Reading Obfuscated Code

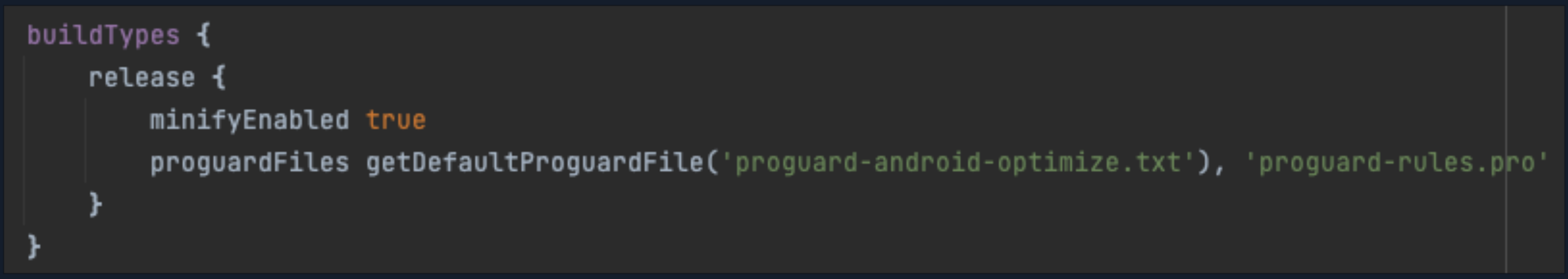
Obfuscation is a process used to transform an app's executable code and assets into a format that is difficult to understand when reverse engineering. While the obfuscated app remains functionally identical to the original, the underlying structure, variable names, method names, and other identifiable elements are changed to enhance the application's security. Among the most common reasons for someone to use obfuscation in their application are:

Reason	Description
`Protect Intellectual Property`	Developers spend significant time and resources building unique features and algorithms. Obfuscating the app could help in safeguarding this intellectual property from competitors and malicious actors.
`Enhance Security`	Apps often contain sensitive information, such as API keys, authentication mechanisms, and proprietary algorithms. By obfuscating the code, the required reverse engineering time could be significantly increased, which eventually reduces any potential security risk.
`Reduce Binary Size`	Some obfuscation tools can shrink the size of the final APK, which can lead to quicker download times and reduced storage requirements.

Even though code obfuscation can introduce maintainability issues and sometimes reduce performance overhead, it is still a preferred solution to enhance the application's security. There are several commercial and open-source tools offering code obfuscation, and below, we can find some of the most common techniques used to achieve this.

Technique	Description
`Name Obfuscation`	This is the most common technique. Descriptive class, method, and variable names are replaced with short, meaningless names, making the code harder to read and understand.
`Control Flow Obfuscation`	Alters the app's control flow by introducing fake loops, bogus switch statements, and unreachable code.
`Repackaging`	Changes the package of a class. This disrupts the logical organization that packages provide. By rearranging existing classes into different or even entirely new package structures, the inherent context provided by a package name is lost, making it harder for someone to understand the overall organization and functionality of the decompiled code.
`String Encryption`	Encrypts string constants used in the app. These strings are decrypted at runtime, making static analysis more challenging.
`Class Encryption`	Encrypts certain classes or methods, decrypting them only when they're needed at runtime.
`Dummy Code Insertion`	Introduces code that doesn't affect app functionality but confuses decompilers and reverse engineers.

The Android ecosystem provides several tools for obfuscation, the most widely used being R8 (introduced in 2018) and ProGuard (its predecessor). Both are integrated by default in Android Studio, and to enable code obfuscation, the value `minifyEnabled` must be `true` under the `Gradle Scripts` -> `build.gradle` file.



```
buildTypes {
    release {
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
    }
}
```

