Informe Auditoría de aplicación Android

Autor:

Ángel Luis Rodríguez Martín-Palomino

Índice de contenidos

Resumen ejecutivo.....**3**

Resumen Técnico (Análisis de vulnerabilidades).....**3**

Uso indebido de credenciales e información hadcodeada.....3

Evidencias.....*3*

Score CVSS.....*6*

Recomendaciones.....*7*

Criptografía insuficiente.....7

Evidencias.....*7*

Score CVSS.....*9*

Recomendaciones.....*10*

Filtración de datos (Log inseguro).....10

Evidencias.....*10*

Score CVSS.....*11*

Recomendaciones.....*12*

Índice de imágenes

Imagen 1. Actividad principal de la aplicación.....3
Imagen 2. Actividad principal vista desde el AndroidManifest.xml.....4
Imagen 3. Código fuente de la MainActivity.....4
Imagen 4. Código de la función attemptLogin().....5
Imagen 5. Código de la función checkPassword().....5
Imagen 6. Decodificación de la contraseña en base64.....5
Imagen 7. Uso de crackstation para obtener contraseña origianl a partir de md5.....6
Imagen 8. Código de funciones que desencriptan la ruta de la URL con AES.. .8
Imagen 9. Código de la función decryptAES().....8
Imagen 10. Detección de la vulnerabilidad de padding por MobSF.....8
Imagen 11. Descifrado de ruta de la función getFile() con AES/CBC en CyberChef.....9
Imagen 12. Descifrado de ruta de la función getFiles() con AES/CBC en CyberChef.....9
Imagen 13. Codigo de la función onPostExecute que realiza el logging inseguro.....11
Imagen 14. Entrada con información sensible de un usuario en logs del sistema.....11

Resumen ejecutivo

La aplicación auditada, `ctfapp`, con nombre de paquete `com.rooted.ctfapp`, presenta vulnerabilidades de impactos medios, altos y hasta críticos como son el grabado «*a fuego*» de contraseñas en el código fuente, el uso insuficiente o deficiente de la criptografía y la filtración de datos sensibles de los usuarios a través de los logs del sistema. Estas vulnerabilidades pueden poner en peligro la seguridad y privacidad de los usuarios y permitir que actores maliciosos obtengan acceso al contenido privado de la aplicación saltándose los procesos de autenticación.

En los siguientes apartados se detallará en qué consisten las vulnerabilidades encontradas, se mostrarán evidencias de las mismas junto con su impacto en CVSS y se propondrán posibles mejoras y recomendaciones para mitigarlas.

Resumen Técnico (Análisis de vulnerabilidades)

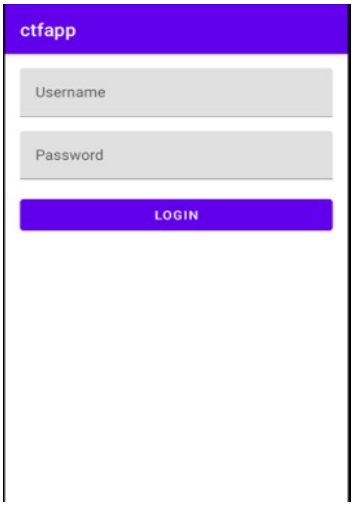
Uso indebido de credenciales e información hadcodeada

La aplicación permite el inicio de sesión a cualquier usuario sin conocer ningún nombre y contraseña previos simplemente obteniendo el fichero `.apk` y accediendo al código fuente de la aplicación. Esto se debe a una gestión deficiente de credenciales donde estás se incluyen en el código directamente al igual que los nombres de usuarios válidos.

Evidencias

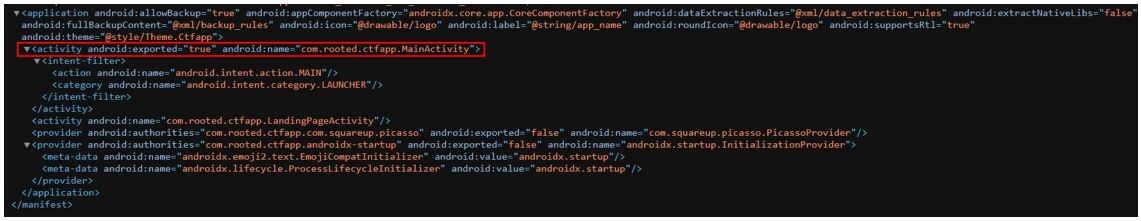
Al abrir la aplicación la página principal que muestra es la siguiente.

