2019

Android ransomware trends and case studies: A reverse engineering approach

Chenliang Xu
Iowa State University

Android ransomware trends and case studies: A reverse engineering approach

by

Chenliang Xu

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Yong Guan, Major Professor
Daji, Qiao
Jaeyoun Kim

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this thesis. The Graduate College will ensure this thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2019

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank my committee chair, Guan Yong, and my committee members, Daji Qiao, and Jaeyoun Kim, for their guidance and support throughout the course of this research.

In addition, I would also like to thank my friends, colleagues, the department faculty and staff for making my time at Iowa State University a wonderful experience. I want to also offer my appreciation to those who were willing to participate in my surveys and observations, without whom, this thesis would not have been possible.

Acknowledgments

ABSTRACT

With the rapid development of science and technology, the mobile device is becoming more and more powerful. However, technology is a two-edged sword, mobile devices also bring security risks. Malware or ransomware is not just for PC, but also a big threat to mobile device security. Because of these malicious applications, the user's mobile device may be locked, files may be encrypted, and even personal information can be exposed in danger. Therefore, more researches and analysis on currently popular ransomware are necessary. This paper is going to conclude the taxonomy of Android ransomware in terms of the types of trend of Android ransomware, the locking/unlocking techniques, and the password-generate techniques of Android ransomware. We also performed both statistic and dynamic analysis on three typical Android applications that carry ransomware, using reverse engineering approach. Furthermore, since there are a great number of ransomware that we found are packed by third-party packer companies, this paper will include two separate chapters talking about a few approaches on unpacking and recovery, to support our security experiments. Lastly, in order to support some dynamic experiments in our team, this paper is going to contribute a general approach with a simple example showing how to recover an unpacked app to make it run as normal.

CHAPTER 1. INTRODUCTION

Ransomware is one type of malware that intrudes the system using the malicious code, it can encrypt critical files, threaten users and request money for decryption, and give monetary damage to users (En.wikipedia.org, 2019) (Song, Kim and Lee, 2016). It was widely spread mainly on PC and website in the last decade. With the rapid growth of mobile device market such as smartphone, tablet, smart watch, the target of ransomware has been extended to the mobile device in the recent years. As Table. 1 shows, Android has taken 85.0% of Smartphone OS Market Share in 2017 Q1 (IDC: The premier global market intelligence company, 2019). The reason that Android OS is such popular can include (1) Global partnerships and large installed base; (2) Powerful development framework; (3) Open market for distributing apps (Yang, 2015). Being a popular OS also expose itself in danger, according to a mobile threat report, there is an estimate 75 % increase in Android mobile malware encounter rates in the United States compared to 2013 (Lookout.com, 2019). According to ESET LiveGrid, the number of Android ransomware detections has grown in year on year comparisons by more than 50%, with the largest spike in the first half of 2016 (Lipovsk, Tefanko and Brania, 2017). Nowadays, people rely on mobile devices such as smart phones and tablets much more than their personal computers and choose to store critical personal information such as bank account, credit card information and personal photos in the mobile device. Therefore, it is necessary to have more people working on the analysis of Android ransomware applications to prevent loss of data and leak of personal information.

Table 1. Smart Device OS Market Share				
Period	**Android**	**iOS**	**Windows Phone**	**Others**
2016Q1	83.4%	15.4%	0.8%	0.4%
2016Q2	87.6%	11.7%	0.4%	0.3%
2016Q3	86.8%	12.5%	0.3%	0.4%
2016Q4	81.4%	18.2%	0.2%	0.2%
2017Q1	85.0%	14.7%	0.1%	0.1%

Since the first appearance of Android ransomware in 2012, it has been evolving into two major types and a lot of families in five years, one of the major types is screen locking, and the other one is file encryption. This paper is going to analyzing three cases of Android ransomware, the first two are lock-screen ransomware, which basically locks the device on a certain page, and ask the victims to pay the ransom for unlocking. The third case is an Android "WannaCry" ransomware, which encrypts all the media files using AES cipher, and also charges the victim money for decryption. All of these three ransomwares have been successfully cracked and tested on Android simulator. After the real-case analysis, we also concluded some general solution and suggestions to prevent some regular Android ransomware which reduces the risks of accidentally installing fake applications that carry ransomware and losing important data.

In section II, we are going to introduce the taxonomy of ransomware application, including the trend of Android ransomware, the principle of device locking, and the different unlocking methods that are commonly used. In section III, we will describe three different ransomware cases, from both static and dynamic aspects, analyzing and showing the specific solution to each case. In section IV, we are going to do a short discussion about what we

observed from the three cases. In section V, we will provide some possible general solutions about preventions that concluded from others' work.

In the next section, we are going to present some related research or analysis performed on both PC and Android ransomware.

CHAPTER 2. RELATED WORKS

Android Defender

Android Defender was one of the first ransomware family found on Android device in mid-2013, which is a classic

fake antivirus (FakeAV) application which is found to be a ransomware. The FakeAV software is a type of scam using malware that intentionally misrepresents the security status of a computer and attempts to convince the user to purchase a full version of the software in order to remediate non- existing infections (Symantec Security Response, 2019). This application performs a fake virus scan and keeps informing the user that there are threats on the device, by showing the real name of files in the SD card to make it more believable. The warning from this ransomware application is endless until the user chooses to pay $40 to $100 for an activation code to remove “threats”. This ransomware is kind of “soft” locking the victim’s devices by continuously popping out the warning, which practically makes the device unusable. According to a full analysis that was performed by Naked Security (Ducklin, 2019), this ransomware is just simply picking some random virus names to display on the device by using *Math.random()* function, even if the device is certainly uninfected with any virus at all. The activate code is hard-coded and can be easily traced using any Android decompiler. After activating, we can clean it up by simply removing it.

Simplocker

The Simplocker is the first file-encrypting found on Android in May 2014. This ransomware was firstly launched in Rus- sia. After installation, a message written in Russian will be displayed on the screen, a separate thread in the background will start to encrypt files with regular media extensions such as JPEG, JPG, PDF, DOC, TXT, AVI, and MP4 and so

on, as a ransomware, it charges 260 UAH from decryption (Securehoney.net, 2019). Since it is a early development version of crypto ransomware, it is very easy to crack. According to the analysis provided by Secure Honey (Securehoney.net, 2019), since the the cipher key is hard-coded in plaintext, and the decryption function can be found in the same class as the encryption function if using a good decompiler. It is totally possible to write a separate decryption program under any IDE.

WannaCry

In May 2017, a PC ransomware WannaCry was spread widely and quickly on Windows machine all over the world. It makes use of the vulnerabilities in the Windows Server Message Block (SMB) to rapidly spread by worm (Bobao.360.cn, 2019). This ransomware encrypts the system files and then demands various Bitcoins payments equivalent from $300 to $600 for decryption. According to a report provided by 360 Security Team(Blogs.360.cn, 2019), this WannaCry ransomware encrypts the file system recursively using a combination of the RSA and AES encryption algorithms, where the RSA encryption comes directly from the Windows Crypto API, but the AES encryption is implemented by a third-party and is statically linked within this ransomware, which makes it impossible to crack the encryption process without knowing the private key. After encryption, this ransomware fills random bytes to the targeted files with certain extensions, which are considered as super important files, it then moves these files to a temporary working directory and deletes them. After this process, the names of metafiles would be changed, which increase the difficulty to recover for some regular data recovery software. Due to these restrictions, current solutions cannot ensure to recover all the files, but only most part of the files depending on the system environment.

In the next section, we are going to introduce the categories of Android ransomware, including technical principles of locking an Android device, and the unlocking techniques that are commonly used by attackers.

CHAPTER 3. TAXONOMY

Types of Android Ransomware

There are two general categories of ransomware on Android: 1) Lock-screen ransomware, 2) Crypto-ransomware.

In lock-screen types of ransomware, the hijacked resource is access to the compromised system. In file-encrypting "crypto- ransomware" that hijacked resource is the user's files (Lipovsk, Tefanko and Brania, 2017). Both types of the ransomware have been found on the Windows platform previously, and all of them have been causing trouble to both individuals and business.

