ISSN 0798 1015

REVISTA ESPACIOS

HOME Revista ESPACIOS ÍNDICES A LOS AUTORES

Vol. 39 (Nº 11) Año 2018. Pág. 7

Detección de vulnerabilidades en aplicaciones que funcionan sobre el sistema operativo Android, mediante el desarrollo de una aplicación tecnológica

Detection of vulnerabilities in applications that work on the Android operating system, through the development of a technological application

Henry Mauricio VILLA Yánez 1; Andrés Santiago CISNEROS Barahona 2; Pablo Martí MÉNDEZ Naranjo 3; María Isabel UVIDIA Fassler 4; Ciro Diego RADICELLI García 5

Recibido: 04/11/2017 • Aprobado: 25/11/2017

Contenido

RESUMEN:
El artículo muestra los resultados obtenidos al ejecutar en un escenario de pruebas, dos prototipos a fin de encontrar vulnerabilidades en aplicaciones que funcionan sobre el sistema operativo Android, utilizando para esto el método de inducción científica y una investigación de tipo experimental, mediante lo cual se elaboró una guía de mejores prácticas para programación segura en Android, y a su vez se desarrolló una aplicación informática (APSEBI), con dos componentes importantes; el generador y el framework de ejecución de los casos de prueba JUnit, mediante los cuales se pudieron detectar vulnerabilidades basadas en un sistema de mensajería propio de Android (intent), obteniendo como resultado luego de la comparación de los dos prototipos que las vulnerabilidades se reducen si se utilizan las recomendaciones de la guía definida en este estudio al programar.

ABSTRACT:
The article shows the results obtained when executing in a test scenario, two prototypes in order to find vulnerabilities in applications that work on the Android operating system, using for this the method of scientific induction and a type experimental research , through which it was elaborated a guide of best practices for secure programming in Android, and in turn a computer application (APSEBI) was developed, with two important components; the generator and the execution framework of the JUnit test cases, through which vulnerabilities based on an Android messaging system (intent) could be detected, obtaining as a result after the comparison of the two prototypes that the vulnerabilities are reduced if the recommendations of the guide defined in this study are used when programming.
Keywords: Vulnerabilities Android, Operating System

Palabras-Clave: Vulnerabilidades Android, Sistema Operativo Android, Seguridad basada en *intents*, Programación segura

Android, Security based on intents, Secure Programming

1. Introducción

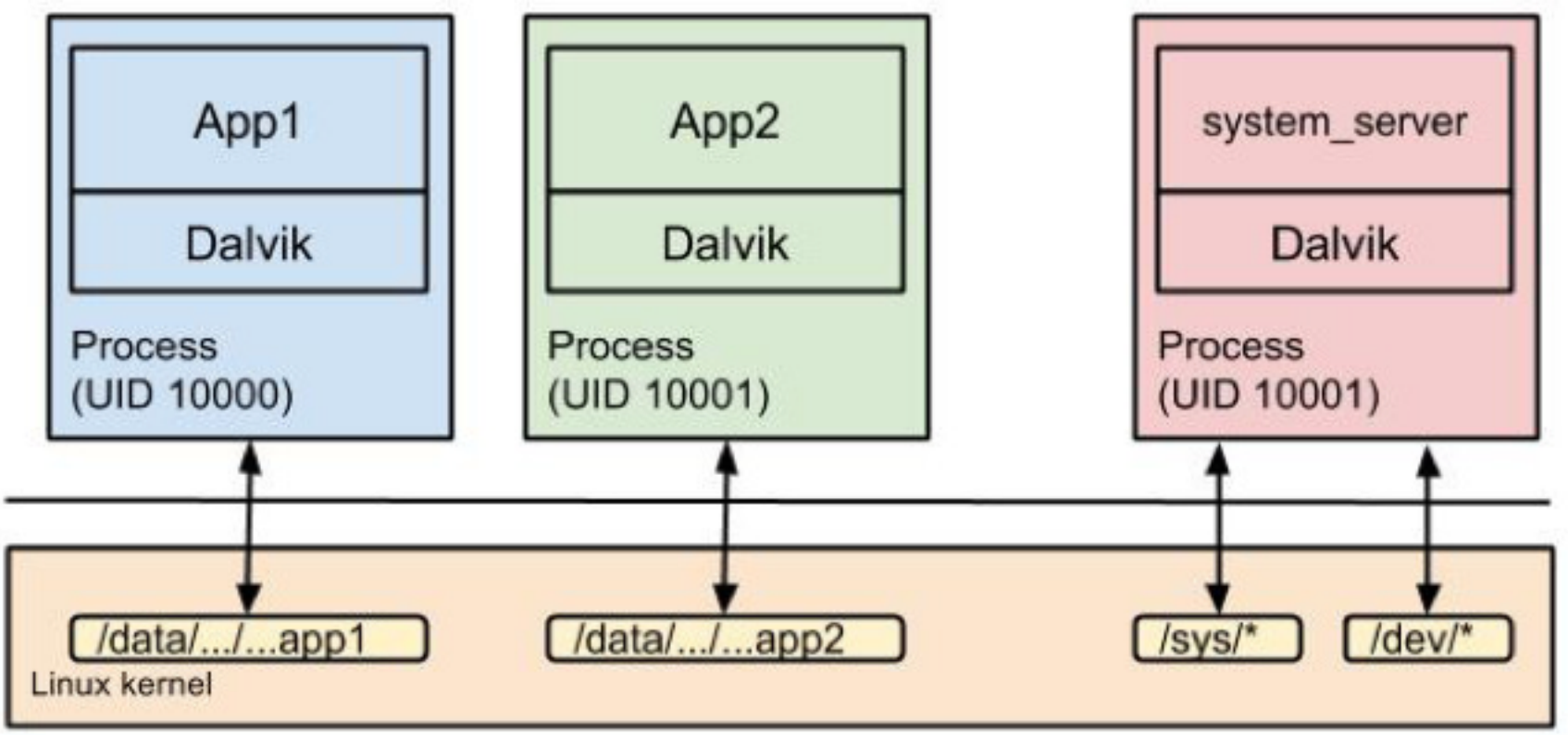
Android es un sistema operativo (S.O.) basado en código Linux y de acceso abierto (*open source*). Sin embargo aunque este aspecto sean considerado como una ventaja, cualquier persona con habilidades de programación podría programar una aplicación y embeber código malicioso como virus, programas espía (*spyware*), gusanos informáticos (*worms*), entre otros dentro del S.O., generando vulnerabilidades de seguridad y por lo tanto un sistema comprometido (Microsoft, 2015); conllevando que al ejecutar estas aplicaciones se tengan fallas en el sistema, tales como el no acceso o la pérdida de datos sensibles de los usuarios (Dhaya, & Poongodi, 2014).

