MARVELDROID WALKTHROUGH

By: 21PC10 - Madumitha G
21PC23 - Renukashree M
21PC11 - Gnanambal M

Power and Space Stones FLAG: Madumitha G
Time and MInd Stones FLAG: Renukashree M
Reality and Soul Stones FLAG: Gnanambal M

MarvelDroid is an Android-based Capture The Flag (CTF) application inspired by the Marvel Universe's Infinity Stones, designed to challenge players' mobile security and reverse-engineering skills. Built with Kotlin and Android SDK, the app features six distinct challenges, each tied to an Infinity Stone and a unique flag in the format InfinityCTF{}.

Here's a list of vulnerabilities associated with each challenge:
Power Stone: Hidden Key in strings.xml (Obfuscation Issue)
Vulnerabilities:
• Static Analysis: Attackers can extract the key from strings.xml using tools like APKTool or jadx.
• Weak Obfuscation: Even if obfuscated, keys can often be reverse-engineered.
• Decompilation: Attackers can inspect res/values/strings.xml and find the hardcoded key.

Space Stone: Fake API Request (Network Log Leak)
Vulnerabilities:
• Logcat Exposure: Flag is leaked in logs, which can be accessed using adb logcat.
• Network Traffic Inspection: Attackers can use Burp Suite or MITMProxy to intercept network logs.
• Lack of Secure Logging: Log.d() statements can expose sensitive data.

Reality Stone: Require Patching the APK
Vulnerabilities:
• Code Modification: Attackers can use Smali Patching to modify the APK and bypass restrictions.
• Frida / Xposed: Dynamic instrumentation can alter runtime behavior.

- Signature Verification Bypass: If not properly implemented, attackers can resign a patched APK and run it.

<u>Soul Stone:</u> Authentication Bypass
Vulnerabilities:

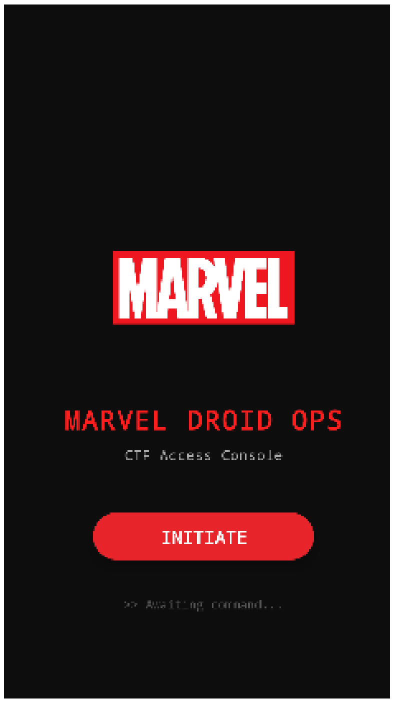
- Hardcoded Credentials: If the app stores login credentials locally, they can be extracted.
- Broken Authentication Logic: Weak or missing validation allows bypassing login screens.
- Token Tampering: If JWT or API tokens are predictable, they can be exploited.
  - Debug Mode Exploitation: If android:debuggable="true", attackers can use adb shell am start to bypass login.

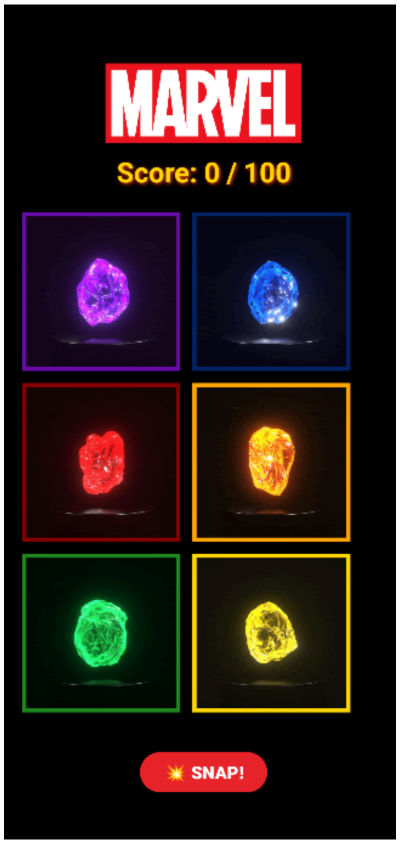
<u>Time Stone:</u> Weak Crypto (Base64 Encoding)
Vulnerabilities:

- Base64 is not encryption: It’s just encoding and can be easily decoded.
- Hardcoded Keys: If weak keys are used (e.g., AES with a static key), attackers can extract them.
- Reversible Encryption: Using weak ciphers (e.g., DES, RC4) makes it easy to brute-force or decrypt.

<u>Mind Stone:</u> Exported Activity Vulnerability
Vulnerabilities:

- Unprotected Exported Activities: If an activity is exported (android:exported="true") and lacks authentication, attackers can start it using adb shell am start.
- Intent Manipulation: Attackers can pass custom intents with malicious data.
- Component Hijacking: If android:exported="true" is set without permission checks, any app can launch it.





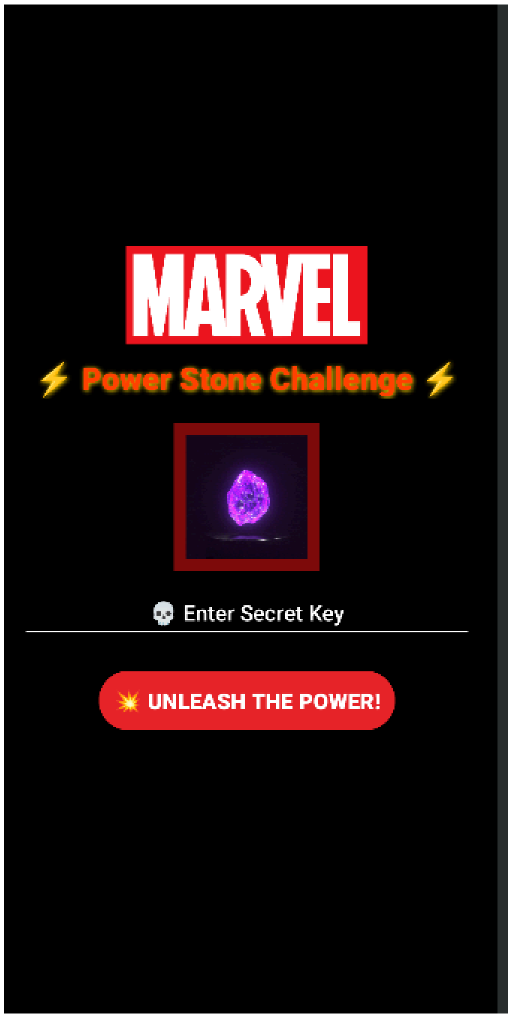
1. **Power Stone Challenge (10 Points)**

Objective: Extract and decode a hidden key stored in strings.xml to get the flag.

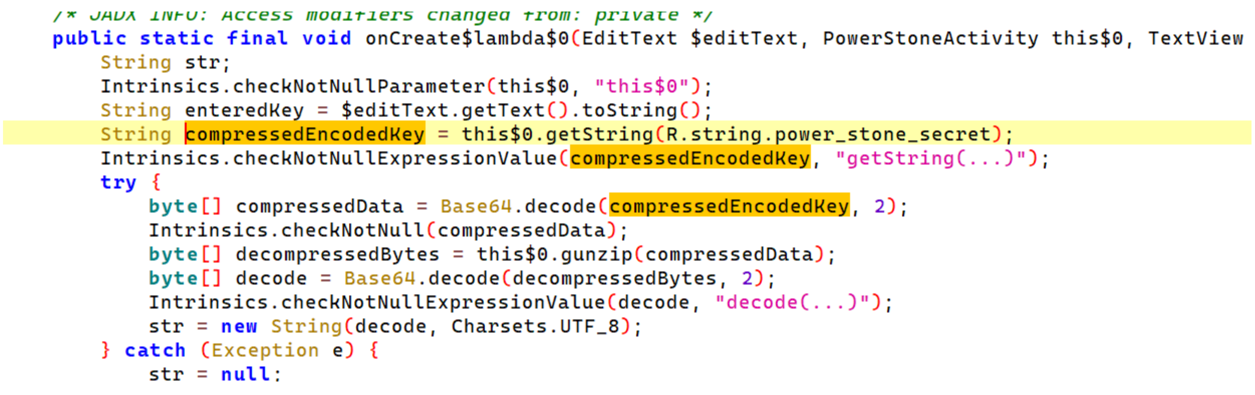
- **Flag Format**: InfinityCTF{power_<6-digit-timestamp>}
- **Points**: 10

Tools Needed

- ADB
- JADX

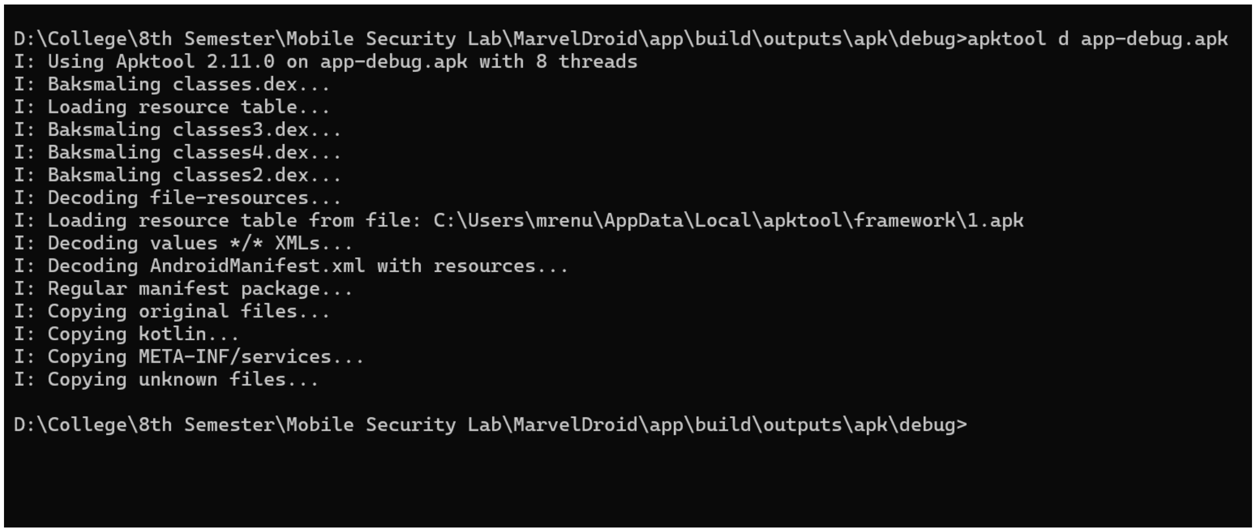


Find the <string> tag with name power_stone_secret. It contains a Base64-encoded, GZIP-compressed, and then Base64-encoded-again value.

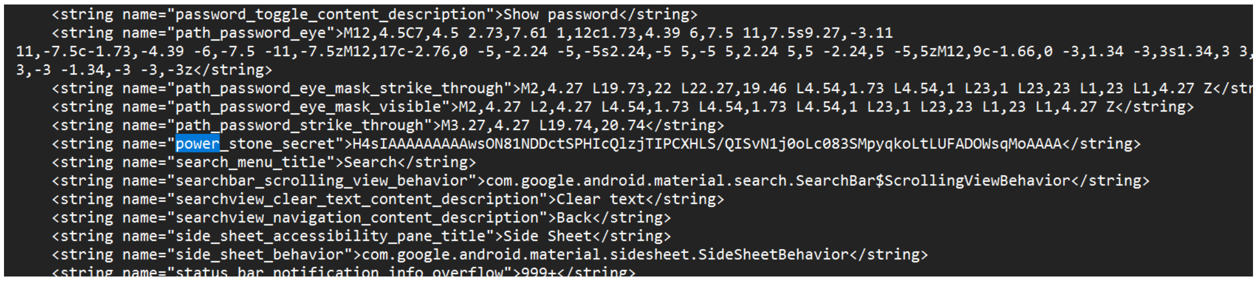


```
/* JADX INFO: Access modifiers changed from: private */
public static final void onCreate$lambda$0(EditText $editText, PowerStoneActivity this$0, TextView
    String str;
    Intrinsics.checkNotNullParameter(this$0, "this$0");
    String enteredKey = $editText.getText().toString();
    String compressedEncodedKey = this$0.getString(R.string.power_stone_secret);
    Intrinsics.checkNotNullExpressionValue(compressedEncodedKey, "getString(...)");
    try {
        byte[] compressedData = Base64.decode(compressedEncodedKey, 2);
        Intrinsics.checkNotNull(compressedData);
        byte[] decompressedBytes = this$0.gunzip(compressedData);
        byte[] decode = Base64.decode(decompressedBytes, 2);
        Intrinsics.checkNotNullExpressionValue(decode, "decode(...)");
        str = new String(decode, Charsets.UTF_8);
    } catch (Exception e) {
        str = null;
```

Use APKTool:
apktool d marveldroid.apk
Navigate to marveldroid/res/values/strings.xml.