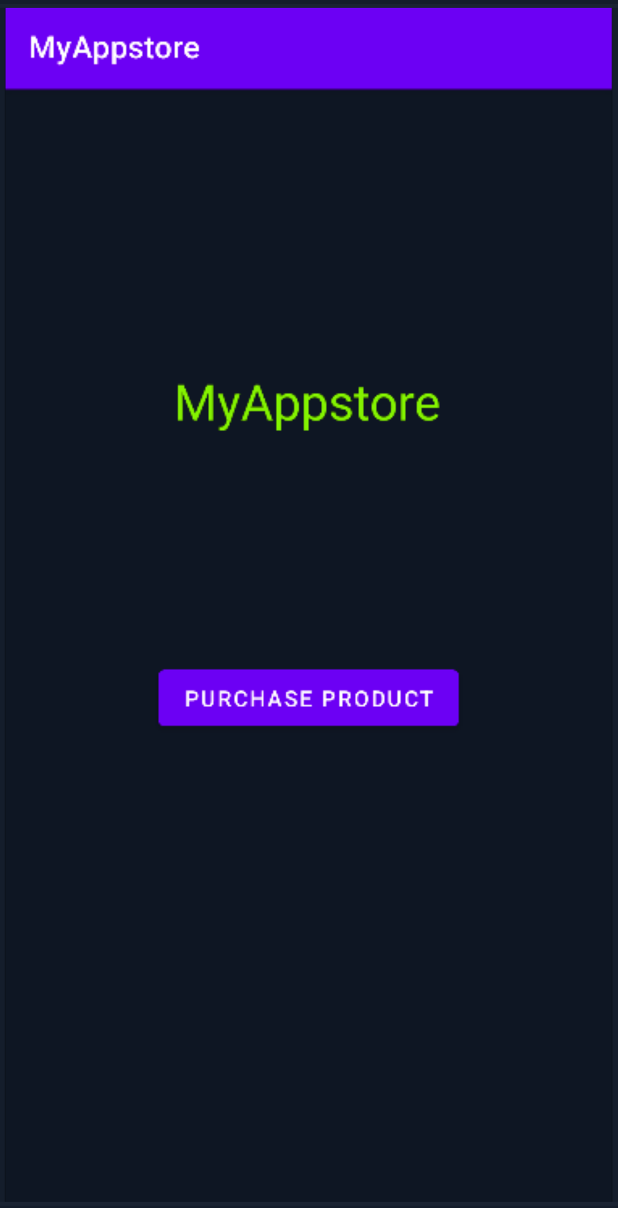
In the `Gradle Scripts` section, under the file `proguard-rules.pro`, you can specify rules to add or remove certain actions during the build process. In the example below, R8 is used. By default, R8 performs both code shrinking (removing unused or unnecessary code) and obfuscation (making the code harder to reverse-engineer). The rules that control how this shrinking and obfuscation occurs are defined in the `proguard-rules.pro` file.

Code: xml

```
# Prevent obfuscation of field names, but allow obfuscation of class and method names
-keep,allowobfuscation class * {
    !private *;
}

-dontoptimize
-dontusemixedcaseclassnames
-dontskipnonpubliclibraryclasses
-verbose
-repackageclasses 'a.b'
-dontwarn javax.lang.model.element.Modifier
```

Following the Control Flow of an Obfuscated App

In this example, we will examine an application created with Android Studio using the embedded R8 code shrinker. While reading the obfuscated code, we will try to understand and follow the flow of the method calls in the app and see if we can find any useful or sensitive information. For this example, we will also compare the pseudocode provided by JADX with the original app's source code in Android Studio. The following is a simple app simulating an app store that provides the user with a button to buy a product.