Además, las facilidades que presta este S.O. podrían incrementar los riesgos de vulnerabilidades (puntos débiles del software) que perjudican la seguridad, integridad, disponibilidad y confidencialidad de la información que se maneja en los dispositivos que utilizan Android. Por ejemplo, este S.O. permite personalizar los teléfonos inteligentes, mediante la instalación de un sin número de aplicaciones que están disponibles en la tienda en línea (Play Store); o en su defecto mediante la instalación por parte del usuario de un archivo de la web conocido como APK (*Application Package File*). Dicha personalización produciría el riesgo de ser víctima de ataques (Chin, Felt, Greenwood, & Wagner, 2011).

Es por esto que en Android para tratar de asegurar la información, se han considerado algunos mecanismos, como: (i) permitir el aislamiento de las aplicaciones al momento de su ejecución, o, ii) establecer los permisos en la ejecución de cada aplicación en Android (Salva & Zafimiharisoa, 2015).

Sin embargo las medidas antes descritas no son suficientes, debido a que como se observa en la figura 1, las aplicaciones que se ejecutan incluso de manera aislada logran comunicarse utilizando un sistema de mensajería de Android denominado *intent*; que permite que una aplicación tenga acceso a recursos y aplicaciones del dispositivo como contactos, GPS, WiFi y demás (Idrees & Rajarajan, 2014). Y debido a que dicho sistema de mensajería se puede monitorear, sería fácil realizar un ataque conocido como de hombre en el medio (*man in the middle*), de tal forma de obtener la información que viaja entre las aplicaciones, para alterar su contenido y poder insertar datos corruptos en la comunicación.

Figura 1
Comunicación entre aplicaciones Android



Fuente: Elaboración propia

Por lo anteriormente mencionado, y en vista de que en el S.O. Android las vulnerabilidades documentadas son relativamente pocas, y que la mayoría de ataques realizados al sistema operativo en mención están embebidas en las aplicaciones instaladas en el dispositivo móvil, considerando además que es de suprema importancia el aseguramiento de las aplicaciones utilizadas por los usuarios en los dispositivos que trabajan sobre Android. El objetivo de la investigación de la cual se derivó este artículo fue la detección de vulnerabilidades en aplicaciones que funcionan sobre el S.O. Android para el aseguramiento de las mismas, desarrollando para ello una aplicación informática llamada APSEBI que permitió escanear dichas vulnerabilidades.

---

2. Metodología

La presente investigación es de tipo experimental ya que se fundamenta en pruebas realizadas en escenarios de laboratorio mediante el uso de prototipos. El prototipo 1 no consideró las recomendaciones de la guía de mejores prácticas para la programación segura en Android; mientras que el prototipo 2 si las consideró. Se utilizó el método de inducción científica, debido a que se analizaron las vulnerabilidades existentes en Android y su relación con la comunicación entre actividades y servicios utilizando intents. Por su parte los instrumentos utilizados fueron los entornos de desarrollo integrado (IDE); Netbeans y Eclipse, para el desarrollo de aplicaciones Android para dispositivos móviles. Eclipse se utilizó además para la creación de las aplicaciones que sirvieron para probar las vulnerabilidades. Se utilizó además el SDK de Java, que no es más que un conjunto de herramientas software que permite al desarrollador crear aplicaciones concretas, es decir, es una interfaz de programación de aplicaciones (API).

2.1. Determinación de los diferentes tipos de vulnerabilidades existentes en Android

Luego de realizar la búsqueda de información acerca de los diferentes tipos de vulnerabilidades que existen en el S.O. Android, se relacionaron las mismas en torno a los cuatro componentes claves de la seguridad de la información.

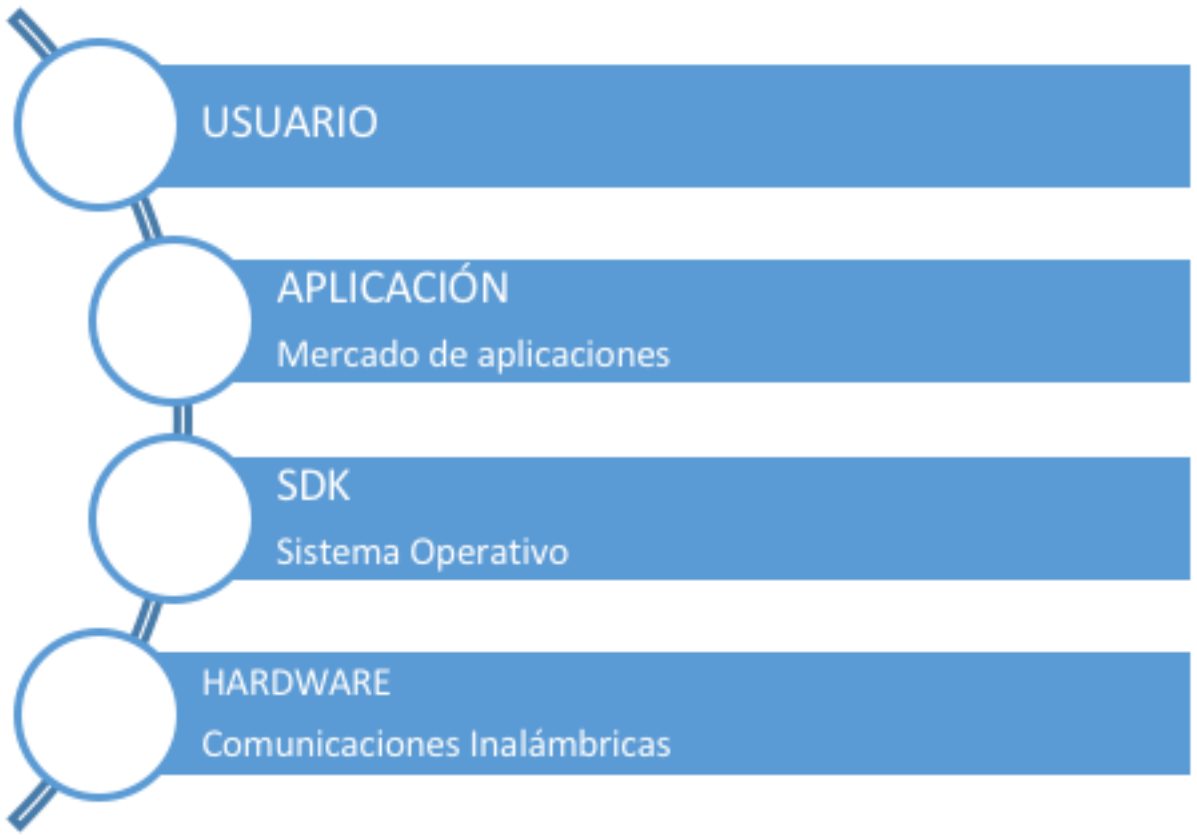
- **Confidencialidad**: se refiere a que debe ser posible que nadie puede interceptar las comunicaciones y apropiarse de información delicada, logrando de esta manera que no se pueda hacer mal uso de la misma. Esta característica denominada también privacidad, asegura que sólo

quien tiene autorización podrá acceder a la información.

