Taller de ejercicios de vulnerabilidades en aplicaciones móviles

Tema 5. Detección de problemas de seguridad en aplicaciones para dispositivos móviles

Índice general

Capítulo 1

Instalación del entorno

En esta sección vamos a instalar y desplegar el entorno de pruebas que utilizaremos a lo largo del taller.

1.1. Kali Linux y Docker

Para la realización del taller necesitaremos algunas herramientas instaladas en Kali Linux (ADB) y Docker para el rápido despliegue de Mobile Security Framework (MobSF) y Drozer. En caso de no disponer de estas herramientas, podéis consultar los recursos de temas anteriores para ello.

1.2. Instalando Genymotion

Vamos a instalar Genymotion para virtualizar dispositivos Android y poder instalar y ejecutar algunas aplicaciones vulnerables. **Genymotion utiliza VirtualBox para funcionar, por lo que es necesario instalarlo previamente.**

Por un lado, si usáis Linux podéis seguir estos pasos para instalar Genymotion: https://linuxhint.com/install_genymotion_android_emuator_ubuntu/. Por otro, si utilizáis Windows como SO principal:

- Descargamos Genymotion desde el siguiente enlace (con o sin VirtualBox): https://www.genymotion.com/download/.
- Realizamos el proceso de instalación habitual de Windows y ejecutamos Genymotion.
- Genymotion nos obliga a registrarnos y a verificar nuestra cuenta. Seguid un proceso de registro/verificación habitual.
- Cuando completemos el registro, nos logueamos en la aplicación (en el formulario de la Figura 1.1). Después nos preguntará por la licencia, elegimos la de uso personal.

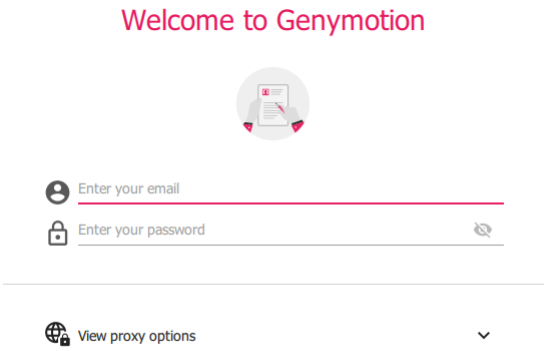

Figura 1.1: Página de login de Genymotion

1.3. Creando el dispositivo virtual

Una vez que accedemos a Genymotion debemos agregar un dispositivo virtual (pulsamos la cruz de arriba a la derecha). Para evitar problemas de compatibilidad con las APKs que utilizaremos, sería conveniente elegir la misma API (23) y dispositivo (Google Nexus 5X) que vemos en la figura 1.2.

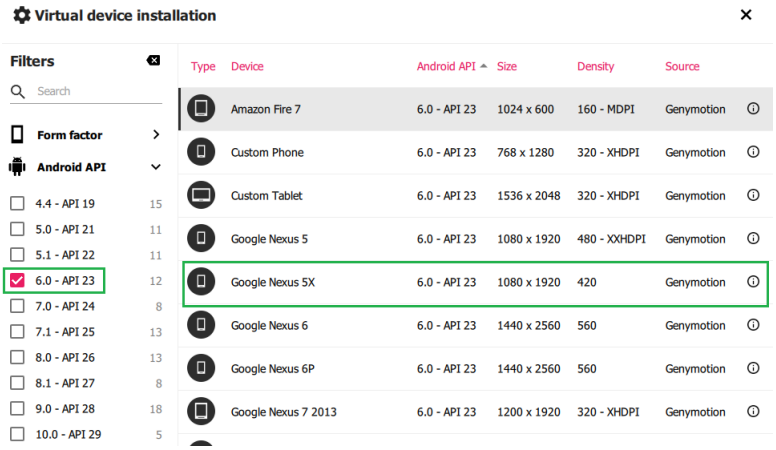

Figura 1.2: Elección del dispositivo a emular en Genymotion

Una vez que elegimos el dispositivo, aparecerá la pantalla de configuración (Figura 1.3). Podéis dejar la configuración del dispositivo virtual por defecto o configurarlo según los recursos disponibles en vuestra máquina física (núcleos CPU, RAM, etc). En la parte inferior podemos encontrar las opciones de red. **Elegimos NAT o bridge según la configuración que le dimos a la máquina Kali Linux en el taller anterior: el objetivo es que todas nuestras máquinas se vean entre sí.**

