Android malware analysis with Radare: Dissecting the Triada Trojan

Published: 2017-11-21 · Archived: 2026-04-10 03:04:45 UTC

I periodically assess suspicious mobile apps in order to identify malicious behavior, get ideas for new product functionality, or implement preflight checks to make sure apps submitted to our automated mobile app security testing solution can be properly assessed. As part of my work as a mobile security analyst at NowSecure, I recently performed Android malware analysis with Radare on a sample of the Triada Trojan.

Radare2 is known to disassemble Linux, Windows, and OSX binaries, but what about Android? Well, it can load Dalvik DEX (odex, multidex), ELF (.so, ART, executables, etc.), Java CLASS files and more!

I wrote this blog post to provide an introduction on how to use Radare for Android malware analysis. After reading this post, you'll understand how to use Radare2 to disassemble Android binaries, how to identify suspicious or malicious app behavior, and some of the benefits and limitations of using Radare2 for this use case.

Identifying a suspicious Android app

Finding a sample of Android malware isn't difficult, but I wanted to dissect something interesting. I searched Koodous, a platform for Android malware research, for "free download" and found an app named "Free Youtube Video Download." The app struck me as suspicious because screenshots were taken from another app, and the official icon was altered. So I clicked the download button and began my analysis.

Intent: Deciding whether a suspicious Android app is malware

A good first step in evaluating a suspicious file is understanding what other people already know about the file by scanning it with an anti-virus solution such as VirusTotal. I uploaded the sample, and VirusTotal reported a detection ratio of 34/55 and identified it as the Triada Android Trojan. So the app was infected.

Keep in mind that anti-virus technology can sometimes fail or return false-positive results because malware evolves faster than anti-virus vendors can develop good heuristics or signatures to filter them out.

To gather more information beyond that provided by anti-virus engines, I examined a YARA rule that detects the sample and uses the Androguard YARA module written by some peeps from the Koodous team.

The sample was discovered in March 2016, and there's still no publicly available technical analysis. At the time, Kaspersky Lab called Triada the most advanced Android malware to date and compared it to Windows malware in terms of complexity. To read more detailed information about the infection techniques used by this particular malware sample, see the blog post "Attack on Zygote: a new twist in the evolution of mobile threats."

As I prepared to dig deeper, I collected more samples that matched the same YARA rule by pasting the rule in the Koodous search box, which resulted in three more .apks. While I had gathered multiple samples that matched the

YARA rule, I couldn't be sure whether or not they were variants of the same family.