Imagen 1. *Actividad principal de la aplicación.*



Por tanto, esta debe de corresponder a la actividad principal `com.rooted.ctfapp.MainActivity` que se nos indica en el fichero `AndroidManifest.xml` (obtenido tras utilizar la herramienta `apktool d` sobre el fichero `.apk`).

Imagen 2. *Actividad principal vista desde el AndroidManifest.xml*



Teniendo acceso al archivo `.apk` se puede utilizar la herramienta `jadx-gui` sobre dicho fichero para obtener el código fuente en formato legible y ahí buscar dicha actividad principal.

Imagen 3. *Código fuente de la MainActivity.*

```
/* loaded from: classes.dex */
public class MainActivity extends AppCompatActivity {
    private boolean loginError = false;
    private TextInputEditText passwordEditText;
    public String username;
    private TextInputEditText usernameEditText;

    @Override // androidx.fragment.app.FragmentActivity, androidx.activity.ComponentActivity, androidx.co
    protected void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(R.layout.activity_main);
        this.usernameEditText = (TextInputEditText) findViewById(R.id.username_edit_text);
        this.passwordEditText = (TextInputEditText) findViewById(R.id.password_edit_text);
        ((Button) findViewById(R.id.login_button)).setOnClickListener(new View.OnClickListener() { // fro
            @Override // android.view.View.OnClickListener
            public final void onClick(View view) {
                this.f$0.m90lambda$onCreate$0$comrootedctfappMainActivity(view);
            }
        });
    }

    /* renamed from: lambda$onCreate$0$com-rooted-ctfapp-MainActivity, reason: not valid java name */
    /* synthetic */ void m90lambda$onCreate$0$comrootedctfappMainActivity(View view) {
        attemptLogin();
    }
```

Como se puede ver en la anterior imagen, al crear la actividad principal con el método `onCreate()` se establece un *event listener* sobre el botón de *login* que se vio en la interfaz de tal manera que al pulsar sobre el se ejecuta la función `attemptLogin()`.

En dicha función se puede ver como se compara que el nombre de usuario introducido en el cuadro de texto coincida con la cadena `john` o `admin` y que la comprobación de la contraseña sea correcta. Es decir, existen sólo dos usuarios, `john` o `admin`, cuyos nombres están escritos directamente en el código en texto plano.

Imagen 4. *Código de la función attemptLogin().*

```
private void attemptLogin() {
    this.username = this.usernameEditText.getText().toString();
    String string = this.passwordEditText.getText().toString();
    if ((this.username.equals("john") || this.username.equals("admin")) && checkPassword(string)) {
        Toast.makeText(this, "Login successful", 0).show();
        Intent intent = new Intent(this, (Class<?>) LandingPageActivity.class);
        intent.putExtra("USERNAME", this.username);
        startActivity(intent);
        finish();
        return;
    }
    this.loginError = true;
    Toast.makeText(this, "Incorrect username or password.", 0).show();
}
```

Además, la función `checkPassword()` compara el valor introducido con otro valor *hardcodeado* en lo que parece ser base64 de la siguiente manera.

Imagen 5. *Código de la función checkPassword().*

```
private boolean checkPassword(String str) {
    return Base64.getEncoder().encodeToString((md5(str) + "nohashnopass").getBytes()).equals("N2JhYzUyYThlZDgzNGNiMGQ2NTFmYTE5ZmUzY2RhZTZub2hhc2hub3Bhc3M=");
}

private String md5(String str) {
    try {
        byte[] bArrDigest = MessageDigest.getInstance(MessageDigestAlgorithms.MD5).digest(str.getBytes());
        StringBuilder sb = new StringBuilder();
        for (byte b : bArrDigest) {
            sb.append(Integer.toHexString((b & UByte.MAX_VALUE) | 256).substring(1, 3));
        }
        return sb.toString();
    } catch (NoSuchAlgorithmException unused) {
        return null;
    }
}
```

El funcionamiento es trivial, se realiza un hash MD5 sobre la contraseña introducida y se añade la cadena `nohashnopass`. Dicho resultado se codifica en base64 y se compara con la cadena `N2JhYzUyYTh1ZDgzNGNiMGQ2NTFmYTE5ZmUzY2RhZTZub2hhc2hub3Bhc3M=`

El uso de un hash MD5 ya es una mala práctica en sí puesto que se han descubierto colisiones, pero además el hecho de que se codifique como base64 significa que se puede decodificar fácilmente como se evidencia en la siguiente imagen.

