-obfuscado por un atacante con las herramientas mencionadas anteriormente.

- M10 - Extraneous Functionality: a veces los desarrolladores se dejan funcionalidades activadas que no estaban planeadas para formar parte de la versión final de la app. Un atacante puede encontrar estas funcionalidades y explotarlas para conseguir privilegios.

  Se recomienda revisar la aplicación en busca de posibles funcionalidades escondidas que se hayan usado para el desarrollo y comprobar que no se revela demasiada información sobre el backend de la app en los logs.

3.3.2. Seguridad en Android

Las aplicaciones de Android están programadas en Java y corren en una máquina virtual dentro del dispostivo. Dado que Android es open source, debe cumplir con los requisitos de una amplia variedad de dispositivos y en el caso de las aplicaciones, no pasan por controles tan minuciosos como en iOS.

3.3.2.1. Medidas de seguridad

Aunque esté basado en el kernel de Linux (Park, 2012), Android implementa sus propios métodos de seguridad que se describen a continuación:

- Process Isolation: previene la interación directa entre aplicaciones dentro de un dispositivo Android, colocando cada una en una dirección de memoria diferente y haciendo sandboxing de su PDU (Process Data Unit). Esto último asigna un UID a cada aplicación y el kernel maneja el acceso a la memoria a través de ese PID. Además se implementa el acceso seguro a funciones del sistema mediante llamadas a una API.

- Secure Boot: mediante boot verification, se garantiza la integridad del sistema operativo, esto es, una cadena de confianza que empieza en una raíz y acaba en la partición de archivos. Durante el boot, cada punto de la cadena verifica la integridad (hash) y la autenticidad (firma) del siguiente punto antes de ejecutarlo.

  La ROM lanza el bootloader y verifica su integridad con la OEM Key (poner definition), un hash del hardware en la ROM, para comprobar que no ha sido modificado. Android también tiene una clasificación de estados del boot que nos dirá el nivel de protección del proceso de inicio.

  - Verde: todas las verificaciones han sido exitosas, desde el bootloader a todas las particiones.
  - Amarillo: el dispositivo está desbloqueado pero la partición ha sido comprobada usando el certificado incorporado y es válido.
  - Naranja: el dispositivo está desbloqueado pero no se ha podido hacer ninguna verificación, por lo que la integridad se deja al usuario.
  - Rojo: el proceso de verificación ha fallado, por lo que el proceso de inicio se ha detenido.

El dispositivo también puede ser bloqueado y desbloqueado, por lo que se usa dm-verity, un driver que verifica todas las particiones del sistema y por último la OEM Key comprueba la imagen del SO cargada por el bootloader (Figuras 3.1 y 3.2).

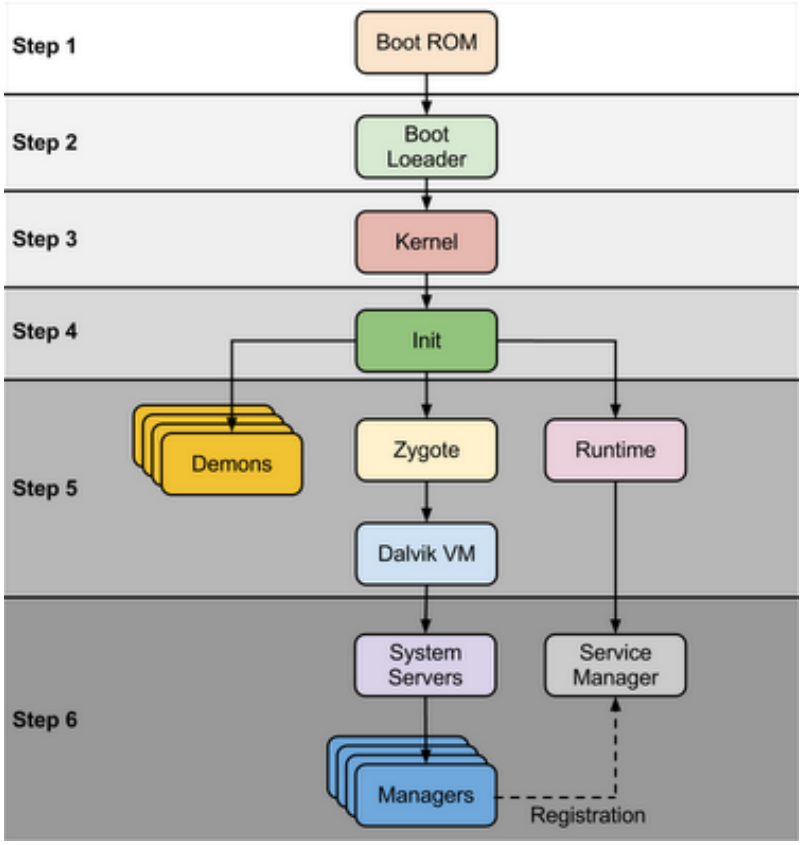


Figura 3.1: Boot flow

- SELinux: es un sistema de control de acceso obligatorio (MAC) de linux que etiqueta los procesos de un dispositivo con privilegios de root para que solo escriba en los bloques de memoria que le permita la política implementada, así se evita que se sobreescriban datos del sistema fuera de un bloque específico. Se basa en sujetos (apps o procesos), objetos (archivos, procesos) y acciones (escribir, leer, etc).

3.3.2.2. Estructura de Android

Android está basado en el kernel de Linux, con un sistema de archivos en su mayoría EXT4 y una arquitectura ARM, aunque algunos soportan Thumb, x86 o MIPS. Eso significa que a la hora de hacer ingeniería inversa sobre una librería, el set de instrucciones será probablemente ARM (Figura 3.3).

- Sistema de archivos: se divide en varias particiones.
  - Boot loader: partición de solo lectura, es el primer código que se ejecuta cuando se enciende el móvil y carga el kernel de Android.
  - Boot: incluye el kernel.

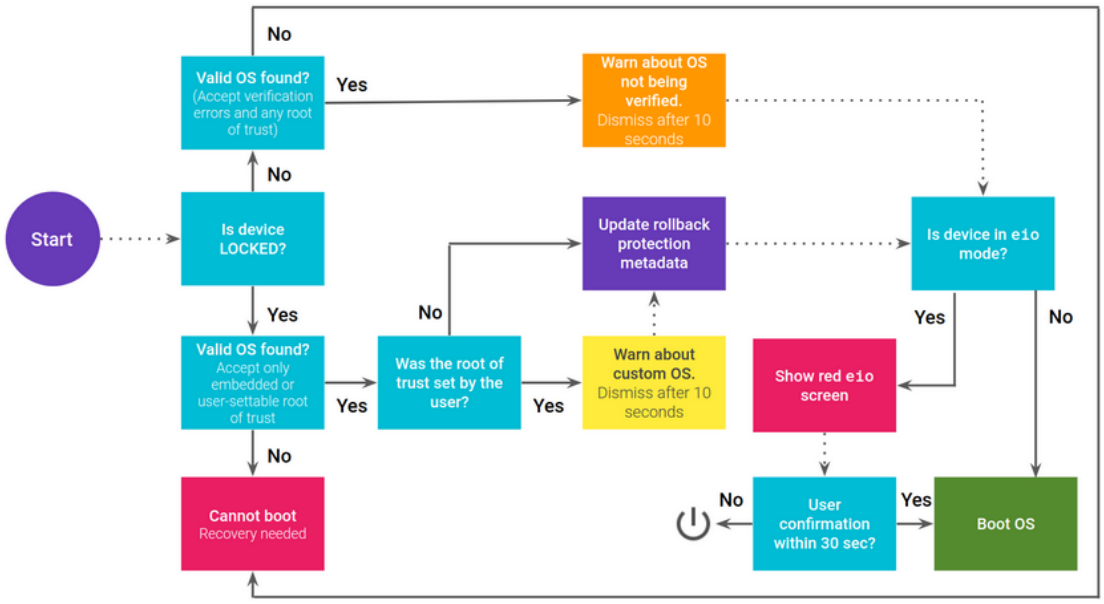


Figura 3.2: Boot flow completo con estados en detalle

- Splash: guarda la imagen cargada cuando se inicia el dispositivo.
- User data: almacena todos los datos del usuario.
- Sistema: incluye librerías, aplicaciones del sistema y el framework de Android.
- Cache: almacena temporalmente archivos usados por distintas apps.

Todos los procesos tienen acceso restringido a determinados directorios.

- Aplicaciones: las apps se compilan en .dex y se comprimen junto con los recursos y el código nativo en el apk. Cuando se instala la aplicación, se descomprime en la máquina virtual de Dalvik o el Runtime de Android y el .dex se precompila como en Odex o Elf. Actualmente se usa Runtime de Android con Elf. El ensamblador de Dalvik se llama smali (Figura 3.4).

  En cuanto a la estructura del apk (Figura 3.5):

  - Manifest: contiene los permisos, servicios y actividades declaradas.
  - Res: contiene todos los recursos de la app, tales como imágenes, iconos, etc.
  - Resources.asrc: versión compilada de los recursos que contiene los ids.
  - Lib: el directorio donde se encuentran las funciones.
  - Código compilado: en formato .dex para Dalvik.
  - META-INF: metadatos y certificaciones.
  - Assets: más recursos adicionales.

3.3.3. Seguridad en iOS

Las aplicaciones de iOS están programadas en Objective-C o Swift. Estas apps son revisadas por equipos especiales de Apple y posteriormente distribuidas a través de la Apple Store.

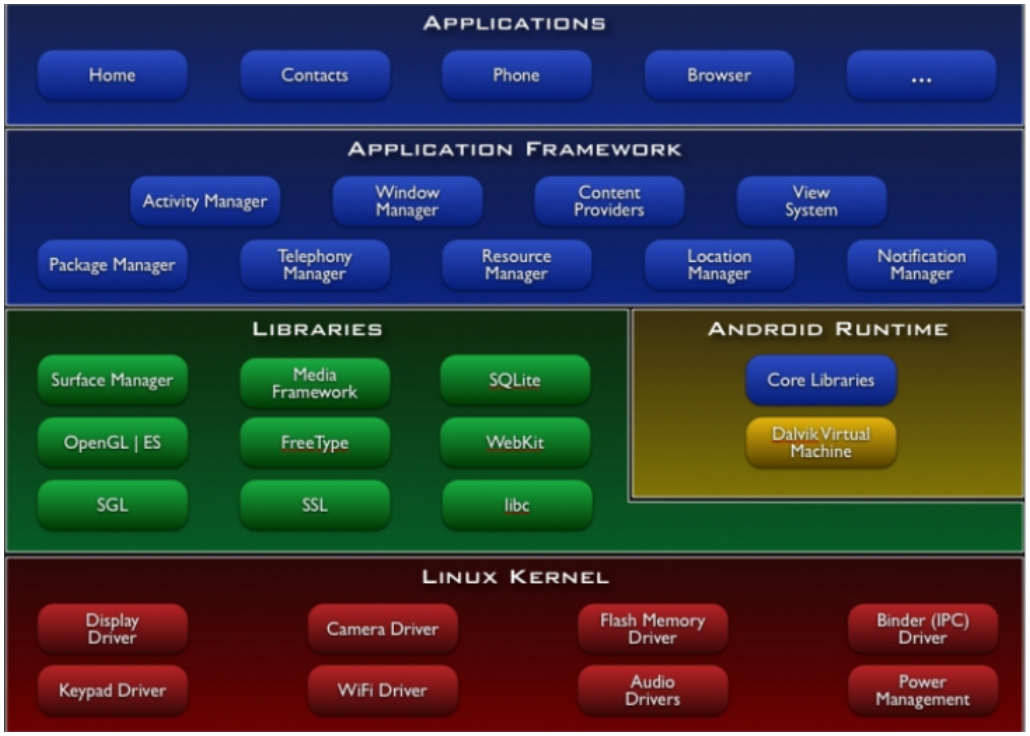


Figura 3.3: Arquitectura Android

3.3.3.1. Medidas de seguridad

La architectura de seguridad de Apple está muy documentada y se actualiza con cada nueva versión. Consiste en seis funcionalidades:

- Hardware security (CryptoEngine): se encarga de generar las claves del dispositivo para proteger la keychain y otros archivos. Las distintas claves que genera se enumeran a continuación:
  - UUID Key: única en cada dispositivo
  - GID Key: clave compartida por procesos similares.
  - Passcode Key: derivada del PIN de usuario. Se almacena hasta que el dispositivo se bloquea.
  - Class Key: para proteger archivos según su criticidad.
  - File Key: clave para archivos específicos.
- Secure Boot: similar al de Android, la ROM contiene código inmutable y el certificado de Apple Root, el cual es implantado creando lo que se llama root of trust. A partir de ahí va punto por punto validando las firmas de cada parte del proceso y si alguna falla, entra en modo recovery.
- Code signing: Apple tiene un sistema de DRM que solo permite que código aprobado por Apple pueda ejecutarse en sus dispositivos.
- Secure Enclave: proceso encargado de cifrar la memoria, generar números aleatorios por hardware y llevar a cabo todas las operaciones de cifrado y gestiones del TouchID.