Exploring the functionality of Android malware

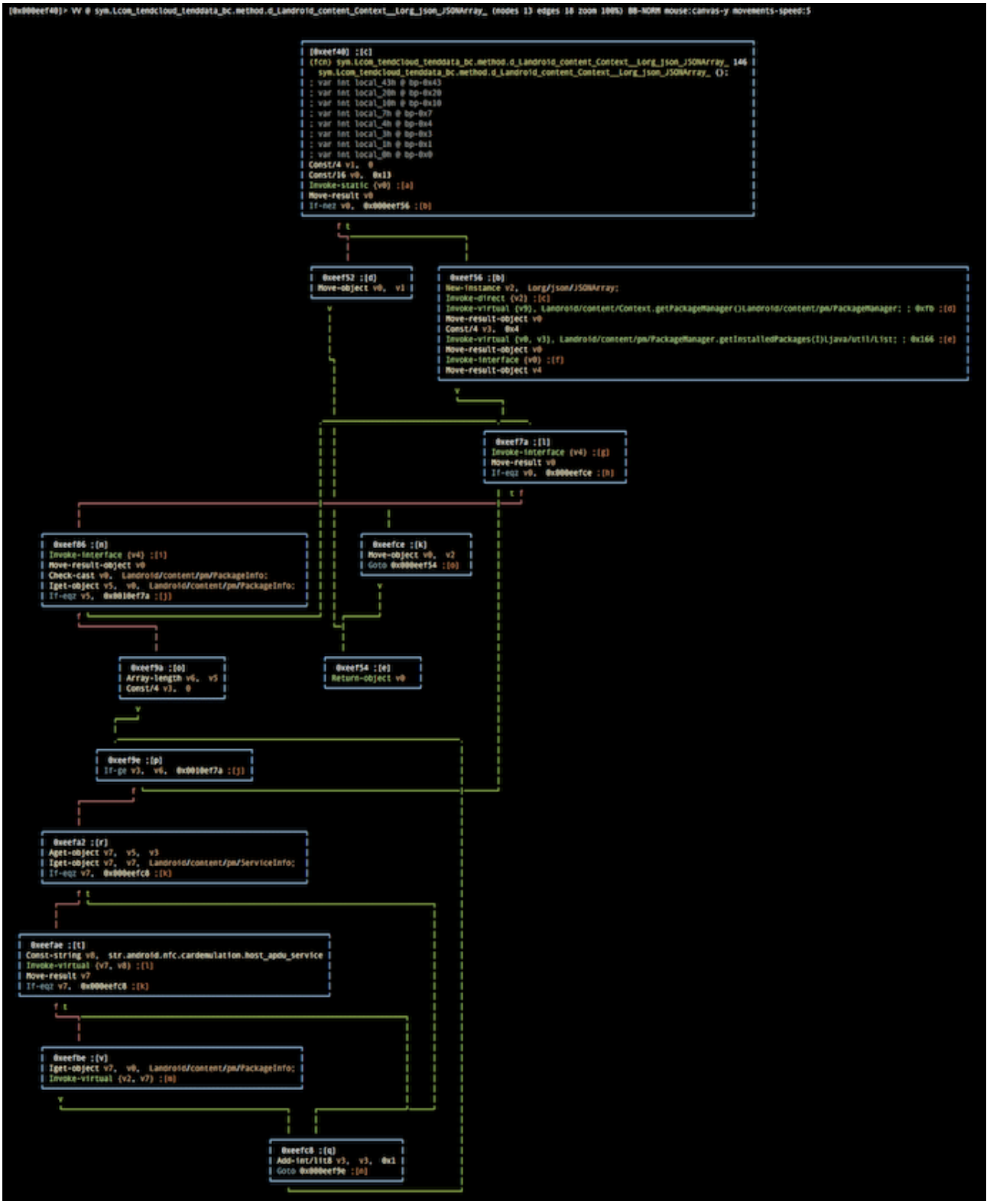
When analyzing a suspicious Android app, you'll evaluate the following factors:

- **Obfuscation techniques** – Obfuscation may be used to legitimately protect intellectual property or to hide malicious code.
- **Excessive permissions** – If the app requests excessive permissions, it may simply be a mistake made by the developer or deliberate and for nefarious purposes.
- **Strange files on the device** – Trash files, binary resources or encrypted payloads.
- **Emulator checks** – Requiring a patch for the app or using an actual device for analysis.
- **ARM-only libraries** – Making it so the app can't be executed on an x86 emulator, for example.
- **Strings pointing to system binaries** – This may be a legitimate root detection mechanism or part of an escalation of privileges exploit.
- **Hardcoded IPs in the binaries** – These may be IP addresses for testing servers forgotten by developers or for command-and-control servers (botnets).
- **Links to APKs** – Developers may have forgotten to remove these links used for testing purposes or the links may point to second-stage payloads.
- **Access to the SMS/messaging functionality** – This may be legitimate and used for two-factor authentication or for malicious purposes such as unauthorized subscription payment services scams or stealing authentication codes for bank payments, etc.
- **Modified legitimate apps** – Bad actors regularly modify legitimate apps to include/hide a malware payload within them.
- **App source or reliability of hosting/connectivity channel** – If the source of an .apk is an alternative Android marketplace or unknown, you can't be sure what review process the app went through.

If the app comes from a suspicious source, you should trust it less than an app that comes from an official marketplace (which will typically document its review process), and it warrants deeper analysis to check for malicious intent. In the same way, if an app asks for more permissions than seems appropriate, avoid installing it and do more analysis. Manually analyzing the files within the .apk is a good place to start and will help you select some metrics to help understand the app's intent and capabilities.

When you analyze a malware sample, determining what code is part of the original app as opposed to the malware itself, the protector, ads, or a tracking system can be a challenge. It's common to find numerous statistics SDKs within the same app collecting data about the device and user behavior. You'll also find boot-time services that launch them after reboot.

In this case, I found the sample generating a JSON file enumerating all the installed apps containing permissions for doing host card emulation (HCE) for NFC. The sample also dynamically loads two encrypted blobs and references another .apk protected with APK Protect. Extracting those pieces requires dynamic analysis, which I plan to cover in a future article.