- **Autenticación:** Hace referencia a la identificación mediante el uso de claves, pasos de testigo (*tokens*), autenticación biométrica, entre otras, para permitir que únicamente el dueño del dispositivo móvil haga uso de las funcionalidades del mismo.
- **Integridad:** trata de que la información de voz y/o datos viaje en la red sin que sea modificada. Es la propiedad que se asegura la no alteración de la información, entendiéndose por alteración al cambio, borrado o sustitución de la misma.
- **No repudio:** Es la característica que se asegura que ninguna parte pueda negar una acción realizada anteriormente. Es decir no debe existir la posibilidad de que el usuario, luego de haber utilizado el dispositivo móvil, niegue luego el uso del mismo. Con esto se garantiza que existan las pruebas necesarias para identificar las acciones realizadas por un determinado usuario.

De las características mencionadas, la más destacable sería la autenticación, debido a que no sirve de mucho que la información obtenida sea confidencial e integra, si resulta que el receptor de la información no es el destinatario correcto (Bavota, Escobar Velásquez, & Linares Vásquez, 2017). Con estas consideraciones, se realizó un análisis de la seguridad de los dispositivos móviles de la manera más eficiente posible, organizando dicho análisis en cuatro apartados (capas): (i) la comunicación inalámbrica, (ii) el sistema operativo, (iii) la aplicación y el usuario, tal como se muestra en la figura 2.

Figura 2
Capas de seguridad en dispositivos móviles



Fuente: Elaboración propia

En cada una de las capas de seguridad se analizaron los principales ataques que se pueden realizar, como por ejemplo enmascaramiento (*masquerading*), ataques de denegación de servicios (*DoS*), escuchas secretas (*eavesdropping*), entre otras. Posteriormente, se procedió con el análisis de las vulnerabilidades encontradas en los intents. Por lo que se examinó la comunicación utilizando intents entre actividades y servicios considerando la integridad de dicha comunicación, mediante el desarrollo de una aplicación que escanee vulnerabilidades basadas en intents en las aplicaciones Android, posteriormente se desarrolló una guía de mejores prácticas para la programación segura en Android, con lo que se pudo demostrar la hipótesis propuesta en la investigación.

2.2. Determinación del formalismo para la escritura de patrones

de vulnerabilidad

Han existido propuestas para representar políticas de seguridad, propiedades o vulnerabilidades, por ejemplo en algunos casos se han usado expresiones regulares, mientras que en otros el uso de lógicas no clásicas (lógicas temporales y deónticas), core typed languages o state machines, fue bastante adecuado. Sin embargo en esta investigación se utilizó el Modelo ioSTS (*input output Symbolic Transition Systems*) para construir los patrones que determinarán si una aplicación es vulnerable o no, sin el riesgo de cometer ambigüedad en el análisis. Villa (2016), afirma que "*el modelo ioSTS permite modelar sistemas críticos o imperativos sin el uso de la recursividad y la comunicación con su entorno, permitiéndoles ser más autónomos"*. En este sentido algunos tipos de formalismos han sido definidos en cuanto a políticas de seguridad se refiere, así en ioSTS, un formalismo está compuesto de variables, que en sí vienen a ser las acciones de entrada y salida que llevan parámetros de comunicación realizadas por acciones, guards y tareas (Bernd Kleinjohann, 2007).

El modelo mencionado es ampliamente usado para la verificación y pruebas en diferentes campos, como el modelamiento de grandes y complejos sistemas, tales como las composiciones de servicios web, los sistemas críticos, las aplicaciones web y de escritorio, entre otros. Por esto el modelo de ioSTS se adaptará muy bien en el caso de que la documentación de Android sea enriquecida o si se descubren nuevas vulnerabilidades.

Los patrones de vulnerabilidad que se determinaron en este estudio, indudablemente ayudaron a describir si los comportamientos de los componentes eran vulnerables o no, en base a la utilización de autómatas simbólicos, mismos que están compuestos por variables y guards sobre variables, que permiten escribir las restricciones sobre las acciones que se puede realizar. Estos patrones se utilizaron para mitigar las vulnerabilidades que se pueden explotar en base a los intents mediante los cuales se lleva información delicada desde una aplicación (actividad) a otra, misma que puede ser monitoreada mediante diferentes técnicas (Bernd Kleinjohann, 2007).

La creación de los patrones estuvo basada en referencia a las aplicaciones y algoritmos existentes en la documentación oficial de Android. Razón por la cual, dicho modelo posee gran flexibilidad y acoplamiento con respecto a los avances que se tendrían en Android, en cuanto a la definición de patrones de vulnerabilidad.

2.3. Desarrollo de la aplicación para pruebas de seguridad basadas en intents (APSEBI)

La aplicación APSEBI (*Aplicación para Pruebas de SEguridad Basadas en Intents*), que se desarrolló en esta investigación, analiza diferentes componentes del proyecto Android, enfocándose en las Actividades (*Activities*) y Servicios (*Services*) del proyecto; además de la relación que tienen estos componentes con el proveedor de contenidos (*ContentProviders*), considerando que éste no puede ser llamado directamente por los intents, pero si puede a través de otros componentes exponer información sensible como datos personales, contraseñas, entre otros.

Para el desarrollo de APSEBI se utilizaron casos de prueba, mismos que fueron generados luego del análisis realizado entre los archivos del mismo proyecto (componentes compilados, manifest) y de patrones que describen modelos de vulnerabilidades específicas en Android (Mouelhi, Fleurey, Baudry, & Le Traon, 2008).

2.4. Arquitectura

De forma general APSEBI está compuesto por dos aplicaciones: (i) el generador de especificaciones (SpecGen); y (ii) la herramienta para testear las aplicaciones Android (Testing Tool). En cuanto a su arquitectura, APSEBI consta de los siguientes componentes:

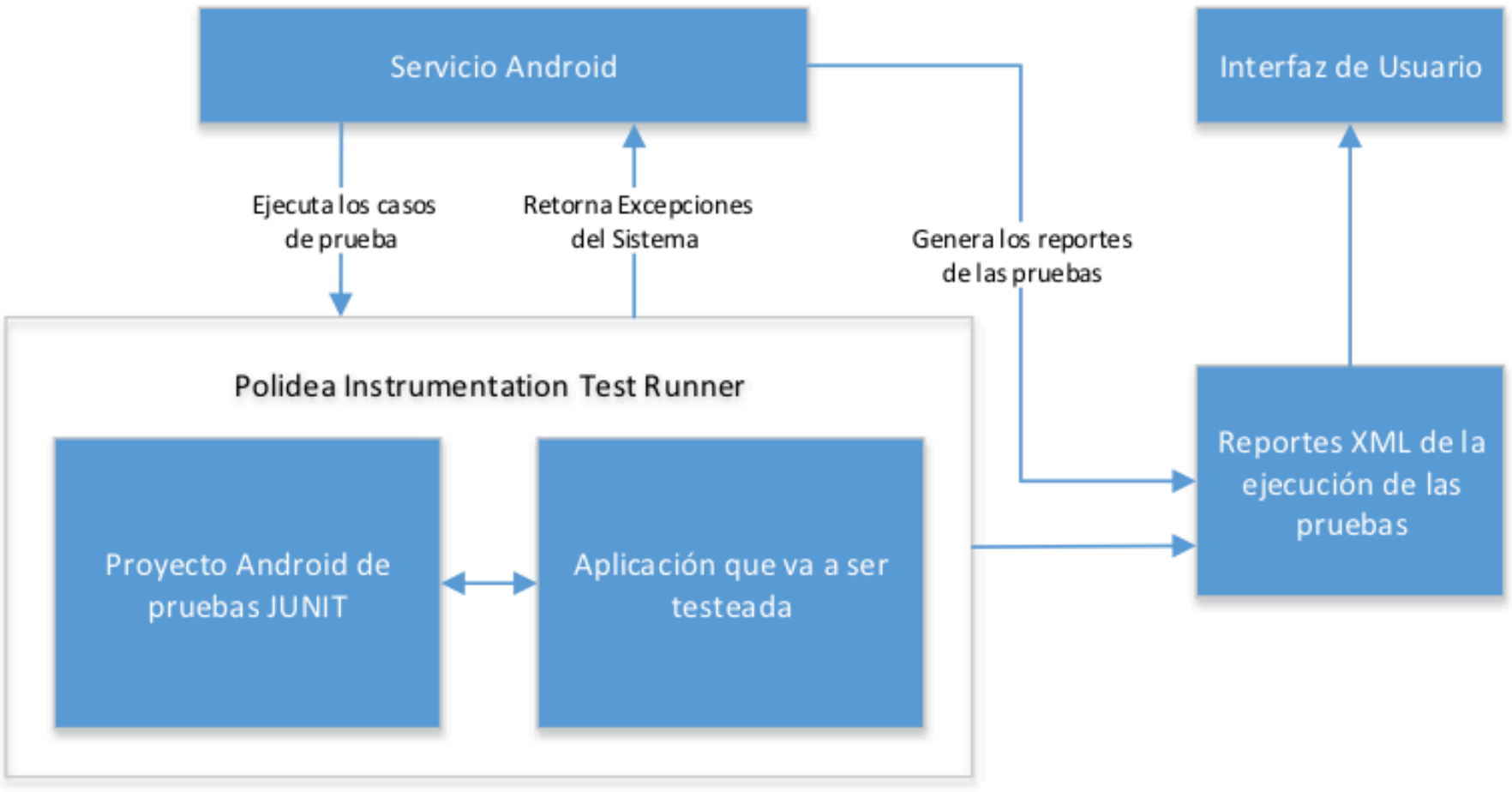
- **Generador de casos de prueba para analizar las aplicaciones**: creado con la arquitectura MCV (*Model View Controller*), su estructura se muestra en la figura 3.

A continuación se describen cada uno de los componentes, tanto en la generación de especificaciones como en la generación de los casos de prueba

- **Patrones de Vulnerabilidad:** los patrones de seguridad de APSEBI fueron creados por expertos de desarrollo en Android, y se basan en las vulnerabilidades que ya han sido descubiertas. En este sentido se tiene el proyecto OWASP (*Open Web Application Security Project*), que se dedica a la creación y mantenimiento de aplicaciones confiables y seguras, contribuyendo de esta manera con la creación de informes de funcionamiento sobre los huecos de seguridad y la mitigación de los mismos en entornos de acceso abierto, como es el caso de Android (TechTarget, 2017).
- **Especificaciones Parciales ioSTS:** se las genera usando la funcionalidad conocida como reflexión de Java, y ayuda a depurar el resultado final de las pruebas realizadas a los componentes, comparándolas con la documentación oficial de Android, con el objetivo de prevenir falsos positivos.
- **Diagrama de clases:**Es la lista los componentes que utiliza la aplicación, se describen además el tipo de componente, y las relaciones que mantienen entre ellos. Las especificaciones parciales ioSTS y el Diagrama parcial de clases son creadas en base a los archivos generados por el mismo proyecto (manifest, componentes compilados).
- **Propiedades de las vulnerabilidades:** *"A la combinación de los patrones de vulnerabilidad, las especificaciones parciales ioSTS y el diagrama de clases se les denomina propiedades de las vulnerabilidades"* (Villa, 2016). Dichas propiedades pueden ser refinadas mediante de los intents (explícitos e implícitos) que recibe la aplicación Android.
- **Casos de prueba ioSTS:** Al definir las propiedades de las vulnerabilidades se obtienen los casos de prueba, mismos que serán completados con valores al ejecutar la aplicación. Finalmente, los resultados de los casos de prueba ioSTS conjuntamente con las transiciones, variables y guards son convertidos a casos de prueba JUNIT que son los que realmente se ejecutan en la aplicación

El framework detalla el funcionamiento de la ejecución de los casos de prueba JUNIT, dicha ejecución se la puede realizar directamente en el dispositivo Android o en un dispositivo virtual de Android (AVD - *Android Virtual Device*), como se muestra en la figura 4.

Figura 4
Framework de ejecución de los casos de prueba de JUNIT



Fuente: Elaboración propia

Dicho Framework está compuesto por la herramienta de ejecución de pruebas Android, que complementada con la instrumentación Polidea Instrumentation permiten crear reportes formato en XML para tener un control y visualización de las vulnerabilidades. La ejecución de los casos de prueba lo realiza un servicio android, que despliega los resultados en el dispositivo o en el AVD. Una vez que todos los casos de prueba han sido ejecutados, el servicio se encarga de mostrar los resultados en pantalla y de generar un reporte detallado de la ejecución de los casos de prueba en un archivo XML. Obteniendo como resultados de la ejecución los siguientes tipos: (i) VUL, cuando encuentra que algún componente es vulnerable a los tipos de ataques descritos en los patrones de vulnerabilidad: (ii) NVUL, si ninguna vulnerabilidad ha sido detectada en la aplicación a analizarse, y (iii) NCONCLUSIVE, cuando un escenario de vulnerabilidad no puede ser testeado.

2.5. Escritura de los Patrones de Vulnerabilidad

Para la escritura de los patrones de vulnerabilidad se han definido algunas notaciones, a fin de determinar el comportamiento de los componentes de la aplicación Android utilizando ioSTS, como se muestra en la tabla 1.