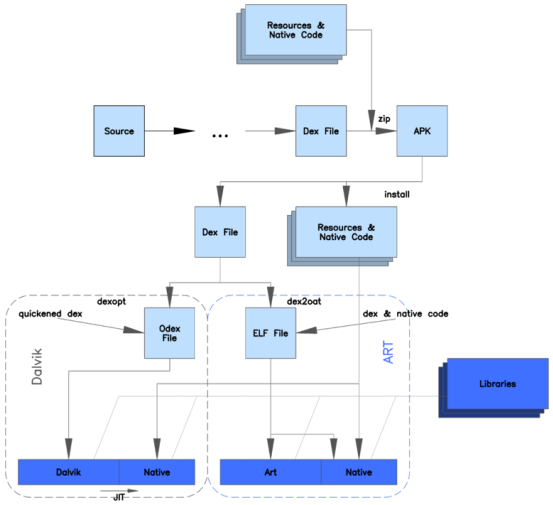


Figura 3.4: Ciclo de vida de aplicaciones

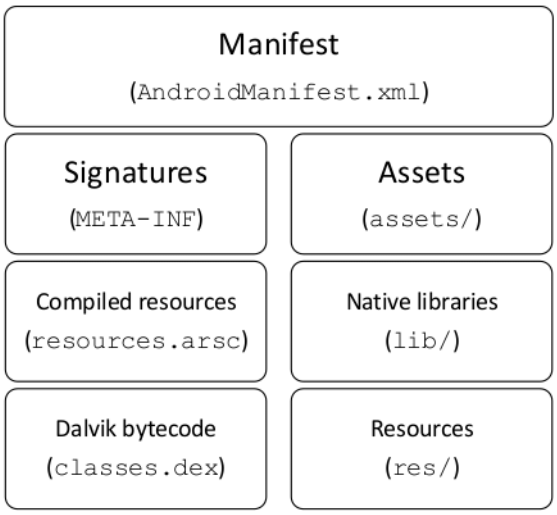


Figura 3.5: Estructura del apk

- Secure Element: encargado de guardar los datos de la tarjeta de crédito en transacciones móviles.
- Secure FileSystem: tiene dos particiones:
  - Partición del sistema: /dev/disk0s1s1 en /, que es montado para solo lectura y contiene los binarios y librerías del sistema, así como el /etc/passwd, etc.
  - Partición del usuario: /dev/disk0s1s2 en /private/var donde todo está cifrado. Contiene información de las aplicaciones, logs y lo más importante, la keychain, donde se almacena de manera segura todos los datos sensibles y gestiona cuando son accesibles.
- Sandboxing: cada aplicación es aislada del resto a nivel del kernel y su objetivo es reducir el daño que se puede producir si una app es comprometida.
- Jailbreak: aunque no es una medida de seguridad en sí, se usa para testear la seguridad

de un dispositivo/aplicación y se basa en deshabilitar la firma de código del sistema para tener acceso como root a todo el sistema de archivos.

3.3.3.2. Estructura de iOS

iOS está basado en Drawin, un SO open source de Unix desarrollado por Apple. El kernel es XNU, un híbrido que combina los kernels de Mach y FreeBSD. Las aplicaciones de iOS son aisladas a nivel del sistema de archivos y están muy limitadas en términos de acceso al sistema a través de la API (Figura 3.6).

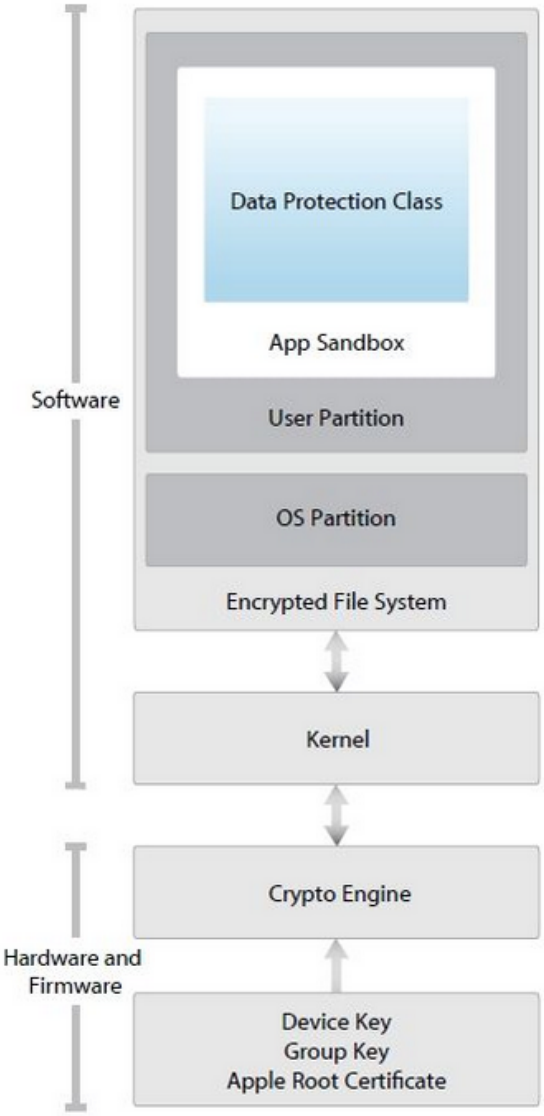


Figura 3.6: Estructura de iOS

- Sistema de archivos: como descrito en el apartado anterior, se divide en dos particiones, una para el sistema donde se almacenan binarios y librerías del SO y otra para el usuario, enteramente cifrada, donde se encuentran todos los datos de aplicaciones, logs y la keychain.

Dentro del sandbox de Las aplicaciones se crean los directorios de /Documents, /Library y /tmp para algunas de sus funciones.

- Aplicaciones: las aplicaciones son comprimidas en un .ipa, un zip que contiene todos los recursos y el código de la app. Los binarios vienen encriptados con el DRM de Apple para evitar que se examinen. Los IPA se crean a partir de XCode (Figura 3.7).

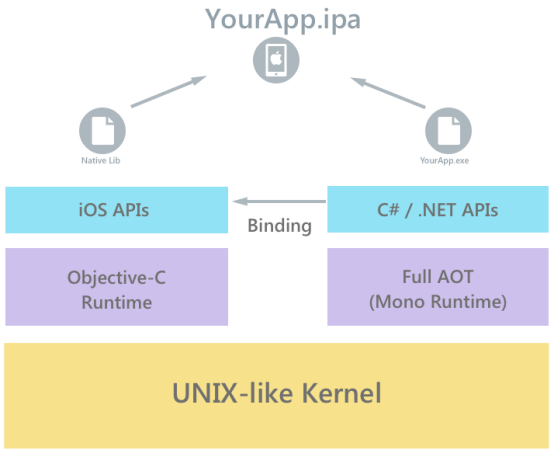


Figura 3.7: Ciclo de vida de aplicaciones iOS

En cuanto a la estructura de un ipa (Figura 3.8):

- Payload: contiene todos los datos de la app.
- Payload/Application.app: contiene los datos de la app en sí con el código compilado y sus recursos estáticos asociados.
- iTunesArtwork: PNG usado como icono de la app.
- iTunesMetadata.plist: contiene info sobre el ID del desarrollador, el ID del bundle, información del copyright, género, nombre de la app, etc.
- WatchKitSupport/WK: es un ejemplo de una extensión del bundle, que en este caso da soporte a las interacciones con AppleWatch.
- META-INF: metadatos y certificaciones.

3.4. Pentesting

Pentesting es la práctica de atacar diversos entornos con la intención de descubrir vulnerabilidades, fallos u otros problemas de seguridad para poder así prevenir ataques externos hacia esos equipos en el futuro (Nuñez, 2018). Podemos distinguir entre tres tipos de pentesting:

- Caja Blanca: el equipo que realiza el pentest lo hace a partir de un usuario de la plataforma con todos los privilegios y conociendo el entorno.
- Caja Gris: el pentest se realiza con un usuario con pocos o ningún privilegio y con algo de información sobre el entorno.

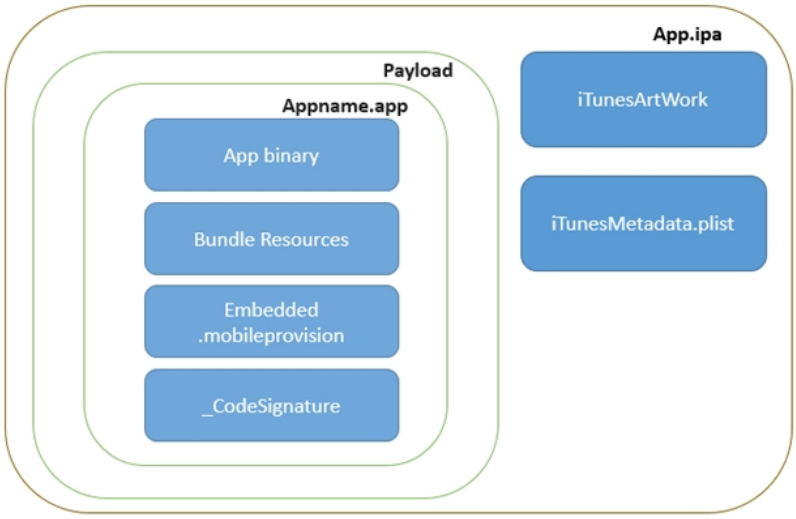


Figura 3.8: Estructura de un ipa

- Caja negra: es el pentest que más se acerca a una situación real en la que el equipo no posee ninguna infomación del entorno y debe realizar una fase de reconocimiento más exhaustiva exponiendo las vulnerabilidades más externas de la compañía como haría un atacante ajeno.

Las pruebas de penetración son llevadas a cabo por el llamado Red Team, el equipo ofensivo que ataca el entorno para destapar vulnerabilidades con el consentimiento de las compañías. A esta práctica también se le denomina Hacking Ético, ya que el objetivo es una buena causa, la de prevenir ataques permitiendo a las compañías arreglar esas vulnerabilidades encontradas.

3.4.1. Fases

Tras el acuerdo entre ambas partes para la realización del pentest sobre la plataforma, se lleva a cabo la prueba siguiendo las fases que se describen a continuación:

- Planificación y reconocimiento: en esta fase se define el alcance que tendrá la prueba y los objetivos de esta, así como los sistemas que van a ser objeto del pentest y los métodos que se van a usar. También la parte interesada proveerá de la información necesaria al encargado de la evaluación, como cuentas de usuario, características de la plataforma (SO, aplicación, red y posibles servicios remotos) Posteriormente se realiza el análisis del entorno, "jugandoçon la plataforma y estudiando su comportamiento así como recolectando información y valorando posibles puntos de entrada y/o vulnerabilidades. También se identifican los datos sensibles que pueden ser accedidos en reposo, en uso o en tránsito.
- Escaneo de vulnerabilidades: en este paso se pone a prueba a la plataforma con los ataques más comunes para evaluar como reacciona y comprobar si es de primeras vulnerable, así como la inspección del código disponible en busca de posibles datos sensibles o útiles para posteriores ataques. En esta fase se suelen usar herramientas automáticas que testean la plataforma con una batería de ataques comunes. Posteriormente se analizarían los resultados.

- Ganando acceso: los testers usan la información de la fase anterior para explotar las vulnerabilidades encontradas y tratar de escalar privilegios en la plataforma, así como analizando las comunicaciones y robando datos para evaluar cuánto daño pueden hacer.

- Mantener el acceso: se comprueba si una vez se ha comprometido la plataforma, un atacante podría mantener el control de manera persistente para posteriores ataques. Se pretende así evitar que las amenazas persistan en un sistema.

- Análisis y reporte: los resultados del pentest son descriptos en un reporte que incluye cada vulnerabilidad encontrada, el riesgo, su categoría, cómo afecta a la plataforma, una descripción detallada de la vulnerabilidad y como se ha reproducido, así como lo que se ha visto comprometido. También se incluyen recomendaciones de cómo arreglar estas vulnerabilidades y evidencias en forma de capturas.

3.5. Pruebas de seguridad en móviles

El pentesting de móvil consiste en testear la seguridad de la forma en la que lo haría un atacante con el objetivo de comprometer una aplicación. Para ello se debe tener un conocimiento amplio de la funcionalidad de la app y el tipo de datos que maneja (el grado de sensibilidad/criticidad). Debemos distinguir entre varios tipos de aplicaciones:

- Nativas: Android y iOS tienen su propio SDK para que los desarrolladores elaboren apps específicas para su plataforma. Garantizan una gran fiabilidad y rapidez, ya que se han diseñado según los principios concretos del SO y las apps pueden acceder directamente a casi todos los componentes del dispositivo. En el caso de Android, provee de dos kits de desarrollo, SDK para Java y Kotlin y NDK para C/C++, esta última para hacer código que acceda a APIs de bajo nivel.

- App Web: estas aplicaciones corren en el navegador del dispositivo y la mayoría están programadas en HTML5. Su integración con los componentes del dispositivo es limitada al correr dentro de un navegador y su performance es menor. Están pensadas para funcionar en varias plataformas por lo que su UI no sigue ningún principio de diseño. La ventaja es la reducción de costes de mantenimiento y la posiblidad de actualizarla sin tener en cuenta las especificaciones de la tienda de una plataforma.

- App Híbrida: la app se ejecuta como una nativa pero varios de sus procesos se ejecutan en la web, lo que se llama WebView. Con un código base se pueden hacer apps que tengan como target varias plataformas con una UI similar a la específica de un entorno.

- App Progresiva: apps que se cargan como una página web, pero se pueden usar offline y acceder a través de un móvil, lo cual solo era tradicionalmente posible si era nativa. Se configura mediante un manifiesto Web App y aunque es soportada por Andorid e iOS, todavía no dispone de todas las funcionalidades del hardware.