Performing a full analysis goes beyond the scope of this blog post, but I will walk through the introductory steps that readers can then use as a starting place for more in-depth analysis. The steps I will cover include the following:

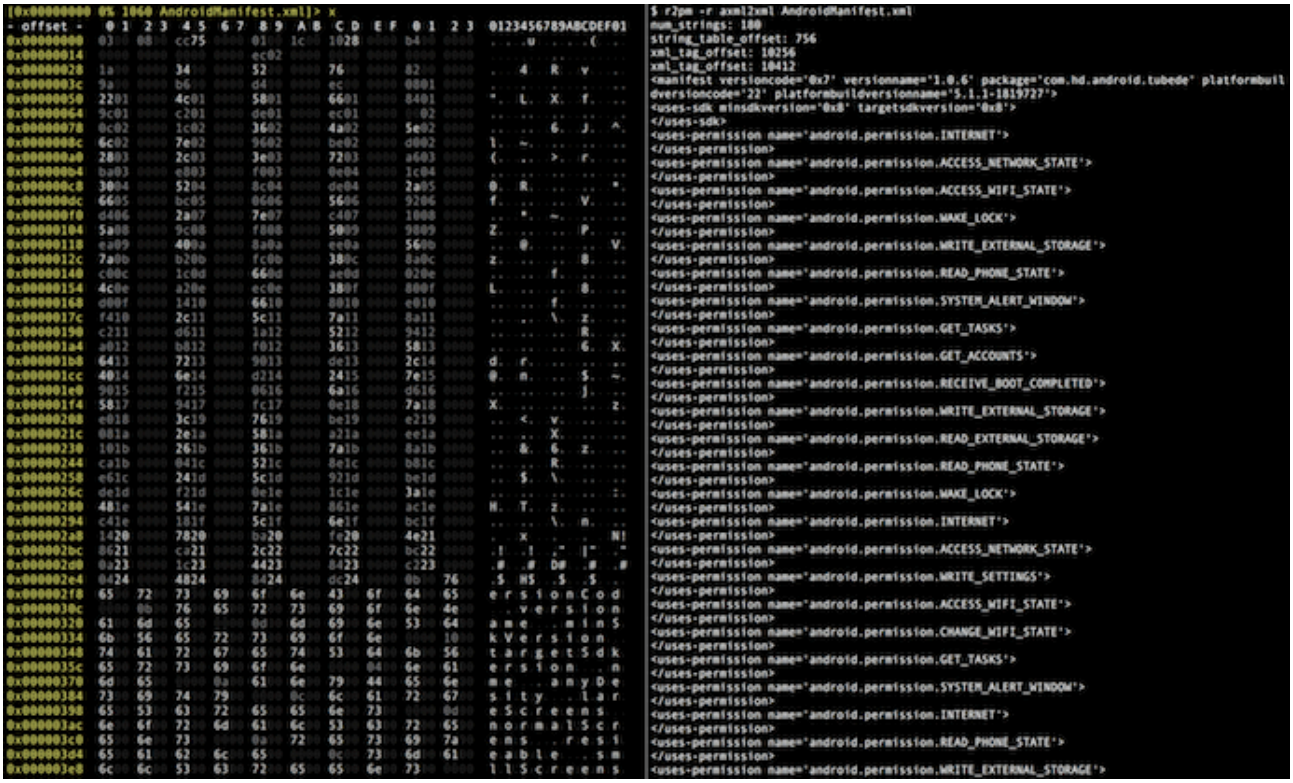
1. Extracting details about the malware like lists of permissions, types of binaries, function names, imported symbols, classes, strings contained in the binaries, etc.

2. Performing static analysis on the suspicious binaries within the app in order to understand what the code is doing with the resources identified in the first step.
3. Emulating the sample in a QEMO-based Android environment in order to trace program execution and identify system events and behaviors.

Most mobile malware will not execute on non-ARM or emulated environments, and so I analyzed the sample on an actual hardware device with clean, reflashed firmware.

AndroidManifest.xml

While its extension is .xml, AndroidManifest.xml is not plain text. It is a binary XML format used only in Android, and you can convert it to a readable format with the Axml2xml.rb Ruby script (also available via r2pm -i axml2xml).



