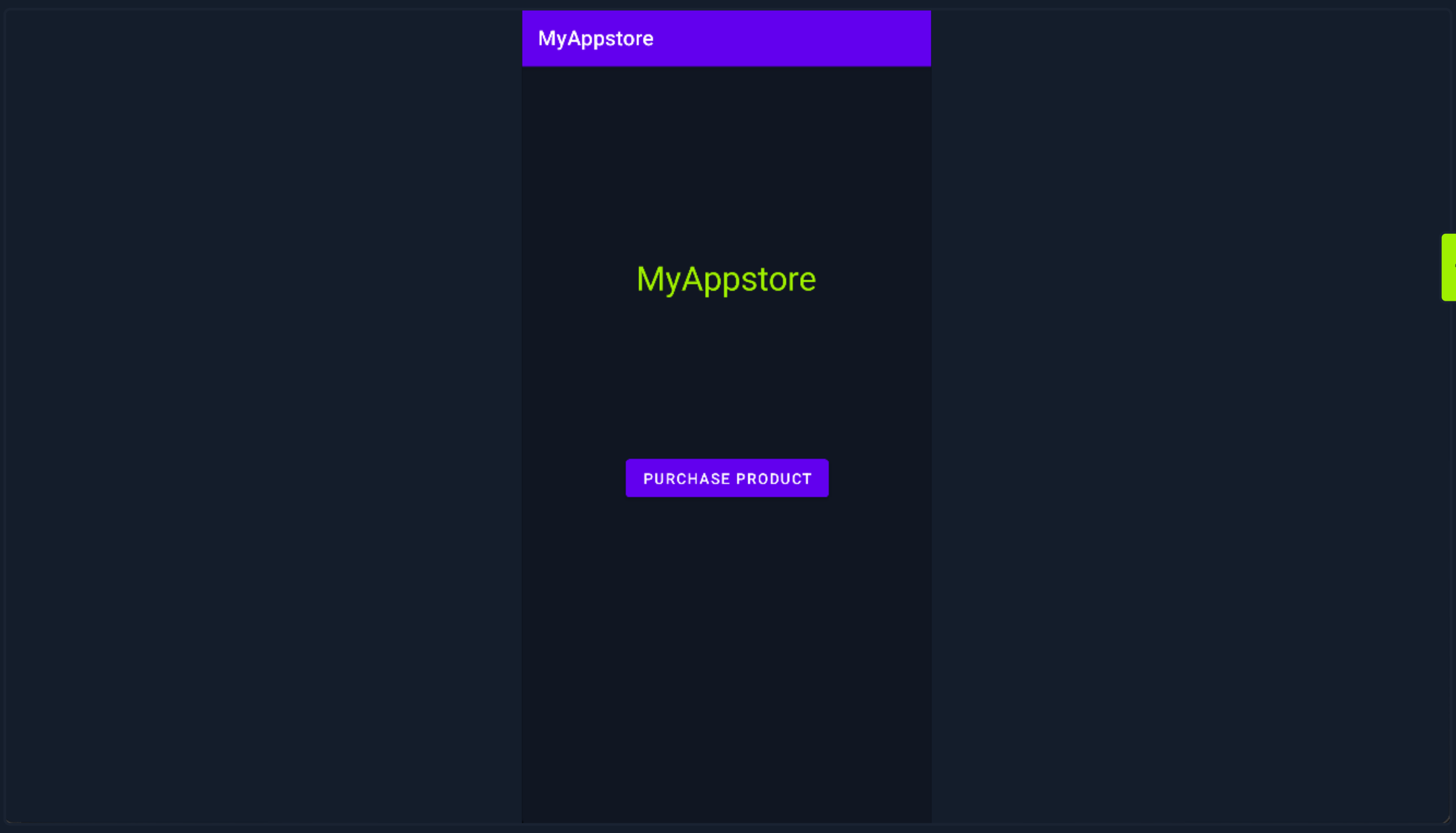
Deobfuscating Code

In the previous section, we explored how to manually analyze obfuscated code by tracing method calls and control flow to better understand an application's behavior and uncover potential security issues. While obfuscation can obscure variable names, method structures, and logic, it does not render the code entirely unreadable. With time and effort, it's often possible to reconstruct the program's functionality—even in heavily obfuscated apps.

