Android Application Audit Report

Author:

Ángel Luis Rodríguez Martín-Palomino

Table of Contents

Executive Summary.....**3**

Technical Summary (Vulnerability Analysis).....**3**

Improper Credential Usage and Hardcoded Information.....3

Evidence.....*3*

CVSS Score.....*6*

Recommendations.....*7*

Insufficient Cryptography.....7

Evidence.....*7*

CVSS Score.....*9*

Recommendations.....*10*

Data Leakage (Insecure Logging).....10

Evidence.....*10*

CVSS Score.....*11*

Recommendations.....*12*

Table of Figures

Figure 1. Main activity of the application.....3
Figure 2. Main activity as seen in the AndroidManifest.xml.....4
Figure 3. Source code of the MainActivity.....4
Figure 4. Code of the attemptLogin() function.....5
Figure 5. Code of the checkPassword() function.....5
Figure 6. Base64 decoding of the password.....5
Figure 7. Using CrackStation to obtain the original password from the MD5 hash.....6
Figure 8. Code of the functions that decrypt the URL path with AES.....8
Figure 9. Code of the decryptAES() function.....8
Figure 10. Detection of the padding vulnerability by MobSF.....8
Figure 11. Decryption of the getFile() function path with AES/CBC in CyberChef 9
Figure 12. Decryption of the getFiles() function path with AES/CBC in CyberChef.....9
Figure 13. Code of the onPostExecute function that performs the insecure logging.....11
Figure 14. Entry containing a user's sensitive information in the system logs..11

Executive Summary

The audited application, `ctfapp`, with package name `com.rooted.ctfapp`, has vulnerabilities of medium, high and even critical impact, such as passwords being "*hardcoded*" into the source code, the insufficient or poor use of cryptography, and the leakage of users' sensitive data through the system logs. These vulnerabilities can jeopardise users' security and privacy and allow malicious actors to gain access to the application's private content by bypassing the authentication processes.

The following sections detail what the vulnerabilities found consist of, show evidence of them along with their CVSS impact, and propose possible improvements and recommendations to mitigate them.

Technical Summary (Vulnerability Analysis)

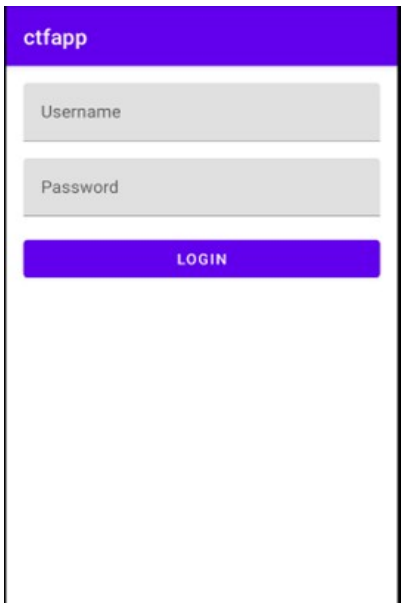
Improper Credential Usage and Hardcoded Information

The application allows any user to log in without knowing any username or password beforehand, simply by obtaining the `.apk` file and accessing the application's source code. This is due to poor credential management, in which credentials are included directly in the code, as are the valid usernames.

Evidence

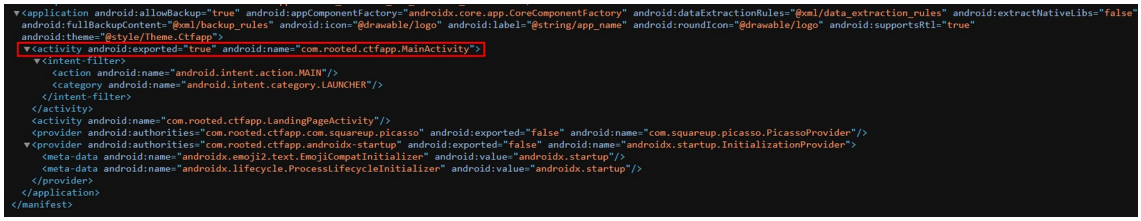
When the application is opened, the main page displayed is the following.

Figure 1. *Main activity of the application.*



Therefore, this must correspond to the main activity `com.rooted.ctfapp.MainActivity` indicated in the `AndroidManifest.xml` file (obtained after running the `apktool d` tool on the `.apk`).

Figure 2. *Main activity as seen in the AndroidManifest.xml*



With access to the `.apk` file, the `jadx-gui` tool can be used on it to obtain the source code in a readable format and look for the main activity there.