You can also use the rafind2 program within Radare to extract strings of interest from the file (note that strings will not work at all):

```
$ rafind2 -ZS android AndroidManifest.xml

$ rafind2 -ZS permissions AndroidManifest.xml

$ rafind2 -ZS intent AndroidManifest.xml
```

Some common permissions used by malware include the following:

```
permission.CHANGE_WIFI_STATE

permission.CHANGE_NETWORK_STATE

permission.INSTALL_PACKAGES

permission.INSTALL_SHORTCUT

permission.SYSTEM_OVERLAY_WINDOW

permission.ACCESS_DOWNLOAD_MANAGER

permission.MOUNT_UNMOUNT_FILESYSTEMS

permission.RECORD_AUDIO

permission.RECEIVE_BOOT_COMPLETED

permission.KILL_BACKGROUND_PROCESSES

permission.ACCESS_MTK_MMHW

permission.DISABLE_KEYGUARD

permission.SYSTEM_ALERT_WINDOW/

permission.GET_TASKS

...
```

Classes.dex

This file contains the Dalvik code of the application. All of an app's original Java code is transpiled into Dalvik and assembled in the DEX file, which runs on a stack-less, register-based virtual machine. Some versions of Android will emulate the code directly from the .dex file, others will translate it to real machine code via JIT (just-in-time) techniques, and most modern versions of Android will precompile most of that code into an AOT (ahead-of-time) executable targeting the ART (Android runtime).

You can use Radare2 to extract information from the .dex file using the following commands:

```
> icq # enumerate classnames

> iiq # imports (external methods)

> ic # enumerate classes and their methods
```

```
> izq # list all strings contained in the program
```

One important hint is to compare the permissions requested by the manifest and the ones used by the app itself. In this case, as long as the application can dynamically load new code we will not see the true purpose unless we perform a dynamic analysis.

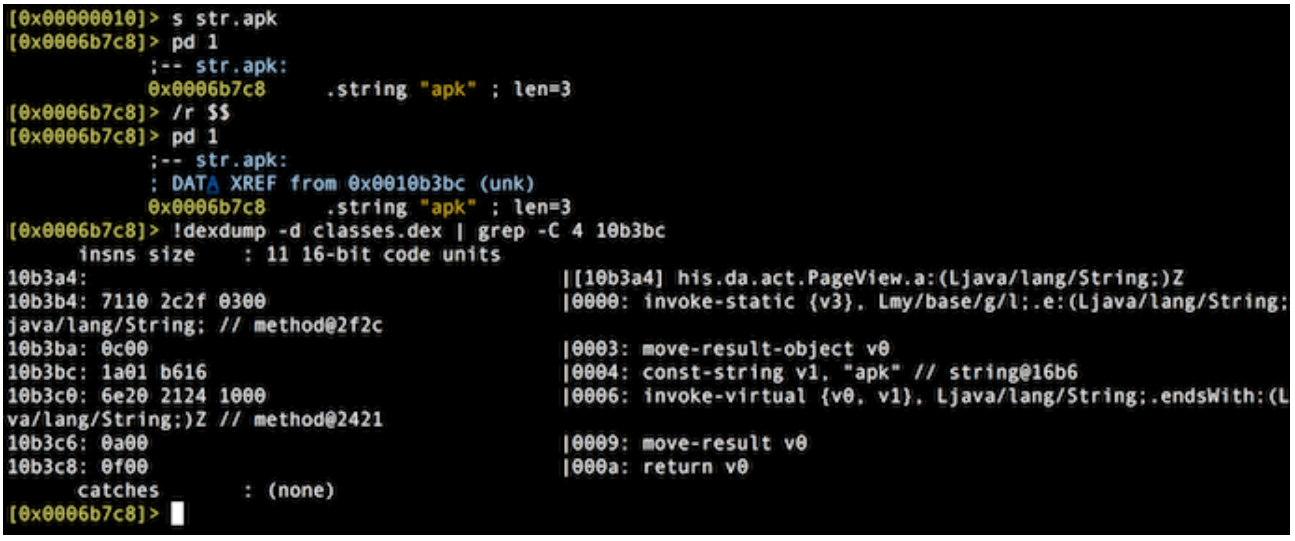
Checking out the imports of the classes.dex will give us some hints about which kind of system APIs the application is using.

```
$ rabin2 -qi classes.dex | grep -i -e sms -e bluetooth -e install -e PackageManager -e Datagram -e T
```

In addition we can use dexdump to enumerate all the intents used in the main dex:

```
$ dexdump -d | grep '"android.' | cut -d , -f 2- | sort -u
```

Once we have a complete disassembly from dexdump, you can use your favorite editor or even plain grep to find string-const calls. But in Radare2 you can also use the /r command to find references to strings or methods in order to identify who is using those suspicious strings and for what purpose.



