Universidad de Buenos Aires

Seguridad de la información

1er cuat. 2014

Tp de Implementación

Integrantes:

Alejandro J. Rebecchi

Jorge Di Vincenzo

Resumen:

El presente trabajo consiste en el desarrollo de dos aplicaciones maliciosas que correrán sobre sistemas operativos Android. Para lograr esto consultamos el libro Android Malware (Xuxian Jiang, Yajin Zhou) en el cual se caracterizan dichas aplicaciones según diferentes aspectos. Principalmente se enuncian las diferentes formas de disparar el *payload* malicioso y los distintos tipos de acciones que puede realizar el mismo. En base a dicha caracterización decidimos desarrollar las siguientes aplicaciones.

En primer lugar una aplicación que se adjunte como un payload malicioso a una aplicación “real” que requiera permisos de acceso a la tarjeta SD y a internet para robar todas las fotos de dicha memoria y enviarlas a un sitio web controlado por nosotros. Para esto se decompilara la aplicación original, se inyectara el servicio y se compilara la aplicacion modificadas para generar un .apk que reúna las características antes mencionadas. De esta forma se persuade al usuario a otorgar los permisos solicitados ya que son acordes al funcionamiento de la aplicación “real”, pero en segundo plano éstos son utilizados para robar las imágenes de la tarjeta y enviarlas utilizando la conexión a internet. Investigaremos sobre la posibilidad de detectar cuando se encuentra conectado a WI-FI para que de ésta forma dichas transferencias no interfieran con el normal funcionamiento de las demás aplicaciones.

Por otro lado se desarrollará una segunda aplicación que consistirá en un teclado virtual que, sin que el usuario sospeche, funcione como keylogger, registrando todas las teclas que fueron presionadas en el mismo y luego enviándolas a través de internet a un sitio web controlado por nosotros. Para la implementación de dicho teclado se consultó la documentación oficial del sitio de android para desarrolladores y aplicaciones existentes de teclaldos para android intentando buscar alguna que tuviera los permisos que requeriamos para no tener que agregarlos y levantar sospechas en los usuarios.

Estas dos aplicaciones trabajaràn de forma independiente al igual que los servicios de servidores que almacenarán la informacion provista por éstas aplicaciones, procesándola para mostrarla de una forma útil al atacante.

Herramientas utilizadas

Para realizar este trabajo tuvimos que utilizar dos tipos de herramientas. En primer lugar necesitabamos encontrar la forma de inyectar código a aplicaciones existentes. Para eso buscamos un tipo aplicaciones llamadas *binders,* que en teoría se encargan de inyectar código a un apk dado. Como no encontramos ninguna que funcionara en el caso general (principalmente no conseguimos ninguna que nos sirviera para implementar el keylogger) decidimos optar por un sistema menos automático en el que tuviéramos un mayor control de cómo se agrega nuestro código a la aplicación. Para eso utilizamos una de las principales herramientas que utilizan los *binders* para hacer su trabajo, los decompiladores/compiladores.

En segundo lugar necesitabamos generar nuestro código y compilar aplicaciones propias para probar independientemente que nuestro

Decompilación y Compilación de aplicaciones

Para compilar y decompilar los archivos *.apk* de las aplicaciones que intentábamos infectar utilizamos una herramienta “APK Multi Tool” (http://www.apkmultitool.com/), la cual es básicamente un script que sirve para automatizar el uso de una serie de aplicaciones utilizadas no sólo para decompilar/compilar archivos *.apk* sino que también nos permite firmarlos e instalarlos en un teléfono/emulador. La herramienta que se encarga propiamente de la decompilación/compilación de las aplicaciones se llama “Apktool 2.0” (https://code.google.com/p/android-apktool/). Esta herramienta permite obtener el código SMALI a partir de una apk compilado, y también permite compilar una carpeta con un proyecto de aplicacion android con archivos en formato SMALI a un archivo *.apk* instalable.