Figure 3. *Source code of the 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();
    }
```

As can be seen in the previous figure, when the main activity is created with the `onCreate()` method, an *event listener* is set on the *login* button seen in the interface, so that pressing it executes the function `attemptLogin()`.

In this function, it can be seen that the username entered in the text box is compared against the string `john` or `admin`, and that the password check must succeed. In other words, there are only two users, `john` or `admin`, whose names are written directly in the code in plain text.

Figure 4. *Code of the attemptLogin() function.*

```
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();
}
```

Furthermore, the `checkPassword()` function compares the entered value with another *hardcoded* value in what appears to be Base64, as follows.

Figure 5. *Code of the checkPassword() function.*

```
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;
    }
}
```

The logic is trivial: an MD5 hash is computed over the entered password and the string `nohashnopass` is appended. The result is Base64-encoded and compared with the string `N2JhYzUyYTh1ZDgzNGNiMGQ2NTFmYTE5ZmUzY2RhZTZub2hhc2hub3Bhc3M=`

Using an MD5 hash is already bad practice in itself, since collisions have been discovered, but on top of that, the fact that it is Base64-encoded means it can be easily decoded, as shown in the following figure.

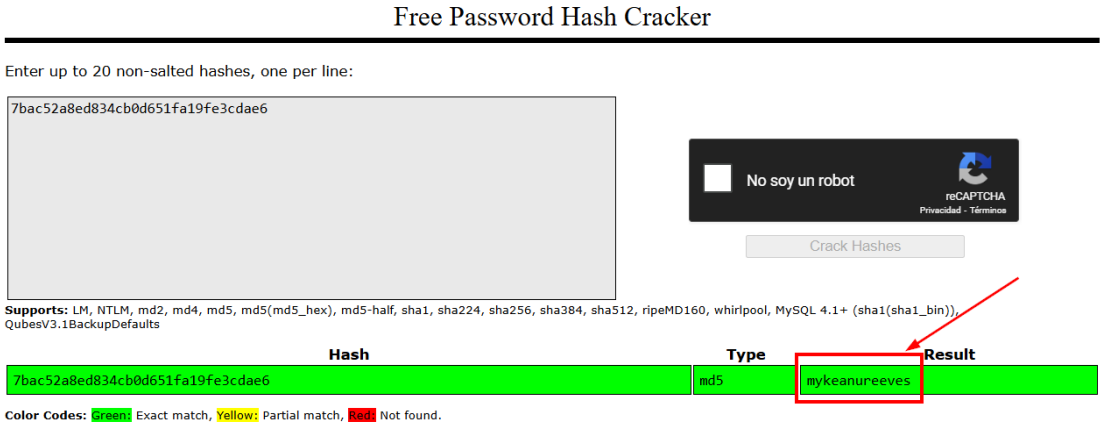
Figure 6. *Base64 decoding of the password.*

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

This yields the string `nohashnopass` together with the MD5 hash of the password (`7bac52a8ed834cb0d651fa19fe3cdae6`), just as seen in the previous code excerpt.

Once this hash is obtained, tools such as *CrackStation*, which hold a table of precomputed passwords for each hash, can be used to obtain the original password value in a matter of seconds.

Figure 7. *Using CrackStation to obtain the original password from the MD5 hash.*



The password is: `mykeanureeves`.

This type of vulnerability usually occurs for several reasons. One of them is that encoding is often confused with encryption. The Base64 algorithm is used for encoding, not for encrypting, which means it can be easily decoded. Encryption, by contrast, requires a key to recover the contents. Therefore, by accessing the source code and understanding the password verification process, the password can be easily obtained, as has been demonstrated.

CVSS Score

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

Metric	Value	Justification
Attack Vector (AV)	**Local (L)**	Requires local access to the device or to the app package.
Attack Complexity (AC)	**Low (L)**	Decompiling an APK and reading the source code is trivial.
Privileges Required (PR)	**None (N)**	No additional privileges are required once the APK has been accessed.
User Interaction (UI)	**None (N)**	No user interaction is required.
Scope (S)	**Changed (C)**	The impact affects the app itself and the linked systems.
Confidentiality Impact (C)	**High (H)**	Credential material is accessed.
Integrity Impact (I)	**High (H)**	Possible tampering with services through access to administrative accounts.
Availability Impact (A)	**High (H)**	Could lead to password changes if the credentials are abused.

Recommendations

Including sensitive information in the source code renders the authentication flow useless, since application usernames and passwords become accessible.

This is partly due to what has already been mentioned: the Base64 algorithm encodes but does not encrypt. It is very important that developers understand this difference, so it is recommended to provide some secure development training so that the team understands the security impact of poor development practices such as *hardcoding* sensitive information in the code, even when Base64 is used.

It would also be advisable to redesign the way registration information is stored, for example, by using databases to store user registration data and thus avoid

exposing usernames and passwords in the code. This database should also store passwords using a secure *hashing* algorithm (not MD5) and use a random *salt* instead of the constant string `nohashnopass`, since, in addition to storing the password securely, tools such as CrackStation would not be able to return results.

Insufficient Cryptography

The use of keys that are too short and easily predictable, combined with the lack of secure storage for them (they appear in the source code), means that information that was originally meant to be confidential can be decrypted. This is the case of the URL paths the application contacts to obtain information from the server.

Additionally, insecure features are used in the configuration of the encryption algorithm, such as the *padding*.

Evidence

When accessing the source code of the activity shown after login, two functions were identified that contact https://uclm.ctfapp.online:8000 to retrieve files. In these functions, the above domain is concatenated with the result of decrypting an AES-encrypted string with a password (`secretpassword12`)

Figure 8. *Code of the functions that decrypt the URL path with 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]);
}
```