El pentest se divide en dos secciones de análisis, estático y dinámico, detallados a continuación:

3.5.1. Análisis estático

Consiste en la inspección del código fuente para buscar fallos de seguridad en la implementación. Por un lado se buscan métodos y llamadas a funciones que resulten interesantes para hacer hooking (interceptar dichas funciones para alterar su comportamiento) durante el análisis dinámico que impliquen APIs o accesos a bases de datos, así como fallos en el diseño o la lógica. Mientras, también se inspeccionan los binarios de la app manualmente o automáticamente para comprobar si hay datos sensibles o útiles para la explotación posterior.

En esta fase se incluye analizar los frameworks y librerías usados para comprobar que sus versiones no tienen vulnerabilidades conocidas.

La revisión es en su mayor parte manual y requiere que el tester tenga conocimientos de programación para que detecte de manera más eficiente las vulnerabilidades que puede haber en el código. También se hace uso de herramientas que automatizan la búsqueda y el análisis, pero no sustituyen el trabajo manual. Es un proceso que consume mucho tiempo del pentesting y es la parte más tediosa, sobre todo si se analiza una aplicación grande.

Algunas de las herramientas que se utilizan en esta fase son:

- Descompresores: dado que los apk e ipa son paquetes comprimidos, para extraer su contenido se usan herramientas típicas para descomprimir archivos, como unzip o apktool.
- Decompiladores: en el caso de Android, necesitamos herramientas para decompilar las clases de la app y hacerlas legibles para el usuario, como dex2jar.
- Desambladores: herramientas que se utilizan para reversing del binario y las librerías de la app para buscar datos sensibles "hardcodeados"(def) y comprender la lógica de la aplicación, tales como IDA Pro y Radare2.
- Editores de texto: para inspeccionar el código y los recursos, lo más sencillo de usar son editores universales como Sublime3 o VSCode, donde se puede abrir el directorio completo de la app.
- Escáner automático: en algunos casos se lanza un escáner con herramientas automáticas como MobSF para que realice un análisis exhaustivo de la parte estática.

Más adelante se describirá en más detalle las herramientas específicas para tests de seguridad en Android.

3.5.2. Análisis dinámico

Consiste en evaluar la seguridad de una aplicación en tiempo real, es decir, durante su ejecución. Se inspeccionan tanto la capa de aplicación como el backend y las APIs que se usan. También se incluye el análisis de las comunicaciones de la app con todos los servicios a los que está conectada. Se pretende comprobar si la app es vulnerable a la hora de almacenar y enviar datos, en la autenticación y autorización, en las configuraciones del servidor, etc. Durante esta parte del análisis se lleva acabo el tampering, lo cual consiste en modificar funciones interesantes que se han encontrado durante la fase estática para comprobar si

el cambio de comportamiento hace a la app vulnerable, fuzzing para evaluar como la app reacciona a input aleatorio y ver si saltan errores o se comporta de manera inesperada y la inspección de todos los directorios y archivos de la app por si guarda información sensible durante su ejecución. Algunas de las herramientas que se usan durante esta fase del análisis se describen a continuación:

- Capturador y modificador de comunicaciones: herramientas que interceptan las requests que la app hace al backend y/o a las APIs y permite enviarlas modificadas. Se usan para analizar los datos que se envían y hacer fuzzing. Las herramientas más usadas para tal propósito son Burpsuite y OWASP ZAP.

- Frameworks para hooking e inyección de código: permiten hookearse a funciones de la app y modificar su comportamiento durante su ejecución. La más usada para todas las plataformas es FRIDA.

- Browser de bases de datos: herramienta que se utiliza para inspeccionar las bases de datos de SQLite3

OWASP también ha definido una checklist[1] que seguir durante los pentests de aplicaciones móviles que permite a los testers de seguridad comprobar todas las funcionalidades que son potencialmente vulnerables y así contribuir a que las aplicaciones que salen al mercado sean lo más seguras posibles (Figura 3.9).

Detailed Verification Requirement	Level 1	Level 2	Status	Testing Procedure(s)
Architecture, design and threat modelling				
All app components are identified and known to be needed.	✓	✓		Architectural Information
Security controls are never enforced only on the client side, but on the respective remote endpoints.	✓	✓		Injection Flaws (MSTG-ARCH-2 and MSTG-PLATFORM-2)
A high-level architecture for the mobile app and all connected remote services has been defined and security has been addressed in that architecture.	✓	✓		Architectural Information
Data considered sensitive in the context of the mobile app is clearly identified.	✓	✓		Identifying Sensitive Data
All app components are defined in terms of the business functions and/or security functions they provide.		✓	N/A	Environmental Information
A threat model for the mobile app and the associated remote services has been produced that identifies potential threats and countermeasures.		✓	N/A	Mapping the Application
All security controls have a centralized implementation.		✓	N/A	Testing for Insecure Configuration of Instant Apps (MSTG-ARCH-1, MSTG-ARCH-7)
There is an explicit policy for how cryptographic keys (if any) are managed, and the lifecycle of cryptographic keys is enforced. Ideally, follow a key management standard such as NIST SP 800-57.		✓	N/A	Cryptographic policy
A mechanism for enforcing updates of the mobile app exists.		✓	N/A	Testing enforced updating (MSTG-ARCH-9)
Security is addressed within all parts of the software development lifecycle.		✓	N/A	Security Testing and the SDLC
A responsible disclosure policy is in place and effectively applied.		✓	N/A	
The app should comply with privacy laws and regulations.	✓	✓		
Data Storage and Privacy				
System credential storage facilities need to be used to store sensitive data, such as PII, user credentials or cryptographic keys.	✓	✓		Testing Local Storage for Sensitive Data (MSTG-STORAGE-1 and MSTG-STORAGE-2)
No sensitive data should be stored outside of the app container or system credential storage facilities.		✓		Testing Local Storage for Sensitive Data (MSTG-STORAGE-1 and MSTG-STORAGE-2)
No sensitive data is written to application logs.	✓	✓		Testing Logs for Sensitive Data (MSTG-STORAGE-3)
No sensitive data is shared with third parties unless it is a necessary part of the architecture.	✓	✓		Determining Whether Sensitive Data is Sent to Third Parties (MSTG-STORAGE-4)
The keyboard cache is disabled on text inputs that process sensitive data.	✓	✓		Determining Whether the Keyboard Cache Is Disabled for Text Input Fields (MSTG-STORAGE-5)
No sensitive data is exposed via IPC mechanisms.	✓	✓		Determining Whether Sensitive Stored Data Has Been Exposed via IPC Mechanisms (MSTG-STORAGE
No sensitive data, such as passwords or pins, is exposed through the user interface.	✓	✓		Checking for Sensitive Data Disclosure Through the User Interface (MSTG-STORAGE-7)
No sensitive data is included in backups generated by the mobile operating system.		✓	N/A	Testing Backups for Sensitive Data (MSTG-STORAGE-8)
The app removes sensitive data from views when moved to the background.		✓	N/A	Finding Sensitive Information in Auto-Generated Screenshots (MSTG-STORAGE-9)
The app does not hold sensitive data in memory longer than necessary, and memory is cleared explicitly after use.		✓	N/A	Checking Memory for Sensitive Data (MSTG-STORAGE-10)
The app enforces a minimum device-access-security policy, such as requiring the user to set a device passcode.		✓	N/A	Testing the Device-Access-Security Policy (MSTG-STORAGE-11)
The app educates the user about the types of personally identifiable information processed, as well as security best practices the user should follow in using the app.		✓	N/A	Testing User Education (MSTG-STORAGE-12)
No sensitive data should be stored locally on the mobile device. Instead, data should be retrieved from a remote endpoint when needed and only be kept in memory.		✓	N/A	
If sensitive data is still required to be stored locally, it should be encrypted using a key derived from hardware backed storage which requires authentication.		✓	N/A	
The app's local storage should be wiped after an excessive number of failed authentication attempts.		✓	N/A	

Figura 3.9: Captura Mobile Pentest Checklist - OWASP

3.5.3. Pruebas de seguridad en juegos Android

A pesar de que no se trata como una disciplina a parte, en los juegos hay que considerar hacer pruebas de seguridad en elementos que son característicos de este tipo de aplicaciones y que no se contemplan en un pentesting normal. Además hay que tener en cuenta que

[1] `https://github.com/tanprathan/MobileApp-Pentest-Cheatsheet/blob/master/Mobile_App_Security_Checklist-English_1.2.xlsx`

actualmente muchos juegos de móvil están construidos a partir de motores de videojuegos como Unity, lo cual obliga a testear también el código asociado a las librerías de Unity.

El ataque más común en los juegos es el cheating, en su mayoría desarrollado a partir de la ingeniería inversa. Se busca modificar variables críticas del juego para tener ventaja, por ejemplo cambiando variables que controlan el dinero que posee el jugador, lo cual hace perder beneficio a las desarrolladoras. Para ello existe mucho software especializado (CheatEngine, GameGuardian) y tutoriales de cómo reproducir tales ataques, lo cual hace necesario testear el código a fondo para calcular su vulnerabilidades a este tipo de ataques.

Muchos juegos también incluyen compras dentro de la app, por lo que si las comunicaciones no son seguras, existen bugs de lógica o se almacena información bancaria en las bases de datos SQLite3 generadas, el juego puede ser vulnerable al robo de información de tarjetas de crédito y su posterior explotación.

Por último, los juegos son muy susceptibles a tener el código no obfuscado y muy sencillo de debuggear y deducir su comportamiento, ya que se el objetivo principal es el entretenimiento del usuario para obtener más beneficio sin tener en cuenta la seguridad.

Actualmente, con motores como Unity se producen muchos juegos de manera sencilla sin requerir de unos conocimientos altos en desarrollo de aplicaciones, por lo que si el juego incluye una modalidad online, trata con los datos de muchos usuarios e incluye compras en la aplicación, es un asset crítico en un dispositivo.

Capítulo 4

Descripción del Trabajo

En este capítulo se desarrolla la implementación de la herramienta propuesta, empezando por el estudio de las tecnologías investigadas y su posterior descarte o uso en la solución. También se muestran resultados finales de las distintas funcionalidades de la herramienta en varios casos de estudio y sus ventajas respecto a soluciones actuales en el mercado.

4.1. RAGnarok

Radare APK Cybersecurity es una herramienta que se usa desde la terminal, lo cual significa que no posee una interfaz gráfica. Consiste en la implementación de la librería de Radare2, r2pipe, en un entorno de python3 para automatizar y facilitar la fase de análisis estático en pruebas de seguridad en aplicaciones Android.

4.2. Tecnologías usadas

A continuación de describen las tecnologías seleccionadas para la implementación de la herramienta RAGnarok.

4.2.1. APK-Anal

APK-Anal[1] (Figura 4.1) es una herramienta sencilla que usa la librería de Radare2, r2pipe, para automatizar el análisis estático de aplicaciones Android y junto a Apktool y APKiD asiste al tester en el proceso. Su nombre se debe a una de las librerías usadas por Radare2, anal.

Es la base de la que parte la herramienta propuesta, reutilizando gran parte de las funciones. Está desarrollada en python y utiliza argparse para gestionar los comandos y se ha probado en distintos análisis de malware en Android.

[1] `https://github.com/mhelwig/apk-anal`

4.2.1.1. Funcionalidades

Al ser una solución sencilla, no cuenta con muchas funcionalidades, pero cumple con unos mínimos que la hacen idónea para construir a partir de lo que ofrece:

- Detección de Root: comprueba si la aplicación hace algún check de si el dispositivo está rooteado.
- Detección de emuladores: mira si la app tiene alguna función que detecte si se está usando en un emulador del dispositivo.
- Archivos poco comunes: analiza si existen archivos sensibles que no tuviesen que estar en el directorio de la app.
- URLs e IPs: comprueba si existen URLs sensibles o IPs de algún servicio expuestas en el código.
- Funciones de accesos de la API (cámara, nfc, gps..): comprueba los permisos de la app y los accesos.

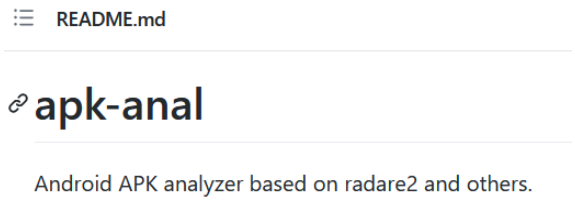


Figura 4.1: Pip command

4.2.2. MobSF

Mobile Security Framework[2] (Figura 4.2) es una herramienta para automatizar el análisis estático y dinámico de aplicaciones para malware o pruebas de seguridad.

Contiene una plataforma creada en un servidor local, containerizado o externo en la que se sube un apk para ser analizado. Dependiendo del tamaño tarda unos minutos y genera un reporte con el análisis estático del código de la aplicación listando las vulnerabilidades, su nivel de riesgo y la descripción de estas. Es muy útil para realizar un primer barrido de las vulnerabilidades más comunes en apps y provee de una API para poder utilizarla de manera externa desde cualquier programa.

La API permite hacer llamadas a las funciones del servidor de MobSF que se tenga levantado para hacer el análisis. Para el trabajo propuesto, se integra dicha API y se usan las funciones de subida de archivos, escaneo y reporte en pdf.

MobSF está desarrollado en python y corre en Windows y Linux. Está bien documentado y se actualiza con frecuencia.

[2] `https://mobsf.github.io/docs/#`

Figura 4.2: MobSF Logo

4.2.3. APKtool

Apktool[3] es una utilidad en línea de comandos que descomprime un apk y decodifica los recursos para que sean legibles a casi su forma original. También permite reconstruir el apk con las modificaciones que se hagan.

En el análisis estático se usa para poder inspeccionar los archivos de recursos en busca de información útil y/o sensible de la app. En el trabajo propuesto se utiliza por un lado para obtener el .dex y así poder tratarlo con Radare2 y por otro para un posterior análisis de los recursos.