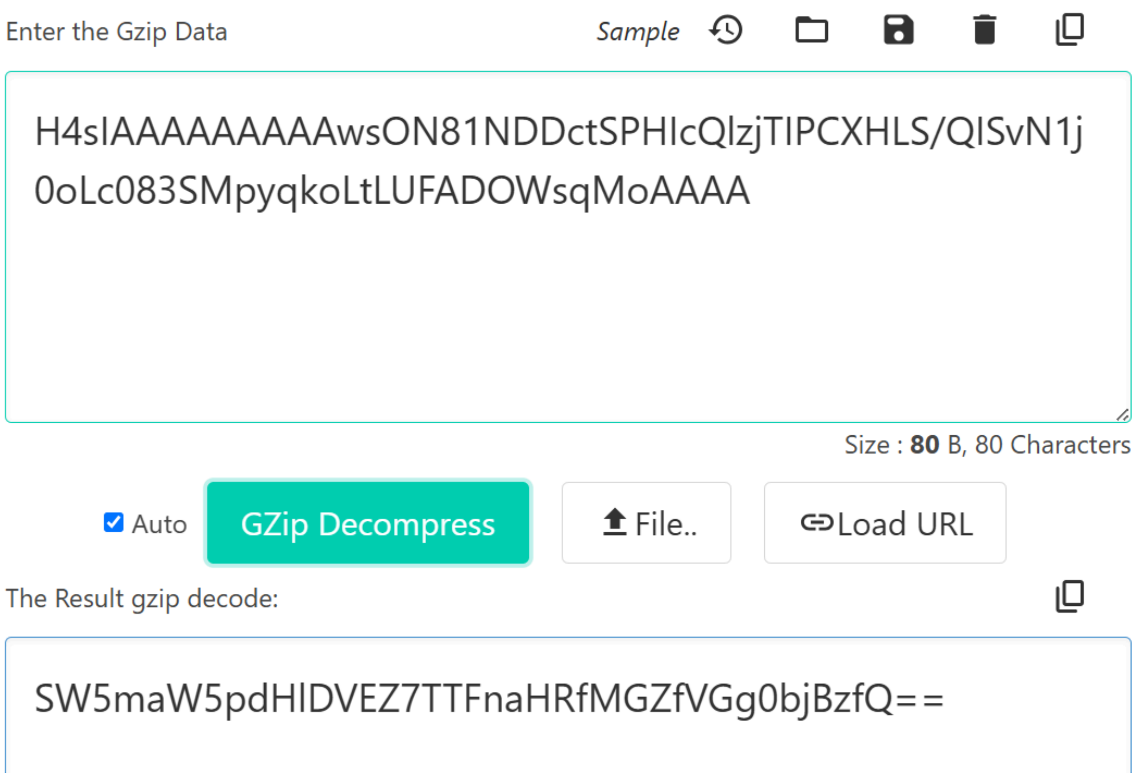


```
D:\College\8th Semester\Mobile Security Lab\MarvelDroid\app\build\outputs\apk\debug>apktool d app-debug.apk
I: Using Apktool 2.11.0 on app-debug.apk with 8 threads
I: Baksmaling classes.dex...
I: Loading resource table...
I: Baksmaling classes3.dex...
I: Baksmaling classes4.dex...
I: Baksmaling classes2.dex...
I: Decoding file-resources...
I: Loading resource table from file: C:\Users\mrenu\AppData\Local\apktool\framework\1.apk
I: Decoding values */* XMLs...
I: Decoding AndroidManifest.xml with resources...
I: Regular manifest package...
I: Copying original files...
I: Copying kotlin...
I: Copying META-INF/services...
I: Copying unknown files...

D:\College\8th Semester\Mobile Security Lab\MarvelDroid\app\build\outputs\apk\debug>
```



```
    <string name="password_toggle_content_description">Show password</string>
    <string name="path_password_eye">M12,4.5C7,4.5 2.73,7.61 1,12c1.73,4.39 6,7.5 11,7.5s9.27,-3.11
11,-7.5c-1.73,-4.39 -6,-7.5 -11,-7.5zM12,17c-2.76,0 -5,-2.24 -5,-5s2.24,-5 5,-5 5,2.24 5,5 -2.24,5 -5,5zM12,9c-1.66,0 -3,1.34 -3,3s1.34,3 3,
3,-3 -1.34,-3 -3,-3z</string>
    <string name="path_password_eye_mask_strike_through">M2,4.27 L19.73,22 L22.27,19.46 L4.54,1.73 L4.54,1 L23,1 L23,23 L1,23 L1,4.27 Z</str
    <string name="path_password_eye_mask_visible">M2,4.27 L2,4.27 L4.54,1.73 L4.54,1.73 L4.54,1 L23,1 L23,23 L1,23 L1,4.27 Z</string>
    <string name="path_password_strike_through">M3.27,4.27 L19.74,20.74</string>
    <string name="power_stone_secret">H4sIAAAAAAAAAwsON81NDDctSPHIcQlzjTIPCXHLS/QISvN1j0oLc083SMpyqkoLtLUFADOWsqMoAAAA</string>
    <string name="search_menu_title">Search</string>
    <string name="searchbar_scrolling_view_behavior">com.google.android.material.search.SearchBar$ScrollingViewBehavior</string>
    <string name="searchview_clear_text_content_description">Clear text</string>
    <string name="searchview_navigation_content_description">Back</string>
    <string name="side_sheet_accessibility_pane_title">Side Sheet</string>
    <string name="side_sheet_behavior">com.google.android.material.sidesheet.SideSheetBehavior</string>
    <string name="status_bar_notification_info_overflow">999+</string>
```

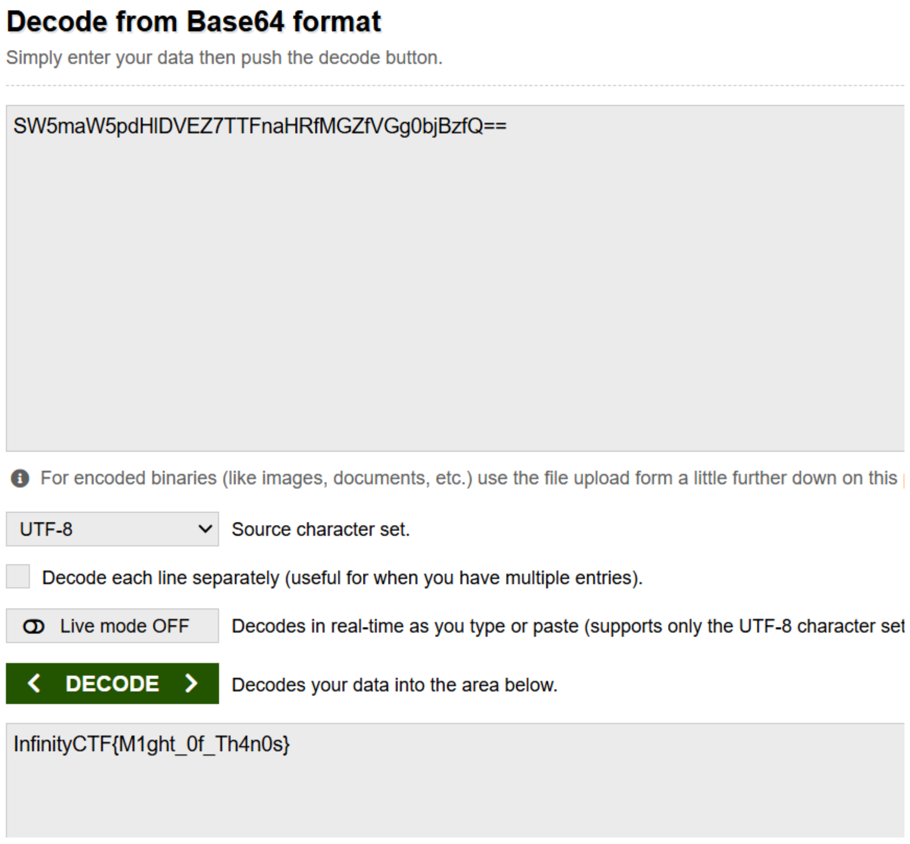
Copy the string and first decompress using GZIP



H4sIAAAAAAAAAwsON81NDDctSPHIcQlzjTIPCXHLS/QISvN1j0oLc083SMpyqkoLtLUFADOWsqMoAAAA

SW5maW5pdHlDVEZ7TTFnaHRfMGZfVGg0bjBzfQ==

Now decode the gzip decompressed string using base64 to get the flag

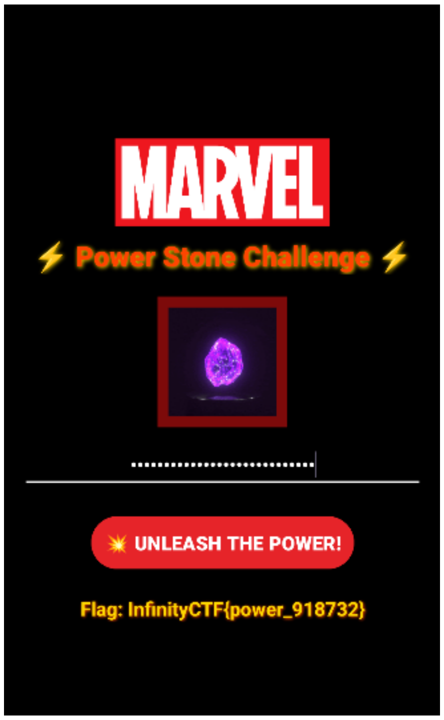

Decode from Base64 format
Simply enter your data then push the decode button.

SW5maW5pdHlDVEZ7TTFnaHRfMGZfVGg0bjBzfQ==

For encoded binaries (like images, documents, etc.) use the file upload form a little further down on this
UTF-8 Source character set.
Decode each line separately (useful for when you have multiple entries).
Live mode OFF Decodes in real-time as you type or paste (supports only the UTF-8 character set
DECODE Decodes your data into the area below.

InfinityCTF{M1ght_0f_Th4n0s}

FLAG : InfinityCTF{M1ght_0f_Th4n0s}



Flag: InfinityCTF{power_918732}

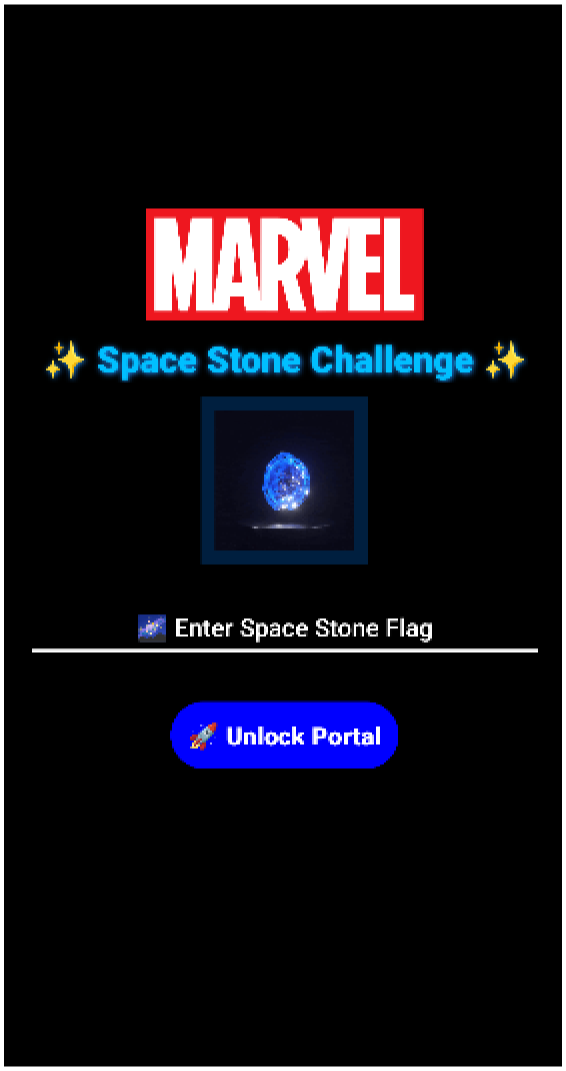
2. Space Stone Challenge (20 Points)