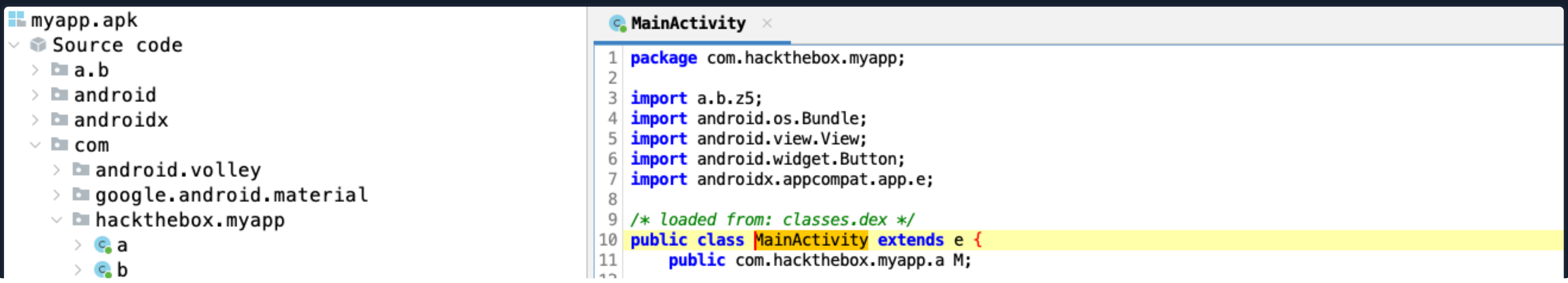


Let's use JADX to decompile the APK and read its source code.

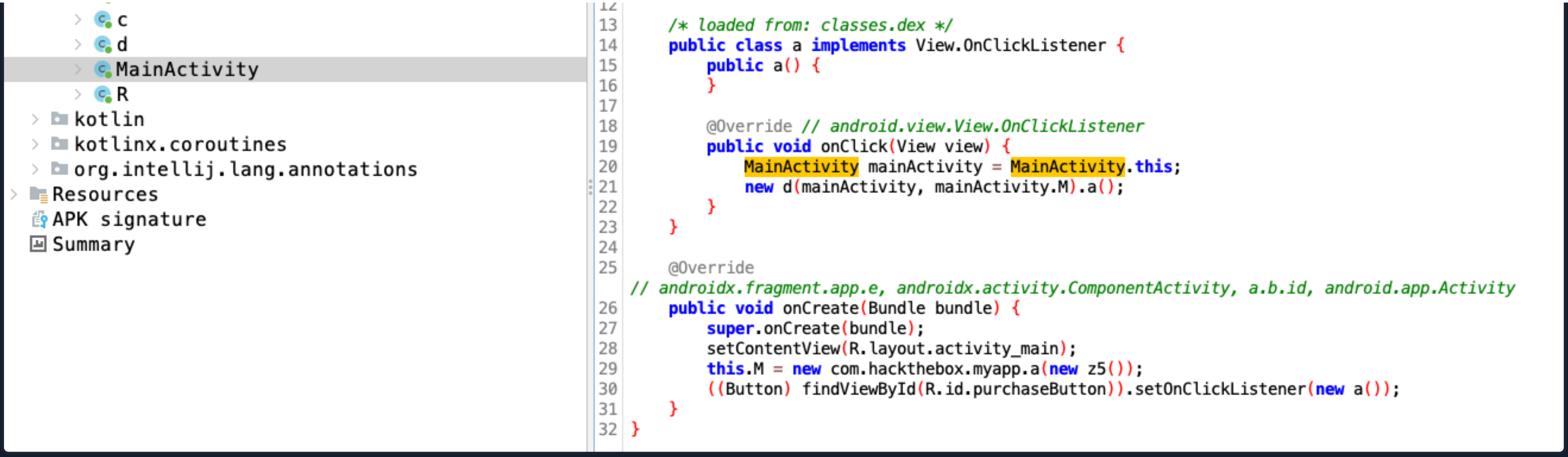
Reading Obfuscated Code

```
rl1k@htb[/htb]$ jadx-gui myapp.apk
```

Under `Source code` -> `com` -> `hackthebox.myapp`, we can see the `MainActivity` class.