Trend of Android Ransomware

The Android ransomware trend is shown in the Figure 1. We can see the first ransomware on Android was found since 2012, there was a fake antivirus application called Fake AVs, which performs a fake scan on the device, and then charge money for removing threats. In 2014, the new generation Android police ransomware came out, which is also known as lock-screen ransomware, and it is the primary type of ransomware on the market so far. In the same year, the first file-encrypting crypto-ransomware appeared as Android Simplocker. Android ransomware is continuously evolving and forming new families years after years.

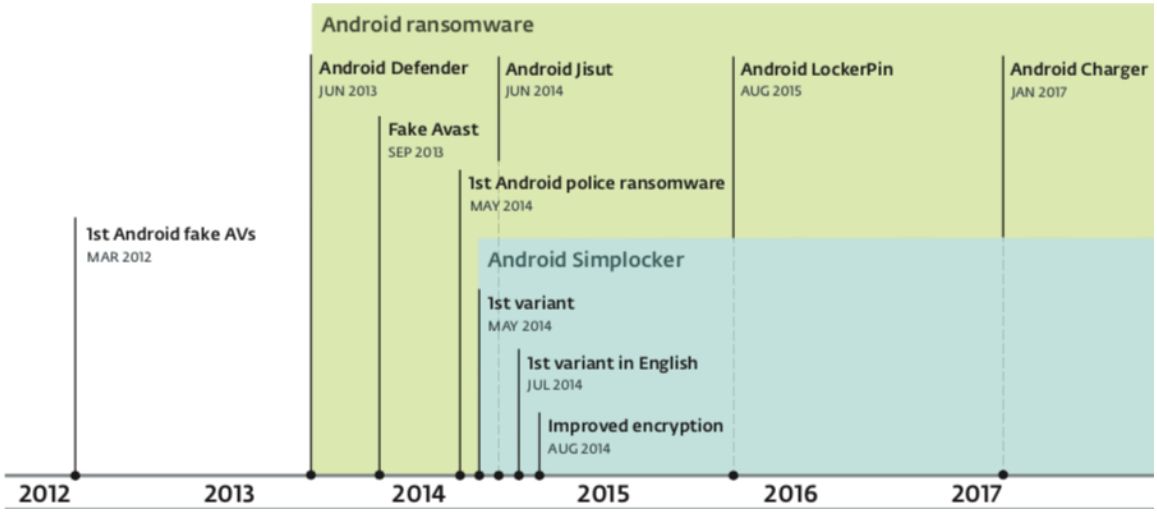

Figure 1. Android ransomware chronology

Most of currently existing Android ransomware applications are lock-screen ransomware because this type of ransomware is relatively easy to implement, possible to reproduce, and pretty effective. What it can do is to lock the screen, make the device totally unusable, then charge money for unlocking service. On the other hand, crypto-ransomware is not so popular until the middle of 2017, right after the burst of WannaCry ransomware on PC, some ransomwares are found to use the similar strategy to encrypt user's files, and charge money for decryption, just like a mobile WannaCry. We cannot conclude the techniques of crypto-ransomware because there are so different and there are limited samples, but we are going to analyze one crypto-ransomware case in the Case Studies section. In the next a few paragraphs, we are going to go over different technology principles of lock-screen types of ransomware that conclude by (Blogs.360.cn, 2019).

Techniques of Lock-screen Ransomware

Attackers can make use of WindowManager.LayoutParams flags attribute. By creating a floating window using *ad- dView()* method, setting the flags attribute of

WindowMan- ager.LayoutParams, such as “FLAG FULLSCREEN”, and “FLAG LAYOUT IN SCREEN”, together with a permission “SYSTEM ALERT WINDOW”, can bring and lock the float- ing window to top in full screen, which locks the screen and makes the device not work properly.

Lock-screen can also be realized by monitoring the top activity by TimerTask, if it is detected is not the lock- screen activity itself, I will restart and set the addFlag to FLAG ACTIVITY NEW TASK, which can overwrite and kill the process of the original activity, then the lock-screen activity is locked at the top, and thus hijacked the device. Fortunately, Android 5.0 or above versions have taken a protective mechanism to prevent this attack.

Another way to lock a screen is to simply shield the home/return button, volume button, and other virtual buttons by rewriting the onKeyDown method, which results in the effect of no response upon pressing button, to lock the screen.

If a device has not been set an unlocking password yet, by inducing the user to activate the Device Manager/root permisson during installation, it can force the device to set an unlock password without noticing the user, causing the user not able to unlock the device. if the device has been previously set a password, with the root permission, it can also modify the */data/system/password.key* file to remove or change the password.

Unlocking Techniques

After locking the device, the attackers will charge money for unlocking, there are a few ways that are commonly used for unlocking.

1. The most straight forward way is to enter the unlock password to unlock, which means the user redeems the unlock code or password by paying money, and then enter the code directly in the lock-screen page to unlock the device. This is one of the most common ways to unlock.

2. There are a few cases that use SMS or network to unlock. SMS control unlock, that is, by receiving the specific SMS number or SMS content to unlock the device remotely. This unlock method may reveal the attacker's phone number.
3. For the network unlocking, attackers use anonymous communication technologies such as Tor browser to remotely control unlocking service in order to hide their personal information. This anonymous technique was originally designed to protect the privacy, but now it is abused by a lot of malware.
4. Some attackers even made the unlocking applications for specific ransomware, to reduce the complexity of unlocking process.

In the next section, we will present case studies of three different Android ransomware applications. For each of them, we will describe how it affects your device, the process of cracking it, and an effective solution.

Password Generate Techniques

Among these unlock method, the most common way is to enter a password, so here I conclude some password generate methods

1. The password can be just hardcoded in plaintext in the code, which will be very easy to find.
2. The password can be generated with a specific random serial number, which most likely the attackers will let victims know the serial number and ask to pay the ransom along with the serial number, so that they will send you back the password that corresponds to the serial number.
3. The serial number and password can be both generated randomly, with no relations, but stored as a key-value pair, and this pair may be encrypted and sent to the attackers through emails or HTTP requests.

CHAPTER 4. CASE ANALYSIS

This section is going to perform statistic and dynamic analysis on three currently existing Android ransomware applications using reverse engineering approach. For statistical analysis, we will be using a decompiler tool called Jeb (Pnfsoftware.com, 2019), which is a powerful tool that can converts bytecode to java and extracts all the classes, manifest, certificate and all the assets files. We also use one graphic tool from Soot (GitHub, 2019), which is one basic package of a taint-analysis tool Flowdroid (Secure Software Engineering, 2019). Soot is a Java optimization framework, which provides intermediate representations for analyzing and transforming Java bytecode. to generate control flow graphs for some functions. For dynamic analysis, we use the Bluestack (Bluestacks - The Best Android Emulator on PC as Rated by You, 2019) to simulate Android environment. All the ransomware samples are downloaded from Janus, which is a mobile security platform that allows users to upload suspicious application and perform security scan to determine if the application is safe or not. It also keeps updating their database and encourage security experts and reverse engineer to contribute.

A. SB Modifier

1) Ransomware Description: SB Modifier is an Android game modifier that was supposed to be a third-party application that allows users to modify the game data. However, this sample APK was injected malicious resources and repackaged. Fortunately, the encryption method is relatively easy, so it is a good place to start.

When installing SB Modifier, it will ask the user the administration permission, and once you allow this permission, your device will restart and gets locked by then. On the lock screen, you can find a unique serial number (It has been tested that the serial number is

unique for a device, and will not change upon restarting the device), an input box which is extremely hard to find, an OR code for payment, and some contact information. Like other lock-screen application, all the buttons except the power button has been tampered, so you can go nowhere.

2) Analysis: Our goal here is to find the password using reverse engineering approach with the de-compiler tool Jeb. After importing this APK to Jeb, we found there are only a few ordinary classes such as *BuildConfig.java*, *R.java*, *a.java*, *b.java*, and *c.java*, and there is no malicious or suspicious code founded in this classes after simply scanning through each class. We then took a close look at these classes and found in class *c.java*, there is a suspicious line of code in the *private void d( )* function shows in Figure 2.

```
private void d(String arg13) throws IOException {
    FileOutputStream v3 = new FileOutputStream(arg13);
    InputStream v2 = this.getAssets().open("ijm-x86.so");
    byte[] v4 = new byte[1024];
    int v5;
    for(v5 = v2.read(v4); v5 > 0; v5 = v2.read(v4)) {
        v3.write(v4, 0, v5);
    }

    v3.flush();
    v2.close();
    v3.close();
}
```

Figure 2. void d( ) function

The author was trying to open a library file *"ijm-x86.so"* which is located in the Asset directory. It seems to be normal to utilize functions from the library, however, this could be a good place to store the malicious code. Therefore, we located this library file and decompile it again using Jeb, there we found a group of very typical locking-screen virus classes shows in Figure 3.