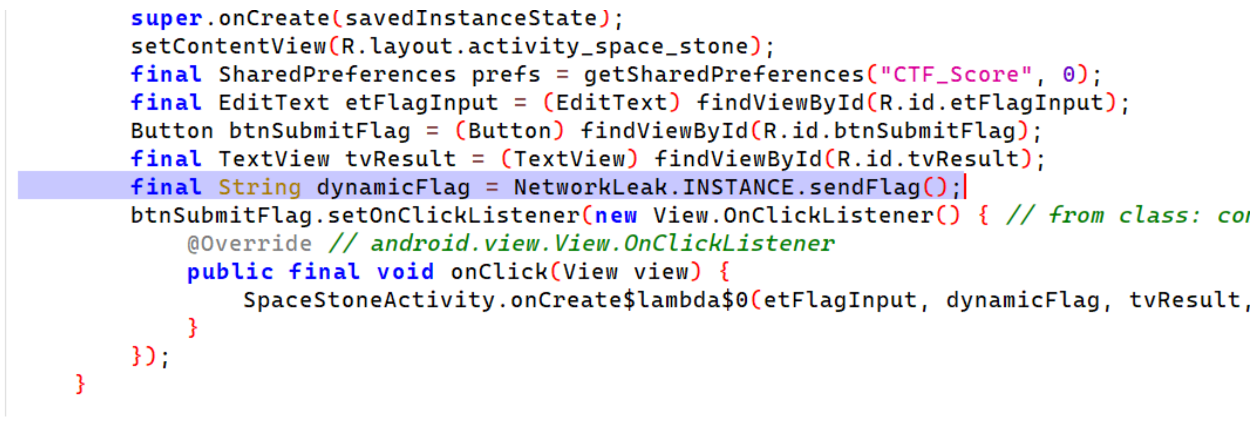
Objective: Capture a flag leaked via a fake API request in network logs.

- **Flag Format**: InfinityCTF{<8-char-UUID>}
- **Points**: 20

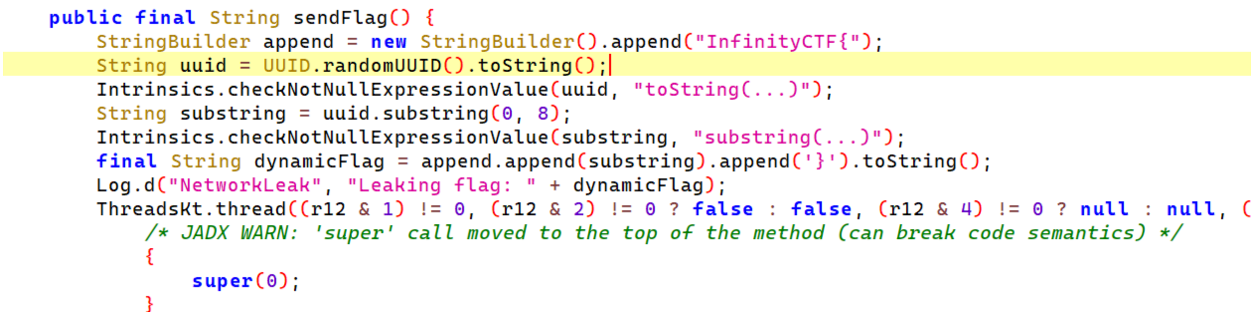
Tools Needed

- ADB (for logcat)




```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_space_stone);
        final SharedPreferences prefs = getSharedPreferences("CTF_Score", 0);
        final EditText etFlagInput = (EditText) findViewById(R.id.etFlagInput);
        Button btnSubmitFlag = (Button) findViewById(R.id.btnSubmitFlag);
        final TextView tvResult = (TextView) findViewById(R.id.tvResult);
        final String dynamicFlag = NetworkLeak.INSTANCE.sendFlag();
        btnSubmitFlag.setOnClickListener(new View.OnClickListener() { // from class: co
            @Override // android.view.View.OnClickListener
            public final void onClick(View view) {
                SpaceStoneActivity.onCreate$lambda$0(etFlagInput, dynamicFlag, tvResult,
            }
        });
    }
```

The dynamic flag is generated using `UUID` every time the activity is started.



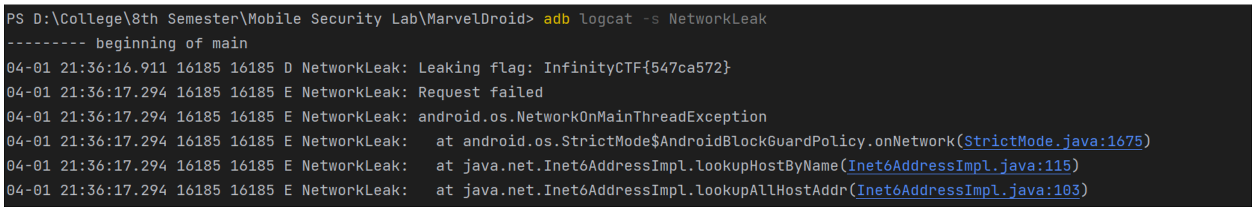
```java
    public final String sendFlag() {
        StringBuilder append = new StringBuilder().append("InfinityCTF{");
        String uuid = UUID.randomUUID().toString();
        Intrinsics.checkNotNullExpressionValue(uuid, "toString(...)");
        String substring = uuid.substring(0, 8);
        Intrinsics.checkNotNullExpressionValue(substring, "substring(...)");
        final String dynamicFlag = append.append(substring).append('}').toString();
        Log.d("NetworkLeak", "Leaking flag: " + dynamicFlag);
        ThreadsKt.thread((r12 & 1) != 0, (r12 & 2) != 0 ? false : false, (r12 & 4) != 0 ? null : null, (
            /* JADX WARN: 'super' call moved to the top of the method (can break code semantics) */
            {
                super(0);
            }
```

The `sendFlag()` function in the `NetworkLeak` object takes care of sending the flag over a network to an external server (in this case, a dummy server like https://eonpdxpb2zf5esp.m.pipedream.net).

Monitor Logs with Logcat:

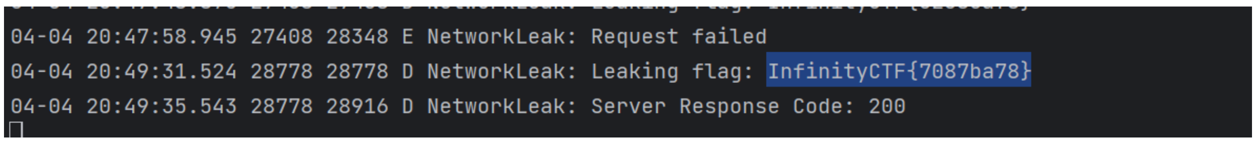
- **Connect the device to your computer and run**

  adb logcat -s NetworkLeak



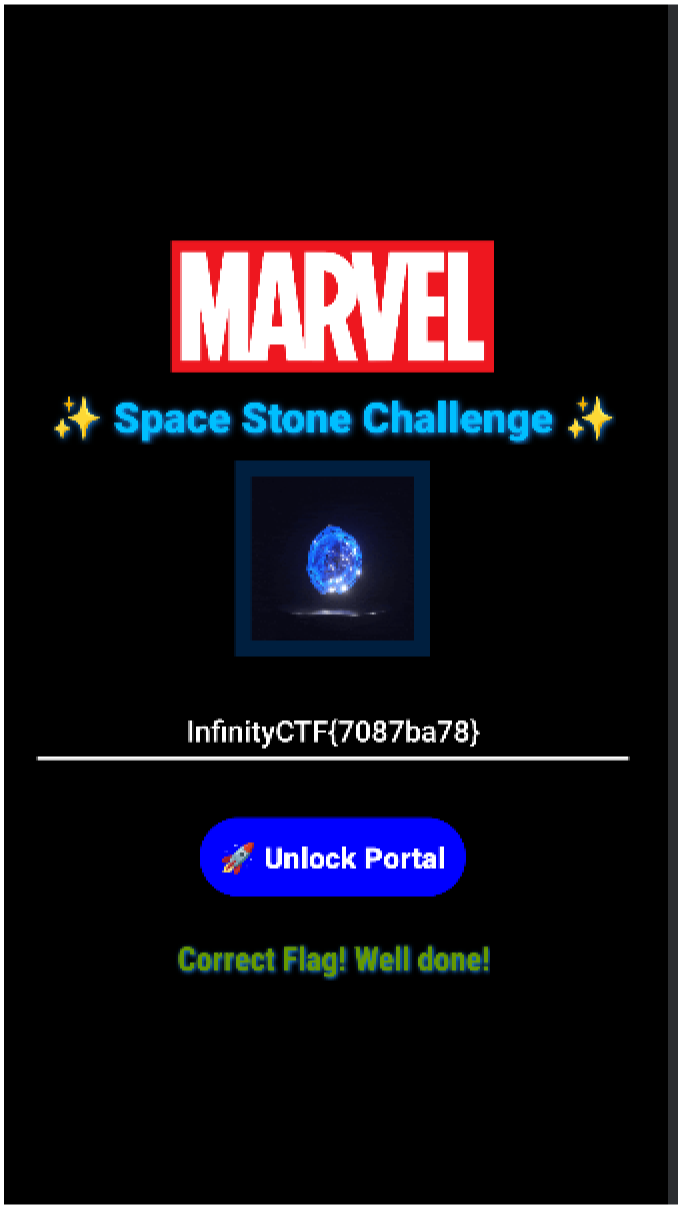
```
PS D:\College\8th Semester\Mobile Security Lab\MarvelDroid> adb logcat -s NetworkLeak
--------- beginning of main
04-01 21:36:16.911 16185 16185 D NetworkLeak: Leaking flag: InfinityCTF{547ca572}
04-01 21:36:17.294 16185 16185 E NetworkLeak: Request failed
04-01 21:36:17.294 16185 16185 E NetworkLeak: android.os.NetworkOnMainThreadException
04-01 21:36:17.294 16185 16185 E NetworkLeak:   at android.os.StrictMode$AndroidBlockGuardPolicy.onNetwork(StrictMode.java:1675)
04-01 21:36:17.294 16185 16185 E NetworkLeak:   at java.net.Inet6AddressImpl.lookupHostByName(Inet6AddressImpl.java:115)
04-01 21:36:17.294 16185 16185 E NetworkLeak:   at java.net.Inet6AddressImpl.lookupAllHostAddr(Inet6AddressImpl.java:103)
```

Look for the flag that gives the response as server response code 200



```
04-04 20:47:58.945 27408 28348 E NetworkLeak: Request failed
04-04 20:49:31.524 28778 28778 D NetworkLeak: Leaking flag: InfinityCTF{7087ba78}
04-04 20:49:35.543 28778 28916 D NetworkLeak: Server Response Code: 200
```

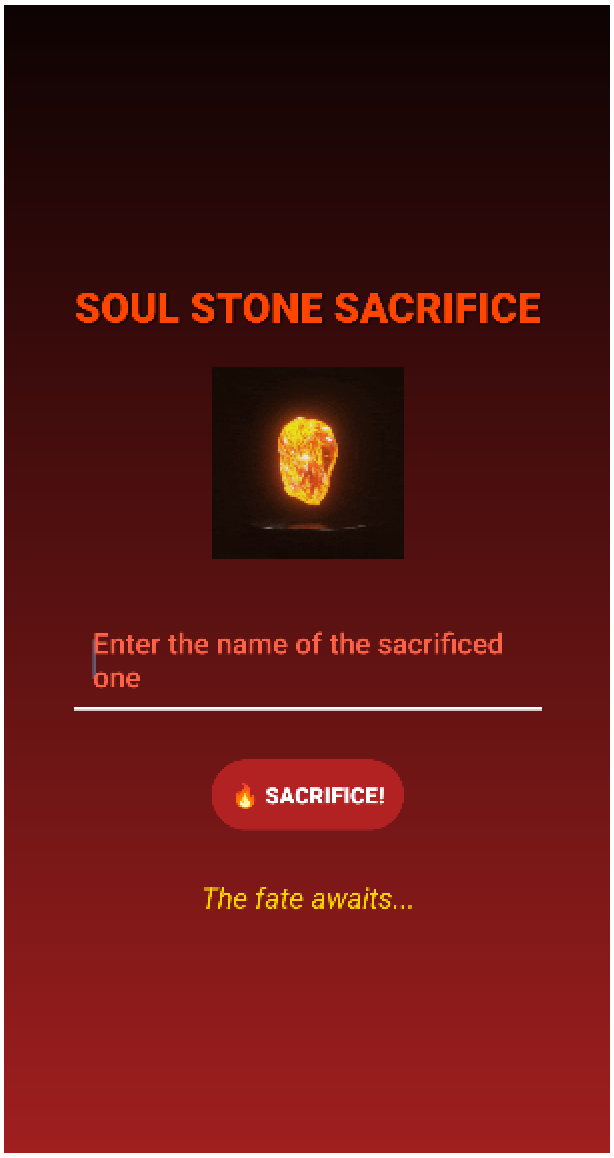
Submit that flag in the console



InfinityCTF{7087ba78}

Correct Flag! Well done!