El codigo SMALI

Los programadores Android escriben sus aplicaciones en lenguaje Java. Luego se utilizan compiladores como el SDK de Android integrado con Eclipse para compilar las aplicaciones desde ese código convirtiendo el código java en ejecutables dalvik (archivos .dex). La máquina virtual de dalvik (dalvikvm) en Android puede correr dichos ejecutables directamente. Lamentablemente realizar el proceso inverso no es posible, por lo cual la alternativa es extraer un código intermedio, de características similares al código ASSEMBLER en algunos aspectos. Éste último es el código smali, el cual afortunadamente respeta la estructura de archivos del proyecto original, lo cual lo hace un poco más fácil de entender al analizarlo, pero realizar grandes modificaciones directo sobre este código sería una tarea por lo menos desafiante. Lo que suele hacerse en muchos casos para evitar tener que escribir directamente nuestra funcionalidad en SMALI es generar otra aplicación en código java, compilarla un archivo *.apk,* decompilarla para obtener el código SMALI y luego copiar la parte que nos interesa de este código a la aplicación que intentamos modificar.

Agregando servicios a una aplicación

Android, a la hora de desarrollar una aplicación, nos da la posibilidad de agregarle servicios a la misma. Éstos consisten en rutinas que se ejecutan de una forma bastante independiente del resto de la aplicaciòn pero compartiendo los mismos permisos.

Para ejecutar dichas rutinas se suscribe al servicio a uno o varias de las señales que nos ofrece la API de Android para detectar eventos del sistema y/o la aplicación a la que está asociado el servicio. Para la aplicación los principales eventos que tenemos son: *onCreate*(), *onStart*(), *onRestart*(), *onResume*(), *onStop*(), *onPause*() y *onDestroy*(). Para el sistema los de mayor interés para este tipo de aplicaciones son: BOOT_COMPLETED, SMS_RECEIVED, ACTION_MAIN, UMS_CONNECTED, CONNECTIVITY_CHANGE, PICK_WIFI_WORK, USER_PRESENT, y WAP_PUSH_RECEIVED entre otros.
Para crear un servicio lo primero que debe hacerse es crear una clase que extienda de *android.app.Service* y definir nuestra funcionalidad dentro de dicha clase.

```
public class MyService extends Service {
	...
}
```

Luego debemos registrar dicha clase como servicio de nuestra aplicaciòn en en archivo *Manifest.xml* en la raíz del proyecto de la misma.

```
<application>
	...
		<service android:name="com.example.target.MyService"></service>
	...
 </application>
```

Por último se debe agregar al main activity para ser cargado, por ejemplo, al crear el proceso:

```
protected void onCreate(Bundle savedInstanceState) {
	...
		startService(new Intent(this, MyService.class));
	...
}
```

De esta forma al iniciar la aplicación se inicia el servicio *MyService* automáticamente.

MP3 Downloader con “*Thief service*”

Idea general

Generar una aplicacion que funcione como troyano para dispositivos con sistema operativo Android, a partir de una programa original descargado de Internet. Dicha aplicación es el resultado de adjuntar a la aplicacion real, un servicio desarrollado por nosotros que busque y envíe todas las fotos almacenadas en la memoria SD del dispositivo a un sitio web propio.

La aplicacion resultante funcionara como troyano brindando la funcionalidad de la aplicación original pero corriendo en background el servicio que funcionaria como payload enviando las imágenes de la memoria SD.

Para no generar en el usuario, sospechas sobre la legitimidad de la aplicacion, se decidió implementar un servicio independiente de la interfaz. De este modo la unica modificacion introducida a la aplicacion original es una linea de código que crea y ejecuta una instancia de nuestro servicio, minimizando la posibilidad de que el usuario sospeche ni de los permisos solicitados, ni de la interfaz, ni del comportamiento o el funcionamiento de la aplicacion.