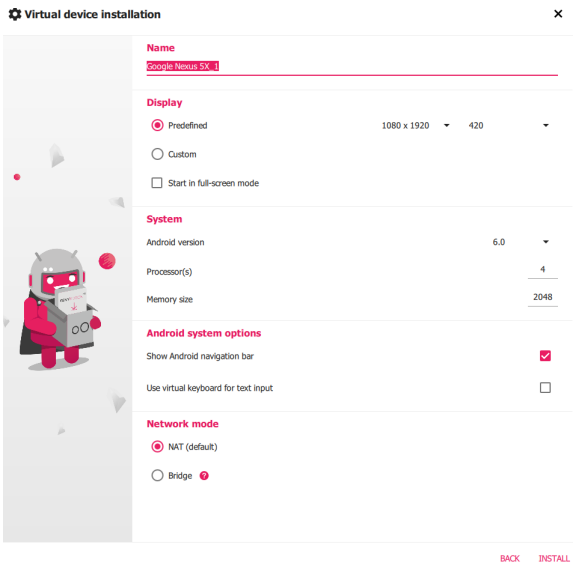

Figura 1.3: Pantalla de configuración del dispositivo virtual

Finalmente, ya podemos arrancar el móvil virtual (doble click en el dispositivo creado en Genymotion). Una vez en ejecución, vamos a instalar las aplicaciones vulnerables que utilizaremos en el taller (APKs disponibles en el Aula Virtual). Para ello, basta con arrastrar las APKs a la pantalla del móvil virtual (Figura 1.4).

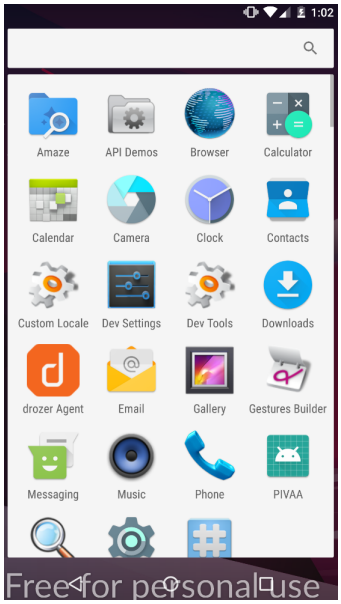

Figura 1.4: Aplicaciones vulnerables instaladas

Capítulo 2

Purposefully Insecure and Vulnerable Android Application

En este capítulo vamos a explotar vulnerabilidades de *Purposefully Insecure and Vulnerable Android Application* (PIVAA).

2.1. Decompilado y análisis previo

El primer paso será analizar el APK con MobSF. Para ello, primero desplegamos y ejecutamos el framework utilizando Docker:

```
docker pull opensecurity/mobile-security-framework-mobsf

docker run --rm -p 8000:8000 -d opensecurity/mobile-security-
    framework-mobsf
```

Listing 2.1: MobSF con Docker

Una vez levantado el contenedor, vamos al navegador y accedemos a la interfaz web de MobSF. Aquí arrastraremos el APK de PIVAA. Después de un breve periodo de tiempo, aparecerá la página principal del análisis de MobSF (similar al de la Figura 2.1). En esta página, podemos encontrar multitud de información relacionada con el APK analizado. En concreto, aparte de las actividades que componen la aplicación (10), nos pueden interesar los servicios, broadcast receivers y content providers. En la Figura 2.1 vemos cómo la aplicación utiliza y exporta uno de cada (los tres vulnerables). Por otro lado, la sección *DECOMPILED CODE* (Figura 2.2) nos muestra el AndroidManifest.xml, el código fuente y el código Smali del APK analizado.

