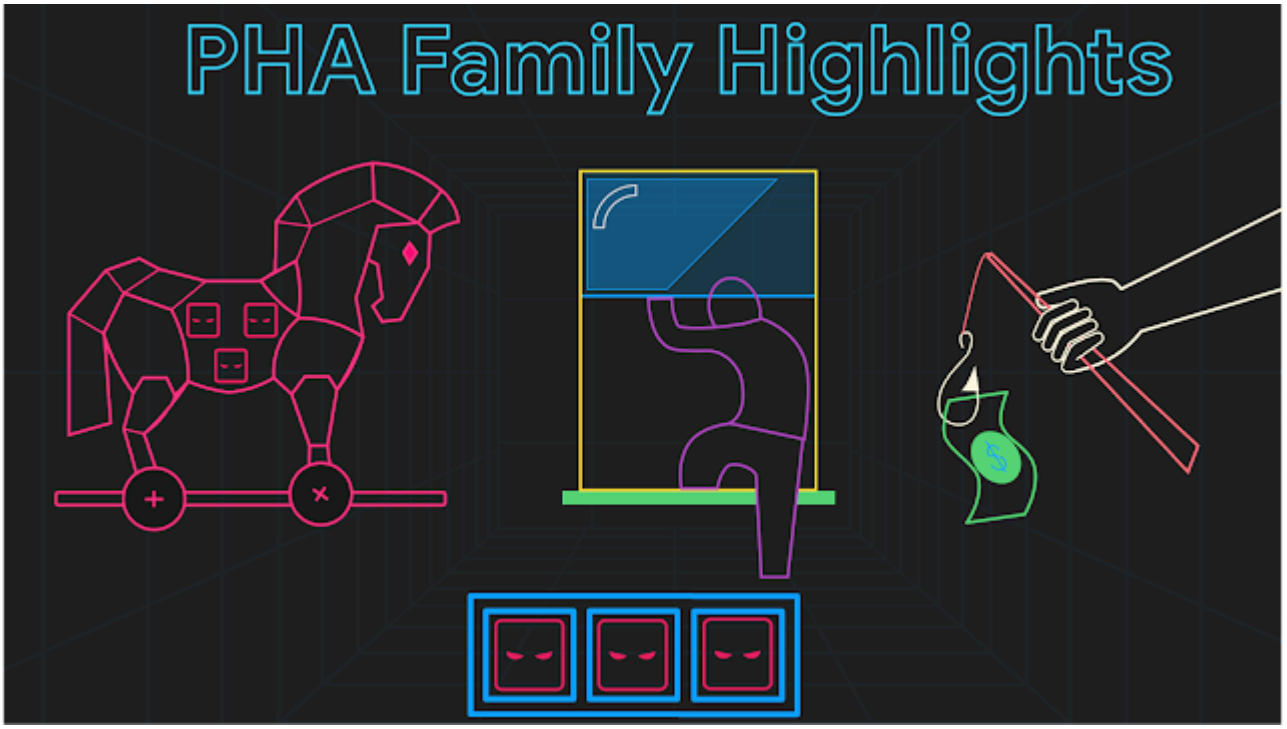
PHA Family Highlights: Triada

By Posted by Lukasz Siewierski, Android Security & Privacy Team

Published: 2019-06-06 · Archived: 2026-04-02 10:36:37 UTC

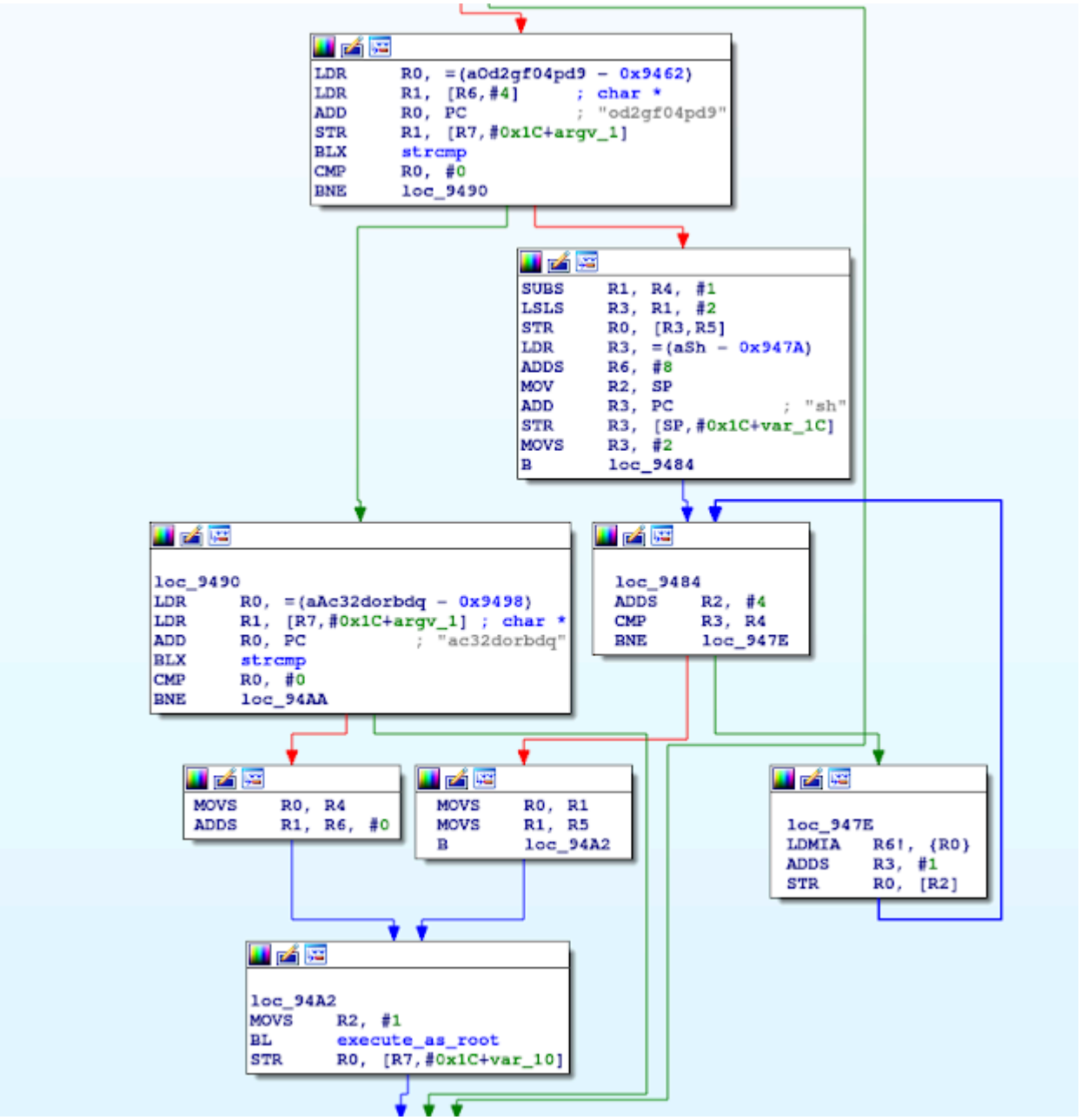


We continue our **PHA family highlights** series with the Triada family, which was first discovered early in 2016. The main purpose of Triada apps was to install spam apps on a device that displays ads. The creators of Triada collected revenue from the ads displayed by the spam apps. The methods Triada used were complex and unusual for these types of apps. Triada apps started as rooting trojans, but as Google Play Protect strengthened defenses against rooting exploits, Triada apps were forced to adapt, progressing to a system image backdoor. However, thanks to OEM cooperation and our outreach efforts, OEMs prepared system images with security updates that removed the Triada infection.