4.2.4. Radare2

Radare2[4] es una herramienta compuesta por un conjunto de librerías y programas para realizar tareas de bajo nivel como ingeniería inversa, debugging, explotación, etc que puede ser automatizado en cualquier lenguaje de programación.

Tiene la ventaja de tener plugins para varias herramientas como Frida y Ghidra que asisten en el proceso de ingeniería inversa en aplicaciones.

4.2.4.1. Funcionalidades

Radare2 tiene una amplia variedad de funcionalidades, entre las cuales se usan para el proyecto las siguientes:

- Extracción de binarios: se extraen los binarios de la aplicación para comprobar si hay información sensible como credenciales o API Keys.
- Extracción de clases y sus métodos: filtra las clases de la aplicación buscando funciones interesantes que puedan aportar valor a la parte de análisis dinámico posterior.
- Extracción de imports: comprueba los imports de la app en busca de posibles vulnerabilidades cuando se importan funcionalidades o librerías.
- Extracción de símbolos: analiza si existen símbolos de debug o que aporten información sensible al tester.

[3] `https://ibotpeaches.github.io/Apktool/install/`
[4] `https://r2wiki.readthedocs.io/en/latest/home/radare2-python-scripting/`

4.2.4.2. R2pipe

R2pipe es la API de Radare2 para llamar a sus funciones desde código como si se invocasen desde la línea de comandos devolviendo un string como resultado. Las dos funciones más importantes son:

- r2.open(): sirve como punto de partida para abrir un proceso, ya sea un script, un binario, un servidor http,etc.
- r2.cmd(): se pasa el comando que se quiere correr en el proceso abierto las veces que se quiera.

4.2.5. Dependency-check

Utilidad de OWASP[5] para hacer un escaneo de las dependencias de un apk para comprobar que no son vulnerables y/o están desactualizadas.

Consiste en una gran base de datos del NVD que hace un recorrido por los CVE y genera un reporte en html del resultado del escaneo (más adelante se desarrolla).

4.2.6. ApkID

ApkID[6] es una herramienta que escanea un fichero .dex, .apk o un directorio y extrae información como el compilador, si usa obfuscador de código y cuál es, etc. Nos permite detectar si la aplicación ha sido modificada y compilada con un compilador no standard y si por tanto, puede tratarse de malware.

4.3. Desarrollo de la herramienta

A continuación se expone todo lo relacionado con el desarrollo del trabajo y la arquitectura del mismo.

4.3.1. Objetivos propuestos

Inicialmente la herramienta tenía que cumplir unos requisitos mínimos para poder llevarse a cabo y tener una utilidad más allá de la que ofrecen soluciones similares.

- Estar desarrollada en Python: python es un lenguaje interpretado que contiene muchos módulos relacionados con seguridad y utilidades que permiten desarrollar herramientas de manera más eficiente y sencilla. Su aprendizaje es muy rápido y su implementación está muy apoyada por la extensa documentación existente.
- Ejecutarse a partir de la línea de comandos: la herramienta no tiene que implicar una interfaz de usuario y ser sencilla de ejecutar, teniendo como objetivo ser un apoyo del proceso.

[5] `https://owasp.org/www-project-dependency-check/`
[6] `https://github.com/rednaga/APKiD`

- Automatizar tareas: se busca que esta herramienta automatice parte del proceso de análisis estático reduciendo el esfuerzo del tester de realizar un análisis manual preciso, por ello ejecutando un comando, se tiene que iniciar un proceso concreto de manera automática.
- Ser un apoyo: ninguna herramienta puede sustituir al 100 % el trabajo humano y su capacidad de decisión racional, por ello su objetivo principal es el de apoyo y su uso no debe sustituir las pruebas manuales.
- Orientada a Android: dado que Android y iOS tienen características que los diferencian a la hora de programar y comprimir las aplicaciones y almacenar los datos, para esta herramienta se ha priorizado que sea solo válida para tratar aplicaciones Android, siendo el SO móvil más utilizado y generalista actualmente en Europa (referencia de esto).
- Especializada en juegos: aunque dicha herramienta pueda ser generalista, en este proyecto se busca una especialización en el caso de los juegos buscando funciones típicas y que pueden ser vulnerables como la gestión de la economía en el juego.

4.3.2. Funcionalidades propuestas

La herramienta se planteó para cubrir no solo el grueso de un análisis estático, sino también la parte inicial de preparación del apk para su análisis. Estas funcionalidades son:

- Descomprimir el apk: dado que un apk es como un archivo comprimido zip, el programa lo cambiará de extensión a .zip y realizará la extracción del contenido obteniendo las clases de la aplicación en formato .dex.
- Decodificar recursos: con Apktool se decodifica el contenido del apk para que sea legible y pueda estudiarse su contenido.
- Extraer código fuente de Java: convertir las clases en .dex a clases de código fuente de Java para analizar y modificar la aplicación.
- Navegar por las clases de Java: usar la herramienta para desplegar las clases en Java extraídas y poder navegar por ellas y filtrar las funciones más interesantes.
- Filtrar y extraer las clases, métodos, strings y símbolos interesantes: utilizando un diccionario customizado y Radare2, integrado con la librería r2pipe, el programa iterará por el código de la aplicación sacando por pantalla aquello que ha filtrado de manera ordenada con un comando para cada filtrado.
- Detección de librerías y frameworks vulnerables: análisis de las versiones de los frameworks y librerías integrados en la aplicación para determinar si son vulnerables y/o están desactualizados con la herramienta de OWASP Dependency-check.
- Búsqueda automática del offset de los strings encontrados para su análisis: un comando que permita ir al offset donde se encuentra el string filtrado y customizar la cantidad de líneas que se muestran.
- Mejoras de implementación: usar otras librerías más eficientes y más "limpias"para algunos de los procesos implementados.

4.3.3. Funcionalidades actuales

Casi todas las funcionalidades propuestas se han cumplido en la herramienta final, aunque no todas han tenido el resultado esperado. Los únicos puntos que no han podido llevarse a cabo son: la navegación por el código fuente de Java, lo cual se ha determinado que sería más eficiente realizarlo de manera manual con una UI como JD-GUI; y la búsqueda automática de los offsets de los strings filtrados, por su complejidad de implementación y por no ser una funcionalidad que aporte automatización al proceso.

En cuanto a las mejoras de las funcionalidades base:

- Uso de *delegator* y *subprocess*: para ejecutar los comandos que llaman a las herramientas de apktool y apkid y que usen las instalaciones del sistema en vez de una ruta predefinida, se utilizan las librerías de delegator y subprocess combinadas.
- Uso de la librería Click: para sustituir argparse y poder añadir comandos más customizables y visuales, se implementa la librería Click.
- Simplificación del diccionario: se ha reducido el número de strings del array usado para realizar las comprobaciones de texto, ya que había demasiadas que no aportaban nada interesante para el usuario.

En el caso del punto de filtrar elementos, se ha llevado a cabo en solo un comando, lo cual muestra la información muy extendida, haciéndola difícilmente navegable y sin matices de color que permitan diferenciar mejor los resultados.

4.3.4. Arquitectura de la herramienta

RAGnarok está alojado en un repositorio de Github y se compone de una carpeta principal (Figura 4.3) que contiene las herramientas de las que depende su funcionamiento (Figura 4.4):

- Scripts: contiene el script mobsf.py que integra la API de MobSF con RAGnarok, el script principal de la herramienta, ragnarok, que maneja todas las funcionalidades y un shell script, setup.sh, que se encarga de instalar la herramienta en el entorno para poder ejecutar el comando desde cualquier ubicación.
- Dependencias: contiene la carpeta de la herramienta de la que depende la comprobación de dependencias: dependency-check.
- Archivo de requerimientos: contiene el archivo de texto requirements.txt para que se puedan instalar las librerías de las que depende la herramienta para funcionar correctamente.

En el archivo de setup.sh podemos encontrar los pasos necesarios para instalar la herramienta en el dispositivo (se introduce con más detalle en el Capítulo A).

La herramienta contiene una serie de opciones añadidas al comando principal para realizar las distintas funciones necesarias para el análisis, las cuales se implementan con la librería Click.

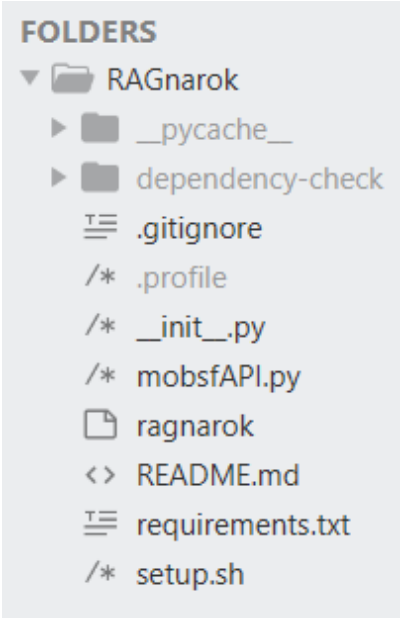


Figura 4.3: Estructura de las carpetas y archivos

4.3.4.1. MobSF API

La integración de MobSF se hace a través de un script que inicialmente define el servidor donde se levanta la herramienta (para RAGnarok se usa una dirección local: 127.0.0.1 en el puerto 8000) y la API key que genera MobSF para nuestro uso. En el Capítulo A se desarrollan más las funciones de la herramienta.

4.3.4.2. Apktool

El script requiere tener Apktool instalado (apt install apktool) y se ejecuta cogiendo el parámetro que es el apk que queremos decodificar para un posterior estudio.

4.3.4.3. Radare2

El esqueleto principal de RAGnarok es r2pipe (Figura 4.5), una integración de Radare2 con python. Anteriormente se ha explicado su funcionamiento, y en la herramienta se utiliza para hacer análisis estático del apk, en concreto de strings, clases y métodos, librerías importadas y símbolos.

Los resultados de este análisis se imprimen en la consola.

4.3.4.4. Dependency-check

Herramienta para analizar las dependencias usadas en busca de vulnerabilidades de OWASP (descripción en la siguiente sección).

Al igual que Apktool, se añade el path en el script principal y se llama a la herramienta por comando con el archivo como parámetro.

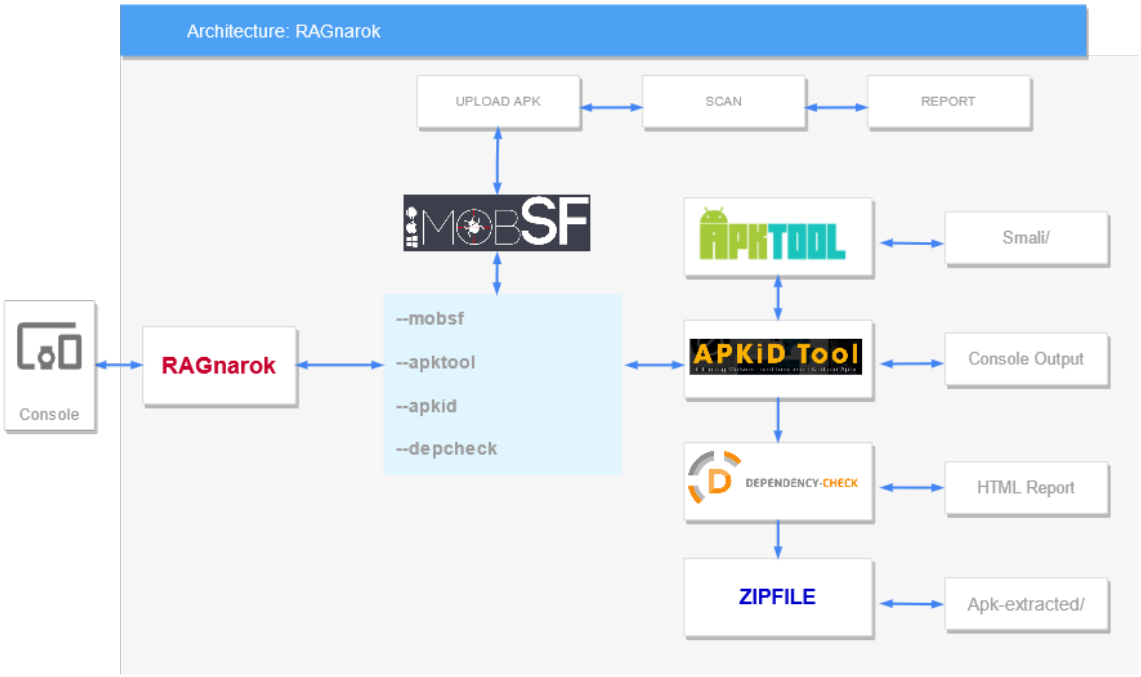


Figura 4.4: Arquitectura de la herramienta

```
# Do searches with radare2
def analyse(checks,r2p):
    result = {}
    if "strings" in checks:
        result["strings"] = r2_check_strings(checks["strings"],r2p)
    if "methods" in checks:
        result["methods"] = r2_check_classes_and_methods(checks["methods"],r2p)
    if "imports" in checks:
        result["imports"] = r2_check_imports(checks["imports"],r2p)
    if "symbols" in checks:
        print("holi")
        result["symbols"] = r2_check_symbols(checks["symbols"],r2p)
    return result
```

Figura 4.5: Función analyse de RAGnarok

La salida es un html con el reporte de los resultados en la carpeta raíz del proyecto.

4.3.4.5. ApkID

Se instala apkID con el archivo de requerimientos y es otra de las opciones modificadas en el proyecto, mejorando la ejecución del comando y la impresión de la salida de resultados.