Aplicación original

Respecto de la aplicación original usada como fachada para adjuntar nuestro servicio, se buscó una que por un lado brinde una funcionalidad que resulte útil al usuario, y que por otro lado nos resulte útil para generar el troyano final, esto es, que solicite al usuario los permisos de acceso a Internet y de lectura a la memoria SD, que son los dos permisos mínimos necesarios para que nuestro servicio funcione correctamente.

La aplicación elegida fue "Simple MP3 Downloader 2.0.1": una aplicación para búsqueda, descarga, y reproducción de música en formato MP3. Al momento de instalarse, esta aplicación solicita al usuario:

-Permiso de acceso a Internet para descargar los archivos MP3 (nuestro servicio lo necesita para el envio de las imagenes encontradas)

-Permiso de lectura de la memoria SD para poder reproducir los archivos MP3 y leer unos archivos que la aplicacion usa para llevar registro de los archivos que se descargaron y los que se están descargando (el servicio lo necesita para leer las imagenes de la SD del dispositivo)

-Permiso de escritura en la memoria SD para guardar los archivos descargados (aprovechamos este permiso para generar un archivo que el servicio utiliza para llevar registro de las imagenes enviadas y evitar varios envíos de la misma de imagen)

La aplicación "Simple MP3 Downloader 2.0.1" puede descargarse desde el siguiente link http://m.apps.store.aptoide.com/app/market/org.golden.simpledl/201/5770978/Simple%20mp3%20Downloader)

Servicio malicioso:

La implementación del servicio que se desarrolló para adjuntar como payload a la aplicación anterior, cuenta con dos clases. Una implementa el servicio propiamente dicho (clase ThiefService). La otra es una clase auxiliar que se encarga del envio de las imagenes a nuestro sitio web (clase MultipartServer).

El funcionamiento de la clase ThiefService consta básicamente de dos pasos.

Busqueda:

Primero, el servicio recorre completamente (y en forma recursiva) la estructura de directorios de la memoria SD del dispositivo. La forma de recorrer el árbol de directorios es Depth-First, recorriendo en cada paso el contenido del primer directorio encontrado, y verificando la extensión de todos los archivos encontrados en cada directorio. Si la extensión de un archivo encontrado es JPG, GIF, BMP, PNG, o GIF, se guarda el nombre del archivo y la ruta completa en un lista de strings privada.

Envio y registro de envíos:

En el segundo paso, el servicio itera la lista de imágenes encontradas en el paso anterior y para cada una de ellas se verifica si ya ha sido enviada. En caso de que ya haya sido enviada, se omite un nuevo envio de la imagen. En cambio, si la imagen nunca fue enviada, se envía al sitio de posteo de imágenes y se registra la imagen como enviada.

Para llevar registro de las imágenes enviadas, se decidió crear un archivo dentro de la carpeta donde la aplicación original guarda sus archivos (Carpeta de nombre “simplemp3” en la raíz de la SD). Para este archivo se eligió en nombre “downloaded_files” para que aparente ser de la aplicacion original, ya que en la misma carpeta la aplicacion guarda el archivo "downloading_files" donde lleva registro de los archivos MP3 que actualmente se estan descargando.

Almacenar los nombres de las imagenes en texto plano en nuestro archivo de registro de imagenes enviadas resultaría sospechoso, por lo que se decidió que el servicio guardara una línea (para cada imagen enviada) con un hash de la ruta completa concatenada con el nombre del archivo. El hash se genera usando la función criptográfica de hash SHA-1 que provee java.

Para evitar el envio reiterado de la misma imagen el servicio verifica, para cada imagen encontrada, que el hash de la ruta concatenada con el nombre de la imagen encontrada no haya sido registrado en el archivo de imágenes enviadas.