Without needing to look inside the `decryptAES()` function, it can be assumed that the first argument is the encoded string and the second the decryption key.

However, relevant information can be obtained from the function, such as the mode used (CBC) and the use of the *padding* scheme PKCS5Padding.

Figure 9. *Code of the decryptAES() function.*

```
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;
    }
}
```

This mode together with PKCS7Padding represents a very high vulnerability, as it is susceptible to *padding oracle* attacks. This attack allows an attacker to decrypt data without knowing the key used, so it would not even be necessary to access the source code.

Figure 10. *Detection of the padding vulnerability by MobSF.*



Nevertheless, using the *online* tool CyberChef, we can also obtain the decrypted text of each function and thereby find out which web resource is accessed remotely.

Figure 11. *Decryption of the getFile() function path with AES/CBC in CyberChef*

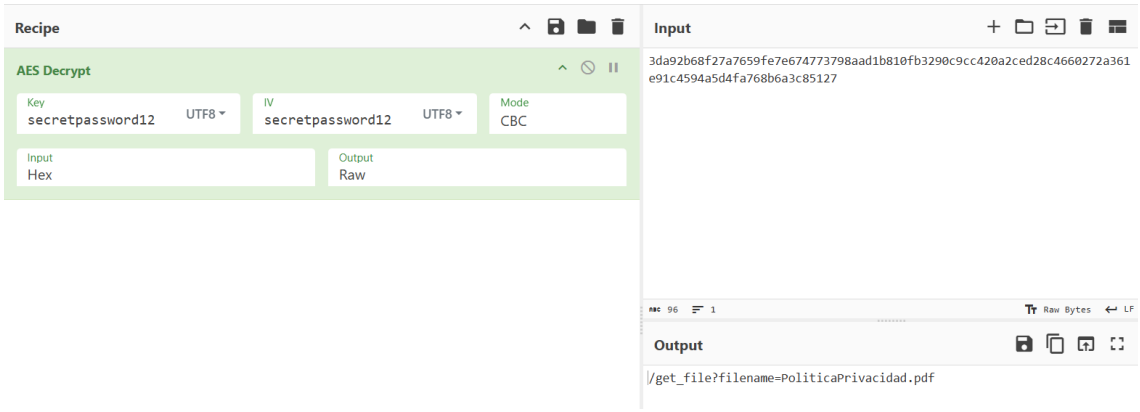
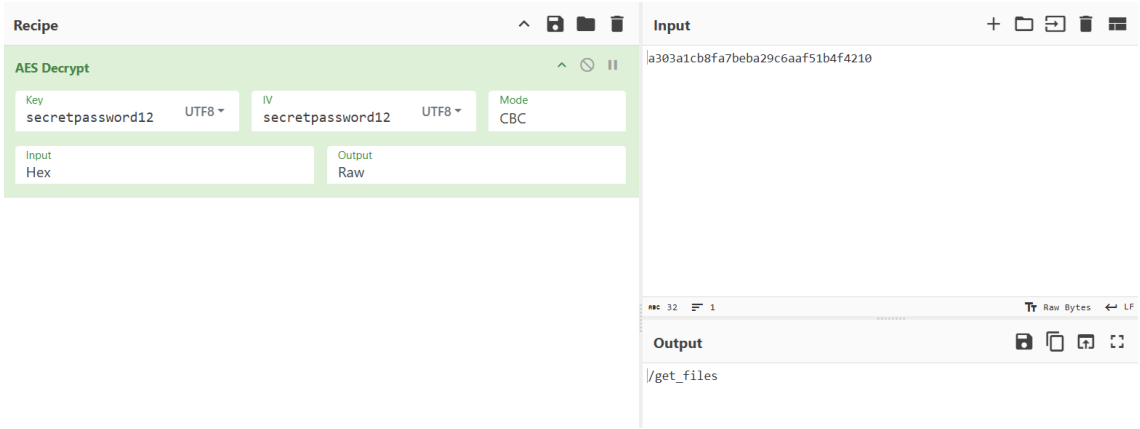