Tabla 1
Notaciones ioSTS

Notación	**Significado**
AuthActtype	Conjunto de acciones de un determinado tipo de componente
ACTr	Conjunto de las acciones de los intents que requieren una respuesta
ACTnr	Conjunto de acciones de los intents que no es requerida una respuesta
?intent(Cp, a, d, c ,t, ed)	Un intent compuesto por: ***Cp:*** llamada del componente, ***a:*** una acción, ***d:*** datos, ***c:*** categoría de la acción, ***t:*** tipo de dato y, ***ed:*** datos extras
C	Conjunto de categorías Android
T	Conjunto de tipos Android
U R I	Conjunto de URI encontradas en una aplicación y completadas con otras URIs a lazar
RV	Conjunto de valores predefinidos y aleatorios
I N J	Conjunto de inyecciones SQL y XML

!Display	Lo que muestra y despliega en la pantalla del dispositivo o emulador un intent
!Running	El servicio que está en ejecución
!ComponentExp	Excepción creada por un componente
!SystemExp	Excepción creada por el sistema
?call(Cp, requ est, tableU R I)	El componente ContentProvider llama con una petición a la tabla URI
?callResp(Cp,resp)	El componente ContentProvider responde con ***resp*** el contenido de la respuesta solicitada

Fuente: elaboración propia

A continuación, se procede a ampliar las notaciones descritas, con el objetivo de entender de mejor las mismas. Así, considerando que los componentes se comunican a través de intents, se construyó la ecuación 1:

$$intent\ (Cp, a, d, c, t, ed) \quad (1)$$

En donde: : ss el componente que se invoca; : ss la acción que va a ser ejecutada; : ss un dato expresado como una URI; : ss la categoría de un componente de la cual se extrae información adicional acerca de la acción que se va a ejecutar; : es una extensión específica de los datos que se envían en el intent (MIME type); : representa los datos extras que se envían con el intent.

Conociendo además que las acciones de los intents tienen diferentes propósitos, como por ejemplo: la acción VIEW que permite que una actividad muestre algo en la pantalla, y la acción PICK que permite escoger un ítem de entre las opciones, pudiendo retornar de esta manera una URI del componente que se está llamando.

Bajo este precedente, se ha divido el conjunto de acciones (ACT) en dos categorías: ACTr que es el conjunto de acciones que requieren una respuesta y ACTnr, cuando no se requiere respuesta. Además se ha denotado con C, al conjunto de categorías predefinidas en Android y con T el conjunto de tipos. Así, por ejemplo *?intent(Act, PICK, content://com.android/contacts, DEFAULT,,); r*epresenta la llamada de una actividad ACT la cual le permite al usuario escoger un contacto inicialmente guardado en una lista de contactos.

Es importante mencionar además que, al momento de ejecutar las pruebas en las aplicaciones, los componentes de Android pueden crear excepciones que se agrupan en dos categorías: las excepciones que son creadas por el mismo S.O. Android, las cuales son representadas con !SystemExp; y las excepciones que son creadas debido a la colisión entre componentes, que son representadas con !ComponentExp.

Todas las definiciones detalladas anteriormente, tienen su correspondiente función codificada en APSEBI, por ejemplo, la acción !Display(A) está representada por la función Display(), misma que retorna verdadero si una interfaz ha sido mostrada en el dispositivo o emulador (AVD).

Esta relación entre las acciones y el código Java hace que sea mucho más sencillo la construcción de los patrones de vulnerabilidad y por ende de los casos de prueba dentro de APSEBI. Además, dichas acciones pueden ser actualizadas en cualquier momento en base a las nuevas vulnerabilidades encontradas. Así en vez de definir una vulnerabilidad de cada especificación parcial que se genera, es decir, escribir una vulnerabilidad para cada una de las

especificaciones parciales creadas, lo que se realiza es definir patrones de vulnerabilidades que engloben en su mayoría las vulnerabilidades de los componentes Android basadas en intents. En términos generales, un patrón de vulnerabilidad describe un conjunto amplio de vulnerabilidades que permitirán evaluar los diferentes componentes de la aplicación. En el caso de que se requiera, se puede crear patrones específicos para un determinado componente o aplicación.

Por lo anteriormente mencionado, es importante entonces definir primero la notación a ser utilizada en la creación de los patrones de vulnerabilidad. Así un patrón de vulnerabilidad se denota con V, y el mismo está compuesto por las acciones que ejecutan los componentes, a este conjunto de acciones Δ_v, conocidas como acciones del patrón de vulnerabilidad, se las define como $AuthAct_{type}$.

De esta manera, los patrones de vulnerabilidades se han escrito específicamente para las actividades y servicios de la aplicación. Estos patrones tratan de vulnerabilidades relacionadas a la integridad de los datos que manejan las aplicaciones. El primer patrón es sencillo y está relacionado con la integridad en actividades, denotando la vulnerabilidad relacionada con la integridad. El objetivo es chequear si una actividad llamada con intents compuestos con datos maliciosos no puede alterar el contenido de una tabla de base de datos manejada por el ContentProvider. Por su parte el segundo patrón de vulnerabilidad está relacionado con la integridad de los servicios, y tiene como objetivo chequear si la estructura de la base de datos manejada por el ContentProvider no es alterada (modificación de los atributos de nombres, eliminación de tablas, etc.), a través de una instrucción SQL injection enviada por una actividad que reciba intents. La principal diferencia en el patrón de vulnerabilidad 2, radica en la forma de llamar al ContentProvider, puesto que lo hace con las tres primeras transiciones, con el objeto de extraer la estructura de la base de datos que está almacenada en la variable de origen. En este caso, luego del envío del intent compuesto por SQL injections, el componente no es vulnerable si la estructura de la base de datos es igual a la original (para esto se usa un predicado definido como equalstructure). Lo anteriormente descrito, permitió la realización de la guía de mejores prácticas para la programación segura en Android. Como se muestra en la figura 5.

2.6. Desarrollo de la Guía de mejores prácticas para la realización de una programación segura en Android

Figura 5
Estructura y caracterización de la Guía de mejores prácticas

Fuente: elaboración propia

En la Guía de mejores prácticas, se establecen los lineamientos que deberían seguir los desarrolladores de software para crear aplicaciones Android seguras. A continuación se detallan los mismos.

No se debe almacenar información confidencial en un dispositivo de almacenamiento externo (tarjeta SD) sin que antes sea cifrado: Para solucionar esta vulnerabilidad se debe crear el archivo con permisos de MODE_PRIVATE para que otras aplicaciones no puedan acceder al archivo. O en el caso de que se tenga que almacenar información en la tarjeta SD, es importante encriptar la información que se almacene. Además, al momento de encriptar la información se debe usar métodos que no puedan ser descifrados muy fácilmente.