La clase MultipartServer utiliza el método POST para enviar por HTTP las imagenes encontradas por el servicio. Las imágenes son enviadas al sitio http://cheverebe.pythonanywhere.com/file/, sitio hosteado específicamente para recibir imágenes via POST en el parámetro de nombre “media”.

Desarrollo, pruebas y herramientas utilizadas

El desarrollo del servicio se hizo integramente en la version de Eclipse incluida en el Android SDK (http://developer.android.com/sdk/index.html). Las pruebas durante el desarrollo se corrieron sobre una instancia de AVD (Android virtual Device) incluidas en el mismo kit de desarrollo Android.

Para adjuntar el servicio a la aplicacion original se uso la herramienta “Apktool 2.0” (https://code.google.com/p/android-apktool/). Esta herramienta se uso unicamente para decompilar y recompilar los apk necesarios para generar la aplicacion final.

Para firmar el apk resultante se usaron las herramientas "keytool" (para la generacion del keystore propio) y "jarsigner" para autofirmar el apk. Tanto la herramienta "keytool" como la herramienta "jarsigner" las provee el JDK de java.

Las pruebas definitivas se hicieron utilizando dispositivos físicos reales (celulares y una tablet) con sistema operativo Andoid. Para verificar si el troyano funcionaba correctamente, se hosteo la pagina http://cheverebe.pythonanywhere.com/allfiles/ que muestra todas las imagenes posteadas hasta el momento.

Generacion del troyano

El primer paso en la generacion del troyano final, fue el desarrollo del servicio que explora la SD del dispositivo y envia las imagenes Generar el apk del servicio malicioso.

Para que las pruebas de desarrollo sean más ágiles, se decidió crear un proyecto de los que trae como ejemplo el Android SDK (Un MainActivity sin mas funcionalidad que un "Hello world" e incluir nuestro servicio como una clase más del proyecto. Esto evito tener que adjuntar el servicio a una aplicacion ya compilada, evitando el proceso de decompilacion / compilación en cada prueba. De este modo solo era necesario ejecutar desde Eclipse el proyecto para ver si el servicio funcionaba de la forma esperada, permitiendo ademas "debugear" facilmente la aplicacion para analizar la ejecucion.

Habiendo comprobado que el servicio funcionaba de la forma esperada en nuestra aplicacion de prueba, se decompiló el apk de la aplicacion original (usando Apktool) para obtener el código smali, al cual adjuntarle nuestro servicio.

Luego, para adjuntar nuestro servicio a la aplicacion original, necesitabamos el codigo smali de nuestro servicio, para poder incorporarlo a la aplicacion original decompilada. Para esto se genero el apk de nuestra aplicacion y de compiló usando nuevamente Apktool.

Para poder recompilar la aplicacion original en una versión modificada que sea capaz de correr una instancia de nuestro servicio fue necesario copiar las clases smali decompiladas del servicio, a la carpeta con el codigo smali decompilado de la aplicacion original, y ademas

modificar el manifest de la aplicacion original decompilada para declarar nuestra clase ThiefService como servicio de la aplicacion.

La unica modificacion al codigo smali de la aplicacion original fue el agregado de unas líneas de código que ejecutan una instancia nuestro servicio. Era necesario agregar esta llamada en el método onCreate del Activity principal de la aplicacion original, para que el servicio corra cada vez que la aplicacion es ejecutada. La línea de código java a insertar era

```
startService(new Intent(this, ThiefService.class));
```

pero obviamente necesitabamos la instrucción en código smali para insertarlo en el smali del MainActivity original decompilado. Estas linea de codigo smali las conseguimos del MainActivity de nuestra aplicacion de prueba. Las cuatro líneas smali que levantan la instancia de nuestro servicio son::

```
new-instance v0, Landroid/content/Intent;
const-class v1, Lcom/sim/golden/service/ThiefService;
invoke-direct {v0,p0,v1},Landroid/content/Intent;-><init>(Landroid/content/Context;Ljava/lang/Class;)V
invoke-virtual{p0,v0},Lcom/sim/golden/activity/MainActivity;->startService(Landroid/content/Intent;)Landroid/c
ontent/ComponentName;
```