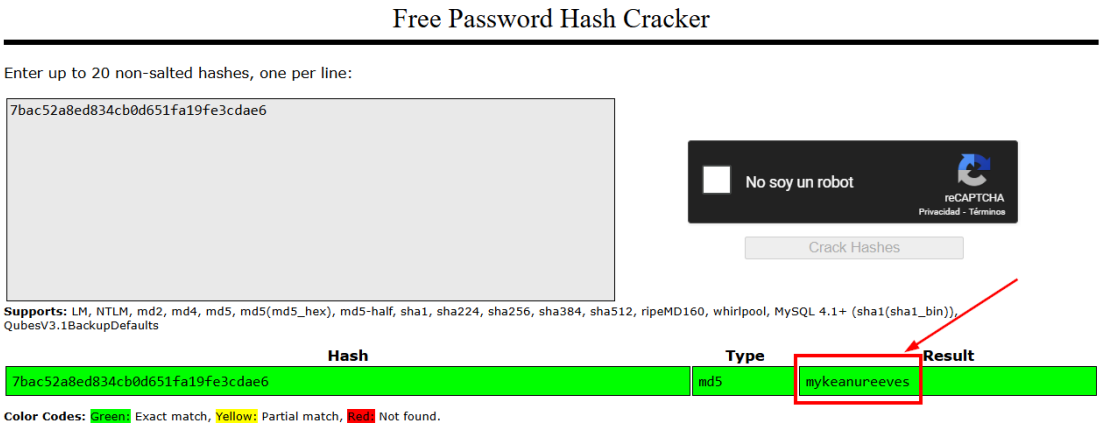
Imagen 6. *Decodificación de la contraseña en base64.*

```
$ echo "N2JhYzUyYThlZDgzNGNiMGQ2NTFmYTE5ZmUzY2RhZTZub2hhc2hub3Bhc3M=" | base64 -d
7bac52a8ed834cb0d651fa19fe3cdae6nohashnopass
```

Por tanto, se obtiene la cadena `nohashnopass` junto con el hash md5 de la contraseña (`7bac52a8ed834cb0d651fa19fe3cdae6`), tal y como se vio en el extracto de código anterior.

Una vez se tiene este hash, se pueden utilizar herramientas como *Crackstation* que tienen una tabla de contraseñas precalculadas para cada hash y en cuestión de segundos se tiene el valor original de la contraseña.

Imagen 7. *Uso de crackstation para obtener contraseña origianl a partir de md5.*



La contraseña es: `mykeanureeves`.

Este tipo de vulnerabilidades suele ocurrir por varios motivos. Uno de ellos es que se suele confundir la codificación con el cifrado. El algoritmo de base64 sirve para codificar y no para cifrar, lo que quiere decir que se puede decodificar fácilmente. En el caso del cifrado se requiere de una clave para poder recuperar los contenidos. Por tanto, accediendo al código fuente y entendiendo el proceso de comprobación de la contraseña, esta se puede obtener fácilmente tal y como se ha demostrado.

Score CVSS

CVSS v3.1: CRÍTICO | 9.3 – AV:L/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H

Métrica	Valor	Justificación
Vector de Ataque (AV)	**Local (L)**	Requiere acceso local al dispositivo o al paquete de la app.
Complejidad del Ataque (AC)	**Baja (L)**	Descompilar un APK y leer código fuente es trivial.
Privilegios Requeridos (PR)	**Ninguno (N)**	No se requieren privilegios adicionales una vez accedido al APK.
Interacción del Usuario (UI)	**Ninguna (N)**	No se requiere intervención del usuario.
Ámbito (S)	**Cambio (C)**	El impacto afecta a la propia app y a los sistemas vinculados.
Impacto en Confidencialidad (C)	**Alto (H)**	Se accede a material de credenciales.
Impacto en Integridad (I)	**Alto (H)**	Posible alteración de servicios mediante acceso a cuentas administrativas.

Métrica	Valor	Justificación
Impacto en Disponibilidad (A)	**Alto (H)**	Podría derivar en cambios de contraseña si se abusan las credenciales.

Recomendaciones

La inclusión de información sensible en el código fuente hace que el flujo de autenticación sea inservible ya que se tienen acceso a usuarios de la aplicación y contraseñas.