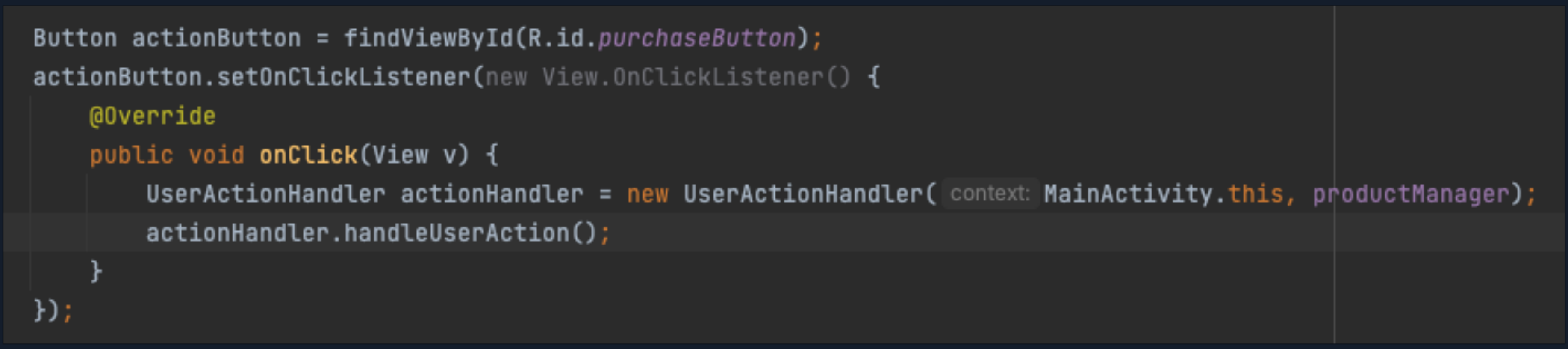


```
package com.hackthebox.myapp;

import a.b.z5;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import androidx.appcompat.app.e;

/* loaded from: classes.dex */
public class MainActivity extends e {
    public com.hackthebox.myapp.a M;
```



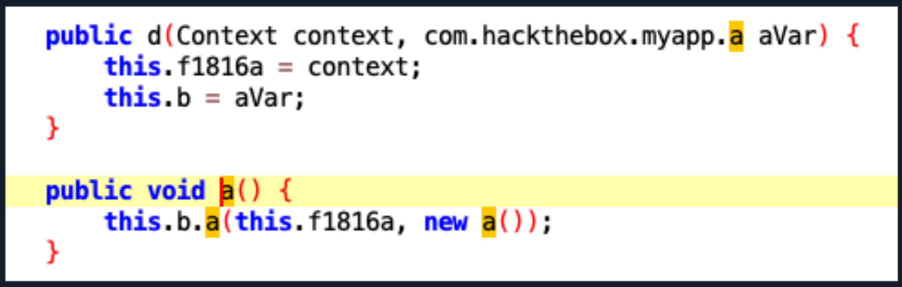
```
/* loaded from: classes.dex */
public class a implements View.OnClickListener {
    public a() {
    }

    @Override // android.view.View.OnClickListener
    public void onClick(View view) {
        MainActivity mainActivity = MainActivity.this;
        new d(mainActivity, mainActivity.M).a();
    }
}

@Override
// androidx.fragment.app.e, androidx.activity.ComponentActivity, a.b.id, android.app.Activity
public void onCreate(Bundle bundle) {
    super.onCreate(bundle);
    setContentView(R.layout.activity_main);
    this.M = new com.hackthebox.myapp.a(new z5());
    ((Button) findViewById(R.id.purchaseButton)).setOnClickListener(new a());
}
}
```

Reading the source code of the `MainActivity` class reveals the `onClick(View view)` method, which is most likely the `purchase product` button. Also, its content seems to call the method `a()` of the class/object `d()` in the line `new d(mainActivity, mainActivity.M).a();`. Having a look at the original app's source code reveals that the `onClick()` method will create the object `UserActionHandler`, which takes two parameters and then calls the method `handleUserAction()`.