History of Triada

Triada was first described in a blog post on the Kaspersky Lab website in March 2016 and in a follow-up blog post in June 2016. Back then, it was a rooting trojan that tried to exploit the device and after getting elevated privileges, it performed a host of different actions. To hide these actions from analysts, Triada used a combination of dynamic code loading and additional app installs. The Kaspersky posts detail the code injection technique used by Triada and provide some statistics on infected devices at the time. In this post, we'll focus on the peculiar encryption routine and the unusual binary files used by Triada. Triada's first action was to install a type of superuser (su) binary file. This su binary allowed other apps on the device to use root permissions. The su binary used by Triada required a password, so was unique compared to regular su binary files common with other Linux systems. The binary accepted two passwords, `od2gf04pd9` and `ac32dorbdq`. This is illustrated in the IDA

screenshot below. Depending on which one was provided, the binary either 1) ran the command given as an argument as root or 2) concatenated all of the arguments, ran that concatenation preceded by `sh`, then ran them as root. Either way, the app had to know the correct password to run the command as root. This Triada rooting trojan was mainly used to install apps and display ads. This trojan targeted older devices because the rooting exploits didn't work on newer ones. Therefore, the trojan implemented a *weight watching* feature to decide if old apps needed to be deleted to make space for new installs. Weight watching included several steps and attempted to free up space on the device's user partition and system partition. Using a blacklist and whitelist of apps it first removed all the apps on its blacklist. If more free space was required it would remove all other apps leaving only the apps on the whitelist. This process freed space while ensuring the apps needed for the phone to function properly were not removed. Every app on the system partition had a number, or *weight*, associated with it. The weight was a sum of the number of apps installed on the same date as the app in question and the number of apps signed with the same certificate. The apps with the lowest weight were installed in isolation (that is, not on a day that the device system image was created) and weren't signed by the OEM or weren't part of a developer bundle. In the weight watching process, these apps were removed first, until enough space was made for the new app.



su binary accepts two passwords

In addition to installing apps that display ads, Triada injected code into four web browsers: AOSP (`com.android.browser`), 360 Secure (`com.qihoo.browser`), Cheetah (com.ijinshan.browser_fast), and Oupeng (`com.oupeng.browser`). The code was injected using the same technique described in our blog post about the Zen PHA family and in previously mentioned Kaspersky blog posts. The web browser injection was done to overwrite the URLs and substitute ad banners on websites with ads benefiting the Triada authors. Triada also used a peculiar and complex communication encryption routine. Whenever it had to send a request to the Command and Control (C&C) server, it encrypted the request using two XOR loops with different passwords. Because of XOR rules, if the passwords had the same character in the same position, those characters weren't encrypted. The encrypted request was saved to a file, which had the same name as its size. Finally, the file was zipped and sent to the C&C server in the POST request body. The example below illustrates one such request. The yellow bytes are the zip

file's signature of the central directory file header. The red bytes show the uncompressed file size of 0x0952. The blue bytes show the file name length (4) and the name itself (2386, a decimal version of 0x0952).

```
09 00 00 50 4B 01 02 14 00 14 00 08 00 08 00 4F ...PK.........O
91 F3 48 AE CF 91 D5 B1 04 00 00 52 09 00 00 04 ..H........R....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00 32 33 38 36 50 4B 05 06 00 00 00 00 01 00 01 .2386PK.........
00 32 00 00 00 E3 04 00 00 00 00       .2.........
```

The underlying data protocol changed periodically. It was either a simple JSON, a list of key-value pairs similar to the properties file, or a proprietary format as shown below.

```
[collect_Head]device=Nexus 5X
[collect_Space]xadevicekey=xxxxx
...
[collect_Space]collentmod=opappresultmode
[collect_Space]registerUser=true
[collect_End]
```

When Triada was discovered, we implemented detection that removed Triada samples from all devices with Google Play Protect. This implementation, combined with the increased security on newer Android devices, made it significantly harder for Triada to infect devices.