2.2. Main Activity (Login)

Tras realizar el primer análisis abrimos la aplicación PIVAA en el dispositivo virtual de Genymotion. La primera actividad que vemos es una pantalla

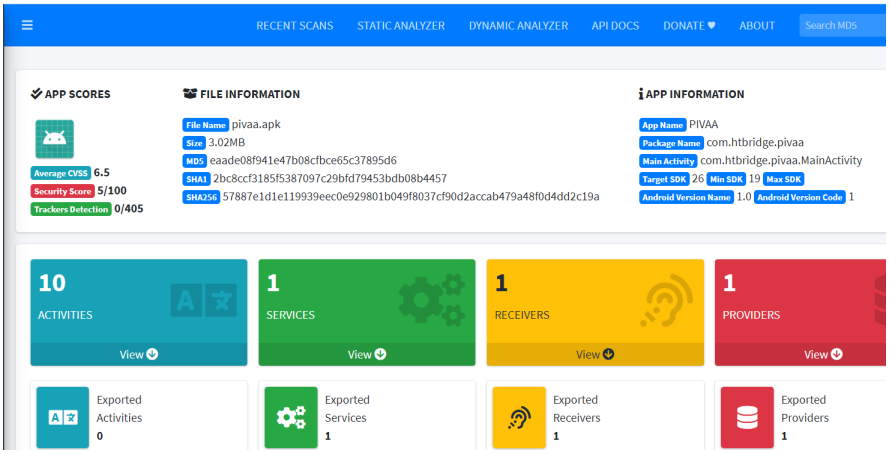

Figura 2.1: Página principal de MobSF

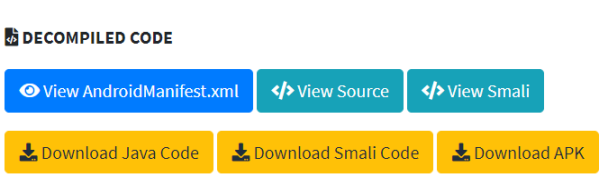

Figura 2.2: Sección *DECOMPILED CODE*

de login (Figura 2.3). Para ver y cambiar entre las diferentes pantallas/actividades de la aplicación, pulsad el botón de menú (tres puntos) de la parte superior derecha.

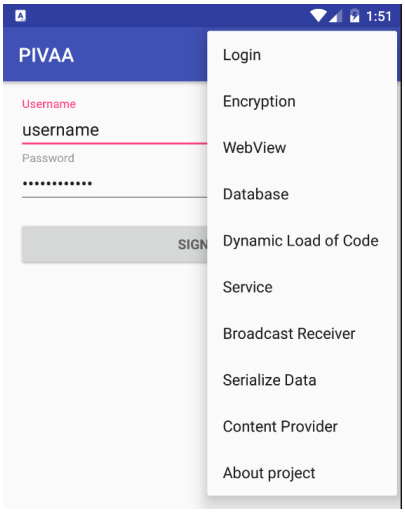

Figura 2.3: Login de PIVAA

Ahora vamos a analizar la actividad relacionada con la pantalla de login. Para ello, vamos a la sección *DECOMPILED CODE* de MobSF, código fuente y buscamos el archivo *MainActivity.java*. La ubicación puede verse en la Figura 2.4.