3. Soul Stone Challenge (15 Points)



Objective: Bypass authentication by activating a backdoor (tap 5 times quickly).

- **Flag Format: InfinityCTF{soul_<6-digit-timestamp>}**
- **Points: 15**

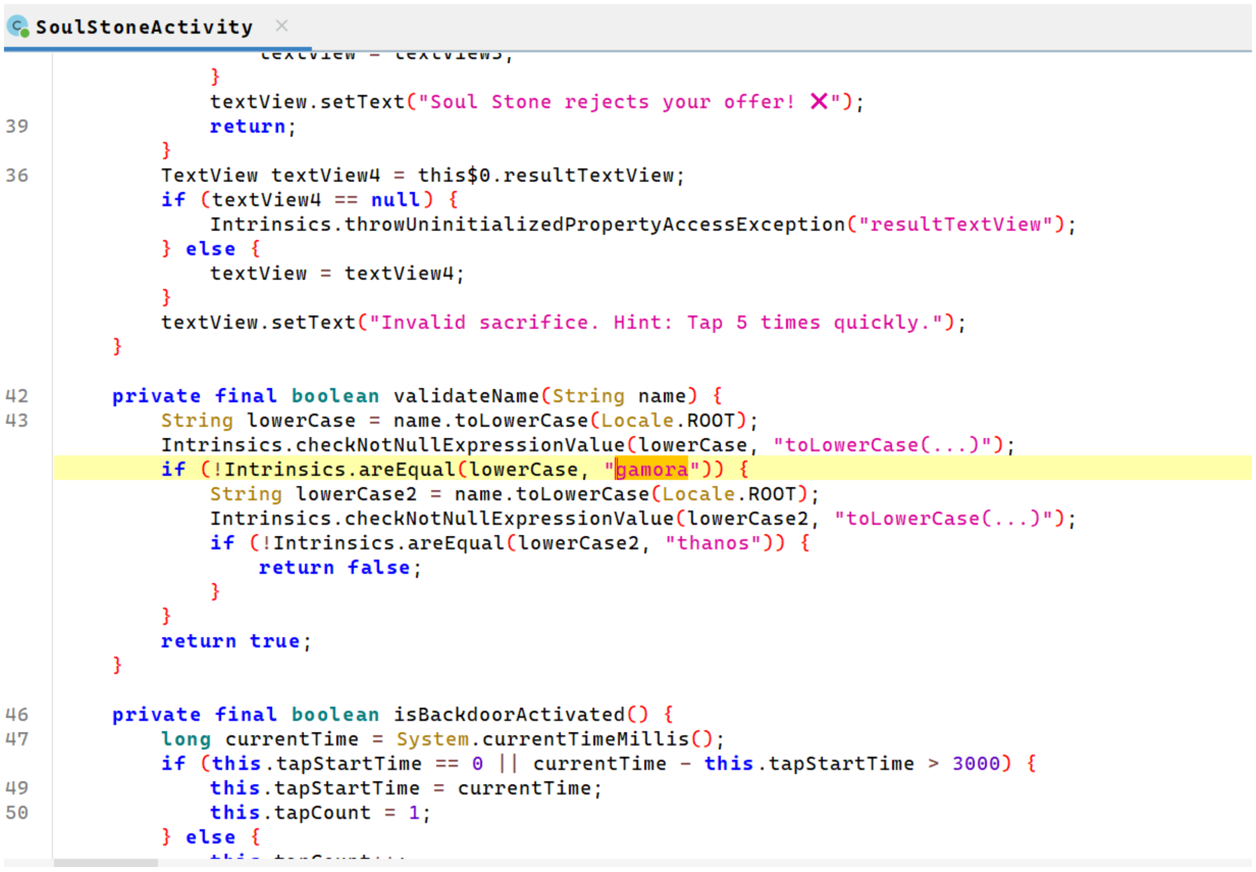
Tools Needed

- **Just the app on a device/emulator.**

Steps to Solve

1. **Open the Challenge:**

- Navigate to the Soul Stone Challenge in the app.

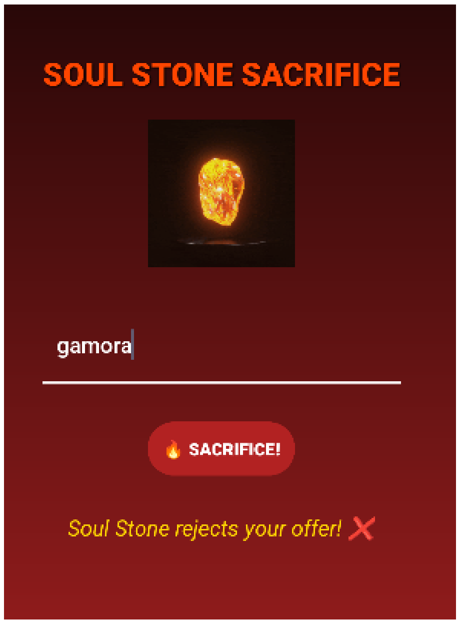


SoulStoneActivity

```
            }
            textView.setText("Soul Stone rejects your offer! ❌");
            return;
        }
        TextView textView4 = this$0.resultTextView;
        if (textView4 == null) {
            Intrinsics.throwUninitializedPropertyAccessException("resultTextView");
        } else {
            textView = textView4;
        }
        textView.setText("Invalid sacrifice. Hint: Tap 5 times quickly.");
    }

    private final boolean validateName(String name) {
        String lowerCase = name.toLowerCase(Locale.ROOT);
        Intrinsics.checkNotNullExpressionValue(lowerCase, "toLowerCase(...)");
        if (!Intrinsics.areEqual(lowerCase, "gamora")) {
            String lowerCase2 = name.toLowerCase(Locale.ROOT);
            Intrinsics.checkNotNullExpressionValue(lowerCase2, "toLowerCase(...)");
            if (!Intrinsics.areEqual(lowerCase2, "thanos")) {
                return false;
            }
        }
        return true;
    }

    private final boolean isBackdoorActivated() {
        long currentTime = System.currentTimeMillis();
        if (this.tapStartTime == 0 || currentTime - this.tapStartTime > 3000) {
            this.tapStartTime = currentTime;
            this.tapCount = 1;
        } else {
```

2. **Activate the Backdoor:**

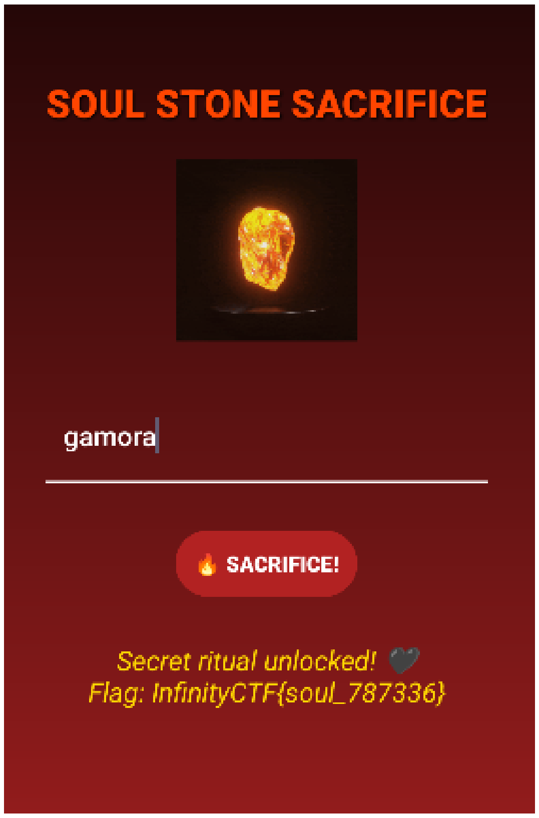
- **Tap the "Sacrifice!" button 5 times within 3 seconds.**



Soul Stone rejects your offer! ❌

-

- **Hint: The app suggests "Tap 5 times quickly" if you enter an invalid name.**

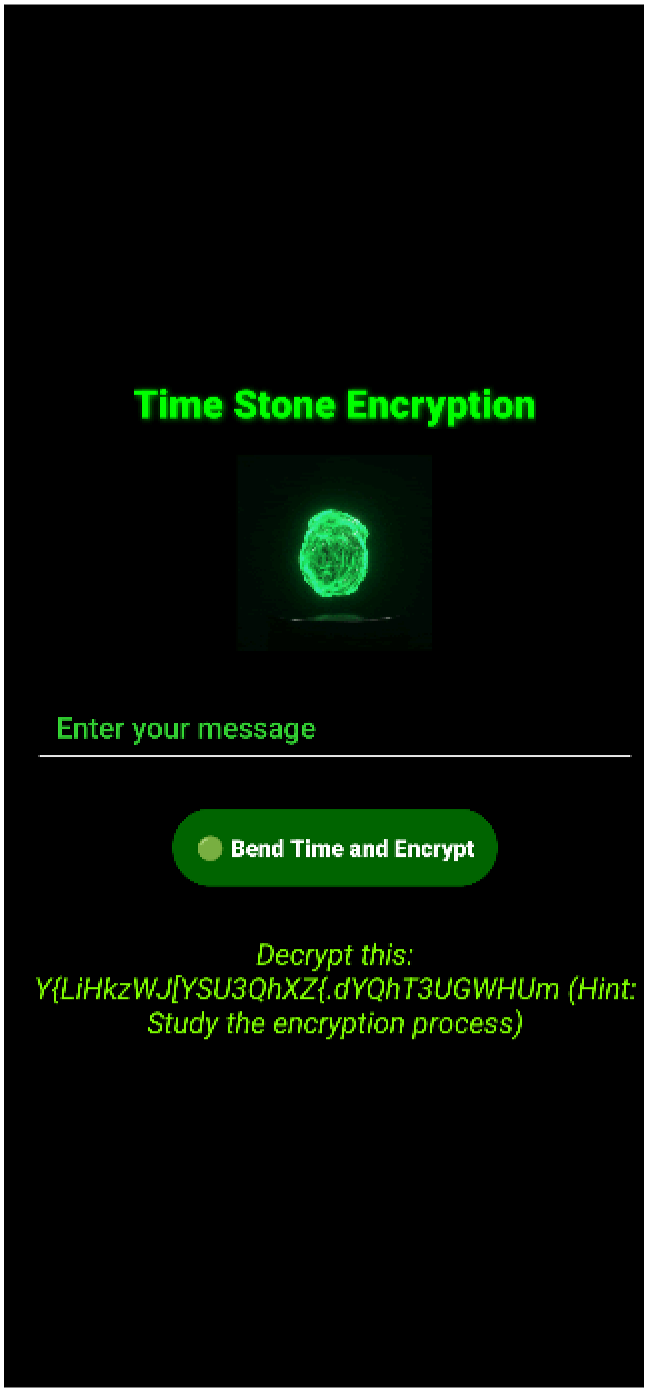


Secret ritual unlocked! 🖤
Flag: InfinityCTF{soul_787336}

3. Get the Flag:

- **After 5 rapid taps, the app displays the flag (e.g., InfinityCTF{soul_789012}) and awards 15 points.**