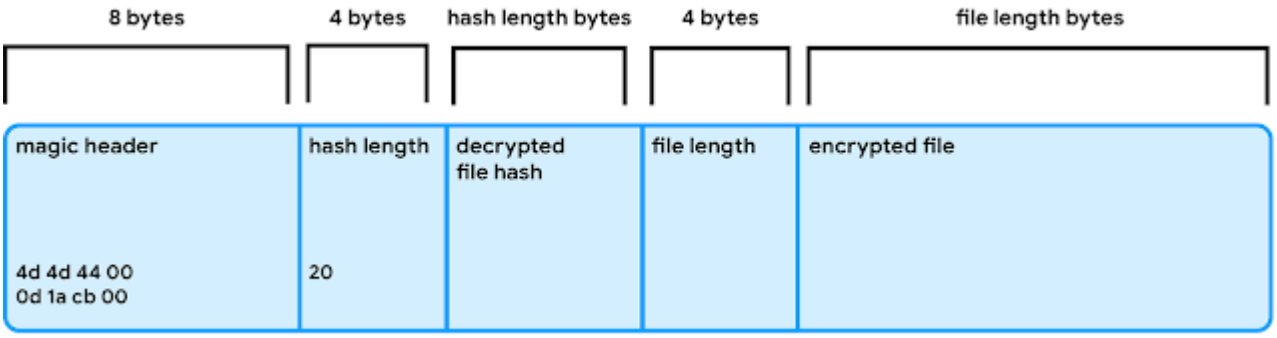
When rooting doesn't work…

During the summer of 2017 we noticed a change in new Triada samples. Instead of rooting the device to obtain elevating privileges, Triada evolved to become a pre-installed Android framework backdoor. The changes to Triada included an additional call in the Android framework log function, demonstrated below with a highlighted configuration string.

```
LABEL+13:
   V18 = -1;
LABEL_18:
   j___config_log_println(v7, v6, v10, v11, "cf89450001");
   if ( v10 )
```

This backdoored log function version of Triada was first described by Dr.Web in July 2017. The blog post includes a description of Triada code injection methods. By backdooring the log function, the additional code executes every time the log method is called (that is, every time any app on the phone tries to log something). These log attempts happen many times per second, so the additional code is running non-stop. The additional code also executes in the context of the app logging a message, so Triada can execute code in any app context. The code injection framework in early versions of Triada worked on Android releases prior to Marshmallow. The main purpose of the backdoor function was to execute code in another app's context. The backdoor attempts to execute