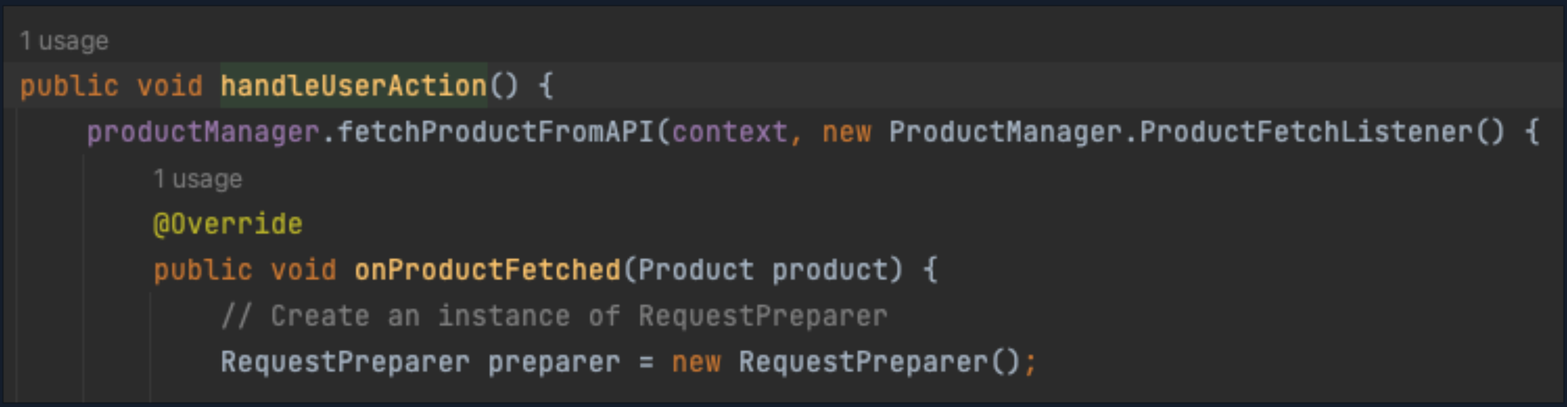
```
Button actionButton = findViewById(R.id.purchaseButton);
actionButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        UserActionHandler actionHandler = new UserActionHandler( context: MainActivity.this, productManager);
        actionHandler.handleUserAction();
    }
});
```

There is a high probablity that the method call `d(mainActivity, mainActivity.M).a();`—revealed in JADX—and the method call `actionHandler.handleUserAction();` in the original source code are the same. Let's double-click on the method `a()` in JADX to see where this method is created in the code.



```
public d(Context context, com.hackthebox.myapp.a aVar) {
    this.f1816a = context;
    this.b = aVar;
}

public void a() {
    this.b.a(this.f1816a, new a());
}
```

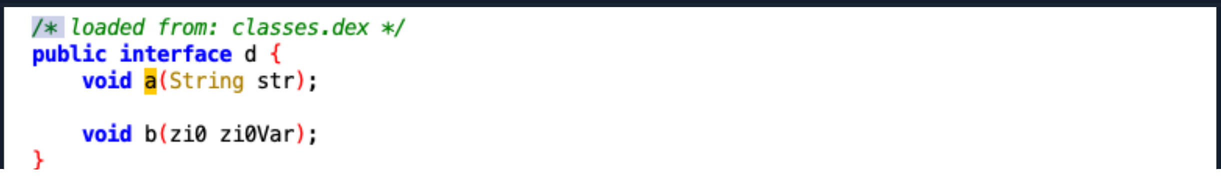
This will redirect us to the class `d{}`, revealing the method `a()` at the end of the file. Following the flow on Android Studio by holding alt and clicking on the `handleUserAction()` method also reveals the following snippet.



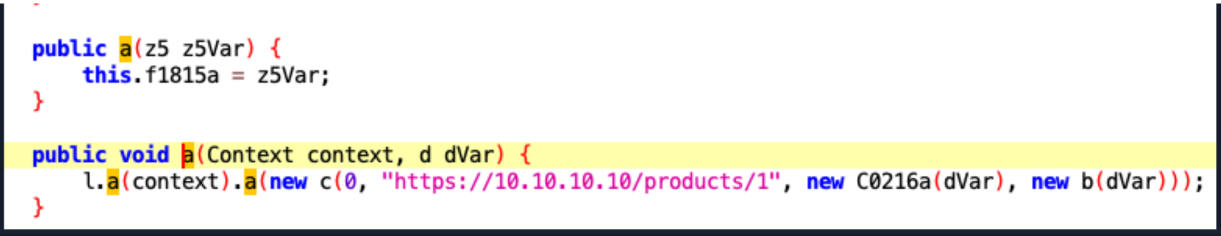
```
1 usage
public void handleUserAction() {
    productManager.fetchProductFromAPI(context, new ProductManager.ProductFetchListener() {
        1 usage
        @Override
        public void onProductFetched(Product product) {
            // Create an instance of RequestPreparer
            RequestPreparer preparer = new RequestPreparer();
```

=

This, in turn, calls the method `productManager.fetchProductFromAPI()`. Likewise, the code in JADX also appears to make a call at the method `b.a(this.f1816a, new a())`. Let's follow the method call again by double-clicking on the method `a()`.