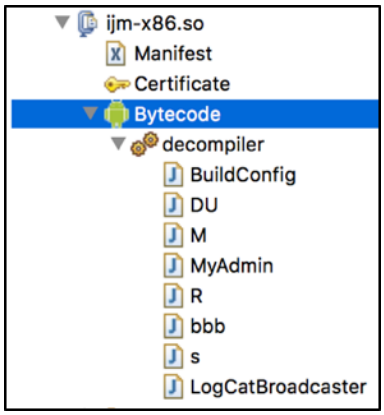

Figure 3. ijm-x86.so

In the s.java class, we found a void function *onCreate( )* (Figure 4), which generates the password.

```
@Override public void onCreate() {
    s v0 = this;
    super.onCreate();
    v0.pass = ((long)(Math.random() * (((double)155))));
    v0.passw = new Long((v0.pass + (((long)7176))) * (((long)7)));
    v0.des = new DU("flower");
    s v3 = v0;
```

Figure 4. void onCreate()

3) Result: The password generation procedure is relatively easy, the “pass” is found to be the unique serial number that randomly generated when this application is installed (It is 79 in our case), the “passw” is the password we are looking for, and it just does a lot of bit math on the serial number, so we can calculate the unlocking password very easily by plugging in the serial number.

This ransomware sample is a very typical one, the encryp- tion algorithm pretty easy, but the actual difficulty of finding the solution is to locate the malicious code.

B. Red Wars

1) Ransomware Description: Red Wars is an application that was designed as a plug-in to some social chatting application such as WeChat, to help user grab Red Pocket (aka money) as fast as it can. As an external plug-in, these applications like Red Wars are required to take the root permission because they need to take control of your device to perform some scripts. Although in the recent Android version, the system will warn you the consequences you will get if you give the root permission, but people download and install this application with their own intentions, so it is highly possible that even a user with a little knowledge could get fooled.

During the process of installation, it asks you a bunch of permission such as root permission, internet permission, and boot receiver permission, if we accept it, then the device will restart and enter a lock screen. There are only an input box and a button on the screen with an anime picture as a background. it looks like a semi-finished product because there is no payment method or contact information on the screen, but it does not affect our unlocking process since we are not going to pay them anyway.

2) Analysis: After importing this application to Jeb, besides the package of malicious classes, we found a separate package that contains a class named "ADRT" (Shows in *Figure 5*), this is an evidence that this application was developed with AIDE (Android Integrated Development Environment), which is an IDE that user can development Android application on an Android device, this environment allows unprofessional people can modify or create android applications, inject malicious code to a popular and widely used application, or simply modify an existing ransomware by changing the key value and contact/payment information to create their own ransomware. This explains why this application looks like a

semi-finished product and some code with weird logic which I will introduce in the next a few paragraphs.

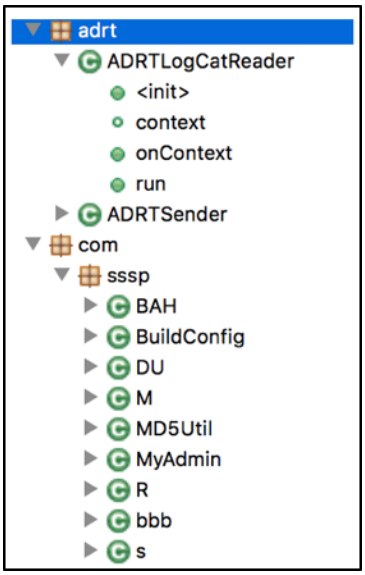

Figure 5. Red Wars classes

In the *MyAdmin.java* class, we found a *CharSequence* (*Figure 6*) function that contains code to change the password:

```
@Override public CharSequence onDisableRequested(Context arg13, Intent arg14) {
    String v7 = M.getsss(BAH.getString(arg13.getResources().openRawResource(2131099650)).
                replaceAll("\n", ""));
    this.getManager(arg13).lockNow();
    this.getManager(arg13).resetPassword(v7, 0);
    return super.onDisableRequested(arg13, arg14);
}
```

Figure 6. CharSequence and onDisableRequested()

First of all, it locks your device when the user requests to disable this application, and it resets the password with String *v7*. Lastly, it calls super and lets system to process this request. There is another function called *onPasswordChanged( )*, which was implemented very similar as *onDisableRequest( )*, both of them are made to prevent user from unlocking.

What we are interested in is the String *v7*. The key of that is the function *getsss( )* in class *M.java*, *getString( )* in class *BAH.java*, and the Raw Resource *2131099650*.

Let's analyze it from inner to outer. To figure out the Raw Recourse, we need to check the class *R.java*. In there, we found an inner class (*Figure 7*)

```
public final class raw {
    public static final int bahk = 2131099648;
    public static final int gdmm = 2131099649;
    public static final int pin = 2131099650;

    public raw() {
        super();
    }
}
```

Figure 7. class raw

We can see that the raw data *2131099650* we found matches the integer *pin*. Thanks to Jeb, we can also explore the resources such as *drawable*, *layout* and *raw* files along with the Apk. In the raw directory, there are 3 text files as the *Figure 8.* shows. What we are interested in is the *pin.txt*, and indeed it stores pin code: *Cj09UWU1bFhl*.

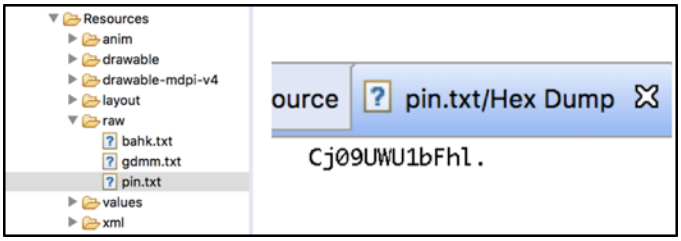

Figure 8. pin.txt

The *getString( )* function in *BAH.java* class is nothing but changing the argument one character by one character to string, so our next goal is to figure out what *getsss( )* does.

In the *M.java* class, we found the String type *getsss( )* function shows in *Figure 9*. There are a few Chinese characters with no special meanings, they are used as input for encoding and decoding only.

```
public static final String getsss(String arg21) {    //arg21 = pin = Cj09UWU1bFhl
    String v2 = new String(Base64.encode("by:彼岸花
                                qq:1279525738".getBytes(), 0));
    CharSequence v3 = v2.subSequence(3, 4);
    CharSequence v4 = v2.subSequence(4, 5);
    return new String(new String(Base64.decode(new StringBuffer().append(new
        StringBuffer(new String(Base64.decode(arg21.replaceAll(v3, "三生石畔
        ").replaceAll(v4, "彼岸花开").replaceAll("三生石畔", v4).replaceAll("彼岸
        花开", v3).toString(), 0))).reverse()).append("").toString().toString(), 0)));
}
```

Figure 9. String getsss()

To make it clear and readable, I rewrite the code in Eclipse. In the following paragraph, I will explain every line with my rewriting and the output I got from Eclipse.

The first line (*Figure 10*) is taking a string's byte value as input, and encoding using Base64 standard.

```
String input = "by:彼岸花 qq:1279525738";
String s1 = Base64.getEncoder().
        encodeToString(input.getBytes(StandardCharsets.UTF_8));
```

Figure 10. Base64 Encoding

The output of s1: "*Ynk65b285bK46IqxIHFx0jEyNzk1MjU3Mzg*"

After that, it takes the sub-sequence from index 3 to 4, and from index 4 to 5 (*Figure 11*), where the begin index is inclusive and the end index is exclusive. Therefore, it turns out that it only takes the single character of index 3 and 4 of the string s1.

```
String s2 = (String) s1.subSequence(3, 4);
String s3 = (String) s1.subSequence(4, 5);
```

Figure 11. Taking Sub-sequence

The output: s2 = "6", s3 = "5".