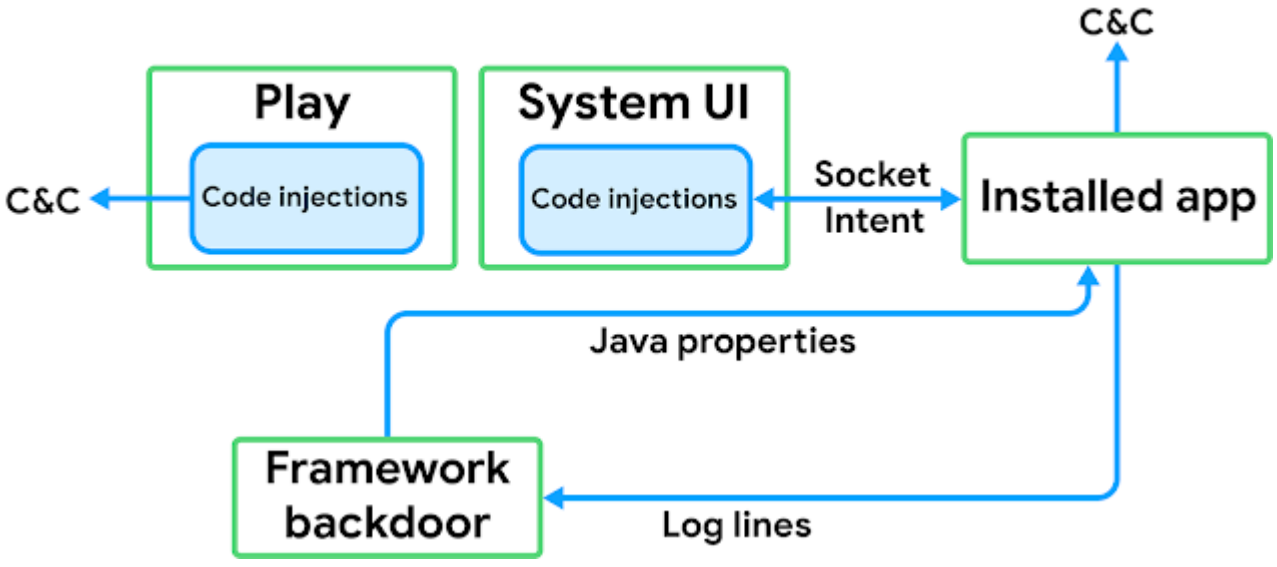
additional code every time the app needs to log something. Triada developers created a new file format, which we called MMD, based on the file header. The MMD format was an encrypted version of a DEX file, which was then executed in the app context. The encryption algorithm was a double XOR loop with two different passwords. The format is illustrated below.



Each MMD file had a specific file name of the format `<MD5 of the process name>36.jmd`. By using the MD5 of the process name, the Triada authors tried to obscure the injection target. However, the pool of all available process names is fairly small, so this hash was easily reversible. We identified two code injection targets: `com.android.systemui` (the System UI app) and `com.android.vending` (the Google Play app). The first target was injected to get the `GET_REAL_TASKS` permission. This is a signature-level permission, which means that it can't be held by ordinary Android apps. Starting with Android Lollipop, the `getRecentTasks()` method is deprecated to protect users' privacy. However, apps holding the `GET_REAL_TASKS` permission can get the result of this method call. To hold the `GET_REAL_TASKS` permission, an app has to be signed with a specific certificate, the device's platform cert, which is held by the OEM. Triada didn't have access to this cert. Instead it executed additional code in the System UI app, which has the `GET_REAL_TASKS` permission. The injected code returned the app running on top (the activity running in the foreground and being actively used by the device user) to other apps on the device. This app was exposed using two methods: an intent or a socket created for this purpose. When an app on the device sent the intent or wrote to a socket created by Triada's code injection, it received the package name of the app running on top. Triada used the package name to determine if an ad was displayed. The assumption was that if the app running on top was a browser, the user would expect to see some ads, so Triada displayed ads from the background. The second injection target was the Google Play app. This injection supported five commands and responses to them. The supported commands are shown below in Chinese, a language that was used throughout the Triada app and injection. English translations are given on the right.

1. 下载请求 2. 下载结果 3. 安装请求 4. 安装结果 5. 激活请求 6. 激活结果 7. 拉活请求 8. 拉活结果 9. 卸载请求 10. 卸载结果	1. download request 2. download result 3. install request 4. installation result 5. activation request 6. activation result 7. pull request 8. pull the results 9. uninstall request 10. uninstall result