Filtering Strings

To filter the output of commands, you can use the ~ operator in the Radare2 shell. The ~ operator is similar to the UNIX grep utility, but it runs internally and doesn't require any separate system process to run.

Using the following string filters you can identify some interesting findings in the classes.dex file:

- /system /data /bin/su …
- http://
- https://
- .apk

- %d.%d.%d.%d
- Install
- SMS
- DexClassLoader InjectCall (*used for Dalvik code injection*)
- application/vnd.android (*mimetype used to spawn .apk installations)*
- == (*embedded base64 resources*)

After executing the string filters on the malware sample, I found some hard-coded IPs, paths to system binaries like su for device root checks, Dalvik code injection calls, the strings commonly used for installing .apks and finally the URL to another file named 300010.apk.

When an anti-virus vendor or a researcher discovers URLs linking to malware, they typically send a take-down notice to the appropriate ISP. The ISP will then contact the owner of the server hosting the malware or directly block those servers. So I was sure to grab the 300010.apk before it was removed so that I could perform further analysis.

Because the .apk is protected with APK Protect, we need to decrypt the dynamically loaded code in order to understand what the app is doing exactly. This Android unpacker tool can be used for that purpose.

In this .dex file, I might also find strings encoded in base64, which can be decoded automatically by setting up the RABIN2_DEBASE64 environment variable or using the command rax2 -D on every single string. It turns out that some base64 encoded strings contain binary data and others show interesting locations inside the binary, like loadLibrary().

```
$ RABIN2_DEBASE64=1 rabin2 -qzz classes.dex
```

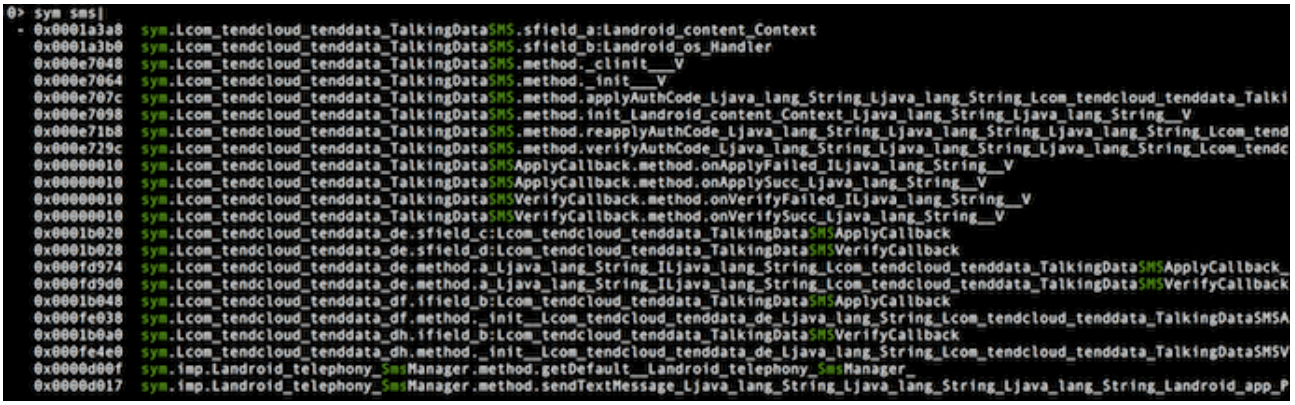
Filtering methods

Radare2's visual mode also supports an interactive heads-up display that will filter out all string, method, and class names from the .dex file. To enter the visual mode, run the "V_" command, then type '_' in visual mode. Next, type text that you want to highlight. When you press <enter>, the UI will jump to that place in the disassembler so that you can view the text in context.

Some relevant text you may want to search for include:

- onReceive (*used by event handlers*)
- Init (*all classes have one of them*)
- Password
- Install
- Dex
- SMS

Most of the time, malware is protected with ProGuard or other obfuscation tools that will make the class/method/field names completely useless. In that case, you need to dig into the disassembly or perform dynamic analysis to recover the original strings and understand the purpose of each method.



Disassembling

Radare2 provides a visual mode (V command) and web user interface (via the =H command) that allows you to use the mouse and get a more interactive view than the just a static prompt.