Esto se debe en parte a lo que ya se ha comentado, el algoritmo de base 64 codifica, pero no cifra. Es muy importante que los desarrolladores entiendan esta diferencia, por lo que se recomienda realizar alguna formación en desarrollo seguro para que el equipo comprenda el impacto en seguridad que pueden tener las malas prácticas de desarrollo como *hadcodear* información sensible en el código, aunque se use base64.

También sería conveniente remodelar la forma en la que se guarda la información de registro, por ejemplo, utilizando bases de datos para almacenar los datos de registro del usuario y así evitar exponer usuarios y contraseñas en el código. Dicha base de datos deberá además guardar las contraseñas utilizando un algoritmo de *hashing* seguro (no MD5) y optar por usar un *salt* aleatorio en vez de la cadena constante `nohashnopass`, ya que además de almacenar de forma segura la contraseña, herramientas como Crackstation no serían capaces de devolver resultados.

Criptografía insuficiente

El uso de claves de longitud demasiado corta y fácilmente predecibles sumado a la falta de un almacenamiento seguro de las mismas (aparecen en el código fuente) supone que se puede descifrar información que en un principio debía de ser confidencial. Es el caso de las rutas de la URL a las que contacta la aplicación para obtener información del servidor.

Adicionalmente, se utilizan funcionalidades inseguras en la configuración del algoritmo de cifrado utilizado como el *padding*.

Evidencias

Al acceder al código fuente de la actividad posterior al inicio de sesión se han identificado dos funciones que contactan con https://uclm.ctfapp.online:8000 para obtener ficheros. En estas funciones se concatena el anterior dominio con el resultado del descifrado de una cadena cifrada en AES con una contraseña (`secretpassword12`)

Imagen 8. *Código de funciones que desencriptan la ruta de la URL con AES.*

```
/* JADX INFO: Access modifiers changed from: private */
public void getFile() throws NoSuchPaddingException, NoSuchAlgorithmException, InvalidKeyException, InvalidAlgorithmParameterException {
    new GetPdfFileAsyncTask("https://uclm.ctfapp.online:8000" + decryptAES("3da92b68f27a7659fe7e674773798aad1b810fb3290c9cc420a2ced28c4660272a361e91c4594a5d4fa768b6a3c85127", "secretpass
}

/* JADX INFO: Access modifiers changed from: private */
public void getFiles() throws NoSuchPaddingException, NoSuchAlgorithmException, InvalidKeyException, InvalidAlgorithmParameterException {
    new GetFilesAsyncTask("https://uclm.ctfapp.online:8000" + decryptAES("a303a1cb8fa7beba29c6aaf51b4f4210", "secretpassword12"), getApplicationContext()).execute(new Void[0]);
}
```

Sin necesidad de entrar en la función `decryptAES()` se puede suponer que el primer argumento es la cadena codificada y el segundo la clave de descifrado.

Sin embargo, en la función se puede obtener información relevante como el modo utilizado (CBC) y el uso del *padding* PKCS5Padding.