En esta líneas smali solo modificamos el nombre de los packages de la aplicacion de prueba por los nombres de los packages de la aplicacion original, esto es

- "com/example/target/MainActivity" por "com/sim/golden/activity/MainActivity"
- "com/example/target/ThiefService" por "com/sim/golden/service/ThiefService"

Una vez modificada la aplicacion original para que ejecute en background el servicio adjuntado como payload, se procedió a recopilarla, usando una vez más la herramienta Apktool y generar el apk final del troyano.

Como último paso fue necesario generar un keytore propio usando la herramienta "keytool" de jdk de java y firmar el apk usando "jarsigner" (también provisto por el jdk de java), para permitir la instalación del apk final en cualquier dispositivo que use Android.

Posibles mejoras

El troyano generado guarda un hash computado sobre la ruta de la imagen concatenada con el nombre de la imagen. Esto implica que si la misma imagen se encuentra en otra ruta el hash sería diferente y la imagen se enviariaría nuevamente.

Se podria mejorar esta situación guardando un hash calculado solo sobre el nombre del archivo, pero aun asi podrian reenviarse imagenes varias veces si se encuentra la misma imagen con varios nombres de archivos diferentes.

Para solucionar este problema se podria guardar un hash calculado sobre el contenido completo de la imagen. De esta forma, aun encontrando el archivo dos veces con distinto

nombre o en una ruta diferente, el resultado del hash calculado sería el mismo, y se enviaría una sola vez cada imagen.

Otra posible mejora sería extender la funcionalidad del troyano para incluir mas formatos de imagenes, e inclusive podrían incluirse formatos de archivos que no sean imagenes, tales como documentos de texto, videos y otros. (PDF, DOC, MP4, etc).

Otra buena mejora que no se hizo por falta de tiempo es el desarrollo de una herramienta que automatice la inclusión del payload. Una vez obtenido el código smali de nuestro servicio y la llamada que ejecuta una instancia del servicio, el proceso de adjuntarlo a un apk cualquiera podria ser automatizado. El proceso sería: decompilar el apk en cuestión, copiar el archivo smali del servicio, declarar el servicio en el Manifest, recompilar, y firmar.

Instrucciones de uso

Para probar la aplicacion generada solo hace falta instalar el apk, cediendo los permisos correspondientes, y al correr por primera vez la aplicacion, todas las imagenes de la memoria SD del dispositivo seran enviadas y podran verse en http://cheverebe.pythonanywhere.com/allfiles/.

A partir de ese primer envio en adelante, cada vez que se ejecute la aplicacion y en la memoria SD se encuentren imagenes nuevas, se enviaran unicamente estas ultimas imagenes.

En el archivo "simplemp3/downloaded_files" podrá verse una línea con un hash por cada imagen enviada y si el archivo es eliminado, la proxima ejecucion de aplicacion reenviará todas las imagenes de la memoria SD del dispositivo, aun las que ya hayan sido enviadas.

Aclaración a tener en cuenta en caso de hacer pruebas en una AVD (Android Virtual Device): Por algun motivo la aplicacion levanta el servicio la primera vez que se ejecuta, pero las siguientes ejecuciones de la aplicacion no disparan la ejecución del servicio. Sí funciona si se apaga la AVD, se enciende nuevamente, y se ejecuta otra vez la aplicacion.

Aparentemente es algun problema que afecta el funcionamiento del servicio únicamente corriendo sobre una AVD, ya que en dispositivos físicos el servicio corre cada vez que la aplicacion se ejecuta, sin necesidad de que el dispositivo sea apagado y encendido.