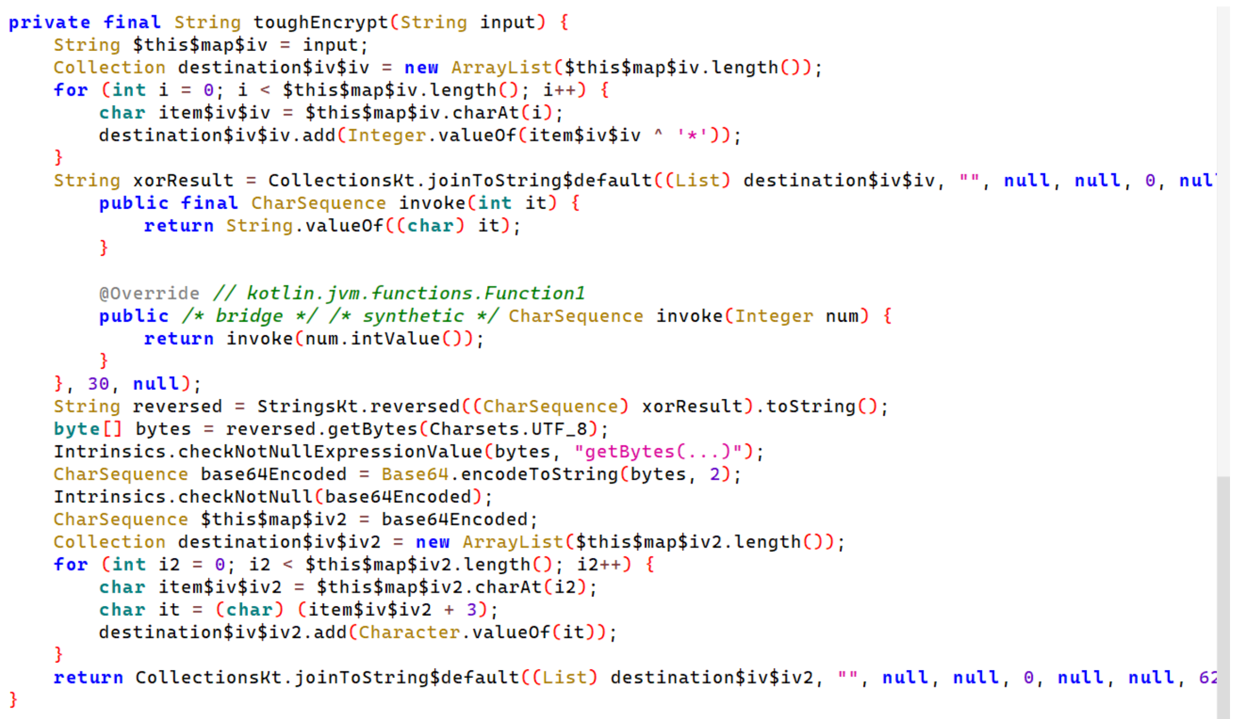
4. Time Stone Activity



Decrypt this: Y{LiHkzWJ[YSU3QhXZ{.dYQhT3UGWHUm (Hint: Study the encryption process)

We get a string that's asked to be decrypted. We also know that the flag is of the format Infinity CTF{}, so that gives the idea about how the string is structured as well.

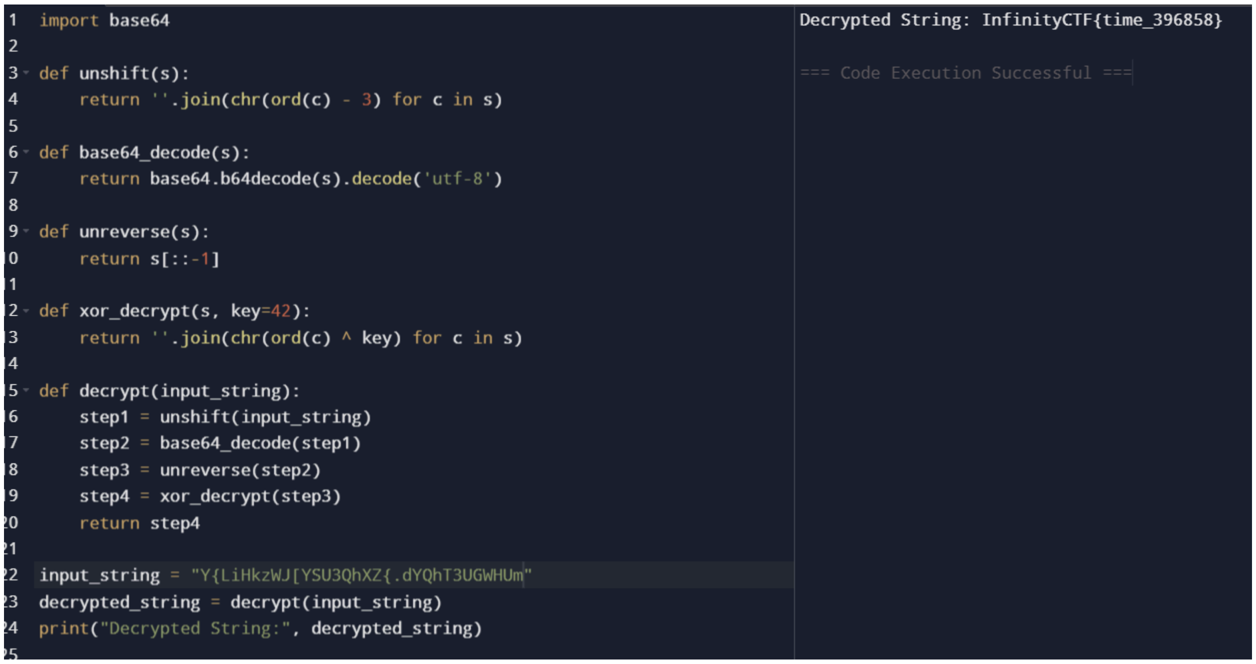
We decompile the APK using JADX and look at the necessary kotlin file.



```
private final String toughEncrypt(String input) {
    String $this$map$iv = input;
    Collection destination$iv$iv = new ArrayList($this$map$iv.length());
    for (int i = 0; i < $this$map$iv.length(); i++) {
        char item$iv$iv = $this$map$iv.charAt(i);
        destination$iv$iv.add(Integer.valueOf(item$iv$iv ^ '*'));
    }
    String xorResult = CollectionsKt.joinToString$default((List) destination$iv$iv, "", null, null, 0, nul
        public final CharSequence invoke(int it) {
            return String.valueOf((char) it);
        }

        @Override // kotlin.jvm.functions.Function1
        public /* bridge */ /* synthetic */ CharSequence invoke(Integer num) {
            return invoke(num.intValue());
        }
    }, 30, null);
    String reversed = StringsKt.reversed((CharSequence) xorResult).toString();
    byte[] bytes = reversed.getBytes(Charsets.UTF_8);
    Intrinsics.checkNotNullExpressionValue(bytes, "getBytes(...)");
    CharSequence base64Encoded = Base64.encodeToString(bytes, 2);
    Intrinsics.checkNotNull(base64Encoded);
    CharSequence $this$map$iv2 = base64Encoded;
    Collection destination$iv$iv2 = new ArrayList($this$map$iv2.length());
    for (int i2 = 0; i2 < $this$map$iv2.length(); i2++) {
        char item$iv$iv2 = $this$map$iv2.charAt(i2);
        char it = (char) (item$iv$iv2 + 3);
        destination$iv$iv2.add(Character.valueOf(it));
    }
    return CollectionsKt.joinToString$default((List) destination$iv$iv2, "", null, null, 0, null, null, 62
}
```

The kotlin file reveals there's a function named toughEncrypt() which takes the input and performs certain operations. The operations include:
Shift each character back by 3.
Base64 decode.
Reverse the string.
XOR each character with 42.
Reversing all these operations will lead us to the flag i.e.
Unshift: Subtract 3 from each character's ASCII value.
Base64 Decode: Decode the result.
Unreverse: Reverse the decoded string.
XOR: XOR each character with 42.

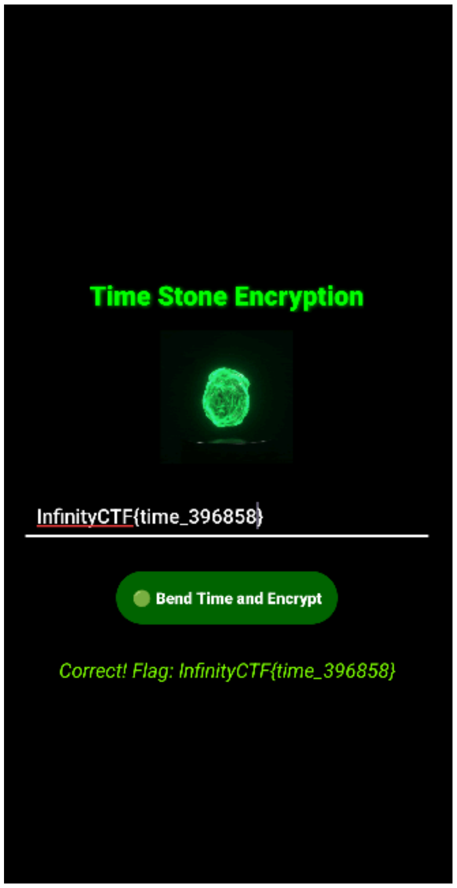
A Python script on that regard can be created undoing these actions to get the flag.



```python
import base64

def unshift(s):
    return ''.join(chr(ord(c) - 3) for c in s)

def base64_decode(s):
    return base64.b64decode(s).decode('utf-8')

def unreverse(s):
    return s[::-1]

def xor_decrypt(s, key=42):
    return ''.join(chr(ord(c) ^ key) for c in s)

def decrypt(input_string):
    step1 = unshift(input_string)
    step2 = base64_decode(step1)
    step3 = unreverse(step2)
    step4 = xor_decrypt(step3)
    return step4

input_string = "Y{LiHkzWJ[YSU3QhXZ{.dYQhT3UGWHUm"
decrypted_string = decrypt(input_string)
print("Decrypted String:", decrypted_string)
```

```
Decrypted String: InfinityCTF{time_396858}

=== Code Execution Successful ===
```