In this section, we shift our focus to deobfuscation, the process of reversing obfuscated code to make it more readable. Rather than manually inspecting every line, we will showcase tools and techniques that automate parts of this process, particularly for reversing string obfuscation. These methods can significantly reduce the time and complexity involved in analyzing protected applications, especially those using open-source libraries to obfuscate their code.

Deobfuscating With Tools

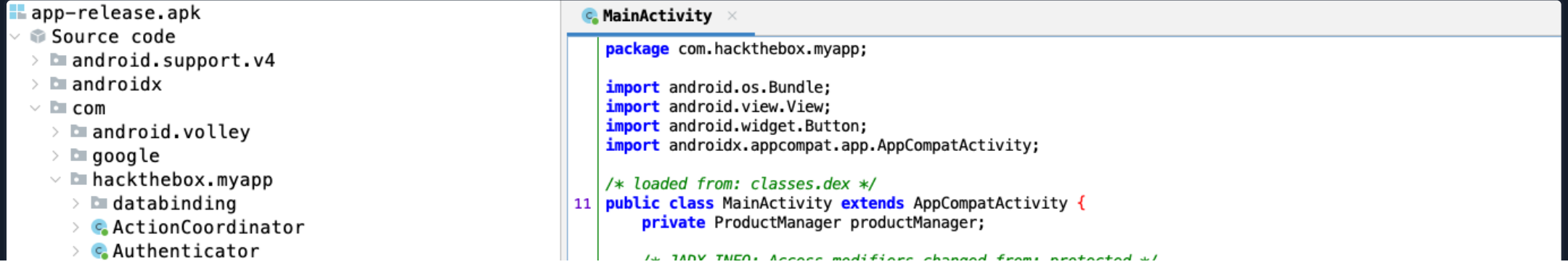
In this example, we will analyze the code of an application that has been obfuscated using an open-source library to conceal sensitive hardcoded strings. This is an application with a single screen that can purchase a product.



Let's use JADX to decompile the application.

Deobfuscating Code

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



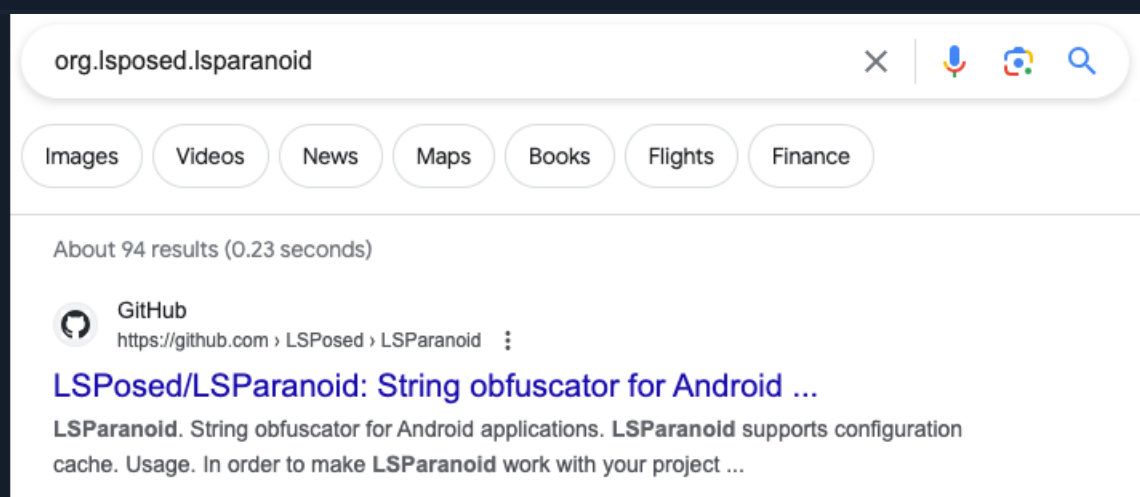
```
BuildConfig
DatabaseHelper
MainActivity
PaymentGateway
PaymentManager
Product
ProductManager
R
RequestPreparer
UserActionHandler
kotlin
kotlinx.coroutines
org
Resources
APK signature
Summary
```

```
    /* JADX INFO: Access modifiers changed from: protected */
    @Override
// androidx.fragment.app.FragmentActivity, androidx.activity.ComponentActivity, androidx.core.app.Compone
ntActivity, android.app.Activity
18  public void onCreate(Bundle bundle) {
19      super.onCreate(bundle);
20      setContentView(R.layout.activity_main);
23      this.productManager = new ProductManager(new Authenticator());
26      ((Button) findViewById(R.id.purchaseButton)).setOnClickListener(new View.OnClickListener() {
// from class: com.hackthebox.myapp.MainActivity.1
            @Override // android.view.View.OnClickListener
28          public void onClick(View view) {
                MainActivity mainActivity = MainActivity.this;
30              new UserActionHandler(mainActivity, mainActivity.productManager).handleUserAction();
            }
        });
    }
}
```