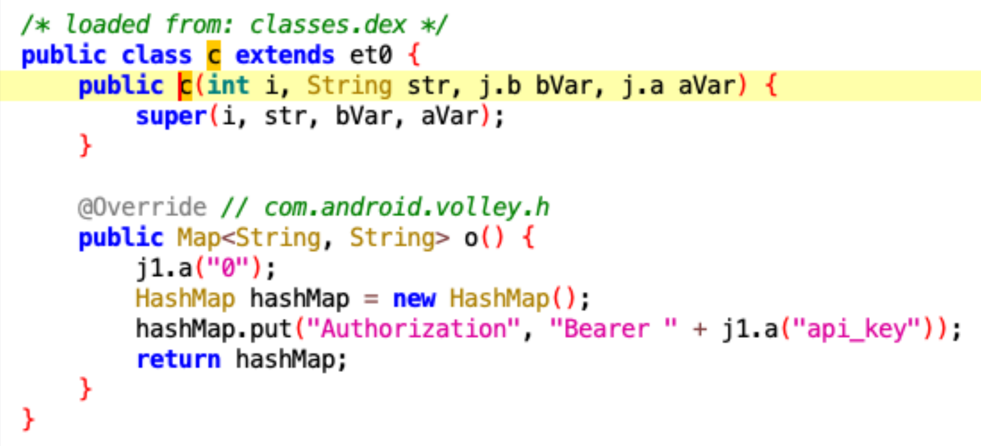


```
/* loaded from: classes.dex */
public interface d {
    void a(String str);

    void b(zi0 zi0Var);
}
```