Limitar la accesibilidad al proveedor de contenidos para no compartir datos confidenciales entre aplicaciones: Para solucionar este inconveniente de seguridad se debe añadir la entrada antes ya detallada android:exported="false" en el archivo AndroidManifest.xml lo que hace al proveedor de contenido privado, para que otras aplicaciones no puedan acceder a los datos.

No permitir que el WebView tenga acceso a recurso local confidencial a través del esquema del archivo: Cualquier URI recibida a través de un intent que no sea confiable (fuera de la aplicación) debe ser validada antes de ser mostrada en el componente WebView. Por ejemplo, el código (file:scheme) comprueba una URI recibida y la abre sólo cuando no es un esquema de archivo, de esta manera se soluciona uno de los problemas que tiene el componente WebView.

No realizar una transmisión tipo difusión (*broadcast*) con información sensible mediante un intent implícito: Si un tipo de intent es transmitido o recibido por la misma aplicación se debe utilizar función LocalBroadcastManager, debido a que de esta manera se asegura que otras aplicaciones no puedan recibir el mensaje de transmisión, lo que reduce el riesgo de fuga de información delicada. En conclusión, en vez de usar Context.sendBroadcast(), se debería usar LocalBroadcastManager.sendBroadcast().

No registrar información sensible o delicada: Si la aplicación incluye una biblioteca de terceros, el desarrollador debe asegurarse de que la biblioteca no envíe (registre) información confidencial al registro de salida (log). Una solución común al desarrollar una aplicación es la de declarar y utilizar una clase de registro personalizado, de modo que este registro sea

automáticamente activado/desactivado basado en la forma de ejecución debug/release. Se recomienda usar ProGuard para borrar llamadas a métodos específicos.

No conceder permisos URI en intents implícitos: Los datos almacenados en el proveedor de servicio de una aplicación, pueden ser referenciados por identificadores URI que se incluyen en los intents. Si el receptor del intent no tiene los privilegios necesarios para acceder a la URI, el remitente del intent puede establecer cualquiera de las siguientes banderas: FLAG_GRANT_READ_URI_PERMISSION, FLAG_GRANT_WRITE_URI_PERMISSION en el intent. Por otro lado, si el remitente ha especificado en el archivo de manifiesto que los permisos de URI pueden concederse, entonces el receptor de la intención será capaz de leer o escribir (respectivamente) datos en la URI. Si un componente malicioso es capaz de interceptar el intent, entonces se puede acceder a los datos en la URI. Los intents implícitos pueden ser interceptados por cualquier componente de modo que, si se desea que los datos sean privados se debería utilizar intents explícitos antes que intents implícitos. Por tanto, una aplicación puede interceptar la intención implícita y tener acceso a los datos en el URI.

No actuar sobre maliciosos intents: Si un receptor de difusión (*Broadcast Receiver*) exportado confía ciegamente en un broadcast intent, este puede realizar acciones inapropiadas como por operar con datos sensibles de la aplicación o en su defecto corromper toda la aplicación, debido a que los receptores a menudo envían datos y comandos a los servicios y actividades. Es por esto que los componentes registrados para recibir broadcast intents con acciones del sistema (DEVICE_STORAGE, ATTERY_LOW, ACTION_POWER_CONNECTED, ACTION_SHUTDOWN, entre otras), son particularmente vulnerables a este tipo de ataques. Razón por la cual se debe limitar la exposición del componente mediante el establecimiento de requisitos de permiso en el manifiesto o dinámicamente controlando la identidad del autor de la llamada.

Proteger los servicios exportados con fuertes permisos: se debe establecer permisos fuertes en el archivo de manifiesto de la aplicación al momento de exportar un servicio. Estos permisos deben estar relacionados directamente a la actividad que vaya a realizar el servicio.

Limitar el acceso a actividades delicadas: En Android, al declarar un intent filter para una actividad en el AndroidManifest.xml, se está diciendo que la actividad puede ser exportada para ser usada por otras aplicaciones. Si la actividad está destinada únicamente para el uso interno de la aplicación que se desarrolla y se declara un intent filter, entonces cualquier otra aplicación, incluyendo malware, puede activar la actividad para un uso no deseado. Por lo tanto, se debería declarar android:exported="false" para la actividad que no se desea exportar.

No liberar aplicaciones que son depurables (*debuggable*): Android permite que el atributo android.debuggable sea establecido con el valor de true en el archivo de manifiesto, de esta manera la aplicación puede ser depurada. Por defecto, este atributo es deshabilitado y asignado el valor de false, pero podría ser establecido verdadero para ayudar con el proceso de "debugging" durante el desarrollo de la aplicación. Sin embargo, una aplicación nunca debería ser descargada con este atributo puesto en verdadero ya que permite que los usuarios obtengan el acceso a detalles de la aplicación que deberían ser mantenidos seguros. Con el atributo puesto en true, los usuarios pueden realizar debug de la aplicación, pero sin el acceso al código de fuente. Por lo que se recomienda, verificar que el atributo android.debuggable tenga el valor de false antes de que la aplicación sea liberada.

Asegurarse de que los datos sensibles sean mantenidos seguros: La seguridad de los datos (para canales de comunicación que no sean intents) puede ser respaldada mediante la creación de un archivo, asignando a la base de datos el permiso MODE_PRIVATE en el almacenamiento interno, pero además se puede encriptar el canal usando técnicas de encriptación, como el uso de una llave de encriptación que sólo pueden conocer la(s) aplicación(es) permitidas. Si este archivo o base de datos se va a guardar en una tarjeta SD. El MODE_PRIVATE, debería ser una constante definida a través de la clase android.content.Context, que debería ser utilizado como parámetro en los métodos openFileOutput(), getSharedPreferences(), y openOrCreateDataBase() (los cuales son definidos

en la clase android.content.Context). En esta solución el archivo es creado usando MODE_PRIVATE, por eso no puede ser accedido por aplicaciones con el mismo ID del usuario como la aplicación que creó el archivo.

No proveer el método de acceso addJavascriptInterface en un WebView que podría tener contenido no confiable: Al permitir a una aplicación usar el método addJavascriptInterface con un contenido no confiable dentro de un WebView deja la aplicación vulnerable a ataques scripting usando procesos de reflexión y de esa forma se permitiría acceder a métodos públicos mediante el uso de JavaScript. El método addJavascriptInterface (Object, String) es llamado desde la clase android.webkit.WebView, así de esta manera los datos delicados y control de aplicación estarían expuestos a ataques de scripting. La solución a esta vulnerabilidad es abstenerse de llamar al método addJavascriptInterface().