Imagen 9. *Código de la función decryptAES().*

```
public static String decryptAES(String str, String str2) throws NoSuchPaddingException, NoSuchAlgorithmException,
    try {
        byte[] bytes = str2.getBytes(StandardCharsets.UTF_8);
        byte[] bArrDecodeHex = Hex.decodeHex(str.toCharArray());
        SecretKeySpec secretKeySpec = new SecretKeySpec(bytes, "AES");
        IvParameterSpec ivParameterSpec = new IvParameterSpec(bytes);
        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        cipher.init(2, secretKeySpec, ivParameterSpec);
        return new String(cipher.doFinal(bArrDecodeHex), StandardCharsets.UTF_8);
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

Este modo junto con el PKCS7Padding supone una vulnerabilidad muy alta ya que se es vulnerable a ataques de *Oracle padding*. Este ataque permite a un atacante desencriptar datos sin conocer la clave utilizada, por lo que no haría ni falta acceder al código fuente.

Imagen 10. *Detección de la vulnerabilidad de padding por MobSF.*

2	The App uses the encryption mode CBC with PKCS5/PKCS7 padding. This configuration is vulnerable to padding oracle attacks.	high	**CWE:** CWE-649: Reliance on Obfuscation or Encryption of Security-Relevant Inputs without Integrity Checking **OWASP Top 10:** M5: Insufficient Cryptography **OWASP MASVS:** MSTG-CRYPTO-3	com/rooted/ctfapp/LandingPageActivity.java	

No obstante, usando la herramienta *online* CyberChef podemos también obtener el texto descifrado de cada función y así poder conocer cuál es el recurso web al que se accede de forma remota.

Imagen 11. *Descifrado de ruta de la función getFile() con AES/CBC en CyberChef*

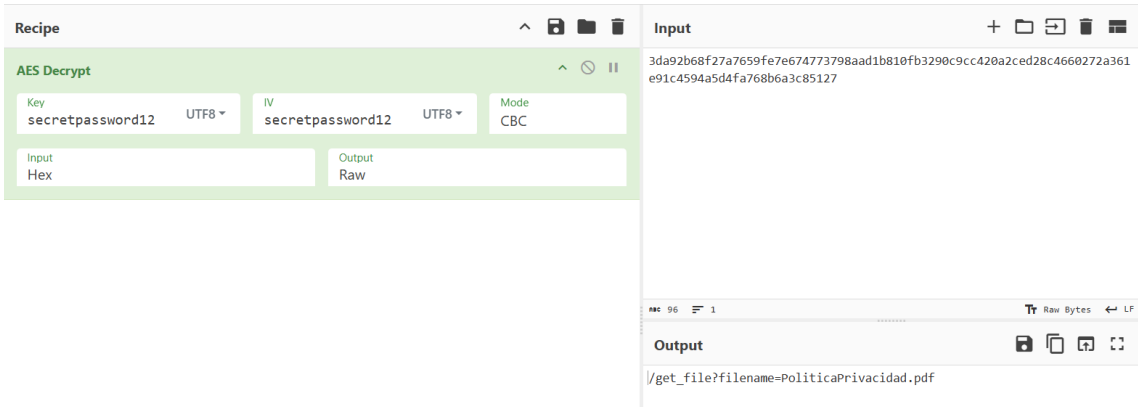
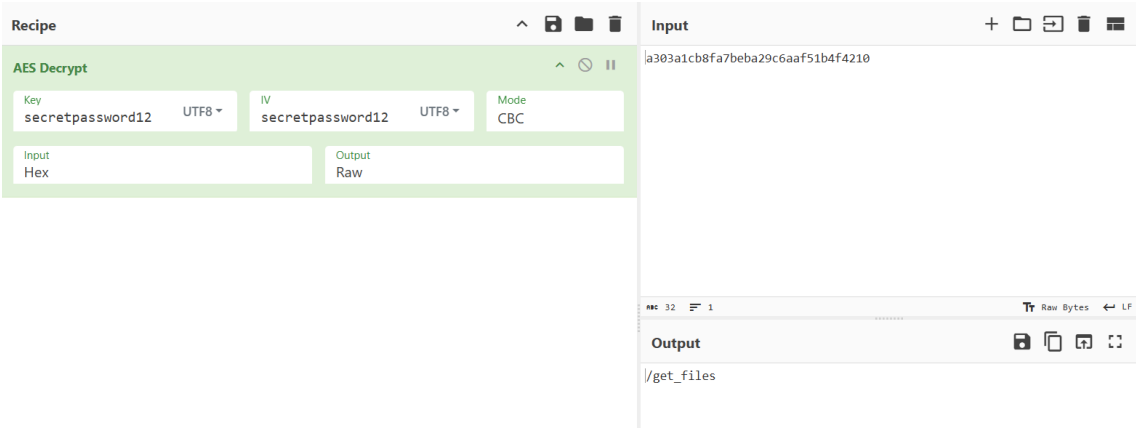


Imagen 12. *Descifrado de ruta de la función getFiles() con AES/CBC en CyberChef*



Score CVSS

CVSS v3.1: ALTO | 7.9 – AV:L/AC:L/PR:N/UI:N/S:C/C:H/I:L/A:N

Métrica	Valor	Justificación
Vector de Ataque (AV)	**Local (L)**	Requiere acceso local al dispositivo o al paquete de la app.
Complejidad del Ataque (AC)	**Baja (L)**	Descompilar un APK y leer código fuente es trivial.
Privilegios Requeridos (PR)	**Ninguno (N)**	No se requieren privilegios adicionales una vez accedido al APK.
Interacción del Usuario (UI)	**Ninguna (N)**	No se requiere intervención del usuario.
Ámbito (S)	**Cambio (C)**	El impacto afecta a la propia app y a los sistemas vinculados.
Impacto en Confidencialidad (C)	**Alto (H)**	Se accede a material confidencial.
Impacto en Integridad (I)	**Bajo (L)**	Se podría modificar la URL contactada.
Impacto en Disponibilidad (A)	**Ninguno (N)**	No hay relación directa que pueda afectar a la disponibilidad del servidor.

Recomendaciones

Para evitar exponer secretos y mejorar el uso de criptografía para proteger la información se recomienda el uso de herramientas como Vault de Hashicorp que permiten una gestión segura y centralizada de secretos, tokens,

direcciones de APIs, certificados y más, basándose en las mejores prácticas y siendo una solución contrastado por el sector.

También se desaconseja totalmente el uso de PKCS5Padding y PKCS7Padding debido a su alto impacto y se recomienda su sustitución.

En esta categoría también entrarían el uso de algoritmos de *hashing* más seguros, evitando MD5 que se vio anteriormente.

Adicionalmente, la mayoría de las recomendaciones sobre material sensible *hadcodeado* del apartado anterior se pueden aplicar en este también.

Filtración de datos (Log inseguro)

En Android los *logs* del sistema son un recurso compartido por todas las apps lo cual implica que cualquier aplicación con el permiso de `READ_LOGS` puede leer los mismos pudiendo derivar en una fuga de datos sensibles del usuario de manera inadvertida si no se controla la información que se registra.

Evidencias

En el código de la aplicación se ha detectado el uso de las funciones de log de Android tanto para *debug* (`Log.d`) como para errores (`Log.e`). Estas funciones se ejecutan en el método `onPostExecute()` de la clase `PostUserInfoAsyncTask` en `LandingPageActivity`, es decir, justo después de contactar con el servidor para actualizar la información del usuario.

Imagen 13. *Codigo de la función onPostExecute que realiza el logging inseguro.*