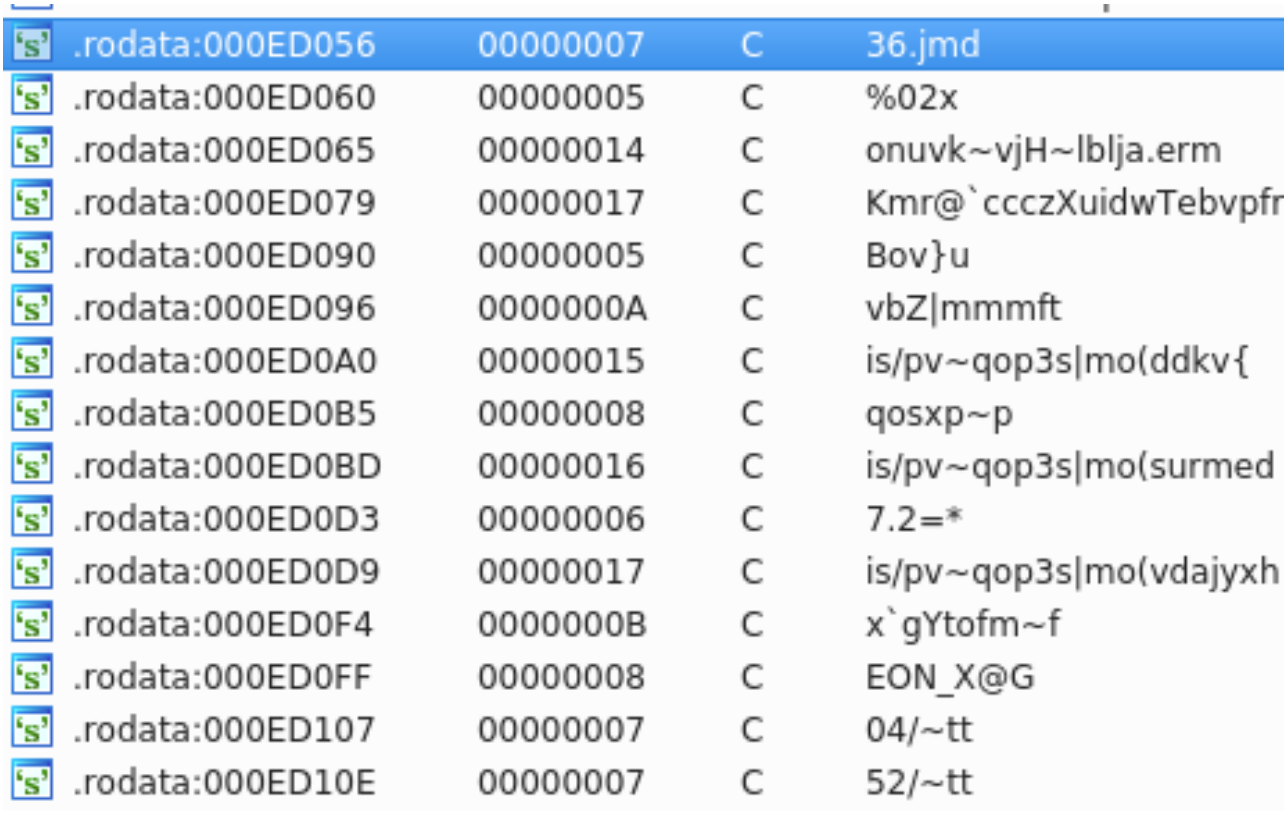
The commands trigger the heartbeat (pull request), download, installation, uninstallation (in the Google Play app context), and activation (the first execution) of the apps. In the Google Play app context, installation meant that Triada didn't have to turn on installation from unknown sources and all app installs looked like they were from Google Play. The apps were downloaded from the C&C server and the communication with the C&C was encrypted using the same custom encryption routine using double XOR and zip. The downloaded and installed apps used the package names of unpopular apps available on Google Play. They didn't have any relation to the apps on Google Play apart from the same package name. The last piece of the puzzle was the way the backdoor in the log function communicated with the installed apps. This communication prompted the investigation: the change in Triada behavior mentioned at the beginning of this section made it appear that there was another component on the system image. The apps could communicate with the Triada backdoor by logging a line with a specific predefined tag and message. The reverse communication was more complicated. The backdoor used Java properties to relay a message to the app. These properties were key-value pairs similar to Android system properties, but they were scoped to a specific process. Setting one of these properties in one app context ensures that other apps won't see this property. Despite that, some versions of Triada indiscriminately created the properties in every single app process. The diagram below illustrates the communication mechanisms of the Triada backdoor.