Swiftkey® Keyboard con Keylogger

Se trata de una aplicación cuyo objetivo es el de capturar las teclas que presiona el usuario y enviarlas a un servidor web controlado por nosotros. El desarrollo de la misma conllevaba dos desafíos principalmente: encontrar la forma de acceder a las teclas que se presionan al escribir en cualquier aplicación que se utilice, y luego enviar dicha información via internet, obteniendo

los permisos necesarios para realizar esta operación; y todo ésto sin que el usuario pueda advertir que se está llevando a cabo una acción maliciosa en su contra.

Luego de hacer una serie de investigaciones logramos recopilar la suficiente información para poder afrontar con dicha tarea. En primer lugar encontramos que el sistema operativo Android permite que se instalen aplicaciones para reemplazar el teclado de uso general que trae por defecto el mismo. Esto era una posible opción para el problema de poder capturar las teclas presionadas en cualquier aplicación, ya que si se logra infectar una aplicación de teclado para android, y ésta es luego instalada en un dispositivo y seleccionada como método de entrada por defecto, será utilizada como teclado en todas las aplicaciones permitiendo tener acceso a las teclas que presiona el usuario en cualquier momento.

Con esta información en mente nos dispusimos a buscar una aplicación de éste tipo de la cual pudiéramos conseguir el archivo binario .apk para infectar y además tratando de privilegiar aquella que tuviera características que favorezcan la solución del resto de los problemas con los que tendríamos que lidiar. Luego de evaluar entre varias opciones (las que incluían a algunos de los teclados mas populares como Google keyboard, Smart Keyboard PRO, AI Type Keyboard Plus y Hacker's Keyboard), finalmente nos decidimos por utilizar el Swiftkey Keyboard.

¿Por que el Swiftkey Keyboard?

Los motivos de la elección de Swiftkey Keyboard para realizar este proyecto fueron varios. En primer lugar, al ser una de las aplicaciones de teclado más populares para android, las posibilidades de que un usuario se interese en instalar la aplicacion deberían ser altas ya que actualmente gran cantidad de usuarios utilizan la aplicación legítima. Si a esto le sumamos que es una aplicación paga (o hasta hace muy poco tiempo lo fué), se incrementan las posibilidades de que un usuario intente obtener directamente el archivo *.apk* de aplicación directamente por fuera del android market, lo que nos da la posibilidad de distribuir la aplicación infectada de forma anónima, sin tener que registrarnos en el market como desarrolladores.

Otro de los motivos fue que ésta aplicación, a diferencia de muchas de las otras, requiere permiso para el uso de internet y permisos para lectura y escritura de archivos desde la tarjeta SD o memoria interna del teléfono. Ésto se debe a que el *Swiftkey Keyboard* permite descargar un diccionario predefinido para el o los lenguajes que se elijan por defecto y ademàs sincronizar las palabras personalizadas agregadas al diccionario con el *Swiftkey Cloud* (lo que permite contar con el mismo diccionario en distintos dispositivos asociados a la misma cuenta de Google). El hecho de contar con dichos permisos en la aplicación original hace que se puedan almacenar las teclas presionadas en caso de ser necesario y también enviar ésta información al servidor web sin necesidad de agregar permisos a la aplicación (lo cual podría levantar sospechas, ya que el usuario puede comprobar los permisos de la aplicacion original en el sitio web del android market), o delegar dicha tarea a otra aplicación que cuente con permisos para el uso de internet.

Infectando el Swiftkey Keyboard.

Una vez elegido la app a infectar quedaba por delante el trabajo más importante, que era el de infectar la aplicación. Nuestro primer objetivo era el de encontrar la forma de capturar las teclas que presiona el usuario utilizando el *Swiftkey Keyboard.* Para esto era necesario recorrer la estructura interna de la aplicación para ver si encontrábamos algún indicio de cómo se hacía para procesar estos eventos, por lo cual procedimos a utilizar el programa *Apk tools 2* para decompilar el apk y obtuvimos la estructura del proyecto con los archivos de código fuente SMALI.