4.3.4.6. Delegator

Se ha añadido la librería Delegator que permite manejar los subprocesos más fácilmente, siendo su aplicación a la hora de ejecutar los comandos que llaman a otras herramientas.

4.3.5. Click

Se ha cambiado **argsparse** por **click** para manejar el comando y sus opciones. Permite un código más ordenado y genera un panel de ayuda automático.

4.3.6. Rich

Para que la salida por consola fuese mucho más clara y se resaltase el texto realmente importante, se ha implementado una librería que colorea la salida según los apartados y la importancia, **rich**.

4.4. Tecnologías descartadas

A continuación se exponen las tecnologías que, tras ser investigadas, se decidió que no formarían parte del proyecto.

4.4.1. MARA Framework

MARA, Mobile Application Reversing Engineering and Analysis (Figura 4.6), es un framework que agrupa varias herramientas existentes para el análisis e ingeniería inversa de apps Android. Intenta detectar las vulnerabilidades establecidas por OWASP en aplicaciones de móvil.

4.4.1.1. Funcionalidades

- Ingeniería inversa de APK: utiliza baskmali y apktool para desensamblar Dalvik a smali bytecode, enjarify para desensamblar Dalvik a Java bytecode y jadx para decompilar el apk en código fuente de Java.
- Deobfuscación de código: utiliza apk-deguard en el caso de que el desarrollador haya obfuscado el código en un intento de complicar el reversing de la app. Dado que muchos utilizan las mismas técnicas para ello, actualmente suele ser sencillo deobfuscarla con herramientas especializadas.
- Análisis del APK: parsea archivos smali con smalisca, dumpea los assets, libs y resources del apk, extrae datos del certificado con openssl, strings y permisos con aapt, identifica clases y métodos con ClassyShark, escanea vulnerabilidades con androbugs, analiza el apk buscando acciones peligrosas con androwarn, identifica compiladores, empaquetadores y obfuscadores con APKiD y extrae URLs, emails, URIs, paths e IPs con regex.
- Análisis del Manifest: extrae información de permisos, intents, servicios, etc, así como comprueba si permite backups, si es debuggeable, si permite enviar secretos y recibir SMS binarios.
- Análisis de dominios: utiliza pyssltest y testssl y hace fingerprinting con whatweb.
- Análisis de seguridad: análisis estático del código fuente.

Figura 4.6: MARA Framework

4.4.1.2. Problemas

Este framework se consideró como punto de partida para la herramienta, dado que contiene muchas funcionalidades que cumplen con los objetivos del trabajo propuesto, pero la herramienta está escrita en bash scripting y requiere una traducción del código actual a python. Su código actual para análisis estático es insuficiente ya que comprueba paths muy específicos que no tienen por qué ser comunes en todas las apps y el resto de la solución no deja de ser un conglomerado de herramientas de seguridad de android que se implementan por separado y se llaman desde el mismo comando.

Tampoco está bien documentada y no se actualiza desde hace dos años. Tras probar su funcionamiento, se produjeron muchos fallos de ejecución que requerían de una dedicación plena que desviaría el trabajo del camino a una solución customizada.

4.4.2. Objection

Objection (Figura 4.7) es un framework basado en Frida que analiza dinámicamente las aplicaciones de móvil. Inicialmente se consideró el implementar a esta herramienta una solución de análisis estático y así englobar ambas fases de un pentesting en un solo framework.

Objection tiene una carácterística muy a su favor y es que funciona con comandos que se autocompletan con tabulador (Figura 4.8) y tiene funcionalidades tanto para Android como iOS. Además tiene una versión de desarrollo y permite customizarla con plugins.

4.4.2.1. Funcionalidades

Entre algunas de sus funcionalidades están:

- Inspeccionar e interaccionar con los sistemas de archivos de Android e iOS: listar el contenido, ver archivos concretos, etc.
- Bypassear el SSL Pinning: muchas aplicaciones llevan funciones de SSL Pinning que devuelven un booleano, lo que permite que automáticamente o manualmente un atacante pueda cambiar la función y hacer que devuelva siempre "true".
- Dumpear keychains: en iOS es donde se guardan algunos datos sensibles.
- Realizar tareas de memoria: como dumpear y parchear la app.

- Explorar y manipular objetos de la pila.
- Descubrir clases y listar sus métodos.
- Interaccionar con bases de datos de SQLite3: evita que el usuario tenga que descargar la base de datos y explorarla con herramientas externas.
- Ejecutar scripts de Frida: cuando se hookea a un método, se puede inyectar código js para cambiar el comportamiento de la app.

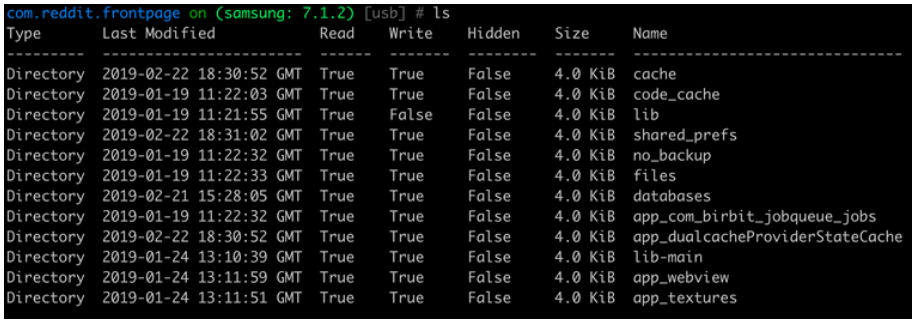


Figura 4.7: Listado de archivos en Objection

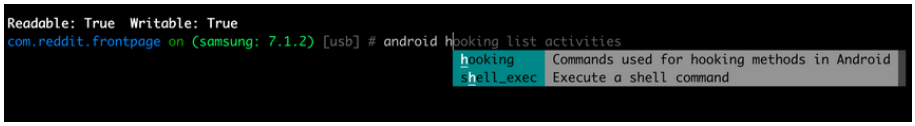


Figura 4.8: Autocomplete de comandos en Objection

4.4.2.2. Problemas

Es una herramienta muy completa y muy bien cerrada, lo que dificulta la tarea de customizar el código al haber una estructura compleja a pesar de estar bien documentada.

Por otro lado, al no aportar ninguna funcionalidad base de análisis estático, implicaba implementar de 0 una solución al respecto e integrarla en una herramienta inicialmente creada solo para análisis dinámico.

4.4.3. StaCoAn

Static Code Analyzer (Figura 4.9) es una herramienta para automatizar el análisis estático de aplicaciones mediantes un servidor y una interfaz de usuario para navegar por el resultado del análisis.

Utiliza diccionarios con palabras clave de vulnerabilidades de OWASP y strings de cosas interesantes como pueden ser passwords, usuarios, emails, etc.

4.4.3.1. Funcionalidades

Sus funcionalidades se centran principalmente en el análisis de palabras con diccionarios del código decompilado y un posterior reporte del resultado mediante una interfaz

navegable.

Figura 4.9: GUI report de StaCoAn

4.4.3.2. Problemas

Lleva mucho tiempo sin actualizarse y sin soporte, por lo cual está muy desactualizado, está poco documentado y el código es confuso. Además, el planteamiento de la herramienta buscada implica no usar GUI y que todo se pueda realizar en la terminal.

Capítulo 5

Conclusiones y Trabajo Futuro

5.1. Conclusión

El presente trabajo sirve como apoyo a la parte de análisis estático de aplicaciones de móvil, no sustituyendo a la revisión manual de un profesional de la seguridad. Su objetivo pretende ser el de proveer al tester de información sobre posibles vulnerabilidades comunes que se captan fácilmente con herramientas automatizadas y que requerirían de mucho esfuerzo humano para detectarlas manualmente.

El proyecto es ampliable y tiene mucho potencial para su uso profesional mejorando algunas funcionalidades: el filtrado de texto, añadir más tools que extraigan más datos del apk y generar un output más colorido para el usuario.

Con el script de setup.sh, la implementación de la herramienta en sí es muy simple, por lo que sirve como base para añadir más funcionalidades con cierta facilidad según necesidad del usuario.

5.2. Trabajo futuro

Según la viabilidad del proyecto se contempla su ampliación integrando una solución que ayude al análisis dinámico.

Anteriormente, se había investigado la herramienta llamada Objection y se había determinado que era una solución puramente de análisis dinámico, además de ser difícil de entender su estructura para realizar cambios. Basándonos en esa herramienta, se podría crear una solución similar mucho más sencilla que cubriese los básicos del análisis dinámico.

Uno de los pilares sobre los que está construido Objection es Frida, y en Radare2 hay un plugin especial para usar Frida, r2frida, que puede utilizarse para el propósito descrito anteriormente, permitiendo hookearse a funciones de la aplicación sin salirnos de la tecnología base sobre la que está construida este proyecto, Radare2 (Figura 5.1).

Por otro lado, también se tiene presente añadir funcionalidades específicas para iOS y que así la herramienta sea multiplataforma, permitiendo que las aplicaciones de iOS también puedan disfrutar de la ventajas que les provee esta herramienta.

```
$ r2 frida:///bin/ls
> \dc          # continue the execution
> \dd          # list file descriptors
> \dm          # show process memory maps
> \dmm         # show modules mapped
> \dl foo.so   # load a shlib
> \dt write    # trace every call to 'write'
> \isa read    # find where's the read symbol located
> \ii          # list imports off the current module
> \dxc exit 0 # call 'exit' symbol with argument 0
```

Figura 5.1: Comandos de r2frida

Por último, dado que muchas veces el contenido sensible pueden ser keys o tokens configurados con caracteres y números randoms o encodeados, se tiene que investigar la manera de poder identificar estos strings y marcarlos como sensibles a la hora de filtrar contenido.

Chapter 6

Conclusions and Future Work

6.1. Conclusions

The current project is used as support for static analysis of mobile applications, not replacing the actual manual review of a security professional. Its objective pretends to provide the tester with common vulnerabilities information captured easily with automated tools which could have required a big human effort to detect them manually.

As the development started too late, the current solution has to improve a lot of functionalities, such as text filter, adding more tools which extract more info from apk and genereting a cleaner output for the user.

The tool it's simple and its implementation is easy, so it provides a base for adding more functionalities easily.

6.2. Future Work

Having into account the viability of the project, it is considered the implementation of a solution which helps with the static analysis.

Previously, it had been investigated the tool named Objection and it had been determined that it was a solution just for dynamic analysis, besides being difficult to understand regarding the architecture when doing modifications. Based on that tool, it could create a similar solution more simple that would cover the basics of the dynamic analysis.

One of the pilars of Objection tool it's Frida and within Radare2 there is an special plugin for using Frida, r2frida, that can be used for the porpuse described previously, allowing methods hooking using the skeleton of this project development, Radare2 (6.1).

On the other side, also it's kept in mind to add specific functionalities for iOS, so as to make the tool multiplatform, allowing iOS apps to enjoy the advantages of this tool.

Finally, since several times the sensitive content could be keys or tokens configured with random or encoded characters and numbers, a way of identifying that kind of strings and mark them as sensitive when filtrating the content, should be investigated.

```
$ r2 frida:///bin/ls
> \dc          # continue the execution
> \dd          # list file descriptors
> \dm          # show process memory maps
> \dmm         # show modules mapped
> \dl foo.so   # load a shlib
> \dt write    # trace every call to 'write'
> \isa read    # find where's the read symbol located
> \ii          # list imports off the current module
> \dxc exit 0 # call 'exit' symbol with argument 0
```

Figure 6.1: Comandos de r2frida

Apéndice A

Anexo A: Diario de desarrollo

El desarrollo se ha dividido en tres fases para poder definir de manera concisa las distintas acciones que se han llevado a cabo.

A.1. Primera fase: investigación y tecnología elegida

Esta fase cubre el proceso de investigar la viabilidad del trabajo, estudiando las herramientas actualmente disponibles que cumplen una función similar, analizando las tecnologías que existen, así como la documentación para poder desarrollar la herramienta propuesta,, definir los requerimientos y funcionalidades mínimos que tenía que cumplir la solución para ser útil y competente en el mundo del pentesting móvil y obtener toda la información posible sobre el estado de la cuestión, siendo el núcleo principal la ciberseguridad en el campo de los videojuegos.

Anteriormente en esta memoria se han descrito las tecnologías elegidas y más adelante se describen también aquellas soluciones que se descartaron, por lo que partiendo desde esa información, se llegó a la conclusión de que, a pesar de que existen soluciones similares a la herramienta propuesta, pocas están actualizadas y/o bien documentadas, contando con aquellas que son open source, dando por hecho de que existen herramientas más eficientes y actualizadas en el mundo empresarial.

La única destacable por ser muy usada en el mundo de la seguridad en aplicaciones de móvil, y cuya implementación y manejo es sencillo y bien explicado, es MobSF, el cual se decide incluir en el proyecto como una segunda opción añadida de análisis estático. Permite comparar resultados con lo obtenido en la herramienta en sí y en el reporte de MobSF, y hacer así un análisis más completo.

Por otro lado, durante la investigación se encuentra una solución que implementa Radare2 con su librería r2pipe para realizar análisis estático, Apk-anal. Esta herramienta, desarrollada en python, no está pulida del todo y requiere de muchas mejoras, lo cual la hace ideal como punto de partida para construir una nueva versión más completa y eficiente que añada más funcionalidades.

A.2. Segunda fase: Implementación

Esta fase se divide en varias secciones para dar más claridad a la implementación de cada funcionalidad y detalles extras del proyecto.