Communication mechanisms of Triada

Reverse engineering countermeasures and development

The Triada backdoor was hidden to make the analysis harder. The strings in the Android framework library that related to Triada activities were encrypted, as shown below.



.rodata:000ED056	00000007	C	36.jmd
.rodata:000ED060	00000005	C	%02x
.rodata:000ED065	00000014	C	onuvk~vjH~lblja.erm
.rodata:000ED079	00000017	C	Kmr@`ccczXuidwTebvpfr
.rodata:000ED090	00000005	C	Bov}u
.rodata:000ED096	0000000A	C	vbZ|mmmft
.rodata:000ED0A0	00000015	C	is/pv~qop3s|mo(ddkv{
.rodata:000ED0B5	00000008	C	qosxp~p
.rodata:000ED0BD	00000016	C	is/pv~qop3s|mo(surmed
.rodata:000ED0D3	00000006	C	7.2=*
.rodata:000ED0D9	00000017	C	is/pv~qop3s|mo(vdajyxh
.rodata:000ED0F4	0000000B	C	x`gYtofm~f
.rodata:000ED0FF	00000008	C	EON_X@G
.rodata:000ED107	00000007	C	04/~tt
.rodata:000ED10E	00000007	C	52/~tt

Android framework strings

The strings were encrypted using the algorithm of two XOR loops. However, the first highlighted string, `36.jmd`, wasn't encrypted. This is the MMD file name string mentioned before. Another anti-analysis measure implemented by the Triada authors was *function padding*, including additional exported functions that don't serve any purpose apart from making the file size bigger and the function layout more random with every compilation. Four types of these functions are shown in the screenshots below.

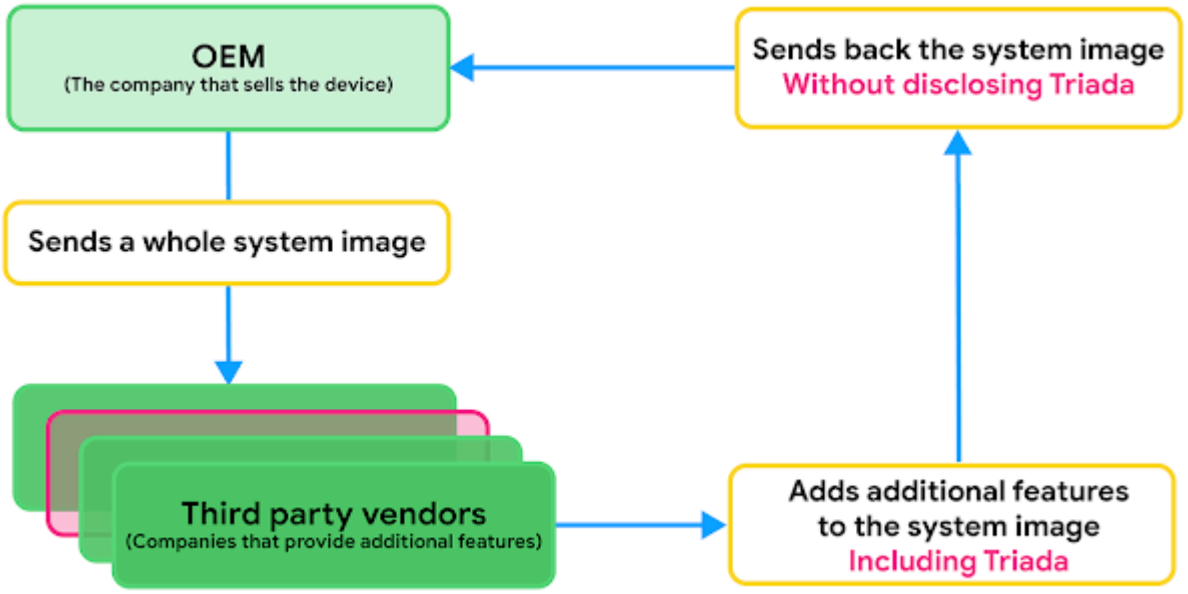
```
                EXPORT _f_rbaid6
                _f_rbaid6
03 48           LDR             R0, =(_GLOBAL_OFFSET_TABLE_ - 0xC14BE)
78 44           ADD             R0, PC  ; _GLOBAL_OFFSET_TABLE_
03 49           LDR             R1, =(dword_106120 - 0xF3E28)
08 18           ADDS            R0, R1, R0 ; dword_106120
41 69           LDR             R1, [R0,#(dword_106134 - 0x106120)]
01 31           ADDS            R1, #1
41 61           STR             R1, [R0,#(dword_106134 - 0x106120)]
70 47           BX              LR
                ; End of function _f_rbaid6

                EXPORT _f_cafbl6
                _f_cafbl6
42 69           LDR             R2, [R0,#0x14]
8A 43           BICS            R2, R1
42 61           STR             R2, [R0,#0x14]
70 47           BX              LR
                ; End of function _f_cafbl6

                EXPORT _f_cafbm5
                _f_cafbm5
C2 68           LDR             R2, [R0,#0xC]
0A 43           ORRS            R2, R1
C2 60           STR             R2, [R0,#0xC]
70 47           BX              LR
                ; End of function _f_cafbm5

                EXPORT _f_cafbm3
                _f_cafbm3
C0 69           LDR             R0, [R0,#0x1C]
70 47           BX              LR
                ; End of function _f_cafbm3
```