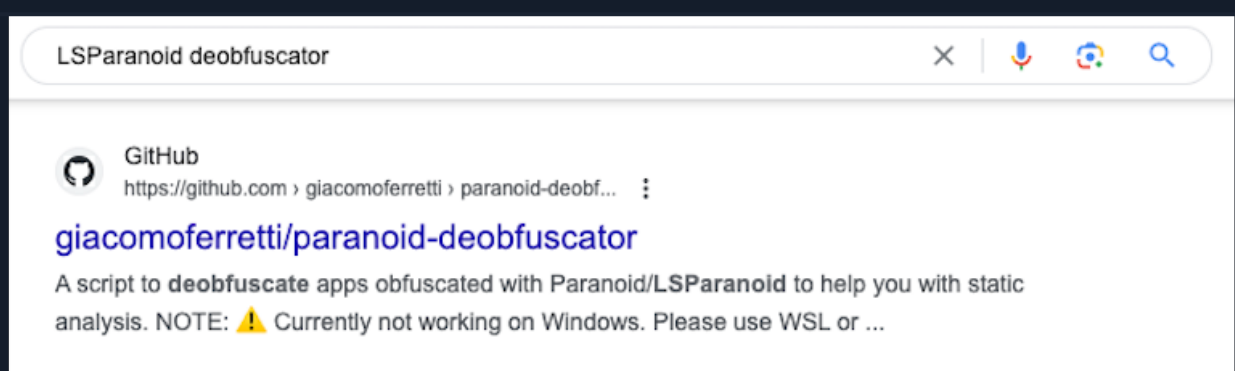
Reading the content of the `MainActivity` reveals the button `R.id.purchaseButton`. Double-clicking the `UserActionHandler` method inside the `onClick()` method reveals the `UserActionHandler` class. Continuing to trace the code as in the previous example leads us to the `Authenticator` class.

```
<:tivity   UserActionHandler   ProductManager   ActionCoordinator   Authenticator >
1  package com.hackthebox.myapp;
2
3  import org.lsposed.lsparanoid.Deobfuscator$myapp$app;
4
5  /* loaded from: classes.dex */
6  public class Authenticator {
7      private static final String API_KEY = Deobfuscator$myapp$app.getString(-280514339328L);
8
9      public String getAPIKey() {
10         return Deobfuscator$myapp$app.getString(-1341465088L);
11     }
12 }
```

The variable `API_KEY` indicates that the app uses an API key to connect with the remote server. As we have seen in previous sections, storing API keys insecurely could lead to unauthorized access to critical systems, data breaches, financial losses, and more. Upon closer examination, we see that the value of the API is not readable and the string is obfuscated. The `Deobfuscator$myapp$app.getString()` method seems to be returning the original value of the obfuscated string, and the line `import org.lsposed.lsparanoid.Deobfuscator$myapp$app;` at the top of the class indicates the obfuscation library. Searching online for `org.lsposed.lsparanoid` reveals the following as the first result.