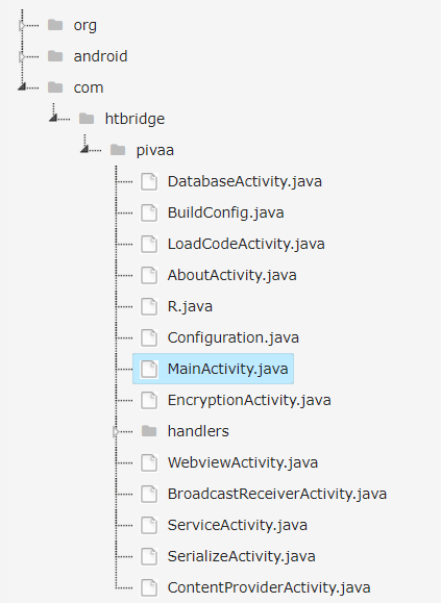

Figura 2.4: Ubicación de MainActivity.java

Vamos a centrarnos en **dos fallos de seguridad** existentes en MainActivity.java. El primer fallo trata del envío de información sensible al log del sistema. Sin embargo, esto no puede verse a simple vista, hay que analizar el proceso (es decir, el código) que sigue un intento de login en la aplicación. Es fundamental detectar que la actividad importa paquetes propios (Figura 2.5). Estos pueden encontrarse dentro de la carpeta 'handlers' del código fuente.

```
import com.htbridge.pivaa.handlers.API;
import com.htbridge.pivaa.handlers.Authentication;
import com.htbridge.pivaa.handlers.MenuHandler;
import com.htbridge.pivaa.handlers.database.DatabaseHelper;
import com.htbridge.pivaa.handlers.database.DatabaseRecord;
```

Figura 2.5: Paquetes propios

Revisando el código, vemos que se crea una tarea asíncrona para intentar el login (línea 200, Figura 2.6). En esta función (UserLoginTask) se llaman otras tres funciones de la clase *Authentication*. Vamos a analizar esta última clase. Si observamos su función *saveCache*, vemos que envía la información de usuario y contraseña al log del sistema. Hemos encontrado el primer fallo de seguridad (**haced una captura de la línea de código vulnerable en cuestión**).

```
public class UserLoginTask extends AsyncTask<Void, Void, Boolean> {
    private final String mPassword;
    private final String mUsername;

    UserLoginTask(String username, String password) {
        this.mUsername = username;
        this.mPassword = password;
        Authentication authCreds = new Authentication();
        authCreds.createLockFile(MainActivity.this.getApplicationCon
        authCreds.saveCache(MainActivity.this.getApplicationContext(
        authCreds.saveLoginInfoExternalStorage(MainActivity.this.get
```

Figura 2.6: Funciones de *Authentication.java* llamadas en el login

Volviendo a la Figura 2.6, el segundo fallo de seguridad lo encontramos en la función *saveLoginInfoExternalStorage*. Si la analizamos, vemos que intenta almacenar información en la tarjeta SD sin cifrar. Esto es peligroso debido a la capacidad de acceso que tendrán las demás aplicaciones con permisos de lectura en la SD. Podemos observar el código de la función en la Figura 2.7.

```
public boolean saveLoginInfoExternalStorage(Context context, String username, String password) {
    if (!isExternalStorageWritable()) {
        return false;
    }
    Log.i("htbridge", "saveLoginInfoExternalStorage: writable, all ok!");
    Log.i("htbridge", "getExternalStorageDirectory = " + Environment.getExternalStorageDirectory());
    Log.i("htbridge", "getExternalStoragePublicDirectory = " + context.getExternalFilesDir(null));
    String filename = context.getExternalFilesDir(null) + "/credentials.dat";
    Log.i("htbridge", "saveLoginInfoExternalStorage: username = " + username + " | password = " + password);
    Log.i("htbridge", "saveLoginInfoExternalStorage: opening for writing " + filename);
    File file = new File(context.getExternalFilesDir(null), "/credentials.dat");
    Log.i("htbridge", "saveLoginInfoExternalStorage: opening for reading " + filename);
    try {
        file.createNewFile();
        FileOutputStream fOut = new FileOutputStream(file);
        OutputStreamWriter myOutWriter = new OutputStreamWriter(fOut);
        myOutWriter.write(username + ":" + password + "\n");
        myOutWriter.close();
        fOut.flush();
        fOut.close();
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}
```