- Instalación de Apk-anal: desde una terminal de Ubuntu, se clona el proyecto de Apk-anal y se instalan los requirements.txt con el comando de la imagen (Figura A.1)

```
pip install -r requirements.txt
```

Figura A.1: Pip command

Estos requierements son: **argparse**, para parsear los argumentos que se pasan por comando; **r2pipe**, pasa usar los comandos de radare2; y **filemagic**, que sirve de apoyo al comando 'file'de Unix.

- Arreglo de integración con Apktool: apk-anal ya viene con el comando apktool integrado, pero no funcionaba correctamente e implicaba que el usuario tuviese apktool instalado en una carpeta concreta y que pasara como argumento el path del .jar de la tool. Para darle solución, ahora se utiliza la librería **delegator.py** con **subprocess** para que se ejecute el comando del apktool instalado en el sistema con **pip install apktool** (como se puede ver en la figura A.2).

```
o = delegator.run(subprocess.list2cmdline([
    'apktool',
    'decode',
    '-f',
    '-r',
    '-o',
    smali_dir,
    apk
]), timeout=60 * 5)
```

Figura A.2: Ejecución Apktool en el código

- Integración de MobSF: el primer paso fue levantar un servicio de MobSF local para poder la integración de esta tool a través de su API con el proyecto. Siguiendo la guia de instalación de MobSF (Figura A.3), se trata de iniciar el servidor a partir de una terminal Ubuntu, subsistema de Windows, que inicialmente da problemas (Figura A.4). La forma de solucionarlo era cambiando la versión de WSL1 a WSL2

Figura A.3: Documentación de MobSF

(Windows subsystem for Linux), ya que la versión 1 tenía un fallo que impedía acceder

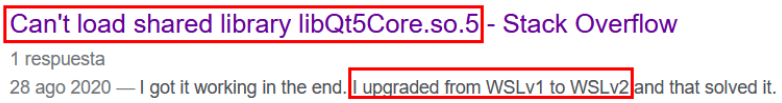


Figura A.4: No se puede cargar la librería

a la librería compartida. Una vez actualizado y cambiada la versión del subsistema, se pudo reiniciar exitosamente MobSF (Figura A.5) en un servidor local (127.0.0.1:8000 por defecto) (Figura A.6).

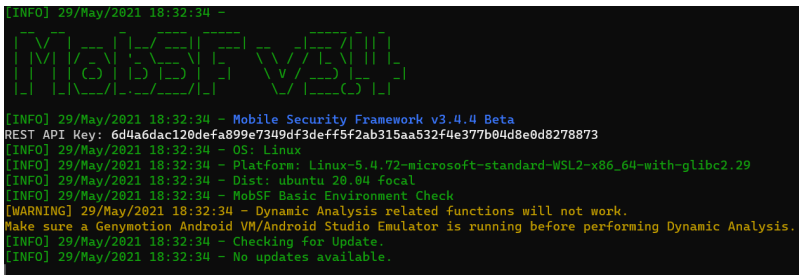


Figura A.5: Inicio de MobSF

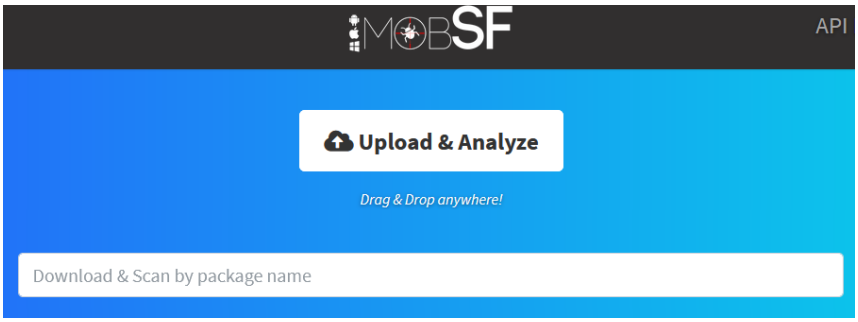


Figura A.6: Pantalla principal de MobSF

El siguiente paso fue integrar su API con el proyecto. Para ello, tras una búsqueda en Internet, había un pequeño script de python base para implementar las funciones de MobSF fácilmente en cualquier proyecto python. Dicho archivo se integra en un script aparte en el directorio raíz, llamado mobsfAPI.py (Figura A.7)

Posteriormente, se importa como módulo en el script principal del programa con todas sus funciones y se implementa un nuevo comando específico para llamar a mobsf (Figura A.8). Los métodos que contiene son: upload(), para subir los archivos pasados como argumento; scan(), que hace el escaneo del apk subido; y por último pdf(), implementado como opción, que genera el report en pdf dentro del directorio del proyecto (Figura A.9). Además se necesita una APIKey que genera el servicio cuando se inicia, que se añade al script de mobSF para que se pase en las requests.

Por último, se prueba la implementación con dos apk: InsecureBank.apk, una aplicación vulnerable de gestión de un banco; y WHAT.apk, un juego indie en Unity con pocas funcionalidades (Figura A.10).

- Llamar a la herramienta como un comando: otra de las mejoras en la implementación base de la herramienta, era la de eliminar de la llamada a la tool el comando python3

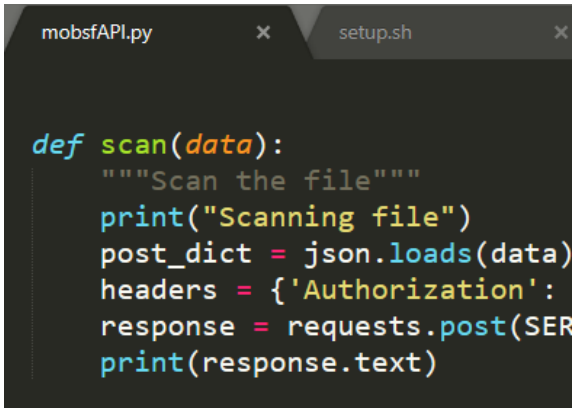


Figura A.7: Implementación de la API

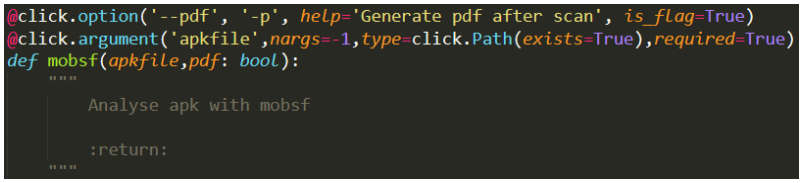


Figura A.8: Comando para MobSF

y la extensión .py. Para ello, se añade en la cabecera del script una línea que obliga a que la ejecución se realice en un entorno python: '#!/usr/bin/python3'.

Seguidamente se cambia el nombre del script a 'ragnarok', sin la extensión .py. Por último, se crea una carpeta /bin en el directorio raíz del usuario y lo añadimos al PATH para que se pueda llamar desde cualquier ubicación (Figura A.11). Esto se mantiene en cuaquier sesión porque en el archivo .profile que se encuentra en el directorio del usuario, viene por defecto que en el caso de que se cree una carpeta bin en esa ubicación, se añada al PATH.

Esto supone que se pueda ejecutar desde cualquier lado, pero los módulos no se importaban bien, dado que el programa se ejecuta desde /bin y los módulos están en el directorio raíz del proyecto. Por ello, en el siguiente paso, se soluciona este problema para cualquier usuario creando un archivo de intalación.

- Hacer un archivo de instalación: para facilitar su instalación y que todo lo hecho anteriormente funcione en otros dispositivos, se ha creado un archivo de instalación, **./setup.sh** (Figura A.12), para que se ejecuten los comandos necesarios para la instalación, siendo estos:

  1. pip3 install -r requirements.txt: línea que instala los módulos especificados en ese archivo txt.
  2. ROOT-DIR=¨$PWD'": guarda en una variable el path del proyecto para facilitar que se importen los módulos cuando el comando se llama desde cualquier lado.
  3. chmod +x racnarok.py / mv racnarok.py racnarok: da permisos de ejecución al script y cambia el nombre quitándole la extensión .py.
  4. sed -i ´/împort os/a ROOT_DIR=$ROOT_DIR racnarok: añade al principio del ejecutable el path del proyecto despúes de la línea indicada.

```
for apk in apkfile:
    data = upload(apk)
    print("[bold green]File downloaded...[/bold green]")
    scan(data)
    click.secho('The file has been scanned', fg='green', bold=True)
    if pdf:
        pdf_gen(data)
```

Figura A.9: Código para MobSF

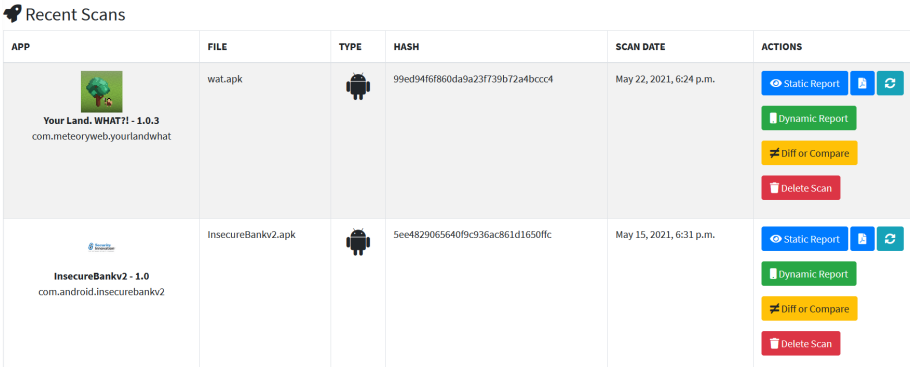


Figura A.10: Vista de los escaneos recientes

```
# set PATH so it includes user's private bin if it exists
if [ -d "$HOME/bin" ] ; then
    PATH="$HOME/bin:$PATH"
fi
```

Figura A.11: Contenido .profile

5. mkdir -p /bin: crea la carpeta /bin en el directorio del usuario logeado.
6. cp racnarok /bin: copia el ejecutable en dicha carpeta.
7. export PATH=$̈PATH:$HOME/bin¨: añade /bin al PATH para que pueda ejecutarse el programa desde cualquier lado.
8. echo éxport PATH=$PATH¨:$HOME/bin to .profile: en el caso de que por defecto en .profile no venga esta línea, se añade para que sea persistente el cambio.

La razón de añadir el path del proyecto en el script, es para que, cuando el comando se ejecuta desde la carpeta /bin, encuentre el módulo de mobSF. Se podría añadir a mano, pero implicaría que no sirviera para instalaciones en otros equipos, obligando a los usuarios a meter dicha línea a mano.

- Integración de **dependency-check** de OWASP: para detectar las dependencias de un apk que puedan ser vulnerables, OWASP pone a disposición de los desarrolladores una herramienta que, o bien se puede integrar durante el desarrollo en Android, o posteriormente para analizar apks que estén actualmente en producción. Dicha herramienta comprueba las dependencias con la base de datos de NVD (National Vulnerability Database (Figura A.13)) y genera un reporte en html con los resultados dentro de la carpeta raíz del proyecto (Figura A.14).

- Añadir diccionarios específicos de juegos: implementar la lectura de archivos de texto que contengan regex comunes relacionados con juegos de móvil para buscar de manera más concreta las vulnerabilidades en el código.

```
#!/bin/sh

pip3 install -r requirements.txt

ROOT_DIR="'$PWD'"


FILE=racnarok.py
if [ -f "$FILE" ];
then
    chmod +x racnarok.py
    mv racnarok.py racnarok
fi

sed -i '/^import os/a ROOT_DIR='$ROOT_DIR racnarok


mkdir -p ~/bin

cp racnarok ~/bin

export PATH="$PATH:$HOME/bin"

echo 'export PATH=$PATH":$HOME/bin"' >> .profile
```

Figura A.12: setup.sh

```
[INFO] Analysis Started
[INFO] Finished Archive Analyzer (0 seconds)
[INFO] Finished File Name Analyzer (0 seconds)
[INFO] Finished Dependency Merging Analyzer (0 seconds)
[INFO] Finished Version Filter Analyzer (0 seconds)
[INFO] Finished Hint Analyzer (0 seconds)
[INFO] Created CPE Index (1 seconds)
[INFO] Finished CPE Analyzer (1 seconds)
[INFO] Finished False Positive Analyzer (0 seconds)
[INFO] Finished NVD CVE Analyzer (0 seconds)
[WARN] Unable to determine Package-URL identifiers for 1 dependencies
[INFO] Finished Sonatype OSS Index Analyzer (0 seconds)
[INFO] Finished Vulnerability Suppression Analyzer (0 seconds)
[INFO] Finished Dependency Bundling Analyzer (0 seconds)
[INFO] Analysis Complete (1 seconds)
[INFO] Writing report to: /[redacted]/dependency-check/bin/./de
```

Figura A.13: Output Dependency-check

- Añadir un output más claro y usar el módulo de python, click: implementar click para tratar los comandos y generar un output más claro y eficiente. El proyecto original implementaba argparse, una librería también válida, pero click incluye funcionalidades útiles como una pantalla autogenerada de ayuda con los comandos implementados y un validador de ruta. En cuanto a la salida, se incluye **rich**, una librería que pone colores a los strings de salida para que su inspección sea más sencilla.

- Banner: se añade un banner que se muestra cada vez que se ejecuta un comando, que incluye el nombre del programa y de la creadora (Figura A.15).