The flag get's solved

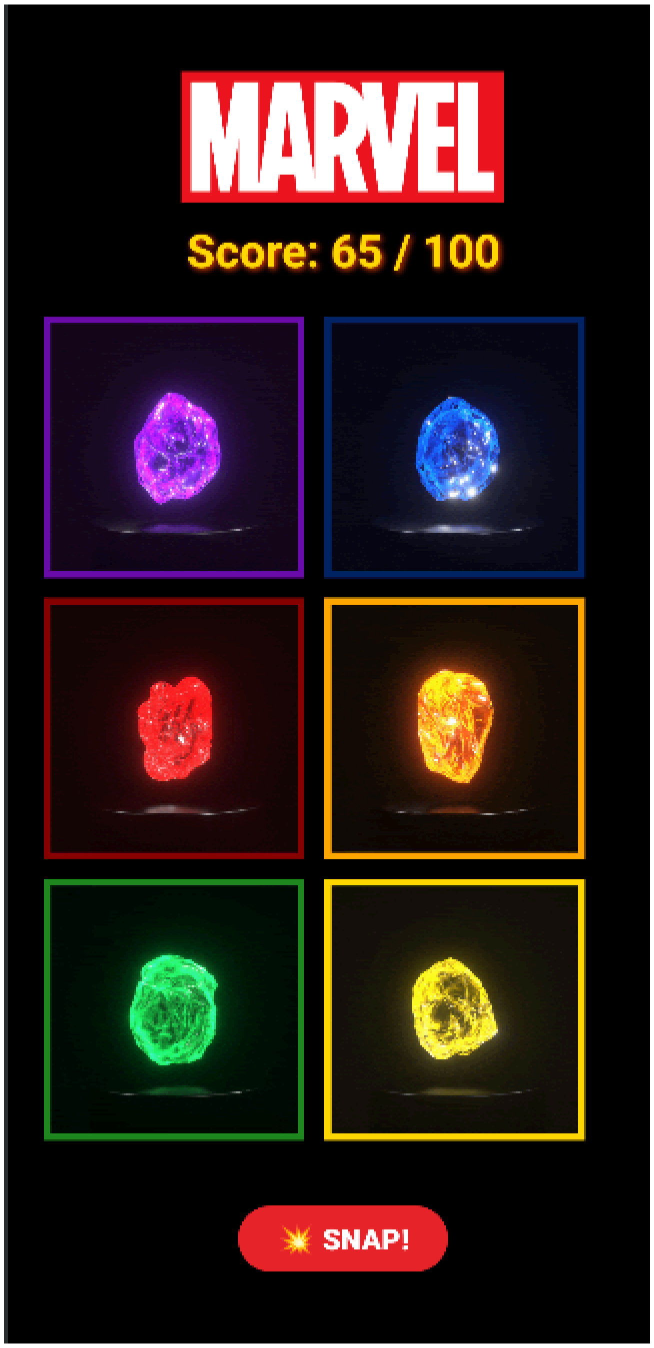


Time Stone Encryption

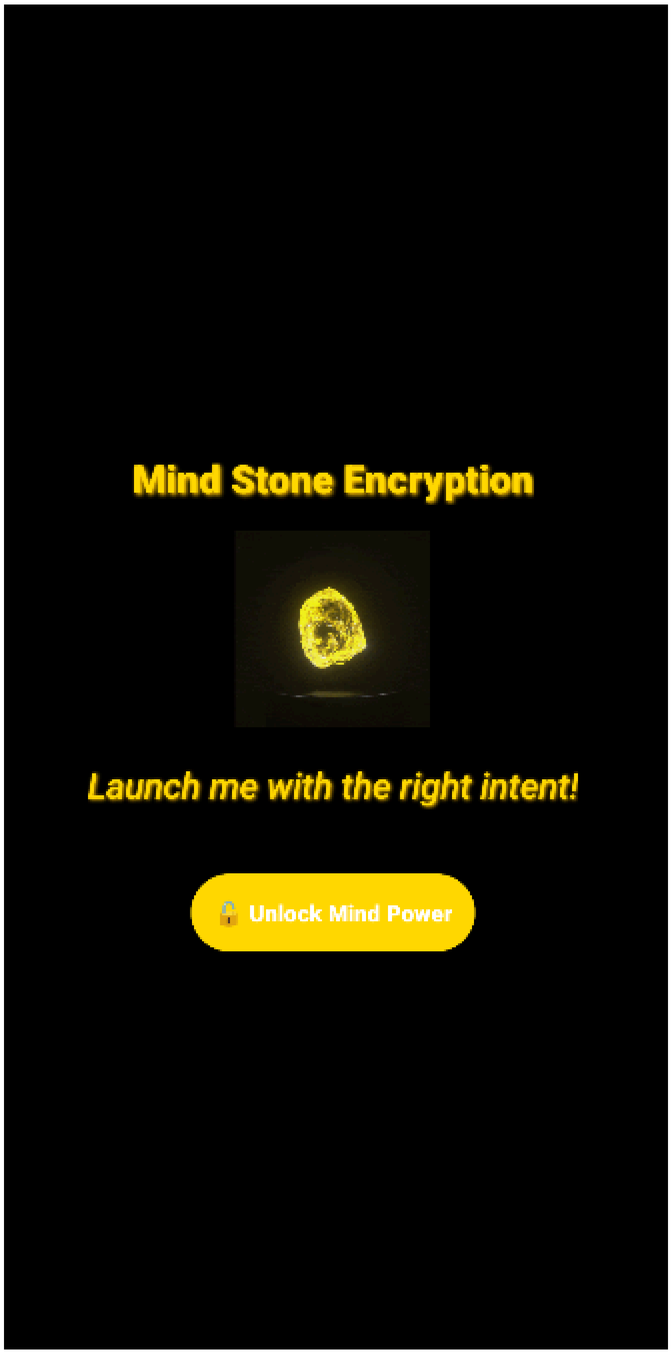
InfinityCTF{time_396858}

Correct! Flag: InfinityCTF{time_396858}

The score also get’s updated thoroughly as

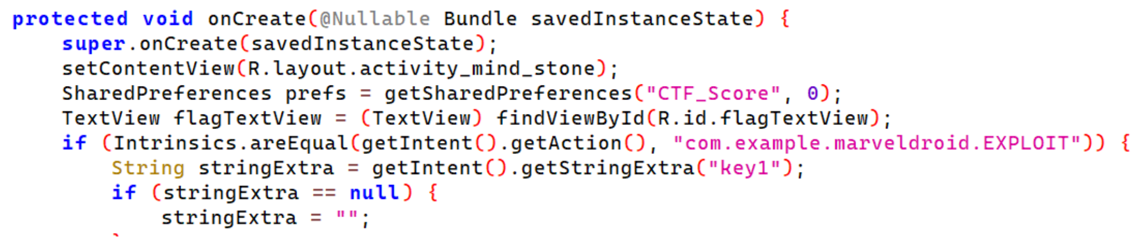


5. MindStoneActivity



Looking at the activity screen gives us the idea that this is related with launching intents.

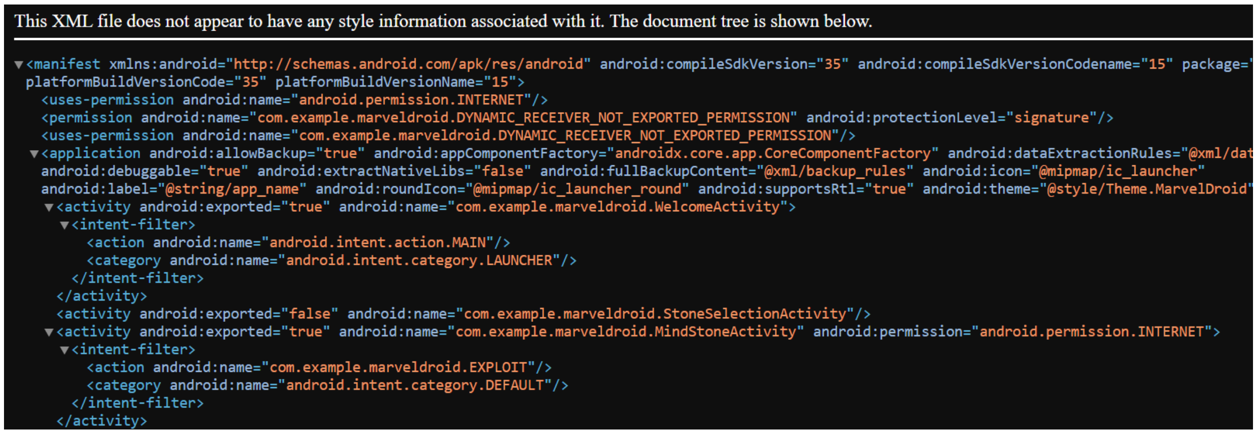
Let’s decompile the apk using jadx and look at the source code file



```
protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_mind_stone);
    SharedPreferences prefs = getSharedPreferences("CTF_Score", 0);
    TextView flagTextView = (TextView) findViewById(R.id.flagTextView);
    if (Intrinsics.areEqual(getIntent().getAction(), "com.example.marveldroid.EXPLOIT")) {
        String stringExtra = getIntent().getStringExtra("key1");
        if (stringExtra == null) {
            stringExtra = "";
```

This line gives us the idea that the activity is invoked with an intent filter com.example.marveldroid.EXPLOIT

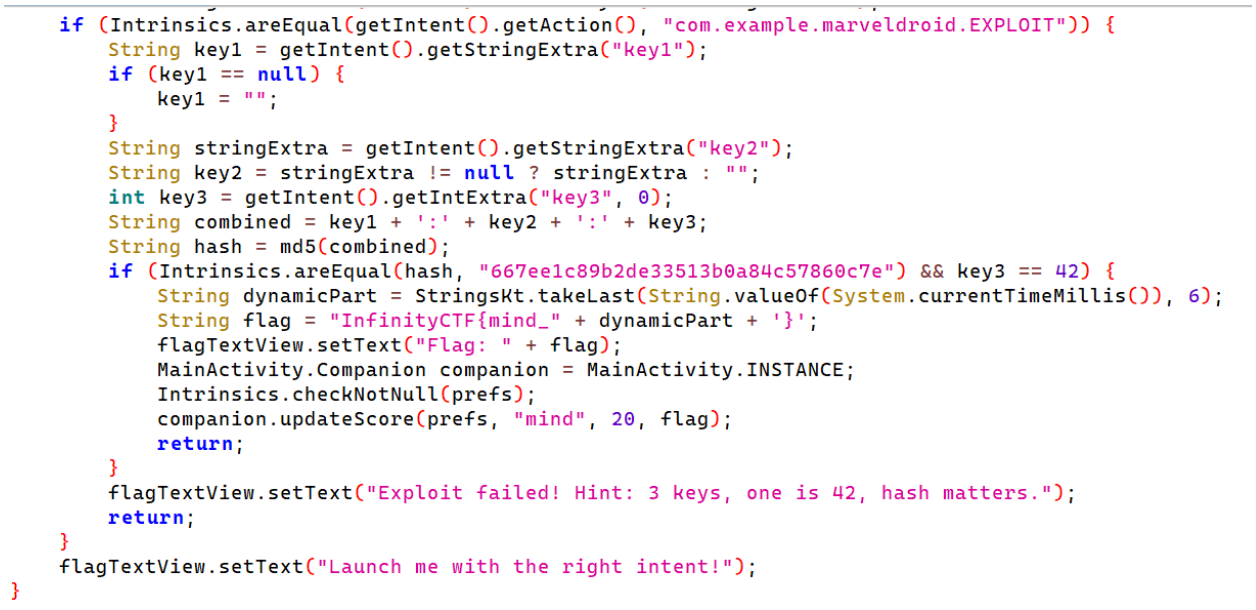
Let's look at the manifest file for the intent filter using apktool.



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<manifest xmlns:android="http://schemas.android.com/apk/res/android" android:compileSdkVersion="35" android:compileSdkVersionCodename="15" package="
 platformBuildVersionCode="35" platformBuildVersionName="15">
   <uses-permission android:name="android.permission.INTERNET"/>
   <permission android:name="com.example.marveldroid.DYNAMIC_RECEIVER_NOT_EXPORTED_PERMISSION" android:protectionLevel="signature"/>
   <uses-permission android:name="com.example.marveldroid.DYNAMIC_RECEIVER_NOT_EXPORTED_PERMISSION"/>
 ▼<application android:allowBackup="true" android:appComponentFactory="androidx.core.app.CoreComponentFactory" android:dataExtractionRules="@xml/dat
   android:debuggable="true" android:extractNativeLibs="false" android:fullBackupContent="@xml/backup_rules" android:icon="@mipmap/ic_launcher"
   android:label="@string/app_name" android:roundIcon="@mipmap/ic_launcher_round" android:supportsRtl="true" android:theme="@style/Theme.MarvelDroid"
  ▼<activity android:exported="true" android:name="com.example.marveldroid.WelcomeActivity">
    ▼<intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
    <activity android:exported="false" android:name="com.example.marveldroid.StoneSelectionActivity"/>
  ▼<activity android:exported="true" android:name="com.example.marveldroid.MindStoneActivity" android:permission="android.permission.INTERNET">
    ▼<intent-filter>
        <action android:name="com.example.marveldroid.EXPLOIT"/>
        <category android:name="android.intent.category.DEFAULT"/>
      </intent-filter>
    </activity>
```

It can be noted that the activity is exported, meaning it can be launched externally with a custom intent.

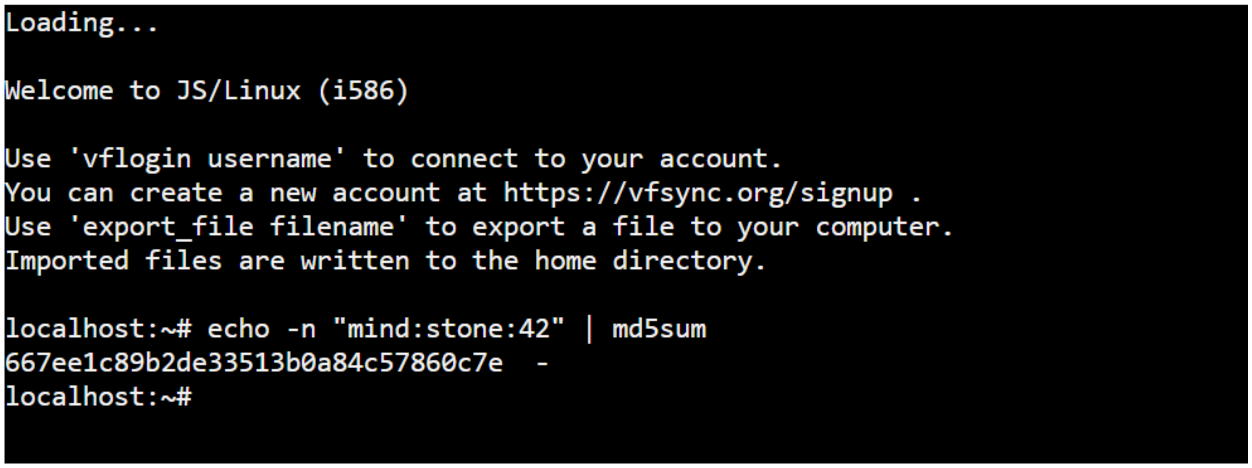