According to the project description, this is a string obfuscator for Android applications called LSParanoid. Searching online for a tool to deobfuscate the app's strings reveals the following:



The first result is a GitHub project named paranoid-deobfuscator, a tool specifically designed to reverse string obfuscation in apps protected with `Paranoid` or `LSParanoid`. To download and install the tool, we can issue the following commands.

> ⓘ **Note:** Newer versions of paranoid-deobfuscator have problems with applications containing multiple getString methods, so for this exercise we will use version 2.0.1.

Deobfuscating Code

```
rl1k@htb[/htb]$ wget https://github.com/giacomoferretti/paranoid-deobfuscator/archive/refs/tags/v2.0.1.zip
rl1k@htb[/htb]$ unzip v2.0.1.zip
rl1k@htb[/htb]$ cd paranoid-deobfuscator-2.0.1
```

```
rl1k@htb[/htb]$ python -m venv .venv
rl1k@htb[/htb]$ source .venv/bin/activate
rl1k@htb[/htb]$ pip install "numpy==1.26.0"
```

Once it's installed, we'll decompile the app using APKTool.

Deobfuscating Code

```
rl1k@htb[/htb]$ apktool d myapp.apk
```

Now, let's try to deobfuscate the source code by running `paranoid_deobfuscator` as a python module.

Deobfuscating Code

```
rl1k@htb[/htb]$ python -m paranoid_deobfuscator -v myapp
```

Finally, we'll build the app using APKTool and read the content of the `Authenticator` class using JADX.

Deobfuscating Code

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

Authenticator

```
1  package com.hackthebox.myapp;
2
3  /* loaded from: classes.dex */
4  public class Authenticator {
5      private static final String API_KEY =
   "xmjPceil0E5ekn6QisfF1XLVSxq3n7HkfK9duVJxaqLPxZ4eB9EiYacvgswubvKZ";
6
7      public String getAPIKey() {
8          return "xmjPceil0E5ekn6QisfF1XLVSxq3n7HkfK9duVJxaqLPxZ4eB9EiYacvgswubvKZ";
9      }
10 }
```

The API key `xmjPceil0E5ekn6QisfF1XLVSxq3n7HkfK9duVJxaqLPxZ4eB9EiYacvgswubvKZ` is successfully deobfuscated.

Deobfuscating Manually

Relying on tools to deobfuscate an application's code isn't always feasible. Thus, learning to do it manually will increase your chances of success. In the following example, we will analyze the source code of an Android app to understand the obfuscation technique in use and manually construct a deobfuscation script. The functionality of this app mirrors that of the previous example.

Examining the `Authenticator` class reveals the line `private static final String API_KEY = Deobfuscator$myapp$app.getString(-280514339328L);`, which was shown in an earlier screenshot. This indicates that the API key is retrieved through a call to the `getString(-280514339328L)` method found in the `Deobfuscator$myapp$app` class. To investigate further, we'll double-click on the `getString()` method to inspect its implementation.