The last line is a very long and complex but ridicules string builder. I will separate it into two parts in my rewriting:

The first part (*Figure 12*) is the inner string buffer, there are two Chinese terms, and I replace the first term with “aaa”, and the second term with “bbb”. This replacement will not affect the output of the code as you will see what it actually does.

```
String s4 = pin.replaceAll(s2, "aaa").replaceAll(s3, "bbb").
        replaceAll("aaa", s3).replaceAll("bbb", s2);
```

Figure 12. Personal Encryption Algorithm

This part replaces all the “6” (s2) in the pin with “aaa”, then replace all the “5” (s3) in the pin with “bbb”, then replace all the “aaa” with “6”, and finally replaces all the “bbb” with “5”. What it does is just swap all the “5” and “6”, and the hilarious point is that there are no “5” or “6” in the pin at all. Thus, the output s4 is still the pin, which is “*Cj09UWU1bFhl*”.

Next, it applies the Base64 decoder of s4 (*Figure 13*), which makes s5 = “==*Qe5lXe*”, then it takes reverse of s5, so s6 = “*eXl5eQ*==”, and finally, it decodes s6 again, so the final string s7 = “*yyyy*”, which is the password to unlock the screen.

```
String s5 = new String(Base64.getMimeDecoder().decode(s4), StandardCharsets.UTF_8);
String s6 = new StringBuffer(s5).reverse().toString();
String s7 = new String(Base64.getMimeDecoder().decode(s6), StandardCharsets.UTF_8);
```

Figure 13. Base64 Encoding

3) Result: So far, we have successfully cracked this lock screening ransomware. We can see this kind of ransomware is easy to reproduce, just by simply changing the pin number, and the non-sense in swapping s2 and s3 indicates this code has been modified and reproduced. This is a very dangerous trend since people with not enough knowledge about software engineering can develop their own ransomware easily, and people who are expert can sell their reproducible ransomware.

C. King of Glory - Beta

1) Ransomware Description: King of Glory is a very famous mobile game in China operated by Tencent Inc, a large number of popularities through all ages and all genders play

this game, there is even e-sports competition holding in a lot of cities. Being such a hot online mobile game, many third- party add-ons and plug-ins applications have been developed in Android market, which requests administrator permission to take control of the device, meanwhile, it leaves the device in danger.

In June of 2017, right after the exposure of WannaCry ransomware in PC, there are a few King of Glory related Android applications named “King of Glory - Assist” or “King of Glory - Beta” spreading in Chinese Android market, which carries the mobile version of WannaCry ransomware. In the following a few paragraphs, I will present the case study of “King of Glory - Beta”.

Before installation, I stored some photos and Microsoft Word documents in the media storage. During the installation, it will ask administrator permission as expected. For the first time opening this application, it will enter a loading screen, and says “First time using, exporting configuration files...” in Chinese. This process takes quite an amount of time, depending on the usage of your storage, because it is actually traversing your directories and performing encryption on your files. When this “configuration” process finishes, it will enter a classic WannaCry page, with some description written in Chinese, as shown in *Figure 14*.

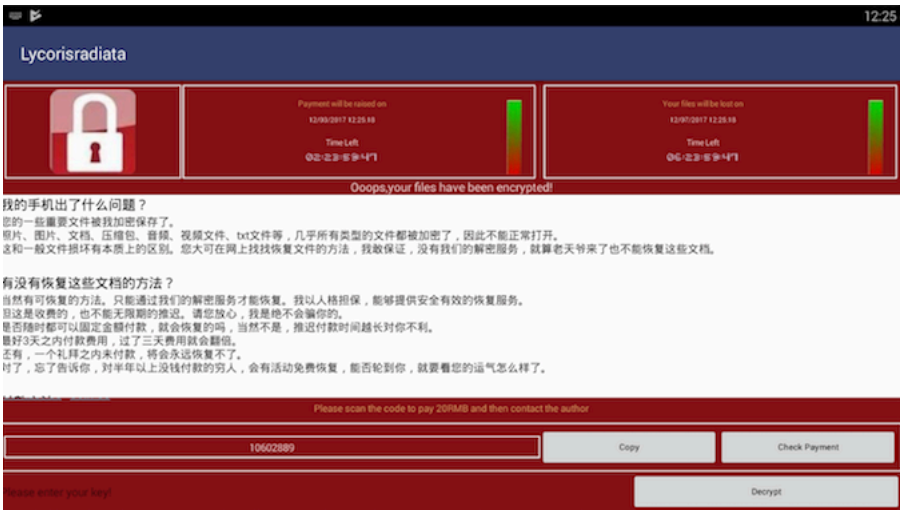

Figure 14. King of Glory - Beta, WannaCry Page

The Chinese description is just a translation of the original English version WannaCry, which basically says that some important files of the device such as photos, pictures, documents, compressed files, music, video files and text files have been encrypted so that you cannot open it anymore. it tries to convince the victims that the only way to recover the files is to contact them and pay for the decryption. It has a three-day countdown to raise the payment, and a seven-day countdown to start deleting the files. There is a pin code in the bottom left, which is 10602889 in this case, and they asked victims to send them the pin code along with the payment so that they will provide the decryption key. The payment can be processed in Alipay, WeChat pay, and QQ pay, which are three major mobile payment applications in China. The developer left their OR codes for those three payments, and charges for 20 Chinese dollars for decryption within three days.

Unlike the other lock-screen ransomware, this WannaCry page will not lock the screen. We can still go back to home screen and run other applications and settings, however, this application will send out a warning message: “Do not close or uninstall this application, otherwise, your files may be lost forever”. Therefore, I switched out without closing it and

checked the Media Manager, where I stored pictures and documents in, and found all the files are simply invisible in Bluestacks simulator, I believe this is because the encrypted files format is unable to view in this Media Manager. Anyway, all the media files are "gone" by now, so we need to proceed to analyze this WannaCry ransomware.

2) Analysis: First of all, I imported this ransomware APK to decompile every class. There are only 7 classes (*Figure 15*) in total, and the naming of the classes is very similar to the other samples I analyzed in previous sections.

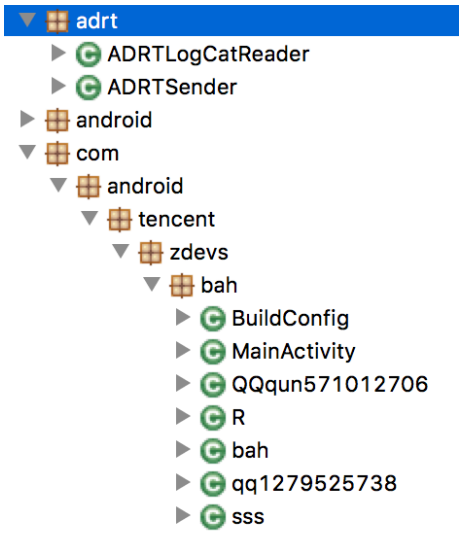

Figure 15. Classes of King of Glory - Beta

Usually, we should check the MainActivity.java class first, but by experience, I went for the sss.java class, because assuming it comes from the same author, then he or she usually store the malicious code in this class. As we expected, in the sss.java class, we found functions such as void delete *Dir()*, void *deleteDirWithFile()*, File *encryptFile()*, and File *decryptFile()*. From these delete functions, we found that it not only deletes files from DCIM (Data Center Infrastructure Management), it also accesses some Cloud drive, for example Baidu Net Disk, and delete files from it. In the encryption function, it takes files as input, and

perform encryption using AES Cipher. In order to solve this ransomware problem, what we are interested most the decryption function.

The decryption function is very long, and there are a lot of "goto" calls with "labels". It might be the author's intention to use "goto", because most Java IDE cannot compile it, and so fewer people can crack his or her decryption method. Or it might be the decompiler issue, when it gets converted from byte code, which has a lot of "goto" and "jump", it will not re-organize to a nice loop with if statements. We can certainly rewrite the encryption code and run some tests in Eclipse, but the relations between each "goto" and "labels" are very complex, it will take a long time to re-organize it by hand, and even if we rewrite it, we cannot ensure the correctness and credibility.