```
    if (Intrinsics.areEqual(getIntent().getAction(), "com.example.marveldroid.EXPLOIT")) {
        String key1 = getIntent().getStringExtra("key1");
        if (key1 == null) {
            key1 = "";
        }
        String stringExtra = getIntent().getStringExtra("key2");
        String key2 = stringExtra != null ? stringExtra : "";
        int key3 = getIntent().getIntExtra("key3", 0);
        String combined = key1 + ':' + key2 + ':' + key3;
        String hash = md5(combined);
        if (Intrinsics.areEqual(hash, "667ee1c89b2de33513b0a84c57860c7e") && key3 == 42) {
            String dynamicPart = StringsKt.takeLast(String.valueOf(System.currentTimeMillis()), 6);
            String flag = "InfinityCTF{mind_" + dynamicPart + '}';
            flagTextView.setText("Flag: " + flag);
            MainActivity.Companion companion = MainActivity.INSTANCE;
            Intrinsics.checkNotNull(prefs);
            companion.updateScore(prefs, "mind", 20, flag);
            return;
        }
        flagTextView.setText("Exploit failed! Hint: 3 keys, one is 42, hash matters.");
        return;
    }
    flagTextView.setText("Launch me with the right intent!");
}
```

From JADX we can view the getIntent().getAction(). Thus we can see that the activity checks for an Intent action(com.example.marveldroid.EXPLOIT) and considers three extras: key1 (string), key2 (string), key3 (int) an MD5 hash of <key1>:<key2>:<key3> must equal 667ee1c89b2de33513b0a84c57860c7e and key3 must be 42.

The condition checks md5("key1:key2:key3") == "667ee1c89b2de33513b0a84c57860c7e" and key3 == 42

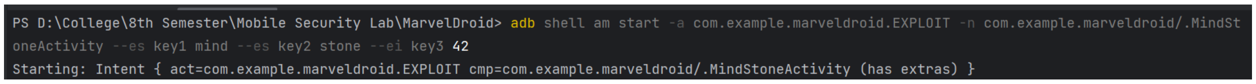
Let's reverse engineer the md5 hash. The code debugged from JADX hints that "mind:stone:42" since key3 == 42 and the challenge is Mind Stone-themed. So let's calculate MD5 hash for this string.



```
Loading...

Welcome to JS/Linux (i586)

Use 'vflogin username' to connect to your account.
You can create a new account at https://vfsync.org/signup .
Use 'export_file filename' to export a file to your computer.
Imported files are written to the home directory.

localhost:~# echo -n "mind:stone:42" | md5sum
667ee1c89b2de33513b0a84c57860c7e  -
localhost:~#
```

This matches the target hash, hence key1 = "mind", key2 = "stone", key3 = 42

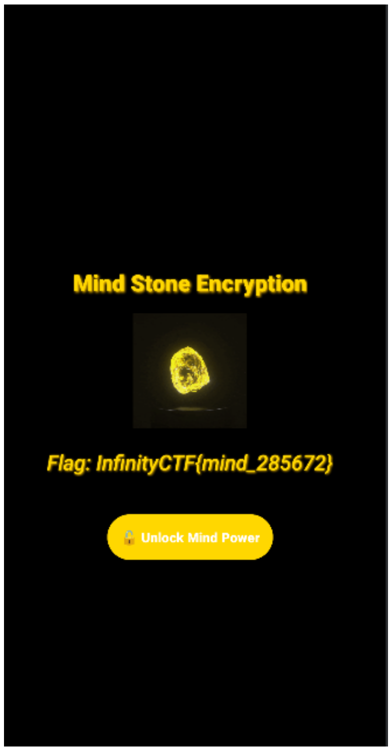
Now let's use adb to send a custom intent



```
PS D:\College\8th Semester\Mobile Security Lab\MarvelDroid> adb shell am start -a com.example.marveldroid.EXPLOIT -n com.example.marveldroid/.MindSt
oneActivity --es key1 mind --es key2 stone --ei key3 42
Starting: Intent { act=com.example.marveldroid.EXPLOIT cmp=com.example.marveldroid/.MindStoneActivity (has extras) }
```

Parameters:

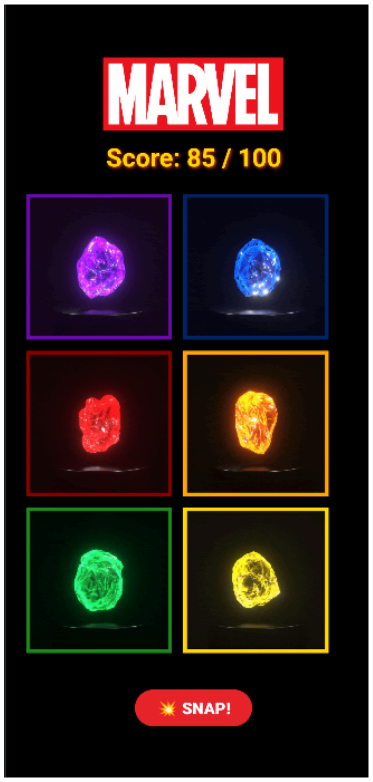
- -a: Action (com.example.marveldroid.EXPLOIT).
- -n: Component name (package + activity).
- --es: Extra string (key1, key2).
- --ei: Extra int (key3).



Mind Stone Encryption

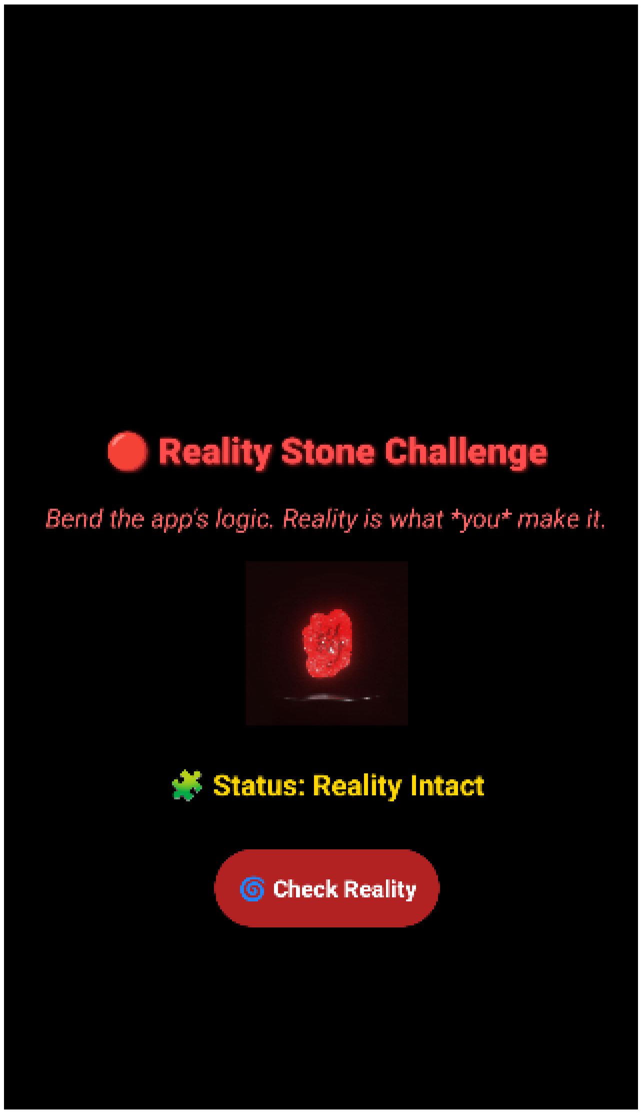
Flag: InfinityCTF{mind_285672}

The score also get's updated accordingly as

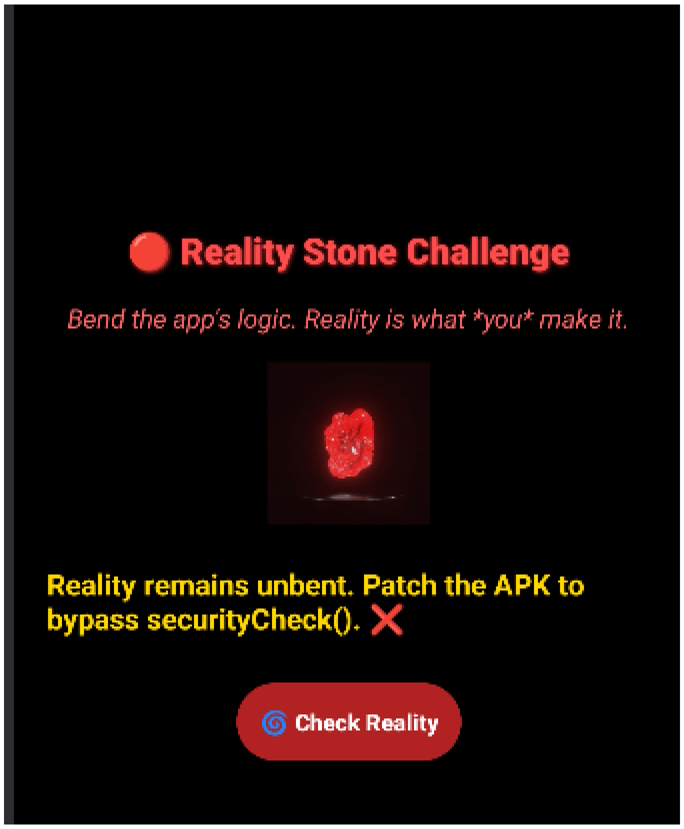


Score: 85 / 100

6. Reality Stone Challenge (15 Points)


🔴 Reality Stone Challenge
Bend the app's logic. Reality is what *you* make it.
🧩 Status: Reality Intact
🌀 Check Reality

This is what the UI is shown and it gives us the clue that we need to bend the app's logic i.e. modify the code.


🔴 Reality Stone Challenge

*Bend the app's logic. Reality is what *you* make it.*

Reality remains unbent. Patch the APK to bypass securityCheck(). ❌

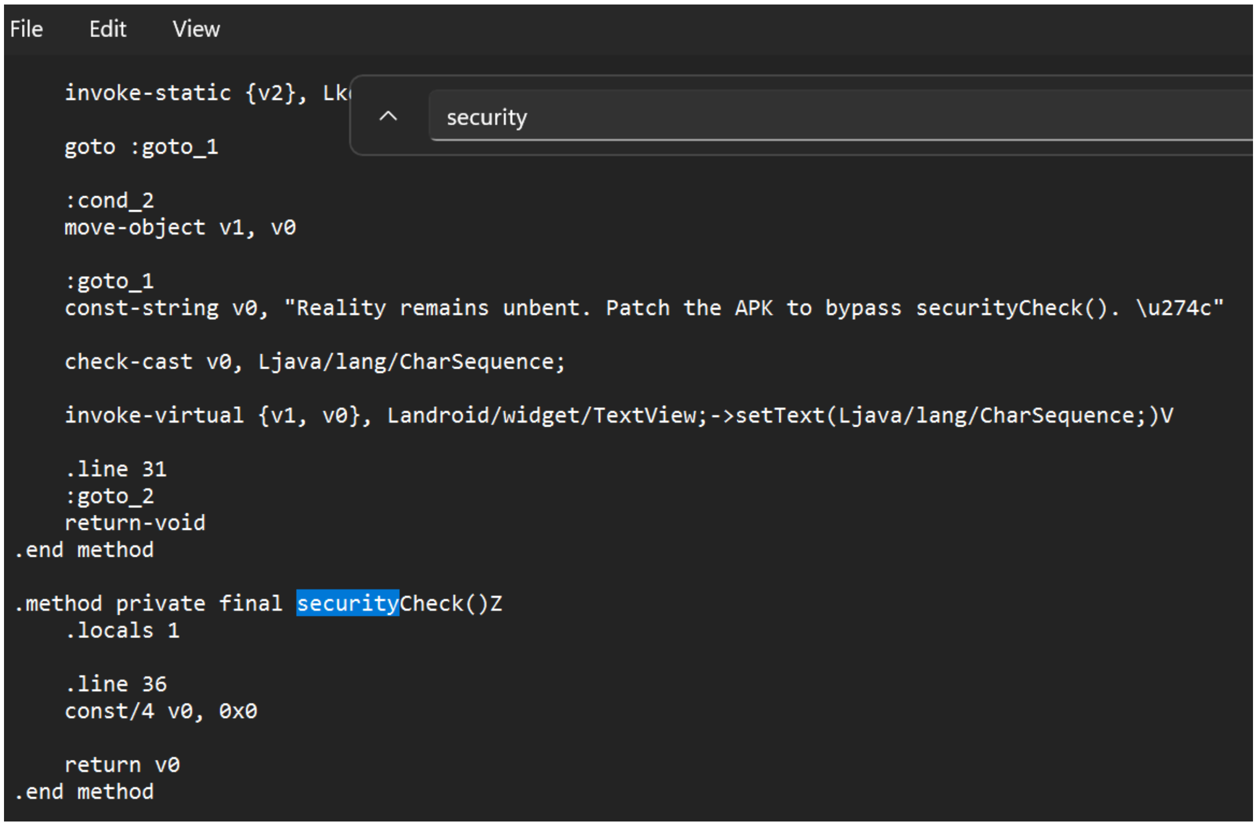
By clicking on the “Check Reality” button, we get a text message saying about patching the APK to bypass securityCheck() which seemingly looks like a function.

Looking at JADX, the function seems to always return False, we need to make it return True.