Figure 12. *Decryption of the getFiles() function path with AES/CBC in CyberChef*



CVSS Score

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

Metric	Value	Justification
Attack Vector (AV)	**Local (L)**	Requires local access to the device or to the app package.
Attack Complexity (AC)	**Low (L)**	Decompiling an APK and reading the source code is trivial.
Privileges Required (PR)	**None (N)**	No additional privileges are required once the APK has been accessed.
User Interaction (UI)	**None (N)**	No user interaction is required.

Metric	Value	Justification
Scope (S)	**Changed (C)**	The impact affects the app itself and the linked systems.
Confidentiality Impact (C)	**High (H)**	Confidential material is accessed.
Integrity Impact (I)	**Low (L)**	The contacted URL could be modified.
Availability Impact (A)	**None (N)**	There is no direct link that could affect the server's availability.

Recommendations

To avoid exposing secrets and to improve the use of cryptography to protect information, it is recommended to use tools such as HashiCorp Vault, which enable secure, centralised management of secrets, tokens, API addresses, certificates and more, based on best practices and being an industry-proven solution.

The use of PKCS5Padding and PKCS7Padding is also strongly discouraged due to their high impact, and replacing them is recommended.

This category would also include the use of more secure *hashing* algorithms, avoiding MD5 as seen earlier.

Additionally, most of the recommendations regarding *hardcoded* sensitive material from the previous section can also be applied here.

Data Leakage (Insecure Logging)

In Android, the system *logs* are a resource shared by all apps, which means that any application with the `READ_LOGS` permission can read them, which may inadvertently lead to a leak of the user's sensitive data if the information being logged is not controlled.

Evidence

The use of Android log functions has been detected in the application's code, both for *debugging* (`Log.d`) and for errors (`Log.e`). These functions are executed in the `onPostExecute()` method of the `PostUserInfoAsyncTask` class in `LandingPageActivity`, that is, right after contacting the server to update the user's information.

Figure 13. *Code of the onPostExecute function that performs the insecure logging.*

```
    /* 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");
        }
    }
}
```

In other words, when the application receives the server's response, it writes it to the system logs. This also happens in the rest of the interactions with the server to retrieve files or user information.

Having this information recorded in the system logs would potentially allow other applications that can read this log to understand the application's behaviour and, more importantly, obtain sensitive information. This is the case of the code shown above, which logs the user information updated on the server.

This information can be viewed using the `logcat` tool from `adb` on a device.

Figure 14. *Entry containing a user's sensitive information in the system logs.*



This is a major problem, since the user's personal data, such as their name, age, occupation and profile picture, is being made accessible.

Apart from users' sensitive information, it is also not advisable to be very verbose when logging application errors (for example, exposing the error's *stack trace*), since these could be studied and exploited by malicious actors to deliberately trigger errors in the application.

CVSS Score

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

Metric	Value	Justification
Attack Vector (AV)	**Local (L)**	The attack can originate from another app on the same device holding the permission `READ_LOGS`.
Attack Complexity (AC)	**Low (L)**	Logs are easily accessible if the permission is granted or if the device is rooted, via `logcat`.
Privileges Required	**Low (L)**	The attacking app needs `READ_LOGS`

Metric	Value	Justification
(PR)		or root access — not elevated privileges.
User Interaction (UI)	**None (N)**	No user interaction is required if accessed with logcat.
Scope (S)	**Unchanged (U)**	The issue remains within the vulnerable app.
Confidentiality Impact (C)	**High (H)**	Exposure of sensitive data.
Integrity Impact (I)	**None (N)**	The logs can be read but cannot be modified.
Availability Impact (A)	**None (N)**	Does not affect the availability of the system or app.

Recommendations

To mitigate this vulnerability, it is recommended not to use the system Log functions to record sensitive information and, where they are necessary, to minimise the information exposed in them.

Automated tools that search for calls to these functions, such as MobSF, can also be used to identify potential information leaks.