Considerar la posibilidad de problemas de privacidad al usar API de geolocalización: La Geolocation API, especificada por W3C, habilita a los navegadores web la capacidad de acceder a la información referente a la ubicación geográfica del dispositivo de un usuario. Para habilitar la geolocalización en una aplicación usando la clase WebView, se deberán conceder los permisos necesarios para el efecto, así como el uso del paquete webkit. Esta solución muestra una interfaz de usuario, que solicita el consentimiento del usuario para el envío de su posición a un servidor web. Dependiendo de la respuesta del usuario, la aplicación puede controlar la transmisión de los datos de geolocalización.

Verificar correctamente el certificado del servidor en SSL/TLS: Las aplicaciones Android que usan los protocolos SSL/TLS para la comunicación segura deberían verificar correctamente los certificados del servidor. Un programador tiene la libertad de personalizar su implementación SSL. El programador debería usar correctamente SSL con el intent de la aplicación. Si el SSL no es usado correctamente, los datos delicados del usuario podrían filtrarse a través del canal de comunicación (SSL vulnerable). La solución compatible podría variar dependiendo de la implementación, pero de forma general se podría usar un certificado de servidor self_signed.

2.7. Ambiente de pruebas

Se estableció un ambiente de pruebas común, en el que se compartieron los dos escenarios establecidos para el análisis de vulnerabilidades basadas en Intents de aplicaciones Android. Las condiciones del ambiente de pruebas para los 2 escenarios fueron:

- Aplicación desarrollada en Android
- Dispositivo móvil no rooteado
- Disposición del código fuente de la aplicación a ser escaneada (no APK)
- Generación del archivo build.xml de la aplicación a ser escaneada.

En cuanto a los escenarios, se tuvieron el escenario 1, en donde se utilizó el Prototipo I, que utilizará la aplicación desarrollada para escanear las vulnerabilidades basadas en intents de un ejemplo en Android, no considerando las recomendaciones de la guía de mejores prácticas; y el escenario 2, en donde en el Prototipo II, la aplicación de escaneo si consideró dichas recomendaciones.

---

3. Resultados

Para el desarrollo de las pruebas se seleccionaron tres aplicaciones de ejemplo, de la documentación oficial de Android, mismas que están publicadas en el Play Store, y que se enuncian a continuación: (i) NotePad; (ii) SearchableDictionary; y (iii) WifiDirectDemo.

Dichas aplicaciones para el desarrollo de las pruebas han sido seleccionadas de acuerdo a los siguientes criterios:

- Número de clases que posee la aplicación.

- Tipo de aplicación.
- Si la aplicación está basada en la documentación oficial de Android.

En la tabla 2 se muestra una comparativa de las aplicaciones antes mencionadas.

Tabla 2
Comparación de las aplicaciones

Criterios	**NotePad**	**Searchable Dictionary**	**WifiDirectDemo**
No. Clases	6	4	5
Tipo de Aplicación	productividad	productividad	productividad
Basada en documentación oficial Android	Si	Si	Si

Fuente: elaboración propia

Establecidos los escenarios, se procedió a realizar las pruebas respectivas con cada una de las aplicaciones.

Aplicación Notepad.-

Tabla 3
Información general de la aplicación NotePad

Nombre de la aplicación	NotePad
Versión	8
Máximo de pruebas por componente	10
Clases	• NoteEditor.java • NotePad.java • NotePadProvider,java • NotesList.java • NotesLiveFolder.java • TitleEditor.java

Fuente: elaboración propia

En la tabla 4 se muestra los resultados del escaneo de los prototipos I y II de NotePad.

Tabla 4
Resultados del escaneo de los prototipos I y II de NotePad

Prototipo I			**Prototipo II**		
VUL	**NVUL**	**INC**	**VUL**	**NVUL**	**INC**
3	41	0	1	43	0

Fuente: elaboración propia

Se observó que al escanear la aplicación con el Prototipo II, que incluyó el uso de la

recomendación número 2 de la "Guía de mejores prácticas para la realización de una programación segura en Android", se disminuyeron las vulnerabilidades basadas en intents en contraste de los resultados obtenidos al escanear la aplicación Notepad con el Prototipo I. Además, se constató que existe todavía una vulnerabilidad no solucionada, debido a que en la guía no se consideró la recomendación para la solución a dicho problema.

Aplicación Searchable Dictionary

En la tabla 5 se muestra la información General de la aplicación Searchable Dictionary.

Tabla 5
Información general de la aplicación Searchable Dictionary

Nombre de la aplicación	Searchable Dictionary
Versión	8
Máximo de pruebas por componente	10
Clases	• DictionaryDatabase.java • DictionaryProvider.java • SearchableDictionary.java • WordActivity.java

Fuente: elaboración propia

En la tabla 6 se muestra los resultados del escaneo de los prototipos I y II de Searchable Dictionary.

Tabla 6
Resultados del escaneo de los prototipos I y II de Searchable Dictionary

Prototipo I			**Prototipo II**		
VUL	**NVUL**	**INC**	**VUL**	**NVUL**	**INC**
4	18	0	1	21	0

Fuente: elaboración propia

Al igual que con la aplicación anterior, al aplicar el Prototipo II, y utilizando la recomendación número 2, se obtuvo como resultado la disminución de vulnerabilidades basadas en intents en contraste de los resultados obtenidos al escanear la aplicación con el Prototipo I de la aplicación Searchable Dictionary. Así mismo tal cual el caso anterior existe una vulnerabilidad no corregida.

Aplicación WifiDirectDemo

En la tabla 7 se muestra la Información General de la aplicación WifiDirectDemo.

Tabla 7
Información general de la aplicación WifiDirectDemo

Nombre de la aplicación	WifiDirectDemo
Versión	8

Máximo de pruebas por componente	10
Clases	• DeviceDetailFragment.java • DeviceListFragment.java • FileTransferService.java • WifiDirectActivity.java • WifiDirectBroadcastReceiver.java

Fuente: elaboración propia

En la tabla 8 se muestra los resultados del escaneo de los prototipos I y II para la aplicación WifiDirectDemo.

Tabla 8
Resultados del escaneo de los prototipos I y II de WifiDirectDemo

Prototipo I			**Prototipo II**		
VUL	**NVUL**	**INC**	**VUL**	**NVUL**	**INC**
2	19	0	0	21	0

Fuente: elaboración propia

De acuerdo a los resultados obtenidos al escanear la aplicación con el Prototipo II, y considerando la recomendación número 2 de la guía, se obtuvo como resultado la disminución de las vulnerabilidades basadas en intents en contraste de los resultados obtenidos al escanear la aplicación con el Prototipo I de la aplicación WifiDirectDemo.

3.1. Comparación de resultados

Se compararon los resultados del escaneo de vulnerabilidades basadas en intents realizado con los prototipos I y II con las tres aplicaciones, como se muestra en la tabla 9.