```
private final boolean securityCheck() {
    return false;
}
```

When you click the "Check Reality" button, it calls securityCheck(). If it returns true, you get the flag; if false, you see "Reality remains unbent."

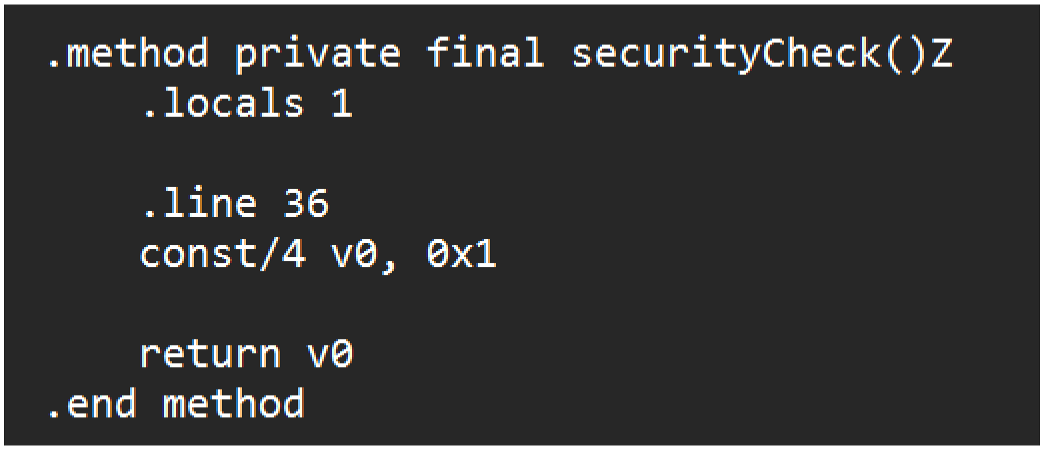
Using apktool to decompile it and looking for the corresponding smali file (RealityStoneActivity.smali), let's look for securityCheck() in it.



File Edit View

security

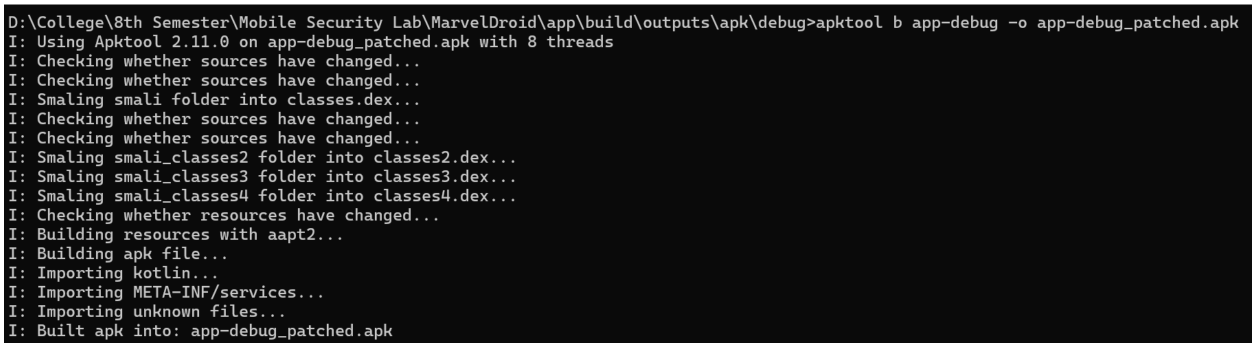
```
    invoke-static {v2}, Lk

    goto :goto_1

    :cond_2
    move-object v1, v0

    :goto_1
    const-string v0, "Reality remains unbent. Patch the APK to bypass securityCheck(). \u274c"

    check-cast v0, Ljava/lang/CharSequence;

    invoke-virtual {v1, v0}, Landroid/widget/TextView;->setText(Ljava/lang/CharSequence;)V

    .line 31
    :goto_2
    return-void
.end method

.method private final securityCheck()Z
    .locals 1

    .line 36
    const/4 v0, 0x0

    return v0
.end method
```



```
.method private final securityCheck()Z
    .locals 1

    .line 36
    const/4 v0, 0x1

    return v0
.end method
```

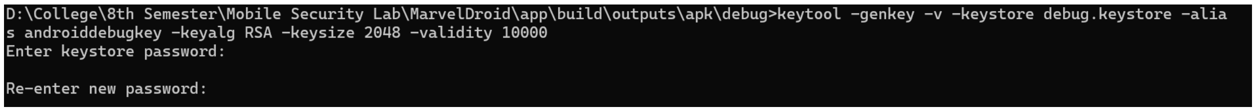
We modify this to always be set to 1 (True).

Now let's rebuild the apk.

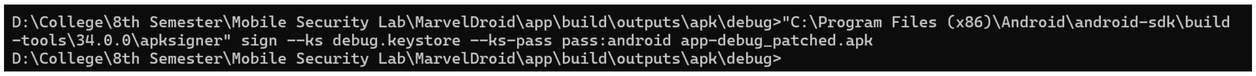


```
D:\College\8th Semester\Mobile Security Lab\MarvelDroid\app\build\outputs\apk\debug>apktool b app-debug -o app-debug_patched.apk
I: Using Apktool 2.11.0 on app-debug_patched.apk with 8 threads
I: Checking whether sources have changed...
I: Checking whether sources have changed...
I: Smaling smali folder into classes.dex...
I: Checking whether sources have changed...
I: Checking whether sources have changed...
I: Smaling smali_classes2 folder into classes2.dex...
I: Smaling smali_classes3 folder into classes3.dex...
I: Smaling smali_classes4 folder into classes4.dex...
I: Checking whether resources have changed...
I: Building resources with aapt2...
I: Building apk file...
I: Importing kotlin...
I: Importing META-INF/services...
I: Importing unknown files...
I: Built apk into: app-debug_patched.apk
```

Let's sign the APK.

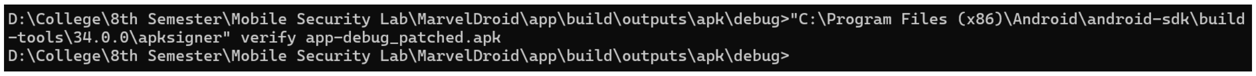


```
D:\College\8th Semester\Mobile Security Lab\MarvelDroid\app\build\outputs\apk\debug>keytool -genkey -v -keystore debug.keystore -alia
s androiddebugkey -keyalg RSA -keysize 2048 -validity 10000
Enter keystore password:

Re-enter new password:
```



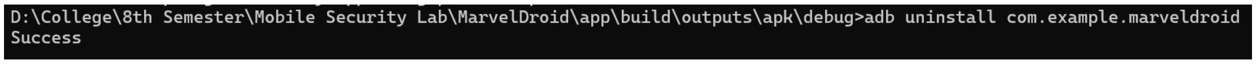
```
D:\College\8th Semester\Mobile Security Lab\MarvelDroid\app\build\outputs\apk\debug>"C:\Program Files (x86)\Android\android-sdk\build
-tools\34.0.0\apksigner" sign --ks debug.keystore --ks-pass pass:android app-debug_patched.apk
D:\College\8th Semester\Mobile Security Lab\MarvelDroid\app\build\outputs\apk\debug>
```

Verifying the signature



```
D:\College\8th Semester\Mobile Security Lab\MarvelDroid\app\build\outputs\apk\debug>"C:\Program Files (x86)\Android\android-sdk\build
-tools\34.0.0\apksigner" verify app-debug_patched.apk
D:\College\8th Semester\Mobile Security Lab\MarvelDroid\app\build\outputs\apk\debug>
```

Let's uninstall the old APK and reinstall the new patched one.

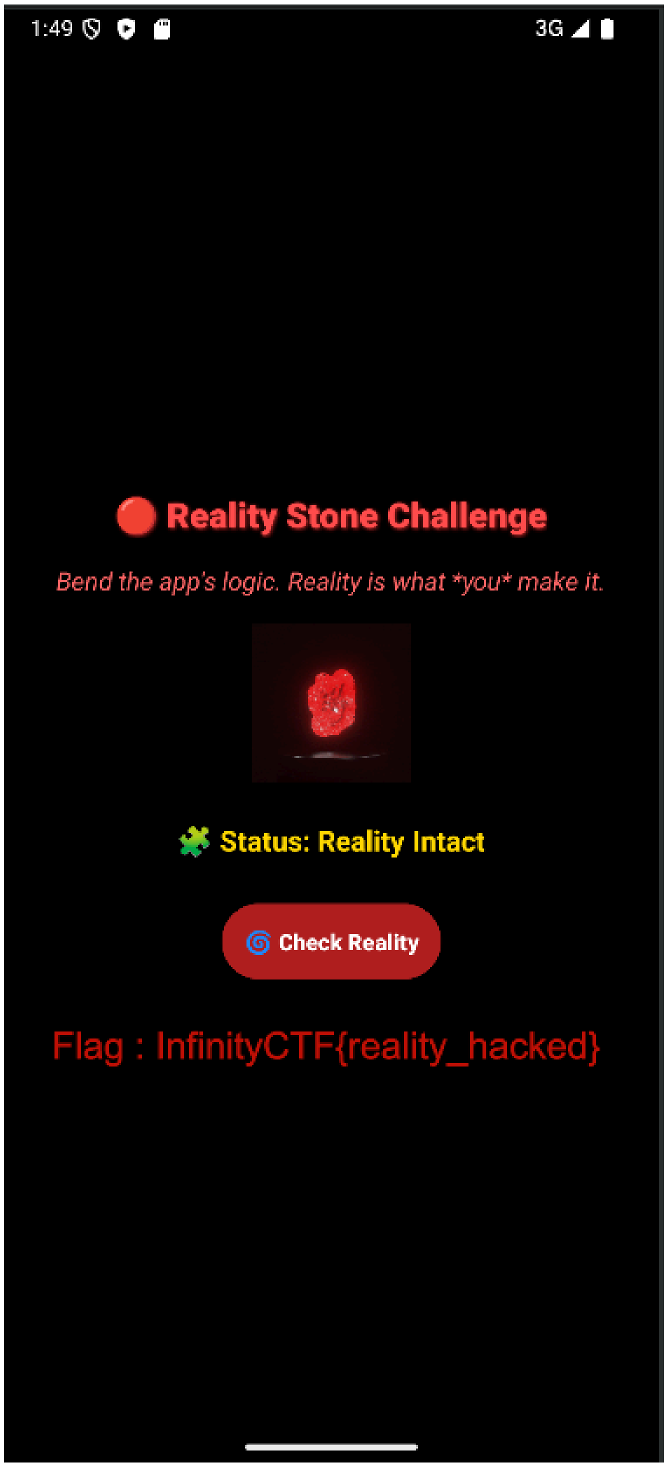


```
D:\College\8th Semester\Mobile Security Lab\MarvelDroid\app\build\outputs\apk\debug>adb uninstall com.example.marveldroid
Success
```

Using Frida to bypass and hook that method and force it to return true.

Starting the app.

Decrypting using python script, we will get the dynamic flag.



1:49 3G

🔴 Reality Stone Challenge

*Bend the app's logic. Reality is what *you* make it.*

🧩 Status: Reality Intact

Flag : InfinityCTF{reality_hacked}