Figura 2.7: Función *saveLoginInfoExternalStorage*

2.2.1. Explotando los fallos de seguridad

En primer lugar, vamos a usar Android Debug Bridge (ADB), una herramienta CLI que te permite comunicarte con un dispositivo móvil Android. Podemos usar el ADB que viene con Genymotion (carpeta 'tools' dentro del directorio de instalación) o instalarlo rápidamente en Kali Linux con *apt install adb*. En este caso lo haremos desde Kali Linux. **NOTA: podemos obtener la IP del dispositivo virtual en la aplicación 'Settings', 'Wi-Fi', icono tres puntos parte superior derecha y 'Advanced' (Figuras 2.8 y 2.9).**

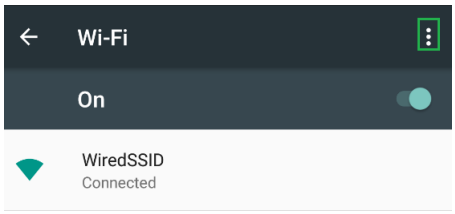

Figura 2.8: Menú desplegable en Settings, Wi-Fi

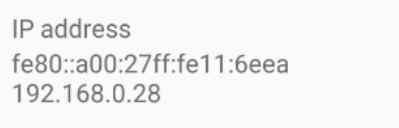

Figura 2.9: IP del dispositivo virtual

```
adb connect IP_ANDROID:5555

adb shell

root@vbox86p:/ # logcat
```

Listing 2.2: ADB y Logcat para leer el log

Como vemos en el Listing 2.2, nos conectamos vía TCP/IP al dispositivo, lanzamos una shell remota interactiva y, una vez dentro del dispositivo virtual, ejecutamos Logcat, una herramienta CLI que vuelca los logs de sistema de Android. Una vez que tengamos Logcat en ejecución, podemos volver al login de la aplicación PIVAA e intentar loguearnos (si observáis otra vez el código de la función *saveCache*, da igual si el usuario y/o contraseña existen). **¿Podéis enseñarme en Logcat el usuario y contraseña que acabáis de introducir?**

El segundo fallo de seguridad que detectamos antes fue que la aplicación almacena las credenciales en el almacenamiento externo sin cifrar, donde pueden ser fácilmente leídas por cualquiera que extraiga la tarjeta o por aplicaciones con permisos de lectura en la SD. Para explotar este fallo, empleamos nuevamente ADB.

```
root@vbox86p:/ # cd /sdcard/Android/data/com.htbridge.pivaa/files

root@vbox86p:/sdcard/Android/data/com.htbridge.pivaa/files # cat
    credentials.dat
```

Listing 2.3: Acceso a las credenciales almacenadas en la sd

Siguiendo los comandos del Listing 2.3, accedemos a la tarjeta SD y leemos los contenidos que se almacenan al usar la función *saveLoginInfoExternalStorage*. Si nos fijamos nuevamente en la Figura 2.7, observamos que

se almacena información en *credentials.dat*, por eso sabemos dónde buscar. **¿Podéis enseñarme el contenido de vuestro *credentials.dat*?**

2.3. Content provider vulnerable

Analizando PIVAA con MobSF, vimos que existía un content provider vulnerable. Vamos a tratar de obtener la información del mismo. Para ello, vamos a utilizar la herramienta Drozer, un framework de pruebas de seguridad para Android. Una de las APKs que hemos instalado en nuestro dispositivo virtual hace referencia al Drozer Agent (servidor embebido) al que nos conectaremos desde fuera mediante la consola Drozer. Tras abrir la aplicación, pulsamos el botón *off* para que cambie a *on* y se ejecute el servidor embebido (normalmente en el puerto 31415). Se verá algo similar a la Figura 2.10.