Siguiendo algunos indicios que nos daban los nombres de los directorios y utilizando herramientas de búsqueda de archivos por nombre llegamos a encontrar el archivo "*\smali\com\touchtype\keyboard\inputeventmodel\events\KeyInputEvent.smali*" el cual parecía que indudablemente era lo que estábamos buscando. Como mencionamos anteriormente leer y/o editar directamente el código SMALI es una tarea bastante compleja, sin embargo podemos extraer la información suficiente escribir en código Java lo que necesitamos que haga e inyectar luego nuestro código en la aplicación víctima. Dándole una mirada rápida al código encontramos que hay un constructor que recibe un parámetro de nombre *InputText*. Lamentablemente no podemos suscribirnos a este evento con un servicio, lo cual haría la tarea mucho más sencilla, por lo que necesitamos inyectar código directamente dentro de dicho constructor. Decidimos por cuestiones de simplicidad y para intentar evitar estropear el correcto funcionamiento de la aplicación original crear una clase nueva que se encargue de hacer el procesamiento que necesitamos de forma asincrónica y sólo agregar una línea que activa dicho mecanismo pasándole la información de la tecla que fue presionada. Por lo tanto sólo tuvimos que agregar dos pequeños bloques de smali al constructor para hacer el llamado a nuestra clase (y su respectivo import).

De esta forma podemos capturar las teclas que presiona el cliente, sin embargo éste no es el único medio de entrada que tiene este teclado. Esto se debe a que esta aplicación en particular también ofrece un sistema de escritura rápida llamado *Flow.* Al utilizarlo el usuario "dibuja" una línea sobre el teclado y el sistema "intuye" la palabra que se deseaba ingresar. En realidad en la mayoría de los casos ofrece varias opciones. Lo mismo sucede cuando se ofrecen opciones "predictivas" para completar la palabra que se está escribiendo. El problema con estos dos casos es que se ingresa toda la palabra al aceptar una de las opciones que se ofrecen en la parte superior del teclado, lo cual no es capturado por el evento keypress. Por dicho motivo nos dispusimos a buscar también el evento que captura la aceptación de una de las opciones ofrecidas.

De manera similar a la que encontramos el *KeyInputEvent.smali* encontramos el archivo *TextInputEvent.smali*, el cual tiene un funcionamiento similar al de ingreso de tecla, por lo tanto agregamos nuestra funcionalidad también a este archivo para poder capturar estos eventos también.

Una vez que tuvimos resuelto el problema de capturar lo que ingresa el usuario nos dispusimos a implementar el mecanismo para enviar la información obtenida al servidor controlado por nosotros. Se nos plantearon en ese momento dos alternativas:

La primera consistía en almacenar las teclas y/o palabras ingresadas en un archivo para luego, periódicamente y/o al detectar que se está conectado a internet, enviar dicho archivo al servidor. Éste mecanismo tiene un problema, ya que al tener que grabar el archivo en la tarjeta, si el usuario lo encuentra probablemente se dará cuenta de que nuestra aplicación es un keylogger. Sin embargo permite que se almacene la información si el usuario no está conectado a la red hasta que vuelva a conectarse. Además facilita la tarea en el servidor al enviar grandes bloques de informacion correspondiente al mismo usuario en un sólo envío, evitando que se tenga que realizar la recopilación a posteriori. Para realizar ésto es necesario registrar el servicio que se encargará de realizar el envío del contenido del archivo al servidor y suscribirlo al disparador del evento de detectar conexión.

La segunda opción era la de enviar el texto ingresado en el momento que se detecta el evento de presionar tecla o aceptar la palabra. Esto tiene una desventaja que es que si el usuario no está conectado en ese momento no se puede enviar dicha información. Sin embargo esto no es 100% verdad, ya que la API de Android nos obliga a realizar todas las tareas que implican conectarse a internet dentro de *AsyncTasks.* Éstas son tareas que se ejecutan de forma asincrónica, por lo cual dicha tarea puede reintentar el envío hasta recuperar la conexión. No es una buena estrategia por un lapso prolongado porque se pueden acumular las tareas en segundo plano provocando un impacto en la performance del sistema.