Authenticator | Deobfuscator$myapp$app

```
1  package org.lsposed.lsparanoid;
2
3  /* loaded from: classes.dex */
4  public class Deobfuscator$myapp$app {
5      private static final String[] chunks;
6
7      static {
8          chunks = r0;
9          String[] strArr = {
   "\uffbf\u0d99e\u0d9e\u0d99\u0d9a2\u0d9a\u0d9a\u0d9ab\u0d9a\u0d9a\u0d99f\u0d99\u0d988\u0d988\u0d99\u0d998\u0d998\u0d9fd\u0d9fd\u0d9a5\u0d9a5\u0d9aa\u0d9aa\u0d9b6\u0d9b6\u0d99e\u0d99e\u0d9ab\u0d9ab\u0d9a7\u0d9a7\u0d9b8\u0d9b8\u0d9bb\u0d9bb\u0d994\uffbf\u0d99e\u0d9e\u0d99\u0d9a2\u0d9a2\u0d9a\u0d9a\u0d9ab\u0d9ab\u0d99f\u0d99f\u0d988\u0d988\u0d998\u0d998\u0d9fd\u0d9fd\u0d9a5\u0d9a5\u0d9aa\u0d9aa\u0d9b6\u0d9b6\u0d99e\u0d99e\u0d9ab\u0d9ab\u0d9a7\u0d9a7\u0d9b8\u0d9b8\u0d9bb\u0d9bb\u0d994"};
10     }
11
12     public static String getString(long j) {
13         return DeobfuscatorHelper.getString(j, chunks);
14     }
15 }
```

The `Deobfuscator$myapp$app` class contains the method `getString()`, which takes one argument, the variable `long j`. This is the value `-280514339328L` that is passed as a parameter when the method is called, likely representing an identifier or key. Inside this method, we can see that there is another method, `DeobfuscatorHelper.getString(j, chunks)`. The values passed when this method is called are `j`—which we identified a

moment ago—and `chunks`, which seems to be the obfuscated string (API key). Let's double-click on the `getString(j, chunks)` method to examine the `DeobfuscatorHelper` class.

```
Authenticator   Deobfuscator$myapp$app   DeobfuscatorHelper

package org.lsposed.lsparanoid;

/* loaded from: classes.dex */
public class DeobfuscatorHelper {
    public static final int MAX_CHUNK_LENGTH = 8191;

    private DeobfuscatorHelper() {
    }

    public static String getString(long j, String[] strArr) {
        long next = RandomHelper.next(RandomHelper.seed(4294967295L & j));
        long next2 = RandomHelper.next(next);
        int i = (int) (((j >>> 32) ^ ((next >>> 32) & 65535)) ^ ((next2 >>> 16) & (-65536)));
        long charAt = getCharAt(i, strArr, next2);
        int i2 = (int) ((charAt >>> 32) & 65535);
        char[] cArr = new char[i2];
        for (int i3 = 0; i3 < i2; i3++) {
            charAt = getCharAt(i + i3 + 1, strArr, charAt);
            cArr[i3] = (char) ((charAt >>> 32) & 65535);
        }
        return new String(cArr);
    }

    private static long getCharAt(int i, String[] strArr, long j) {
        return (strArr[i / MAX_CHUNK_LENGTH].charAt(i % MAX_CHUNK_LENGTH) << 32) ^ RandomHelper.next(j);
    }
}
```

This class contains the method `getString(long j, String[] strArr)`, which is called from the previous class and receives the identifier `-280514339328L` and the obfuscated string as parameters. Inside the method, we notice the following two lines.

Code: java

```
long next = RandomHelper.next(RandomHelper.seed(4294967295L & j));
long next2 = RandomHelper.next(next);
```

These lines use a helper class `RandomHelper` to generate random values based on the input `j`. Reading the rest of the method, we can tell that it is designed to extract and decode strings from the array of obfuscated strings (`strArr`) and eventually return the original string based on the identifier. Let's also look at the `RandomHelper` class by double-clicking on it.

```
Authenticator   Deobfuscator$myapp$app   DeobfuscatorHelper   RandomHelper

package org.lsposed.lsparanoid;

/* loaded from: classes.dex */
public class RandomHelper {
    private static short rotl(short s, int i) {
        return (short) ((s >>> (32 - i)) | (s << i));
    }

    public static long seed(long j) {
        long j2 = (j ^ (j >>> 33)) * 7109453100751455733L;
        return ((j2 ^ (j2 >>> 28)) * (-3808689974395783757L)) >>> 32;
    }

    private RandomHelper() {
    }

    public static long next(long j) {
        short s = (short) (j & 65535);
        short s2 = (short) ((j >>> 16) & 65535);
        short s3 = (short) (s2 ^ s);
        return ((rotl(s3, 10) | (((short) (rotl((short) (s + s2), 9) + s)) << 16)) << 16) | ((short) (((
short) (rotl(s, 13) ^ s3)) ^ (s3 << 5)));
    }
}
```

Now that we know how the deobfuscation mechanism works, we can build a script in Java to deobfuscate the API key. One approach is to copy-paste the code of the JADX classes `Deobfuscator$myapp$app` and `DeobfuscatorHelper` locally. If this method does not work, we can pull these classes directly from the official GitHub repository. For this example, we'll go with the second approach. Below is the class `DeobfuscatorHelper`.