At this point, for analysis purposes, you may want to compare the output of Radare2 with other tools like `dexdump -d`, which is available in the Android SDK. Radare2 is known to be more reliable when tweaking .dex binaries, making it a good alternative when decompilers and transpilers can't be used.

```
$ r2pm -i dex2jar
$ r2pm -r dex2jar classes.dex
```

Once the .dex file is transpiled into Java Classes, we can use Radare2 to disassemble the contents of the JAR. Because having multiple views of the same code and reading assembly is not always the fastest way to understand some parts of a program, you may want to use a decompiler like `jd-gui` which is freely available and can load all the classes into the JAR file simultaneously.

Unfortunately, transpiling from Dalvik to Java bytecode or decompiling Java methods does not always work. Any number of different tools can fail at automated disassembly So an analyst needs to know and understand the risks and weak points of disassembly capabilities within each tool and be able to resolve those problems by hand.

For example, if we try to disassemble the 300010.apk with dexdump we will get a fancy segmentation fault (which has since been fixed in the Android 7 SDK):

```
$ dexdump -d classes.dex  > /dev/null
GLITCH: zero-width instruction at idx=0x0000
GLITCH: zero-width instruction at idx=0x0000
GLITCH: zero-width instruction at idx=0x0000
Segmentation fault: 11
```

Transpiling to Java will partially fail:

```
$ r2pm -r dex2jar classes.dex 2>&1 |grep Error | wc -l
      40
```

Androguard will stop analyzing after an exception:

```
$ androgui.py -i classes.dex
Traceback (most recent call last):
  File "/Library/Python/2.7/site-packages/androguard/core/bytecodes/dvm.py", line 7014, in g
    off = self.__manage_item[ "TYPE_STRING_ID_ITEM" ][idx].get_string_data_off()
IndexError: list index out of range
```

In one case, trying to open the .apk in IDA Pro resulted in 9,382 error messages. In cases such as this, a low level disassembler like Radare2 can help.

Libraries

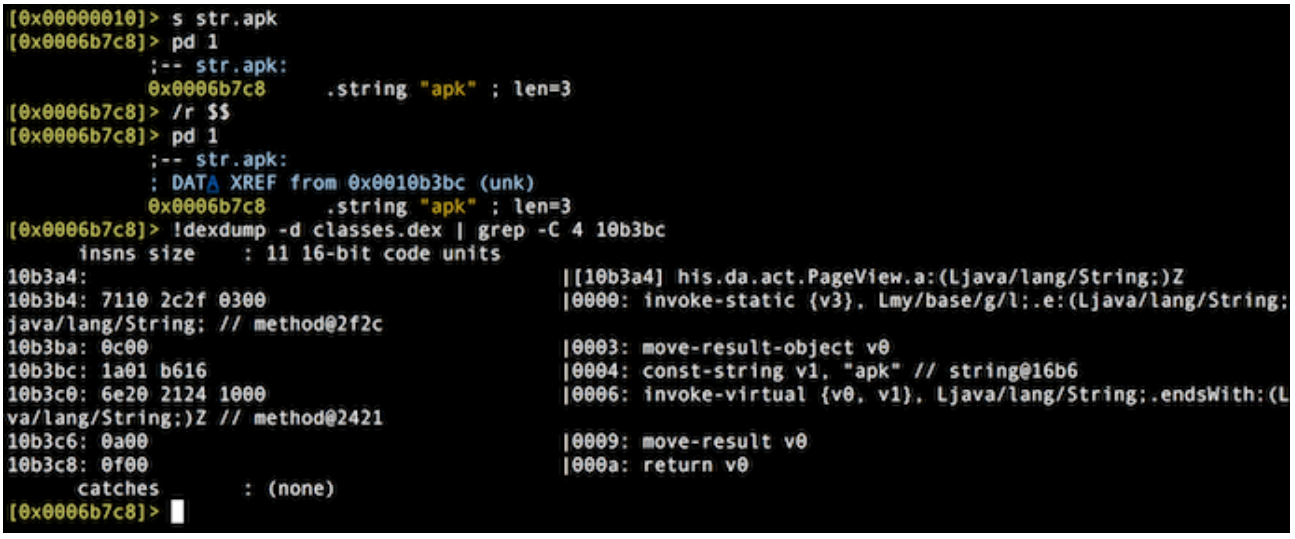
Native libraries are usually interfaced with Java using JNI symbols, which are exposed in the shared library found in the lib/ subdirectory inside the .apk. Those libraries are usually split into different subdirectories depending on the target architecture that is supposed to run (e.g., ARM, x86, ARM64, MIPS, etc.)