Tabla 9
Resultados del escaneo de los prototipos I y II de WifiDirectDemo

APLICACIÓN	**Prototipo I**			**Prototipo II**		
	VUL	**NVUL**	**INC**	**VUL**	**NVUL**	**INC**
NotePad	3	41	0	1	43	0
Searchable Dictionary	4	18	0	1	21	0
WifiDirectDemo	2	19	0	0	21	0

Fuente: elaboración propia

Luego del análisis realizado entre los prototipos I y II aplicados a las 3 aplicaciones, se puede verificar que el prototipo II tiene menos vulnerabilidades, debido a que se incorpora las recomendaciones que se brindan en la guía, lo que implica un mejoramiento en la seguridad de las aplicaciones escaneadas.

4. Conclusiones

Luego del análisis realizado a los diferentes tipos de vulnerabilidades encontrados en Android, se seleccionaron a las de tipo intent, puesto que las mismas permiten una comunicación entre varios componentes de la aplicación Android, lo que las convierte en un componente sensible para recibir ataques.

La aplicación desarrollada escaneó las vulnerabilidades basadas en intents mediante el uso de patrones que facilitaron la especificación de las mismas, para tal fin se utilizaron dos componentes importantes que son el generador y el framework de ejecución de los casos de prueba JUnit.

Mediante la realización de la investigación, se logró desarrollar una guía de mejores prácticas para la realización de una programación segura en Android.

El prototipo II es más seguro debido a que en este prototipo se utilizaron las recomendaciones de la guía de mejores prácticas para la realización de una programación segura en Android.

Referencias bibliográficas

Bavota, G., Escobar Velásquez, C., & Linares Vásquez, M. (2017). An Empirical Study on Android-related Vulnerabilities. En IEEE (Ed.), *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference.* Buenos Aires, Argentina: IEEE. doi:10.1109/MSR.2017.60

Bernd Kleinjohann, L. K. (2007). *From Model-Driven Design to Resource Management for Distributed Embedded Systems.* Braga, Portugal: Springer.

Chin, E., Felt, A., Greenwood, K., & Wagner, D. (2011). Analyzing inter-application communication in android. 9th International Conference on Mobile Systems, Applications, and Services. Berkeley, CA, USA.

Dhaya, R., & Poongodi, M. (2014). Detecting software vulnerabilities in android using static analysis. *Advanced Communication Control and Computing Technologies (ICACCCT), 2014 International Conference.* (pp. 915 - 918). Ramanathapuram, India.

Idrees, F., & Rajarajan, M. (2014). Investigating the android intents and permissions for malware detection. En IEEE (Ed.), *Wireless and Mobile Computing, Networking and Communications (WiMob), 2014 IEEE 10th International Conference.* Larnaca, Cyprus. doi:10.1109/WiMOB.2014.6962194

Microsoft. (2015). *Microsoft Security Intelligence Report*. Recuperado de http://download.microsoft.com/download/1/A/E/1AE5C1D8-8874-481B-94F8-57B41D4E8965/Microsoft_Security_Intelligence_Report_Volume_17_English.pdf

Mouelhi, T., Fleurey, F., Baudry, B., & Le Traon, Y. (2008). A Model-Based Framework for Security Policy Specification, Deployment and Testing. *International Conference on Model Driven Engineering Languages and Systems.* Berlin.

Salva, S., & Zafimiharisoa, S. R. (2015). APSET, an Android aPplication SEcurity Testing tool for detecting intent-based vulnerabilities. *International Journal on Software Tools for Technology Transfer, 17 (2),* 201-221.

TechTarget (2017). *OWASP (Open Web Application Security Project)*. Recuperado de: http://searchsoftwarequality.techtarget.com/definition/OWASP.

Villa (2016). Desarrollo de una aplicación que permita el escaneo de las vulnerabilidades en los dispositivos móviles Android para mitigar los problemas de seguridad. Escuela Superior Politécnica de Chimborazo (ESPOCH), Riobamba, Ecuador.

Anexos

Código de vulnerabilidad de integridad relacionada con los servicios

```
digraph Integrity {

l_0->l_1[label="!intent(action,category,type,uri,extra), [in(type,T+RV(String)) x in(uri,U+RV(String)+INJ) x in(extra,Ve)], *"];

l_1->l_2[label="?running(ServiceS), [in(S.name,ApplicationComponent) x S.isEnable=true],*"];

ll_1->pass[label="?Exception, [*], *"];

l_2->l_3[label=" !q,[data>0], *"];

l_3->pass[label="?open(ContentProviderCP), [in(CP.name,ApplicationComponent) x CP.isEnable=true x data=uri.data], *"];

l_3->fail[label="?open(ContentProviderCP), [in(CP.name,ApplicationComponent) x CP.isEnable=true x data#uri.data], *"];

}
```

Código de vulnerabilidad de integridad relacionado con las actividades

```
digraph Integrity {

l_0->l_1[label="!intent(action,category,type,uri,extra), [in(type,T+RV(String)) x in(uri,U+RV(String)+INJ) x in(extra,Ve)], *"];

l_1->l_2[label="?display(ActivityA), [in(A.name,ApplicationComponent) x A.isEnable=true], *"];

l_1->pass[label="?Exception, [*], *"];

l_2->l_3[label=" !q,[data>0], *"];

l_3->pass[label="?open(ContentProviderCP), [in(CP.name,ApplicationComponent) x CP.isEnable=true x data=uri.data], *"];

l_3->fail[label="?open(ContentProviderCP), [in(CP.name,ApplicationComponent) x CP.isEnable=true x data#uri.data], *"];

}
```

---

1. Profesor y profesional orientado a la Seguridad Informática. Ecuador. Universidad Nacional de Chimborazo. Ingeniero en Sistemas Informáticos. Magíster en Seguridad Telemática. hvilla@unach.edu.ec

2. Profesor y profesional orientado a las Telecomunicaciones y redes. Asesor del Vicerrectorado Académico. Ecuador. Universidad Nacional de Chimborazo. ingeniero en Electrónica y Computación. Magíster en Interconectividad y Redes. ascisneros@unach.edu.ec

3. Profesor y profesional orientado a la Seguridad Informática. Ecuador. Universidad Nacional de Chimborazo. Ingeniero en Sistemas Informáticos. Magíster en Seguridad Telemática. pmendez@unach.edu.ec

4. Coordinadora Nivelación y Admisión. Ecuador. Universidad Nacional de Chimborazo. Ingeniero en Sistemas Informáticos. Magíster en Gerencia Informática. muvidia@unach.edu.ec

5. Profesional orientado a la Telecomunicaciones. Ecuador. Docente investigador Universidad Nacional de Chimborazo. Ingeniero en Sistemas Informáticos. Ph.D. en Telecomunicación. cradicelli@unach.edu.ec

---

Revista ESPACIOS. ISSN 0798 1015
Vol. 39 (Nº 11) Año 2018

[Index]

[En caso de encontrar un error en esta página notificar a webmaster]