Code: java

```
public class DeobfuscatorHelper {
  public static final int MAX_CHUNK_LENGTH = 0x1fff;

  private DeobfuscatorHelper() {
    // Cannot be instantiated.
  }

  public static String getString(final long id, final String[] chunks) {
    long state = RandomHelper.seed(id & 0xffffffffL);
    state = RandomHelper.next(state);
    final long low = (state >>> 32) & 0xffff;
```

```
    state = RandomHelper.next(state);
    final long high = (state >>> 16) & 0xffff0000;
    final int index = (int) ((id >>> 32) ^ low ^ high);
    state = getCharAt(index, chunks, state);
    final int length = (int) ((state >>> 32) & 0xffffL);
    final char[] chars = new char[length];

    for (int i = 0; i < length; ++i) {
      state = getCharAt(index + i + 1, chunks, state);
      chars[i] = (char) ((state >>> 32) & 0xffffL);
    }

    return new String(chars);
  }

  private static long getCharAt(final int charIndex, final String[] chunks, final long state) {
    final long nextState = RandomHelper.next(state);
    final String chunk = chunks[charIndex / MAX_CHUNK_LENGTH];
    return nextState ^ ((long) chunk.charAt(charIndex % MAX_CHUNK_LENGTH) << 32);
  }
}
```

Let's also create the class `RandomHelper`.

Code: java

```
public class RandomHelper {
  private RandomHelper() {
    // Cannot be instantiated.
  }

  public static long seed(final long x) {
    final long z = (x ^ (x >>> 33)) * 0x62a9d9ed799705f5L;
    return ((z ^ (z >>> 28)) * 0xcb24d0a5c88c35b3L) >>> 32;
  }

  public static long next(final long state) {
    short s0 = (short) (state & 0xffff);
    short s1 = (short) ((state >>> 16) & 0xffff);
    short next = s0;
    next += s1;
    next = rotl(next, 9);
    next += s0;

    s1 ^= s0;
    s0 = rotl(s0, 13);
    s0 ^= s1;
    s0 ^= (s1 << 5);
    s1 = rotl(s1, 10);

    long result = next;
    result <<= 16;
    result |= s1;
    result <<= 16;
    result |= s0;
    return result;
  }

  private static short rotl(final short x, final int k) {
    return (short) ((x << k) | (x >>> (32 - k)));
  }
}
```

Finally, let's create a `Main.java` class to initialize the necessary values.

Code: java

```
public class Main {
    public static void main(String[] args) {
        long key = -280514339328L;
        String[] chunks = { "\uffbf\ub795\u17a0\u6064\ud99e\u9a6e\ua1ab\u6067\ud9a2\u9a3d\ua18b\u603b\ud9ab\u9a66\ua1a0\u6038
        };
        String result = DeobfuscatorHelper.getString(key, chunks);
        System.out.println(result);
    }
}
```

In the above snippet, the line `long key = -280514339328L` contains the identification number of the obfuscated string that we can get from the class `Authenticator` when the `Deobfuscator$myapp$app.getString(-280514339328L)` method is called.