Figura 2.10: Drozer Agent

Ahora necesitamos un cliente/consola Drozer para poder interactuar con el servidor embebido. En nuestro caso, lo desplegaremos con Docker en nuestra máquina Kali Linux:

```
docker pull fsecurelabs/drozer

docker run -it fsecurelabs/drozer
```

Listing 2.4: Cliente/consola Drozer con Docker

Para conectarnos al servidor embebido desde la consola Drozer en Docker, necesitamos la IP del dispositivo virtual (ya la tenemos del apartado anterior). Ya solo nos queda conectar mediante el comando del Listing 2.5. **Haced una captura de la salida de la consola de Drozer.**

```
root@85c56e37978a:/# drozer console connect --server 192.168.0.28
```

Listing 2.5: Conexión al servidor Drozer

Si volvemos al análisis que hemos hecho con MobSF, en el apartado de providers podemos ver dónde se implementa el content provider en la aplicación (*com.htbridge.pivaa.handlers.VulnerableContentProvider*). Este content provider tiene una peculiaridad, funciona mediante consultas SQL (en concreto, utiliza SQLite). Como vemos en la Figura 2.11, el content provider llama a *rawSQLQueryCursor*. Esta función está dentro de *DatabaseHelper.java*, que a su vez se encuentra dentro de la carpeta 'database' (también en 'handlers'). En *DatabaseHelper.java*, encontramos la llamada final a SQLite (Figura 2.12), la cual presenta un fallo de seguridad debido al uso de la función *rawQuery* sin ningún tipo de saneamiento de la consulta que se le pasa por parámetro.

```
public Cursor query(Uri uri, String[] projection, String select
    Log.i("htbridge", uri.toString());
    Log.i("htbridge", uri.getLastPathSegment());
    this.db.initDatabaseOuter();
    return this.db.rawSQLQueryCursor(uri.getLastPathSegment());
```

Figura 2.11: Llamada a *rawSQLQueryCursor*

```
public String rawSQLQuery(String query) {
    StringBuilder sb = new StringBuilder();
    SQLiteDatabase db = getWritableDatabase();
    try {
        Log.d("htbridge", "rawSQLQuery: " + query);
        Cursor cursor = db.rawQuery(query, null);
```

Figura 2.12: Llamada a la base de datos

A continuación, usamos la consola de Drozer para encontrar los identificadores de recursos uniformes (URI) que apuntan al content provider. Para ello, ejecutamos el comando del Listing 2.6, que utiliza el módulo *app.provider.finduri* sobre el nombre del paquete de la aplicación PIVAA:

```
dz> run app.provider.finduri com.htbridge.pivaa
```

Listing 2.6: Encontrando los URI del content provider

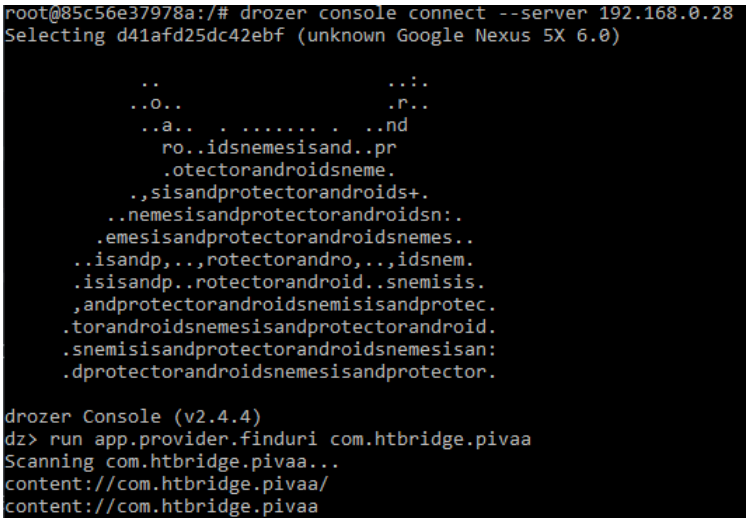

Figura 2.13: URI del content provider

El resultado será el de la Figura 2.13. Con el URI obtenido, podemos inyectar cualquier consulta mediante SQL. Un ejemplo es el del Listing 2.7 (utilizando el módulo de Drozer *app.provider.query*). **¿Podéis enseñarme la salida de dicha consulta?**

```
dz> run app.provider.query content://com.htbridge.pivaa/"select *
    from data"
```