Por cuestiones de tiempo y simplicidad tuvimos que optar por la segunda opción dejando pendiente como una posible mejora de nuestra aplicación implementar la primera de éstas opciones.

El webserver

Si bien el webserver no es era de las partes mas importantes del proyecto, al mandarse el texto ante cada presion de tecla o ingreso de palabra, se le termino delegando una mayor responsabilidad, ya que ahora se debía recopilar todos los mensajes enviados desde la misma dirección ip y mostrarlos juntos para que sean legibles.

El servidor está desarrollado en Python utilizando un Django framework, y básicamente se encargar de guardar en una base de datos todos los requests recibidos con la ip del cliente, la fecha y el texto enviado. Luego para mostrarlos se agrupan por dirección ip y se los ordena por fecha.

Pruebas y resultados

Para probar el funcionamiento de la aplicación primero intentamos instalarla en el emulador de la SDK de Android pero luego de instalarlo, la aplicación fallaba al intentar ejecutarla y se cerraba automáticamente. Por lo tanto las pruebas debieron realizarse únicamente sobre dispositivos reales.

Primero se realizaron pruebas estando conectado el dispositivo a internet mediante WI-FI. El rendimiento fue muy bueno, el texto aparecía casi instantaneamente en el servidor y en el dispositivo no se observó ninguna degradación de la performance.

Luego se realizaron pruebas con el dispositivo conectado mediante 3G, lo cual supone una conexión considerablemente más lenta que por medio del WI-FI. Sin embargo el rendimiento fué prácticamente el mismo que en el caso anterior. Dado que el texto que se envía no es muy extenso la diferencia en el tiempo que toma el envio no parece significativa.

Por último se realizó una prueba con el teléfono completamente desconectado de internet, principalmente para ver si la aplicación funcionaba correctamente a pesar de no poder enviar los requests al server. Afortunadamente ésto fue lo que sucedió, aunque el texto ingresado en ese período no fue capturado por el server y luego no se pudo recuperar.

Algo que notamos al inspeccionar el texto recibido fué que no solo se mostraba una letra o una palabra, sino que aparecían frases enteras. Esto se debe a que el swiftkey realiza las sugerencias en base no sólo de los caracteres actuales, sino que también evalúa la frase entera, lo cual es de mayor utilidad para nosotros, ya que nose evita un poco del trabajo de la reconstruccion de las frases.

Instrucciones de uso

Para probar la aplicación sólo hace falta instalarla en un dispositivo real, ya que por limitaciones de los emuladores, no funciona ni en el emulador de la SDK de Android ni en Genymotion. Durante la instalación se pedirá no solo aceptar los permisos requeridos sino que también se descargará el diccionario para algún lenguaje a elegir, se pedirá activar *Swiftkey* como teclado y elegirlo como predeterminado. Una vez hecho esto tendremos habilitado el Swiftkey con nuestro keylogger para utilizarlo en la aplicación que sea.
Luego para visualizar en el servidor la información enviada al servidor basta con acceder a la siguiente url http://cheverebe.pythonanywhere.com/all/. Ahi se podrá visualizar todos los envios realizados por la aplicación agrupados según la ip de la que se recibió.

Posibles mejoras

Como mencionamos anteriormente queda pendiente la implementación del almacenamiento de la información en un archivo para poder capturar el texto ingresado cuando el dispositivo no está conectado a internet, agregando un servicio que se encargue de enviarlo cuando se detecte que se reestableció la conexión.

Además al ver que se recibe la frase completa se podría implementar un procesamiento del log para evitar las repeticiones y solo guardar la frase final.