A.3. Tercera fase: pruebas y resultados

En esta fase se prueban varios apks vulnerables intencionadamente y algunos juegos de la plataforma apkpure. Dado que el análisis estático de una aplicación no afecta a la integridad de la app y no supone un impacto al resto de posibles usuarios, no se ha

Figura A.14: Reporte de Dependency-check

Figura A.15: Banner de RAGnarok

considerado necesario solicitar permiso para estudiar el código de dichos apks.

Además, el trabajo se realiza con fines académicos, y todo lo extraído tendrá como objetivo su análisis y posterior eliminación y/o comunicación con el equipo de desarrollo, si se considera una vulnerabilidad crítica dentro de la sensibilidad del juego.

Apéndice B

Anexo B: Manual de usuario

A continuación se describe el uso de las funcionalidades de RAGnarok, escogiendo aplicaciones vulnerables que muestran todo el potencial de la herramienta. Dichas aplicaciones son InsecureBankv2 y Pivaa.

B.1. Aplicaciones usadas para el caso de ejemplo

Las aplicaciones elegidas son intencionalmente vulnerables para poder sacar resultados que muestren el potencial de la herramienta.

Por un lado tenemos **InsecureBankv2**[1], que entre otrascosas, se usará para probar el análisis de Radare2.

Por otro tenemos **Pivaa**[2], que se usará para probar el resto de funcionalidades.

B.2. Descargar e instalar la herramienta

RAGnarok puede encontrarse en un repositorio de Github[3] y o bien se puede clonar o bajar el proyecto comprimido directamente (Figura B.1).

También se deben de tener instaladas las herramientas Apktool y ApkID mediante *apt install <name>* y descargarse la utilidad de OWASP, **dependency-check** (Figura B.2).

Una vez se tiene el repositorio en local y las dependencias instaladas, entramos en la carpeta raíz y ejecutamos el script de *./setup.sh*, lo que instalará las dependecias necesarias mediante el fichero de texto *requirements.txt* y hará el comando invocable desde cualquier hubicación del sistema (Figura B.3).

[1] urlhttps://github.com/dineshshetty/Android-InsecureBankv2
[2] urlhttps://github.com/HTBridge/pivaa
[3] `https://github.com/Mariag39/RAGnarok`

B.3. Usar Apktool

Lo primero es ejecutar el comando **ragnarok apktool pivaa.apk** para que se decodifique el contenido del fichero apk (Figura B.4).

Y los contenidos de cada una se ven a continuación en las figuras B.5, B.6 y B.8:

B.4. Usar unzip

Podemos usar la opción unzip si queremos extraer el contenido del apk como si fuera un archivo comprimido normal, aunque estaría desensamblado (Figura B.7 y B.8).

B.5. Análisis de archivo dex con Radare2

Lo siguiente es usar la opción *dex* para que la herramienta use su implementación de Radare2 con *r2pipe* y analice el binario al completo (Figura B.9). Para este análisis se ha usado el archivo *classes.dex* de la aplicación InsecureBankv2.

En la salida de consola, podemos revisar lo que ha detectado el análisis de posibles strings interesantes o vulnerabilidades, como datos sensibles.

En la Figura B.10 podemos comprobar que uno de los strings revela una clave secreta de la aplicación:

Por otro lado también ha encontrado referencias interesantes a ficheros de superuser (Figura B.11):

B.6. Levantar servidor MobSF y subir el apk a la plataforma

Primero necesitamos un servidor MobSF (Figura B.12 y Figura B.13). En el Anexo B (Capítulo B), se explica como levantar tu propio servidor en local para hacer pruebas y como obtener la api key para validar las peticiones que hace la herramienta para subir el apk a la plataforma. Toda la información se puede encontrar en la documentación de MobSF [4]

Una vez el servidor esté en funcionanmiento, desde la herramienta seleccionamos la opción **mobsf** y añadimos como parámetro el fichero apk. Lo siguiente que hará será analizar el archivo durante unos minutos y generar un json y un pdf con el resultado (Figura B.14). También podemos consultar en la plataforma, en la pestaña *Recent Scans*, todos los ficheros que hemos analizado (Figura B.15).

Para esta prueba se ha usado el fichero apk de pivaa. MobSF saca un reporte que analiza las funciones y da información detallada de las vulnerabilidades (Figura B.16).

[4]urlhttps://mobsf.github.io/docs/#/running

B.7. Uso de Dependency-check para análisis de dependencias

Para analizar si las dependencias de un proyecto contienen vulnerabilidades reconocidas, la herramienta cuenta con la opción de **depcheck**, teniendo como parámetro el directorio de la aplicación.

Tras la ejecución del comando y pasados unos minutos, se genera un reporte en html con los resultados del análisis de las dependencias encontradas (Figura B.17).

Para esta prueba hemos usado el directorio de la aplicación Pivaa.

B.8. Uso de ApkID para obtener más información del fichero apk

Para comprobar los compiladores, los paquetes, la obfuscación, etc y saber si el fichero puede contener malware o no ser legítimo, usamos el comando **apkid**, teniendo como parámetro el fichero apk, el archivo dex o un directorio.

La salida que obtenemos se puede observar en la Figura B.18:

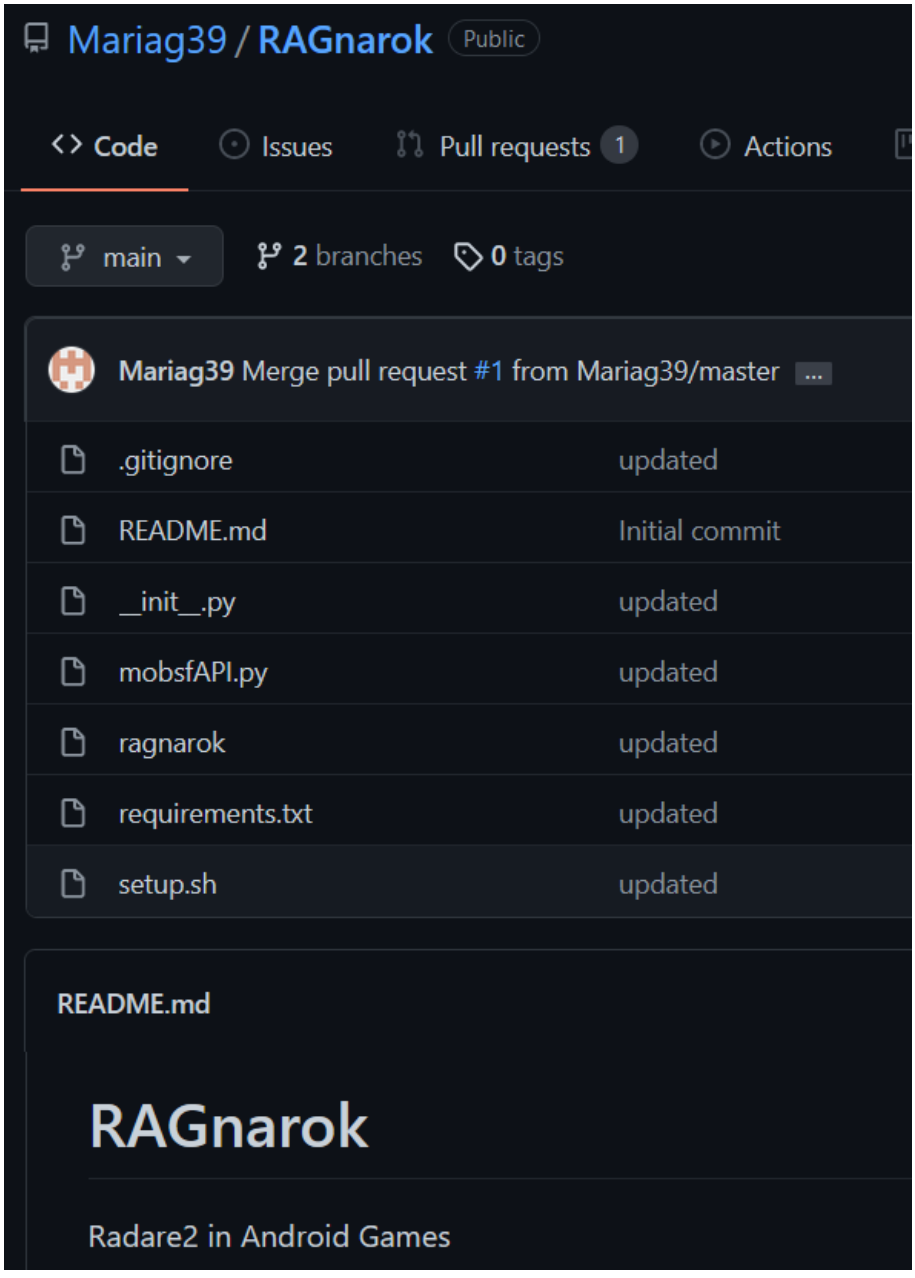


Figura B.1: Repositorio RAGnarok

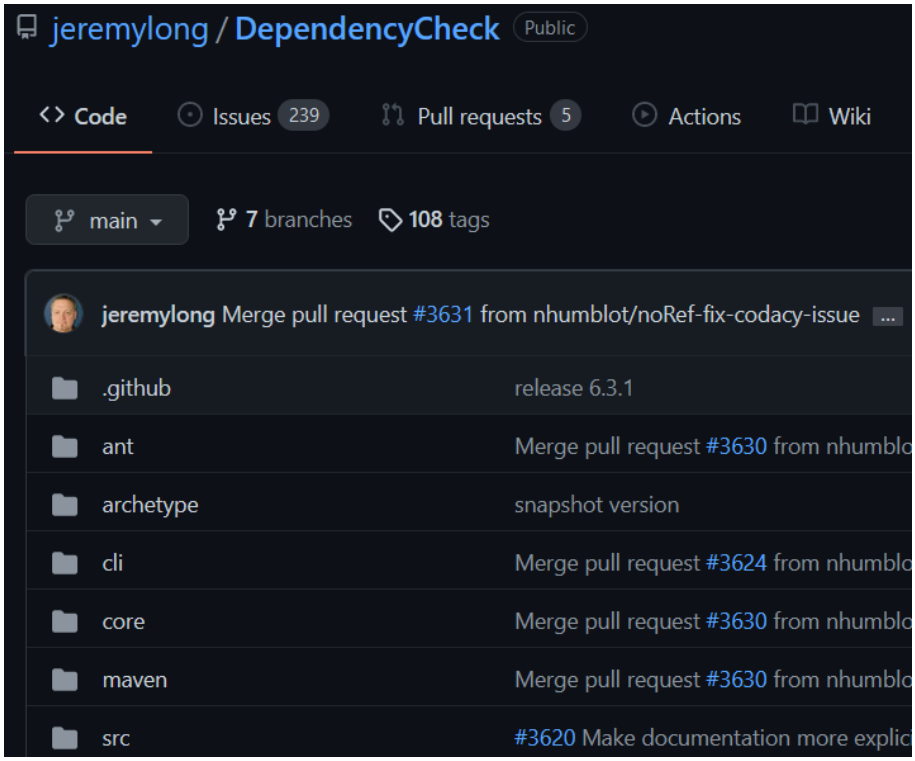


Figura B.2: Repositorio de dependency-check

```
raldu@DESKTOP-Q3LAS1U:/mnt/c/Users/María/RAGnarok/RAGnarok$ ./setup.sh
Defaulting to user installation because normal site-packages is not writeable
Collecting argparse
  Using cached argparse-1.4.0-py2.py3-none-any.whl (23 kB)
Requirement already satisfied: filemagic in /home/raldu/.local/lib/python3.8/s
ne 2)) (1.6)
Requirement already satisfied: r2pipe in /home/raldu/.local/lib/python3.8/site
3)) (1.5.3)
Requirement already satisfied: apkid in /usr/local/lib/python3.8/dist-packages
2)
Requirement already satisfied: click in /home/raldu/.local/lib/python3.8/site-
)) (8.0.1)
Requirement already satisfied: yara-python-dex>=1.0.0 in /home/raldu/.local/li
requirements.txt (line 4)) (1.0.1)
Installing collected packages: argparse
Successfully installed argparse-1.4.0
```

Figura B.3: Salida de la ejecución de setup.sh

```
raldu@DESKTOP-Q3LAS1U:/mnt/c/Users/María$ ragnarok apktool pivaa.apk
   ****************************************

    _____ _____ _____                 _
   |  _  |  _  |  _  |___ ___ ___ ___| |_
   |     -|     |  |  |  _| .'|  _|  _| '_|
   |__|__|__|__|__|__|_| |__,|_| |___|_,_|


   ****************************************
   Radare2 in Android Games
   by: Maria Raldu

[*] Disassembling apkfile with apktool:
The files have been extracted in //mnt/c/Users/María/apk-extracted/smali
```