Listing 2.7: Extracción de información del content provider

2.4. Servicio vulnerable

Otro detalle que podemos obtener con el análisis estático de MobSF es la existencia de un servicio (vulnerable) que se encarga de la grabación de un audio de hasta un máximo de 1 MB (puedes ver el código en el archivo *VulnerableService.java* dentro de la carpeta 'handlers'). **¿Podrías identificar el servicio en el AndroidManifest.xml? ¿Qué atributos tiene configurados?**

El fallo de seguridad de este servicio se debe a que se exporta/se hace visible para las demás aplicaciones y no se especifica ningún permiso en el AndroidManifest.xml para dicho servicio. Por lo tanto, cualquier aplicación puede acceder, ejecutar y abusar del servicio. El comando del Listing 2.8 nos permite ejecutar el servicio de manera inmediata (usa Drozer junto con el módulo *app.service.start* y el componente en cuestión). Finalmente, en la Figura 2.14 vemos el resultado de arrancar el servicio.

```
dz> run app.service.start --component com.htbridge.pivaa com.htbridge
    .pivaa.handlers.VulnerableService
```

Listing 2.8: Arrancando el servicio vulnerable con Drozer

Figura 2.14: Resultado de arrancar el servicio

2.5. Broadcast receiver vulnerable

Finalmente, otro detalle interesante que vimos en MobSF es la existencia de un broadcast receiver vulnerable exportado que no configura ningún permiso. Este fallo de seguridad permitirá que otras aplicaciones activen el broadcast receiver vulnerable mediante el envío de objetos Intent. En concreto, en este ejemplo, otras aplicaciones podrán crear/manipular datos de la aplicación vulnerable.

```
<receiver android:name="com.htbridge.pivaa.handlers.VulnerableReceiver" android:protectionLevel="dangerous" androi
    <intent-filter>
        <action android:name="service.vulnerable.vulnerableservice.LOG" />
    </intent-filter>
</receiver>
<provider android:name="com.htbridge.pivaa.handlers.VulnerableContentProvider" android:protectionLevel="dangerous"
```

Figura 2.15: Definición del broadcast receiver

Si nos fijamos en su definición en el AndroidManifest.xml (Figura 2.15), vemos que cualquier aplicación podría enviar un objeto Intent a esta aplicación cuya acción sea *service.vulnerable.vulnerableservice.LOG* (ya que no se especifican permisos). Si accedemos al código (**¿puedes encontrar y relacionar el código referente al broadcast receiver y el código donde se llama?**) vemos cómo el objeto Intent puede enviar dos campos extra, *location* y *data*: el primer campo determina dónde se almacenará el archivo a crear/manipular, mientras que el segundo determinará su contenido.

El Listing 2.9 muestra cómo explotar esta vulnerabilidad usando Drozer y su módulo *app.broadcast.send*. Como vemos, almacenamos el texto *'nuestro texto'* en el almacenamiento SD externo, creando un nuevo archivo llamado *broadcastfile* (también podríamos manipular algún archivo existente).

```
dz> run app.broadcast.send --action service.vulnerable.
    vulnerableservice.LOG --extra string data "nuestro texto" --extra
     string location "/sdcard/Android/data/com.htbridge.pivaa/files/
    broadcastfile"
```

Listing 2.9: Uso de Drozer para explotar el broadcast receiver

Finalmente, en la Figura 2.16 podemos ver el nuevo archivo y su contenido en la memoria externa del dispositivo. **¿Cómo y con qué herramienta podríamos ver el nuevo fichero? Haced una captura de su contenido**.

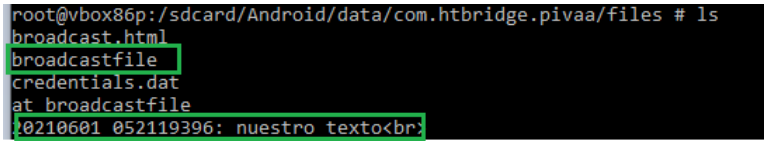

Figura 2.16: Resultado de explotar el broadcast receiver vulnerable