Therefore, in order to see the relations between these "*goto*" and "*labels*", we installed an open-source decompile tool called "*Soot*". One tool of Soot is the CFG Viewer, which allows taking input as Android Apk, and then it will return the control flow graph of every function in that Apk. The complete control flow graph of File *decryptFile()* function is shown in *Figure 16*.

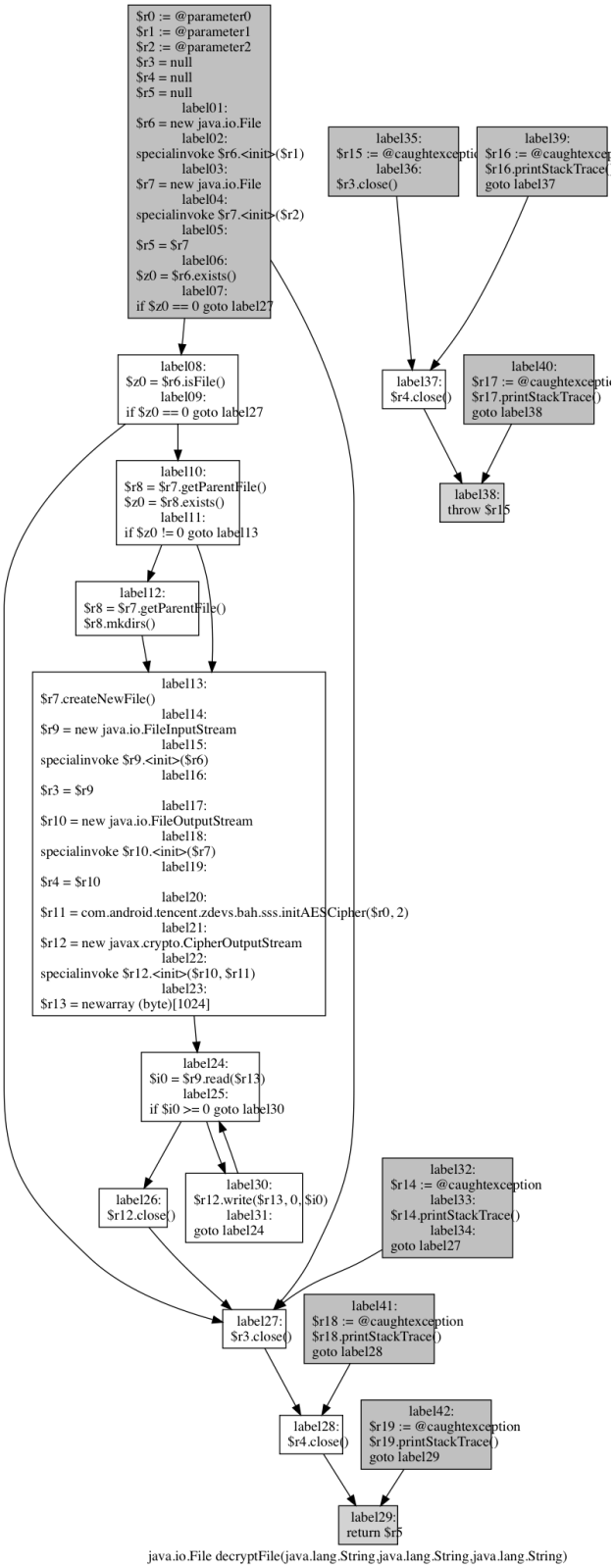

Figure 16. Control Flow Graph of decryptFile()

Now we have the control flow graph of the decryption function, but we still do not understand the meaning of these randomly named variables and arguments, and how the parameters are passed. Thus, I started to analyze the parameter of the decryption function first. As *Figure 17* shows, the decryption function takes three String type arguments as parameters, to find out what are they represent, we need to find where the decryption function is called.

```
public static File decryptFile(String arg26, String arg27, String arg28) {
    FileOutputStream v21_4;
    FileInputStream v21_3;
    int v17;
    byte[] v16;
    CipherOutputStream v15;
    File v8;
    String v2 = arg26;
    String v3 = arg27;
    String v4 = arg28;
    FileInputStream v6 = null;
    FileOutputStream v7 = null;
```

Figure 17. Part of decryptFile()

In the same class, we found a function void *jj()* calls decryption function. It also has three arguments, they are File *arg12*, String *arg13*, and Int *arg14* respectively, as shows in *Figure 18*. From this function, we can see that the *jj()* function is passing its parameters to *decryptFile()*, it converts the File *arg12* to String type, and call *getsss()* on the String *arg13*. The *getsss()* function is just a personal encryption that performs some string modification on the arguments. The third argument of *decryptFile()* is a substring of the file name which from 0 to certain length, depending on the variable *hzs* in the *MainActivity.java* class.

```
public static void jj(File arg12, String arg13, int arg14) {
    File v0 = arg12;
    String v1 = sss.getsss(arg13);
    if(arg14 == 0) {
        String v4 = v0.toString();
        sss.decryptFile(v1, v0.toString(), v4.subSequence(0, v4.length() - MainActivity.hzs));
    }
```

Figure 18. Part of jj()

So far, we knew that the parameters of the decryption function are a String message, a String filename, and a substring of the filename. Thus, we move to *MainActivity.java* class to see how is variable *hzs* is defined.

In the *MainActivity.java* class, we found the *onCreate()* contains the declaration and initiation of *hzs* shows in *Figure 19*. The *hzs* is the length of variable *hz*, and *hz* is a string that was built very complexly. It is not completely shown in *Figure 19* for the string building of *hz*, so I will describe specifically here. Firstly, it calls a *l()* function, which is also a personal string modifier function, it takes Thai characters and converts it to a Chinese sentence, which basically says “Do not uninstall this application, contact xxxx for decryption”. Then append another variable *xh*. Thus, we need to check how is *xh* defined.

```
if(!v3.getString("bah", "").equals("")) {
    v0.xh = v3.getString("bah", "");
}
else {
    v0.xh = new StringBuffer().append("").append((((int)(Math.random() * (((double)1000000))))) + 10000000).toString();
    SharedPreferences$Editor v4 = v3.edit();
    v4.putString("bah", v0.xh);
    v4.commit();
}

MainActivity.hz = new StringBuffer().append(sss.l("[illegible]
MainActivity.m = new StringBuffer().append("").append(Integer.parseInt(v0.xh) * 4 + 3).toString();
MainActivity.hzs = MainActivity.hz.length();
MainActivity.fi = new File(new StringBuffer().append(Environment.getExternalStorageDirectory()).append("/").toString());
if(v3.getInt("cs", 0) >= 2) {
    v0.setTitle("Lycorisradiata");
    v0.getSupportFragmentManager().beginTransaction().replace(2131099760, new qq1279525738()).commit();
    sss.bz(v0);
}
```

Figure 19. Part of onCreate()

Still in the *onCreate()* function, *xh* is defined in an if-else statement. The first statement checks if *xh* is already defined in shared preferences, if not, initiate *xh* as a random number * 1000000 + 10000000 and convert it to string. So far, we can conclude that *hzs* is depending on an random number *xh*, which does not make sense to perform decryption.

Although we got stuck on tracking a parameter of the decryption function, we found quite a few interesting variables defined in the *MainActivity.java* class, especially the randomly defined variable *xh*. When I was running some tests in Eclipse of the process of initializing *xh*, I found the random output is very similar to the pin code that we got on the WannaCry page. If *xh* is the pin, then the if-else statement on generating *xh* makes much more sense, because if the application is first- time open, then it will go to "else" and initialize a new pin, if not, it will check the pin variable which is stored in shared preferences.

Assuming *xh* is the pin code, then we need to find where uses *xh* besides the *hz*. Luckily, just around the initialization of *hz* and *hzs*, there is a string variable *m* defined as *xh* * 4 + 3 and converted to string.