```
    /* JADX INFO: Access modifiers changed from: protected */
    @Override // android.os.AsyncTask
    public void onPostExecute(JSONArray jSONArray) {
        if (jSONArray != null) {
            Log.d("ServerResponse", jSONArray.toString());
            Toast.makeText(this.context, jSONArray.toString(), 0).show();
        } else {
            Log.e("ServerResponse", "Error occurred during the network operation");
        }
    }
}
```

Dicho de otra forma, cuando la aplicación recibe la respuesta del servidor registra la misma en los logs del sistema. Esto también ocurre en el resto de las interacciones con el servidor para obtener archivos o información del usuario.

El que esta información se registre en los logs del sistema permitiría potencialmente a otras aplicaciones que puedan leer este registro, entender el comportamiento de la aplicación y más importante aún obtener información sensible. Es el caso del código mostrado anteriormente que registra la información del usuario actualizada en el servidor.

Esta información se puede ver utilizando la herramienta `logcat` de `adb` en un dispositivo.

Imagen 14. *Entrada con información sensible de un usuario en logs del sistema.*



Esto supone un gran problema ya que se esta haciendo accesible datos personales del usuario como su nombre, edad, profesión e imagen de perfil.

A parte de la información sensible de usuarios, tampoco es recomendable ser muy verboso a la hora de registrar errores de la aplicación (como por ejemplo exponer el *stack trace* del error) ya que estos podrían ser estudiados y aprovechados por actores maliciosos para provocar errores deliberados en la aplicación.

Score CVSS

CVSS v3.1: MEDIO | 5.5 – AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N

Métrica	Valor	Justificación
Vector de Ataque (AV)	**Local (L)**	El ataque puede ocurrir desde otra app en el mismo dispositivo con permisos de `READ_LOGS`.
Complejidad del Ataque (AC)	**Baja (L)**	Los logs son fácilmente accesibles si se tiene el permiso o si el dispositivo está rooteado con `logcat`.
Privilegios Requeridos (PR)	**Bajos (L)**	La app atacante necesita `READ_LOGS` o acceso root — no privilegios elevados.
Interacción del Usuario (UI)	**Ninguna (N)**	No se requiere intervención del usuario si se accede con logcat.
Ámbito (S)	**Sin cambio (U)**	El problema se mantiene dentro de la app vulnerable.
Impacto en Confidencialidad (C)	**Alto (H)**	Exposición de datos sensibles.
Impacto en Integridad (I)	**Ninguno (N)**	Se pueden leer, pero no se pueden modificar los logs.
Impacto en Disponibilidad (A)	**Ninguno (N)**	No afecta la disponibilidad del sistema o app.

Recomendaciones

Para mitigar esta vulnerabilidad se recomienda no utilizar las funciones de Log del sistema para registrar información sensible y en caso de ser necesarias minimizar la información expuesta en las mismas.

También se pueden recurrir a herramientas automatizadas que buscan llamadas a estas funciones de manera automatizada como MobSF para identificar posibles fugas de información.