Figura B.4: Extracción y decodificación apk

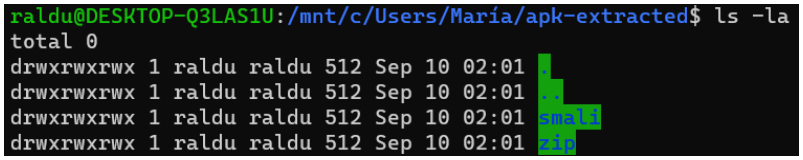


Figura B.5: Contenido directorio *apk-extracted*

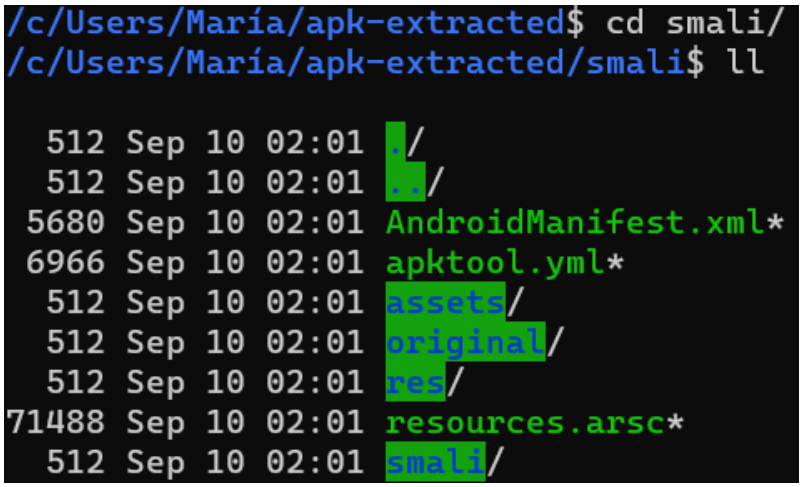


Figura B.6: Contenido *smali*

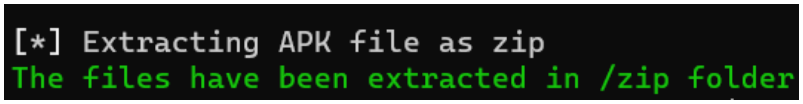


Figura B.7: Comando *zip*

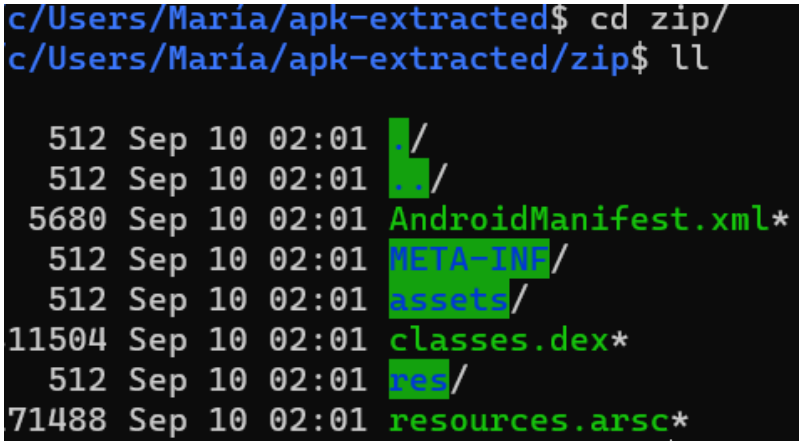


Figura B.8: Contenido *zip*

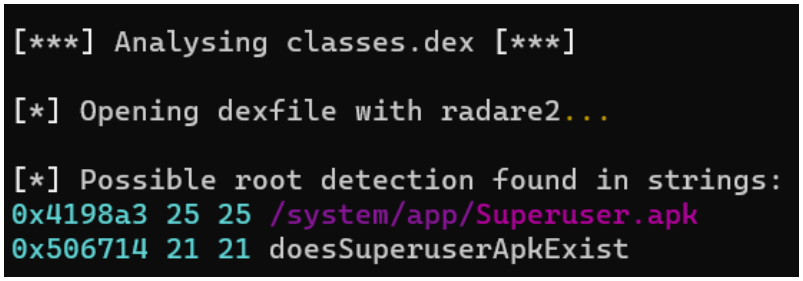


Figura B.9: Ejecución del análisis al archivo dex

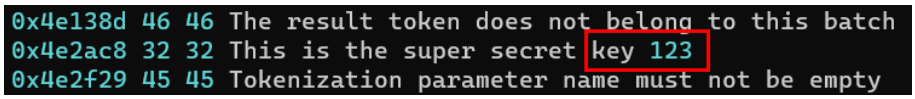


Figura B.10: Clave secreta en strings

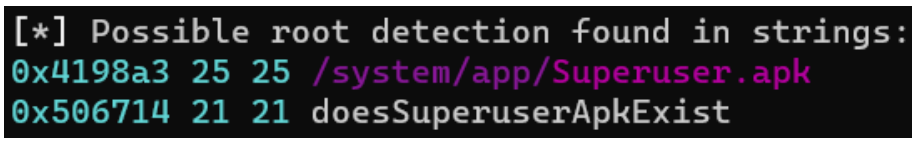


Figura B.11: Ruta a fichero apk de superuser

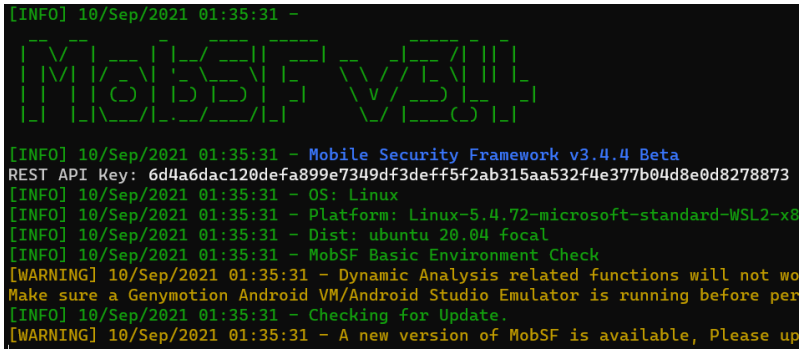


Figura B.12: Ejecución del servidor de MobSF local

Figura B.13: Plataforma MobSF Web

Figura B.14: Ejecución del comando *mobsf*

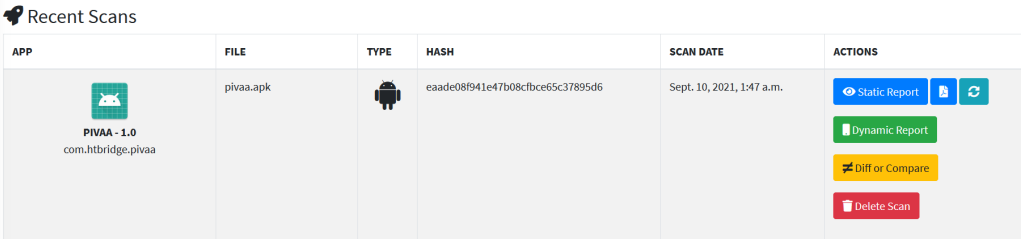


Figura B.15: Pestaña de escaneos recientes

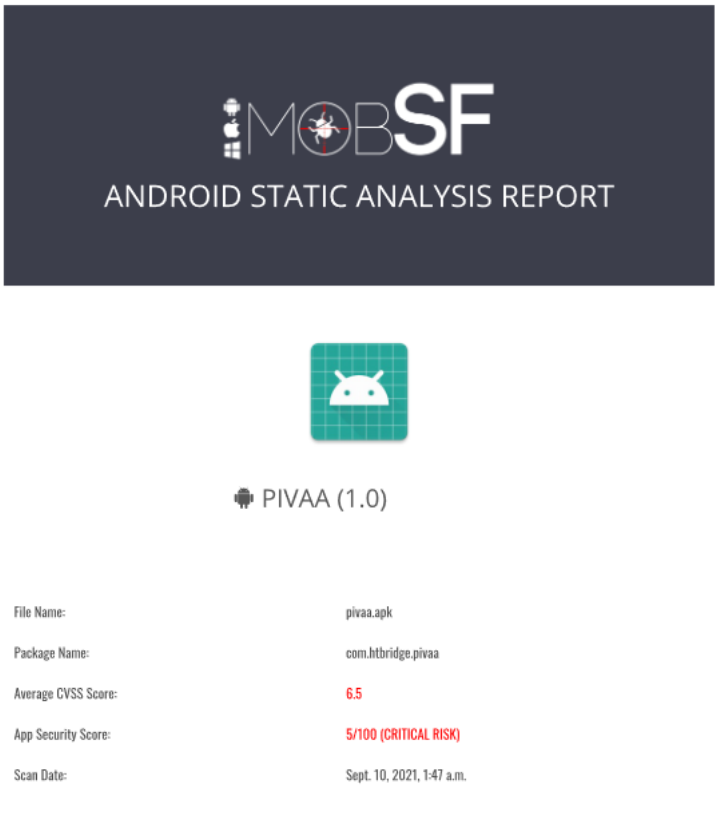
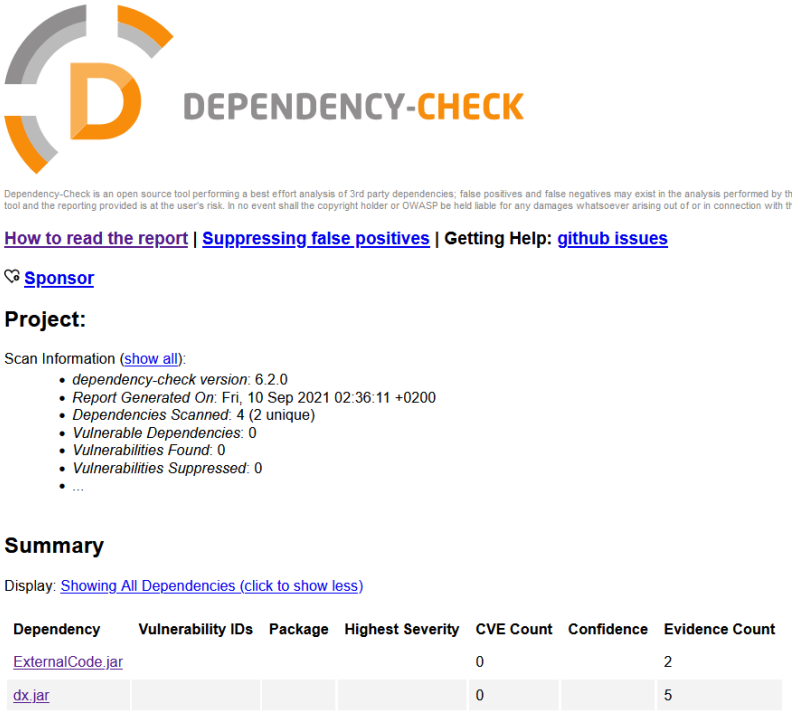


Figura B.16: Reporte en pdf del análisis



DEPENDENCY-CHECK

Dependency-Check is an open source tool performing a best effort analysis of 3rd party dependencies; false positives and false negatives may exist in the analysis performed by th
tool and the reporting provided is at the user's risk. In no event shall the copyright holder or OWASP be held liable for any damages whatsoever arising out of or in connection with th

How to read the report | Suppressing false positives | Getting Help: github issues

Sponsor

Project:

Scan Information (show all):

- *dependency-check version*: 6.2.0
- *Report Generated On*: Fri, 10 Sep 2021 02:36:11 +0200
- *Dependencies Scanned*: 4 (2 unique)
- *Vulnerable Dependencies*: 0
- *Vulnerabilities Found*: 0
- *Vulnerabilities Suppressed*: 0
- ...

Summary

Display: Showing All Dependencies (click to show less)

Dependency	Vulnerability IDs	Package	Highest Severity	CVE Count	Confidence	Evidence Count
ExternalCode.jar				0		2
dx.jar				0		5

Figura B.17: Reporte en html de **Dependency-check**

```
[*] Running APKiD
[+] APKiD 2.1.2 :: from RedNaga :: rednaga.io
[*] InsecureBankv2/app/app-debug.apk!classes.dex
 |-> anti_vm : Build.MANUFACTURER check, Build.MODEL check, Build.PRODUCT check
 |-> compiler : dx (possible dexmerge)
 |-> manipulator : dexmerge
[*] InsecureBankv2/app/app-release.apk!classes.dex
 |-> anti_vm : Build.MANUFACTURER check, Build.MODEL check, Build.PRODUCT check
 |-> compiler : dx (possible dexmerge)
 |-> manipulator : dexmerge
```

Figura B.18: Ejecución del comando de *apkid*

Bibliografía

No documentes el problema; arréglalo.
Atli Björgvin Oddsson

Akamai (2020). State of the internet security in gaming. https://www.akamai.com/newsroom/press-release/state-of-the-internet-security-gaming-you-cant-solo-security.

Cihodariu, M. (2019). Cybersecurity for gamers 101: Gaming malware and online risks. https://heimdalsecurity.com/blog/cybersecurity-for-gamers-101-malware-risks/.

Deyan, G. (2021). Videogames statistics. https://techjury.net/blog/video-games-industry-statistics/.

Editor (2015). Ciberataques a consolas. https://www.welivesecurity.com/la-es/2015/07/02/ataques-juegos-online-mas-grandes/.

Haddix, J. (2016). Owasp top ten mobile. https://owasp.org/www-project-mobile-top-10/.

Miessler, D. (2017). Owasp game security framework. https://es.slideshare.net/danielmiessler/the-owasp-game-security-framework.

Nesterenko, O. (2020). IDC Data videogames grown. https://gameworldobserver.com/2020/12/28/idc-gaming-revenue-2020.

Nuñez, E. A. (2018). Pentesting. https://openwebinars.net/blog/que-es-el-pentesting/.

Park, M. (2012). Mobile application security, the who, how and why. https://owasp.org/www-pdf-archive/ASDC12-Mobile_Application_Security_Who_how_and_why.pdf.

Technologies, P. (2019). Mobile application security threats and vulnerabilities. https://www.ptsecurity.com/ww-en/analytics/mobile-application-security-threats-and-vulnerabilities-2019/.

Townsend, K. (2019). Gamers and gaming security. https://blog.avast.com/cybersecurity-risks-all-gamers-should-know.

Sólo necesitamos tener suerte una vez.

Vosotros necesitáis tener suerte siempre.

Hacker