Now we know *m*, again we need to find where uses *m*. In the *MainActivity.java* class, a *void run()* calls *deleteDr()*, and one of the parameters of *deleteDr()* is *m*. However, this does not help out, because we do not care how it deletes files. After a global searching of variable *m*, we found that in class *qq1279525738.java*, there is a function *void onClick()* checks if the toast text input is equal to *m*, shows in *Figure 20*. Although it does not call the decryption function directly after key match, instead of it, it starts a new thread which performs the decryption. Also, by reading the messages that it prints after key match, we know that the decryption process begins.

```
@Override public void onClick(View arg10) {
    100000004 v0 = this;
    if(v0.this$0.彼岸花) {
        Toast.makeText(v0.this$0.getActivity(), "The decryption has already started! Please don\'t touch it!
    }
    else if(v0.this$0.ed.getText().toString().equals(MainActivity.m)) {
        v0.this$0.彼岸花 = true;
        Toast.makeText(v0.this$0.getActivity(), "The key is correct and the decryption begins!", 0).show();
        v0.this$0.bt.setText("In decryption");
        new Thread(new 100000003(v0)).start();
    }
    else {
        Toast.makeText(v0.this$0.getActivity(), "Key error!", 0).show();
    }
```

Figure 20. Part of onClick()

3) Result: Therefore, the key to decrypt is the pin code * 4 + 3. I have tried with the Bluestacks simulator and it worked, all my media files went back after the decryption. Also, there is a website (Tencent Games, 2019) that Tencent Inc. provides that let users who accidentally installed King of Glory ransomware to input the pin code, and they will output the key to save your device, what they did is just times 4 and plus three.

CHAPTER 5. DISCUSSION

Generally speaking, all of the three ransomware cases are disabling users' devices by either locking the screen, or encrypting media files, and charging money for the password to unlock or decrypt. The malicious code may be injected into a popular application, or a complete fake application only shares the same name and icon with some certain well-known application to induce users to download and install. An interesting thing is that the applications that ransomware tries to foreign are different among countries. In the United States, most ransomware applications are disguised as porn videos, Adobe Flash Player and some system software updates. Whereas in China, ransomware application can be seen as the game plug-in, free Wi-Fi, and "like" gathering tools.

The method to unlock for the three cases are all entering a password that "purchased" from attackers, this is the most straightforward way and the safest way. Moreover, with this unlocking method, a ransomware developer can sell the pro- gram to attackers who have less knowledge about Android development, but they can easily modify the ransomware by changing the pin value, or some input variables that an algorithm used to generate the password, to make a ransomware reproducible and unique to one attacker. What is worse, the ransomware developer usually places an advertisement of their ransomware products and leave their contact information. Since the attackers are spreading the ransomware application over the internet, which actually advertises their ransomware products for free. Thus, not only the original attackers buy ransomware from the developer, even some victims became the attackers after being blackmailed. Therefore, a ransomware development, selling, spreading has already formed a whole industrial chain.

The ransomware industrial chain can be formed because there is a market, not every Android user is an expert in using Android device and has the knowledge of how the ransomware works, so there are still quite amount of people paying for unlocking their device. However, paying is not stopping the attack, but encouraging this ransomware industry. Thus, we should never pay for unlocking, and there are actually other ways to save your device if you are accidentally installed an application with ransomware. In the next section, I am going to conclude some possible general solutions as well as some preventions from other's work.

CHAPTER 6. GENERAL SOLUTIONS AND PREVENTIONS

Since it is impossible to analyze every Android ransomware application to find the key for unlocking or decryption, it is necessary to have some general solutions that can possibly save the device which has already locked. Some general solutions that I concluded as follows.

General Solutions

The first method and is worth to try is to restart the device and delete the ransomware application quickly. However, this method depends on the operating environment of the device, and the implementation of the ransomware, so it only works for a few ransomwares.

Fortunately, Some first aid kits for the mobile device are very powerful right now, although it takes some memory to run and may be placed with some advertisement, it is still worth to install because most of them can detect and stop from installing an application with malware, kill the locking process, and uninstall it automatically.

Entering security mode is also an effective way to remove the ransomware application. What you need to do is to forcibly shut down the device by holding the power button, and then restart the device. Then enter the security mode (different brands and devices may have different ways to enter the security mode, you may need to check the user manual of the device). Find the ransomware application in the setting page, uninstall it and then restart.

For users with some technical background, if the device is rooted and the USB debugging mode is turned on, you can connect the device to a computer and run ADB command. If the ransomware is setting or tampering the unlocking password, then use *rm /data/system/password.key* to remove the password file. For other types of lock-screen

ransomware, you can also use *rm* command to delete the ransomware application installation path.

Preventions

There are also a lot of ways to prevent some regular ransomware from being installed. There are a few details you should pay attention to when downloading and install an unknown application:

First of all, the size of an Android application is an indicator to determine if it contains ransomware, a typical plain ransomware Apk will not be too large, usually less than 5Mb if it is not injected into a regular application. So, if the size of Apk that you download is way less than what it is expected, be careful about this Apk. Secondly, the name of an Android application is also a good clue. Most ransomware applications are disguised as the game plug-in, free WiFi, and "like" gathering tools. Do not simply trust this kind of applications. Last but not least, the permissions request that pops out during application installation is very important. Most ran- somware applications will ask for some sensitive permissions such as "SYSTEM ALERT WINDOW", "WAKE LOCK", "RECEIVE BOOT COMPLETED" and so on. These are the evidence that the package you are about to install may have the malicious program within, and in fact, what most security applications do is trying to analyze permissions and system intent to identify malicious applications (Schmeelk, 2014). Thus, we need to think carefully if the application we are about to install requires these permissions, if not, do not trust them easily, use some security applications to scan it before installing it if you really want that application.

Other suggestions can be: Downloading the package from large and trusted sites, back-up the device regularly, and install security application such as mobile guard.

CHAPTER 7. UNPACKING

As the experiments on Android ransomware move further, we found many ransomware apps are “guarded” or “packed” by some third-party packing services. This issue increases the difficulty of analyzing ransomware apps significantly, as we have to find a way to unpack it in order to decompile it and extract the original code. Therefore, it is necessary to have research on packing, unpacking, as well as the recovery process in order to dive deep in Android ransomware. In the following sections, we are going to give a brief introduction of the packing process, a few explanations of principles of three typical unpacking tools, and also the recovery process.

Introduction to Packing

Code packing was originally designed to protect intellectual property, which is also a double-edged sword to the security community, as the packers are widely used by Android malware or ransomware to hide their malicious code. There are a lot of companies that are providing cloud packing services nowadays, such as Bangcle, Baidu, Alibaba, Qihoo360, etc. The techniques of packing among these companies are not quite identical, but the main concept is the same. The general process of packing is illustrated in Figure 21.

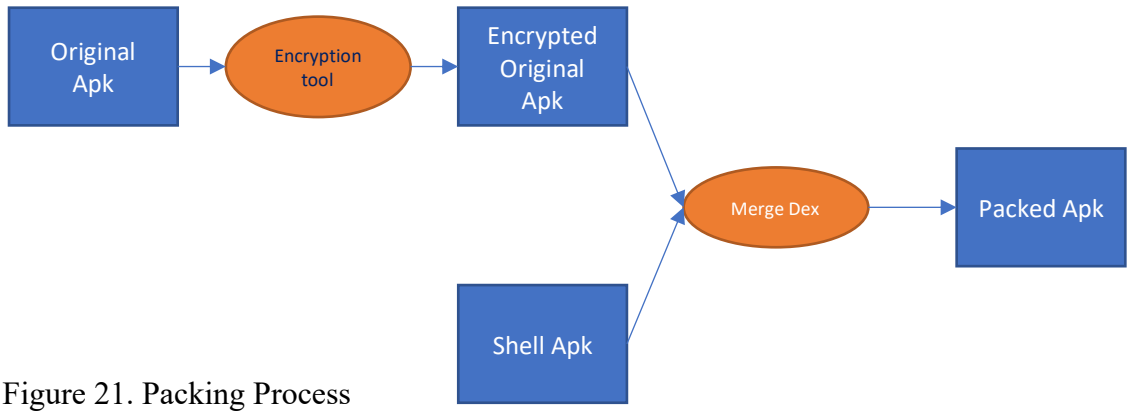

Figure 21. Packing Process

The three objects that we need in the packing process are 1. The original Apk which needs to be packed; 2. The shell Apk, which used for decryption; 3. An encryption tool. The basic process is, first we encrypt the original Apk with the encryption tool, and then merge with the shell Apk to obtain a new dex file. Finally, we can replace the dex file of the shell Apk with the new dex file, which makes the shell Apk be a new Apk. The new shell Apk is called a "packed" Apk, which is responsible for decrypting the original Apk, then dynamically load the Apk and run as normal. The output of this packing process can be illustrated in Figure 22:

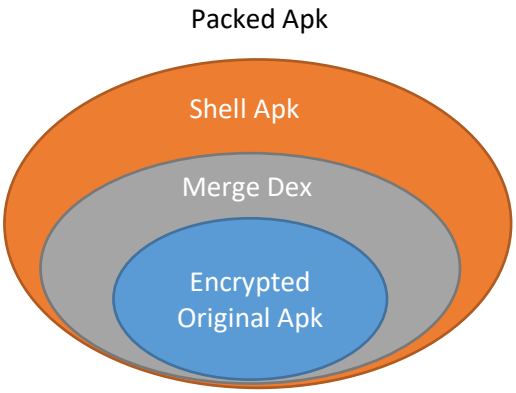

Figure 22. Packed Apk

As our goal is to unpack Apk, and almost every packing process will have its corresponding unpack Apk within, it will make the unpacking process much easier if we can understand how does the shell Apk work from each company.

The general idea of shell Apk is to use java reflection to replace the '*mClassLoader'* in '*android.app.ActivityThread'* to the '*DexClassLoader*' of the original Apk, in this way, it can guarantee that it will load the original program under the shell Apk environment and resources. The process of how the shell Apk works can be demonstrated by Figure 23:

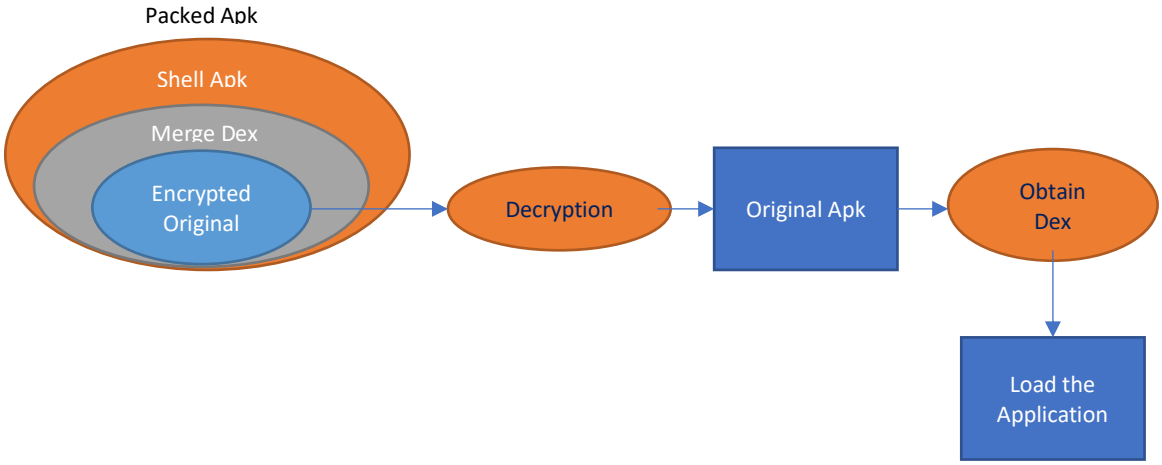

Figure 23. Shell Apk Process

The overall procedures of the shell Apk process can be concluded as follows:

1. Obtain the dex file of the shell Apk
2. Find the original Apk from the dex file, and run the decryption process (the decryption process must be corresponding to the encryption of the packing process, otherwise it will raise dex loading error)
3. Obtain the dex file from the decrypted original Apk
4. Load the Application of the original Apk according to the metadata of '*AndroidManifest.xml*'

Inspiring by the process of the shell Apk, we can think of two approaches to unpack an Apk based on the given unpack Apk: 1) Figure out the encryption & decryption method that each company uses, so we can decrypt the original Apk statistically. 2) Dump the dex file of the original Apk from memory in runtime. There are pros and cons between these two approaches, approach one does not require to run the Apk on device or simulator, which

prevents devices from being in danger since we are dealing with malicious apps, however, to understand or rewrite the decryption method is a difficult and time-consuming work, and it is not a universal way to unpack. On the other hand, approach two is a more general way and most of the recent unpacking tools use this approach, there will be no worries on the decryption as it is unpacking dynamically, so the difficulty of this approach is on system level. First of all, the packed app must be runnable under our system environment, and in order to dump data from system, we might need to grant root permission on device, for some tool that I am about to discuss, we may even need to modify the system code and build a new environment just for unpacking. In the following sections, we will go through three unpack tools, regarding their principle, pros and cons, and also involving experiments that we have done.

Case Study: DrizzleDumper

DrizzleDumper is an unpacking tool based on searching dex files by memory features, developed by (DrizzleRisk, 2017). The pre-requite of this unpacking tool is under root environment, and the idea is to use 'ptrace' to attach the process of the target apk, then performs a feature search among the memory of the target process, once it finds a matching, it will dump the dex files from memory.

The essential part of code is as follows, it firstly attaches pid using ptrace, then perform feature search to find the "magic" memory in that target pid process, and lastly dump dex files after it found matchings.

```
1.  // Attach process id using ptrace, then perform feature search in the target pid
    process, and finally dump dex files if found matchings
2.        if(find_magic_memory(clone_pid, mem_file, &memory, dumped_file_name) <= 0)
3.        {
4.          printf("[*]  The magic was Not Found!\n");
5.          ptrace(PTRACE_DETACH, clone_pid, NULL, 0);
```

```
6.          close(mem_file);
7.          continue;
8.        }
9.        else
10.       {
11.          // dex dump successfully, break the loop
12.          close(mem_file);
13.          ptrace(PTRACE_DETACH, clone_pid, NULL, 0);
14.          break;
15.       }
```

Experiment

Environment:

A rooted device or simulator

Installed Android Debug Bridge (adb)

Installed target packed app, make sure it is runnable on device

Steps:

1. Download drizzleDumper from github
2. adb push drizzleDumper to /data/local/tmp of the device or simulator
3. ./drizzleDumper [target_package_name] [wait_time]
4. Open the target app
5. adb pull the dex files under /data/local/tmp

Notes: wait_time is in second, and we can choose different wait_time (normally 1-3 sec) if the dex was not dumped successfully. This is because the appearance time and lasting time for a dex file are different for different packers.

Results: Apks packed from Qihoo360 can be successfully dumped by this unpacking tool, packers from other company cannot be dumped or dumped an empty dex file.

Pros & cons

The DrizzleDumper is relatively easy to use and to understand the principle, it does the unpack job for Qihoo360 packers, even for the latest version. However, the

DrizzleDumper only works for Qihoo360, which is definitely not a universal unpacking tool that we are looking for.

Case Study: Dex2oatHunter

Dex2oatHunter (Spriteviki, 2016) is another unpacking tool that is based on the source code of Android runtime (ART). It is a modification of Android 4.4 source code mainly in "art/dex2oat/dex2oat.cc"

In ART environment, most codes are compiled ahead-of-time, it transforms byte code to native code, which improves the performance of the Android app. The workflow of compilation is apk → dex → oat, and the idea of Dex2oatHunter is to dump the dex files from memory in the timing just before the dex2oat process happens, to achieve the unpacking purpose. As an example of Android 4.4 source code showing below, in "/art/dex2oat/dex2oat.cc" (line 14). It checks if the dex file has written permission before calling the dex2oat function, and Dex2oatHunter chooses this timing to dump dex files use a given file descriptor.

```
1.  for(constauto& dex_file : dex_files) {
2.      if(!dex_file->EnableWrite()) {
3.          PLOG(ERROR) << "Failed to make .dex file writeable '"<< dex_file->GetLocati
    on() << "'\n";
4.      }
5.      std::stringdex_name = dex_file->GetLocation();
6.      LOG(INFO) << "Finding:dex file name-->"<< dex_name;
7.      // dump dex files for packer: qihoo360
8.      if(dex_name.find("jiagu") != std::string::npos) {
9.          LOG(INFO) << "Finding:dex file from qihoo-->"<< dex_name;
10.         int len = dex_file->Size();
11.         charfilename[256] = {0};
12.         sprintf(filename, "%s_%d.dex", dex_name.c_str(), len);
13.         int fd = open(filename , O_WRONLY | O_CREAT | O_TRUNC , S_IRWXU);
14.         if(fd > 0) {
15.             if(write(fd, (char*)dex_file->Begin(), len) <= 0) {
16.                 LOG(INFO) << "Finding:write target dex file failed-
    ->"<< filename;
17.             }
18.             LOG(INFO) << "Finding:write target dex file successfully-
    ->"<< filename;
19.             close(fd);
20.         } else {
```

```
21.             LOG(INFO) << "Finding:open target dex file failed-->"<< filename;
22.         }
23.     }
24. }
```

Experiment

This tool was abandoned due to it is very time consuming in building source code (will be talked about in pros & cons), while we wanted to make some modification of the original code and test it on a real device.