In this case we have only ARM binaries, which are typically compiled in Thumb2 mode. Radare2 will load those binaries just fine, but you may be interested in switching between ARM and Thumb modes:

```
> e asm.bits=16  # set thumb2 mode
> e asm.bits=32  # set ARM mode
```

There are other variables like asm.cpu that can be useful to get the correct disassembly in some cases, but this will be enough for most situations.

In visual mode we can also use the HUD mode (use the '_' key as explained before) to collect information from the binary, such as symbols (isq), imports (iiq), or strings (izq).



After running those commands, you can make an educated guess about the libraries' purposes:

```
lib/armeabi/libbspatch.so     # bzip2 + binary patch API, used by umeng API to update stuff
lib//armeabi/libcore.so       # upay cryptography (Java_com_lem_sdk_util_CoreEnct_decrypt
lib//armeabi/libmagic.so      # java code injection by using reflection methods
```

At the least, none of them seem related to downloading YouTube videos.

In order to disassemble and understand what the code is doing, you will want to emulate and analyze the code. You can also do this with Radare2. Radare2's emulation capability is implemented on top of ESIL (Evaluable Strings Intermediate Language), which is a completely safe virtual machine that has nothing to do with Dalvik or real hardware at all, so you can consider it as safe as static analysis.

The reason why emulation is necessary to disassemble ARM binaries is because some pointers are computed by more than one instruction. This requires tracking the state changes and resolving the correct strings and method references.

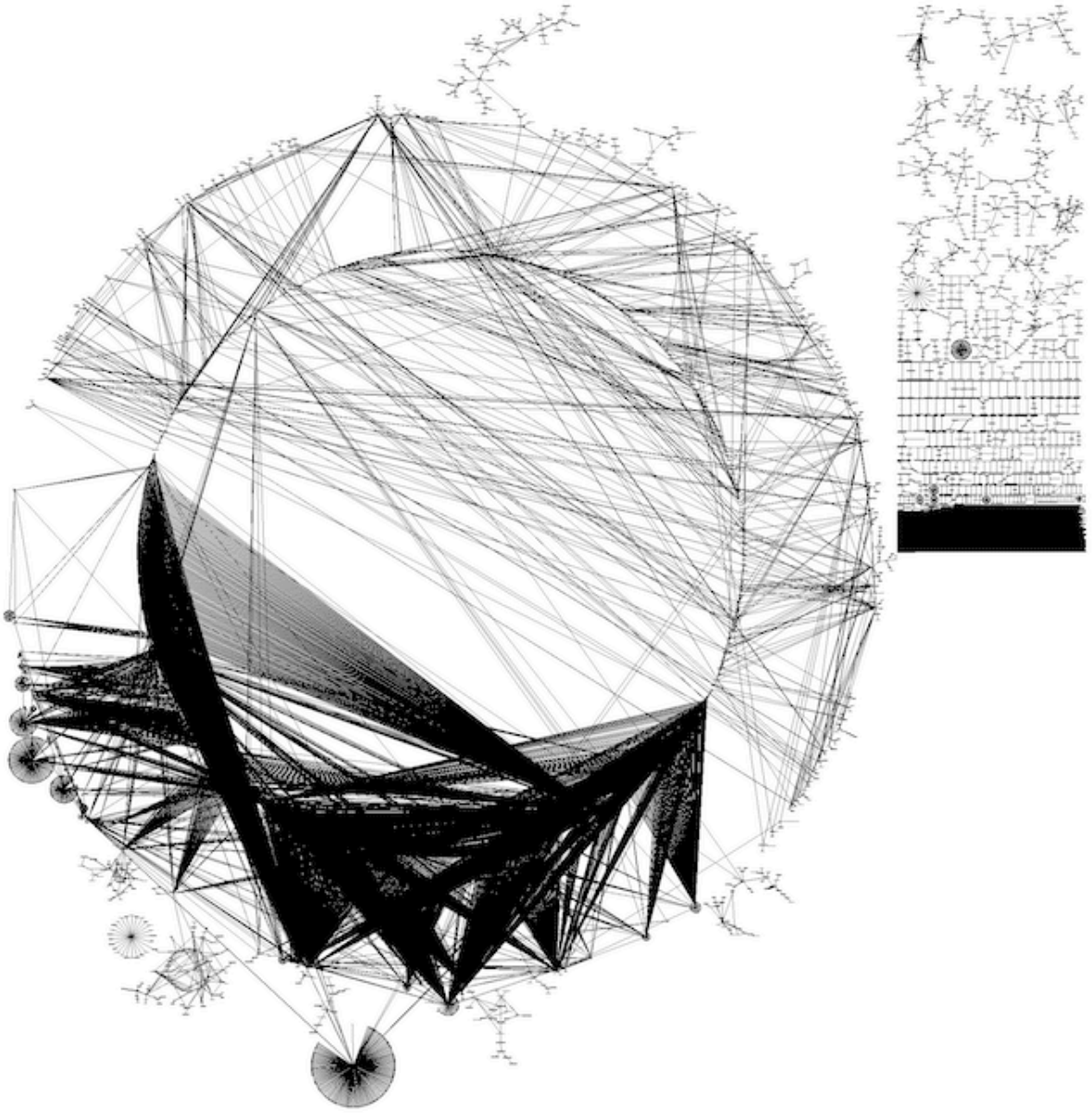
```
> e asm.describe = true   # show description of each ARM instruction
> e asm.pseudo = true     # show pseudo instruction instead of assembly
> e asm.emu = true        # emulate code using ESIL
> e asm.emustr = true     # show string and method referenced in the emu comments
> e anal.hasnext=true     # assume a new function is found after the last one
```