Authenticator

```
1  package com.hackthebox.myapp;
2
3  import org.lsposed.lsparanoid.Deobfuscator$myapp$app;
4
5  /* loaded from: classes.dex */
6  public class Authenticator {
7      private static final String API_KEY = Deobfuscator$myapp$app.getString(-280514339328L);
8
9      public String getAPIKey() {
10         return Deobfuscator$myapp$app.getString(-1341465088L);
11     }
12 }
```

The variable named `chunks` contains the obfuscated string itself, and since its characters are not visible in the `code` mode inside JADX, we have to change to `smali` mode to get the Java unicode escape sequence. The button is located at the bottom of the window in the class `Deobfuscator$myapp$app`.

Authenticator | Deobfuscator$myapp$app

```
22     const-string v2,
"\uffbf\ub795\u17a0\u6064\ud99e\u9a6e\ua1ab\u6067\ud9a2\u9a3d\ua18b\u603b\ud9ab\u9a66\ua1a0\u6038\ud99
f\u9a64\ua1bd\u6068\ud988\u9a3c\ua196\u6042\ud998\u9a5e\ua1b6\u607f\ud9fd\u9a63\ua1f9\u6046\ud9a5\u9a6
b\ua185\u6037\ud9aa\u9a78\ua198\u6044\ud9b6\u9a6c\ua1bf\u6042\ud99e\u9a75\ua194\u603a\ud9ab\u9a4f\ua1f
7\u604b\ud9a7\u9a54\ua1af\u606d\ud9b8\u9a6a\ua1bd\u6079\ud9bb\u9a6f\ua1b8\u6045\ud994\uffbf\ub795\u17a
0\u6064\ud99e\u9a6e\ua1ab\u6067\ud9a2\u9a3d\ua18b\u603b\ud9ab\u9a66\ua1a0\u6038\ud99f\u9a64\ua1bd\u606
8\ud988\u9a3c\ua196\u6042\ud998\u9a5e\ua1b6\u607f\ud9fd\u9a63\ua1f9\u6046\ud9a5\u9a6b\ua185\u6037\ud9a
a\u9a78\ua198\u6044\ud9b6\u9a6c\ua1bf\u6042\ud99e\u9a75\ua194\u603a\ud9ab\u9a4f\ua1f7\u604b\ud9a7\u9a5
4\ua1af\u606d\ud9b8\u9a6a\ua1bd\u6079\ud9bb\u9a6f\ua1b8\u6045\ud994"
23
24     aput-object v2, v0, v1
25
26     return-void
27 .end method
28
29 .method public constructor <init>()V
30     .registers 1
31
32     invoke-direct {p0}, Ljava/lang/Object;-><init>()V
33
34     return-void
```

Code | Smali | Simple | Fallback | Split view

Once the script is ready, we can run the following command to compile and execute it.

Deobfuscating Code

```
rl1k@htb[/htb]$ javac Main.java; java Main

xmjPceil0E5ekn6QisfF1XLVSxq3n7HkfK9duVJxaqLPxZ4eB9EiYacvgswubvKZ
```

The API key `xmjPceil0E5ekn6QisfF1XLVSxq3n7HkfK9duVJxaqLPxZ4eB9EiYacvgswubvKZ` has been successfully deobfuscated.

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 Deobfuscate the source code of the APK found inside the "myapp_deobfuscate_1.zip" archive. What is the API key value?

Submit your answer here...

+10 Streak pts Submit myapp_deobfuscate_1.zip

+ 5 Deobfuscate the source code of the APK found inside the "myapp_deobfuscate_2.zip" archive. What is the API key value?

Submit your answer here...

+10 Streak pts Submit myapp_deobfuscate_2.zip

Previous Next

Cheat Sheet

Go to Questions

Table of Contents

Understanding Smali

Analyzing Application's Source Code

- Reading Hardcoded Strings
- Bad Cryptography Implementation
- Reversing Hybrid Apps
- Reading Obfuscated Code
- Deobfuscating Code

Analyzing Native Libraries

- Reversing Shared Objects
- Reversing DLL Files

Application Patching

- Authentication Bypass
- Modifying Game Apps
- License Verification Bypass
- Root Detection Bypass

Skills Assessment

- Skills Assessment

My Workstation

OFFLINE

Start Instance

∞ / 1 spawns left