Example of function padding

One final interesting feature of Triada worth mentioning is the development cycle. By analyzing subsequent versions of the Triada backdoor (up to 1.5.1) we saw the changes in the code. In the newest version, they substituted MD5 with SHA1. This is used to hash the filenames, which come from a restricted pool of values. The newest version also encrypted the `36.jmd` string and introduced changes to the code for compatibility with Android Nougat. There are also code stubs pointing at the modification of the SystemUI and WebView Android framework elements. We couldn't find the code that was executed by these modifications, just code stubs suggesting more development in the future.

OEM outreach

Triada infects device system images through a third-party during the production process. Sometimes OEMs want to include features that aren't part of the Android Open Source Project, such as face unlock. The OEM might partner with a third-party that can develop the desired feature and send the whole system image to that vendor for development. Based on analysis, we believe that a vendor using the name Yehuo or Blazefire infected the returned system image with Triada.



Production process with malicious party

We coordinated with the affected OEMs to provide system updates and remove traces of Triada. We also scan for Triada and similar threats on all Android devices. OEMs should ensure that all third-party code is reviewed and can be tracked to its source. Additionally, any functionality added to the system image should only support requested features. It's a good practice to perform a security review of a system image after adding third-party code.

Summary

Triada was inconspicuously included in the system image as third-party code for additional features requested by the OEMs. This highlights the need for thorough ongoing security reviews of system images before the device is sold to the users as well as any time they get updated over-the-air (OTA). By working with the OEMs and supplying them with instructions for removing the threat from devices, we reduced the spread of preinstalled Triada variants and removed infections from the devices through the OTA updates. The Triada case is a good example of how Android malware authors are becoming more adept. This case also shows that it's harder to infect Android devices, especially if the malware author requires privilege elevation. We are also performing a security review of system images through the Build Test Suite. You can read more about this program in the Android Security 2018 Year in Review report. Triada indicators of compromise are one of many signatures included in the system image scan. Additionally, Google Play Protect continues to track and remove any known versions of Triada and Triada-related apps it detects from user devices.

Source: https://security.googleblog.com/2019/06/pha-family-highlights-triada.html