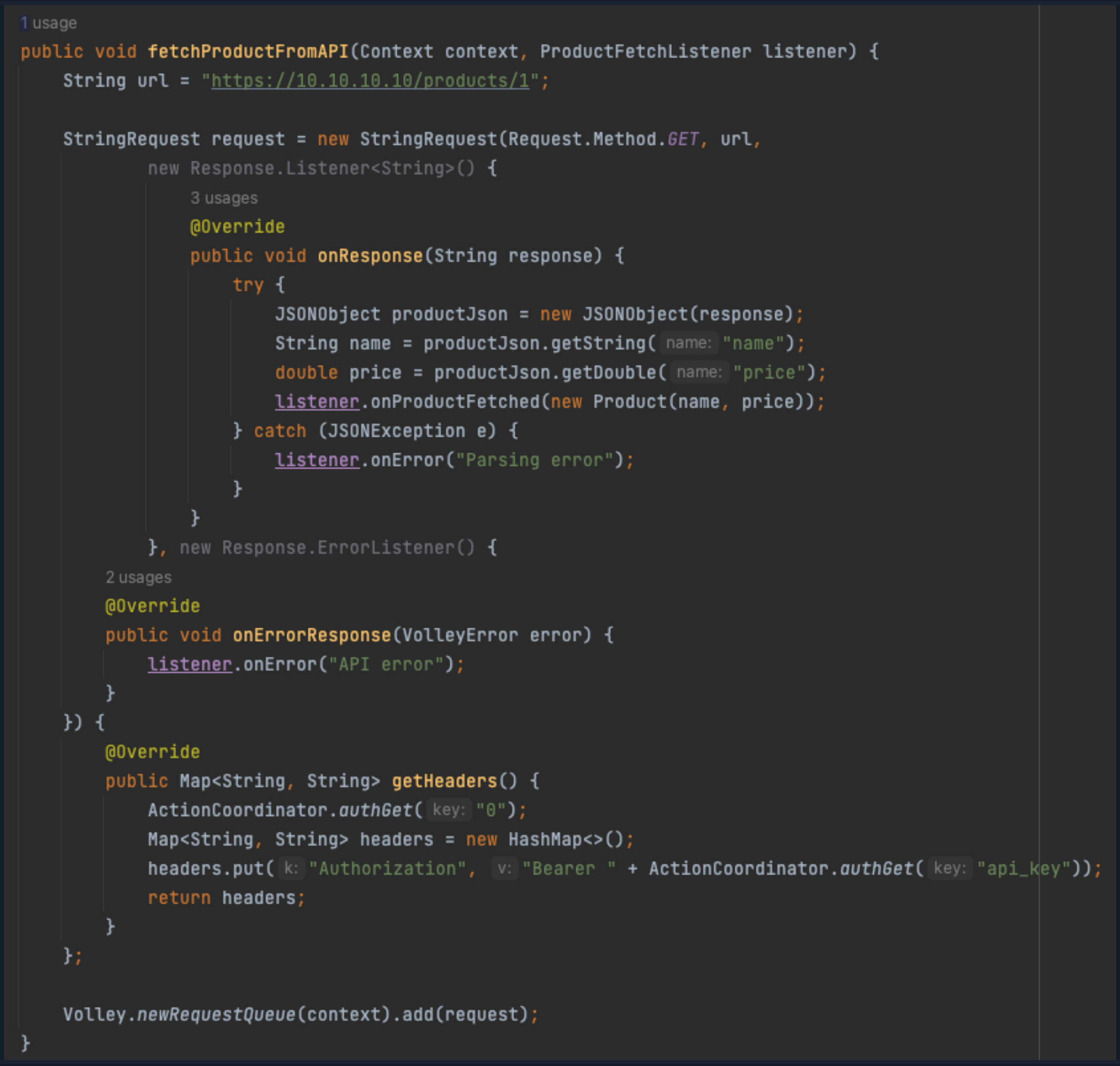
```
public a(z5 z5Var) {
    this.f1815a = z5Var;
}

public void a(Context context, d dVar) {
    l.a(context).a(new c(0, "https://10.10.10.10/products/1", new C0216a(dVar), new b(dVar)));
}
```

This leads us to the method `a()` of the class `a{}`, which looks like it makes a request to the URL `https://10.10.10.10/products/1`. This URL is passed as a parameter to the method `c()`, which, in turn, on double-clicking it, will lead us to the following snippet of code within the same class.



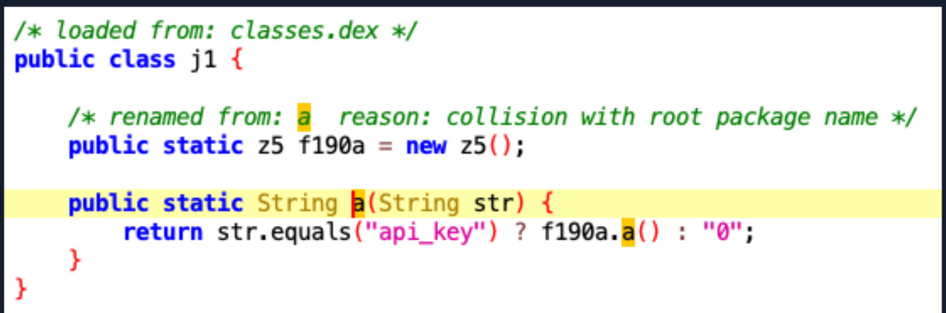
```
/* loaded from: classes.dex */
public class c extends et0 {
    public c(int i, String str, j.b bVar, j.a aVar) {
        super(i, str, bVar, aVar);
    }

    @Override // com.android.volley.h
    public Map<String, String> o() {
        j1.a("0");
        HashMap hashMap = new HashMap();
        hashMap.put("Authorization", "Bearer " + j1.a("api_key"));
        return hashMap;
    }
}
```

In the above method `c()`, we can see that a value with the name `api_key` is also put in a HashMap object, along with other values. Back in JADX, this value is returned by the method `j1.a()`. Having a quick look at the original app's source code, we can confirm that we are following the intended flow of the app.