Some commands that help with malware analysis include:

```
> aa - analyze all public symbols (use aaa or aaaa for more!)
> afr - analyze functions recursively
```

```
> aae - analyze all code references computed with ESIL emulation
> aac - analyze all call destinations as functions
```

In case you're interested in a higher level version of the code, you can use the Retargetable Decompiler package in Radare2 to take advantage of the online decompilation service available at www.retdec.com.



Radial graph of target's full classes.dex references

Assets

An .apk file contains several companion files that are loaded by the app at runtime. Some assets are images, others are XML files describing the user interface. During malware analysis, you may find suspicious companion files that you want to take a closer look at.

You can use the `pm` command in Radare2 or the file tool to guess the filetype with the magic header information on every file:

```
$ find . -type f -exec r2 -qnci~^file -cpm '{}' ';'
$ file assets/*
```

```
$ file assets/*
assets/10011.data:          data
assets/110011.data:         data
assets/ac.png:              data
assets/btn_close_papi.png: PNG image data, 75 x 75, 8-bit colormap, non-interlaced
assets/channel.ini:         ASCII text, with no line terminators
assets/channel.inii:        ASCII text, with no line terminators
assets/close.png:           PNG image data, 64 x 64, 8-bit gray+alpha, non-interlaced
assets/ep:                  data
assets/fx:                  data
assets/lead:                directory
assets/libs:                directory
assets/magic_encrypt.png:   data
assets/oversea:             directory
assets/oylej:               data
$
```

The reason to use the internal magic implementation of Radare2 instead of GNU is mainly because there are known vulnerabilities for libmagic file which may be used to run code or invalidate the analysis. RedHat patched the issue this summer, but some other distributions and Docker images remain vulnerable. Radare2's implementation of libmagic has been taken from OpenBSD, fuzzed, and enhanced. In my experience, Radare2 is also more reliable and easier to modify if needed.

In addition to the print magic (`pm`) command we can use the /m command which will run `pm` on every offset to find known magic signatures on raw files. This is useful for carving files from memory dumps, core images, and more.

When I examined the files contained within this sample of malware, I found some files with the .png extension that seemed to contain encrypted data.

Conclusion: The Triada Android Trojan's capabilities

After using Google to see what other researchers have published about the Tirada Android Trojan, I began to understand the real risks brought about by this malware and how it can harm users:

- Roots the device and modifies the Zygote
- Factory Reset will not remove the malware
- Seems to target pre-KitKat devices but affects all versions
- May ship some modular exploits to raise privileges

At this point, I've explained how to perform some very basic static analysis on a couple targets. As part of my analysis, I've confirmed that this is in fact a modular virus capable of downloading more binaries and applications

at runtime. But, we’re still far from understanding the malware in its entirety due to encrypted components, code loaded at runtime, installation of additional applications, networking communications, and more.