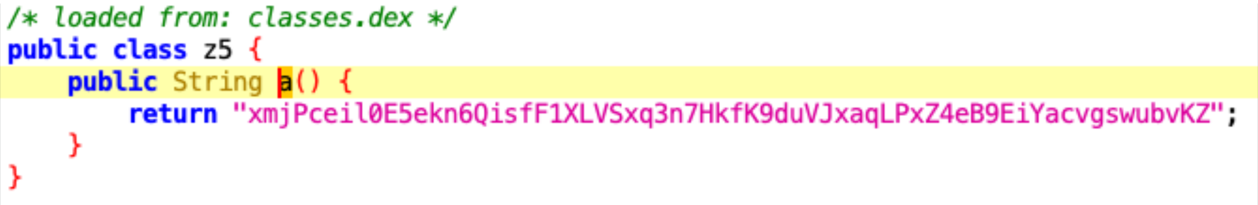
```
1 usage
public void fetchProductFromAPI(Context context, ProductFetchListener listener) {
    String url = "https://10.10.10.10/products/1";

    StringRequest request = new StringRequest(Request.Method.GET, url,
            new Response.Listener<String>() {
                3 usages
                @Override
                public void onResponse(String response) {
                    try {
                        JSONObject productJson = new JSONObject(response);
                        String name = productJson.getString( name: "name");
                        double price = productJson.getDouble( name: "price");
                        listener.onProductFetched(new Product(name, price));
                    } catch (JSONException e) {
                        listener.onError("Parsing error");
                    }
                }
            }, new Response.ErrorListener() {
        2 usages
        @Override
        public void onErrorResponse(VolleyError error) {
            listener.onError("API error");
        }
    }) {
        @Override
        public Map<String, String> getHeaders() {
            ActionCoordinator.authGet( key: "0");
            Map<String, String> headers = new HashMap<>();
            headers.put( k: "Authorization",  v: "Bearer " + ActionCoordinator.authGet( key: "api_key"));
            return headers;
        }
    };

    Volley.newRequestQueue(context).add(request);
}
```

The equivalent `j1.a("api_key")` method seems to be the `ActionCoordinator.authGet("api_key")` one. Double-clicking on the `j1.a()` redirects us to the instance of this method, indicating that we are on the right path.



```
/* loaded from: classes.dex */
public class j1 {

    /* renamed from: a  reason: collision with root package name */
    public static z5 f190a = new z5();

    public static String a(String str) {
        return str.equals("api_key") ? f190a.a() : "0";
    }
}
```

Finally, we can conclude that a string comparison occurs, comparing the name `api_key` with the name returned from the method `f190a.a()`. Let's double-click again on the `a()` method and check if we can read any potential key-value pairs.



```
/* loaded from: classes.dex */
public class z5 {
    public String a() {
        return "xmjPceil0E5ekn6QisfF1XLVSxq3n7HkfK9duVJxaqLPxZ4eB9EiYacvgswubvKZ";
    }
}
```

Eventually, the method `z5.a()` reveals the key `xmjPceil0E5ekn6QisfF1XLVSxq3n7HkfK9duVJxaqLPxZ4eB9EiYacvgswubvKZ`.

Connect to Pwnbox

Your own web-based Parrot Linux instance to play our labs.

Pwnbox Location

UK 28ms

Terminate Pwnbox to switch location

Start Instance

∞ / 1 spawns left

Waiting to start...

Enable step-by-step solutions for all questions

Questions

Cheat Sheet

Answer the question(s) below to complete this Section and earn cubes!

+ 3 What is the value of the hardcoded API key?

Submit your answer here...

+10 Streak pts Submit myapp_obfuscate.zip

Previous Next

Cheat Sheet

? Go to Questions

Table of Contents

Extracting and Enumerating APK Files

Analyzing Application's Source Code

Analyzing Native Libraries

Application Patching

Skills Assessment

My Workstation

OFFLINE

Start Instance

∞ / 1 spawns left