Pros & cons

Dex2oatHunter supports ART, which is the newer version of Android runtime, according to the author's test, it can successfully unpack Qihoo360 and Legu packers due to the special "timing" of dumping dex files, and ideally may work for more company as it can define the packer's package name(line 8). However, the biggest shortage is the time it takes to build source code, and any changes we made to test, require 5-6 hours to build with chances of failures. Another fact is it only support ART, but we are looking for a universal unpacker that should be able to take care of both DVM and ART environment.

VirtualXposed + FDex2

FDex2 is a relatively new unpacking tool based on VirtualXposed framework, which is developed by a forum user in 2018. We cannot find an authorized Github, but it is a brand-new approach of unpacking compares to others, so it is meaningful to talk about its principle, in respect of VirtualXposed and FDex2 separately.

VirtualXposed:

Xposed is a framework for modules that can change the behavior of Android system and apps without touching any APKs. However, there are a lot of limitations such as it is required to unlock the bootloader, and grant root permission of the Android device before

using this framework. The VirtualXposed, on the other hand, it is not necessary to have root permission in order to utilize Xposed. The working principle of VirtualXposed is as the name of it, to create a “virtual” environment in the device, and launch Xposed in this virtual environment, and all the apps or modules that we need will be installed in this virtual environment to make use of Xposed framework. Since it is a virtual environment, any modules (e.g. FDex2) or apps (apps to be unpacked) that be installed into VirtualExposed will not affect the Android system.

FDex2:

FDex2 is a small module that can be run in Xposed or VirtualXposed framework. The idea of this module is, by hooking the *loadClass* method in *ClassLoader*, using reflection to call *getDex* method to obtain Dex object(*com.android.dex.Dex*), and then write the dex object to files. The core code is just a class hook method and is included as follows:

```
1.  public void handleLoadPackage(XC_LoadPackage.LoadPackageParam lpparam) throws Throw
    able {
2.          xsp = new XSharedPreferences("com.ppma.appinfo", "User");
3.          xsp.makeWorldReadable();
4.          xsp.reload();
5.          initRefect();
6.          packagename = xsp.getString("packagename", null);
7.          XposedBridge.log("Set package name: "+packagename);
8.          if ((!lpparam.packageName.equals(packagename))||packagename==null) {
9.              XposedBridge.log("The current package name is null or not matching");
10.             return;
11.         }
12.         XposedBridge.log("Target package name: "+lpparam.packageName);
13.         String str = "java.lang.ClassLoader";
14.         String str2 = "loadClass";
15.
16.         XposedHelpers.findAndHookMethod(str, lpparam.classLoader, str2, String.clas
    s, Boolean.TYPE, new XC_MethodHook() {
17.             protected void afterHookedMethod(MethodHookParam param) throws Throwabl
    e {
18.                 super.afterHookedMethod(param);
19.                 Class cls = (Class) param.getResult();
20.                 if (cls == null) {
21.                     XposedBridge.log("cls == null");
22.                     return;
23.                 }
24.                 String name = cls.getName();
```

```
25.                 XposedBridge.log("Class name: " + name);
26.                 byte[] bArr = (byte[]) Dex_getBytes.invoke(getDex.invoke(cls, new O
    bject[0]), new Object[0]);
27.                 if (bArr == null) {
28.                     XposedBridge.log("The data is null, return");
29.                     return;
30.                 }
31.                 XposedBridge.log("Start writing...");
32.                 String dex_path = "/data/data/" + packagename + "/" + packagename +
     "_" + bArr.length + ".dex";
33.                 XposedBridge.log(dex_path);
34.                 File file = new File(dex_path);
35.                 if (file.exists()) return;
36.                 writeByte(bArr, file.getAbsolutePath());
37.             }
38.             } );
39.     }
```

Experiment

Environment:

An Android device or simulator with system version 4.4 or above

VirtualXposed

Steps:

1. Install VirtualExposed, FDex2, and the app to be unpacked on the Android device
2. Open VirtualExposed, install FDex2 module within the VirtualExposed environment and activate FDex2
3. Install the target app within VirtualExposed
4. Run FDex2 in VirtualExposed, setup the target app to hook its package name
5. Run the target app in VirtualExposed
6. The unpacked dex files will be located in /data/user/0/iv.va.exposed/virtual/user/0/{packagename}

Notes: If the device is rooted, we can retrieve dex files using adb pull, if it is not rooted, then we can use the VirtualXposed built-in File Manager to share files to PC.

Results: Apps that packed by Qihoo360, Baidu, Bangcle are unpacked successfully, apps packed from packers such as Legu, Ijiami cannot be unpacked or output empty dex files.

Pros & cons

The good things about using VirtualExposed and FDex2 are, it can unpack from more packers since it directly hooks the *getDex* method instead of reading dex files from memory, as a lot of packers start to protect the memory during unpacking stage. In addition, because of the framework + module idea, we do not need to compile system, or root device, and if we want to modify or update the module for the latest Android version, we can just make the modifications on the module and packed as an Apk to test, which makes this method extendible. Overall, I believe this is a good approach to dive deep in unpacking in the future.

CHAPTER 8. RECOVERY

In order to verify the unpacked code is identical to the original code, and to support some dynamic experiments that our security group currently running, it is necessary to find a way to recover the unpacked dex files to a runnable Apk. Therefore, this chapter is going to introduce a working method that we have tried to perform a recovery (repack).

In the following a few sections, we are going to conclude the general approach of recover an unpacked android app, and a practical experiment that successfully recovered an app that was packed by Qihoo360 and then unpacked by FDex2.

General Steps

1. *Obtain dex files*

   The first step is to unpack the original packed Apk with one of the unpacking tools that was introduced in the previous chapters and obtain the dex files as the output. We have to be careful about the output dex files as there might be irrelevant or useless dex files depending on different unpacking tools, a good approach to check is to search the dex files content by keywords such as the package name.

2. *Replace dex files*

   Assume you are confident with the dex files, then the next step is to replace the dex files in the original packed Apk with the dex files that we obtained from the last step, and make sure the dex files are renamed identical to the original ones.

3. *Modify AndroidManifest.xml*

   Since the new dex file that we obtained is the one with the shell application code removed, in order to make it work as normal, we have to revise the

AndroidnManifest.xml file such as replacing the application entry point name with the original Apk package name (not the packed package name), and remove some modules or attributes that are apparently used for shell application. This step requires users to have some knowledge of Android development, otherwise, it is difficult to tell which modules to keep or delete.

4. *Repack and sign the Apk*

   There are a lot of tools such as Android Studio and Apktool to sign and pack the Apk, after it is packed successfully, it is ready for installing and testing.

5. *Test and debug*

   If the recovered app is working perfectly after the last step, then this step can be skipped. If unfortunately, the recovered crashed in runtime, it is necessary to debug and repair the app. Most likely, the app is crashed due to some code of shell Apk has been deleted but the main activity is still calling some classes, functions or variables of shell Apk. Therefore, the best approach is to set breakpoints and go over the code in runtime to find out which line of code is causing problems.

Experiment

A useful tool that we found is an Android app named "MT Manager" (MT Manager, 2019), it is a powerful file manager in Android platform, with this app, we can move files, rename files, searching keywords in files, we can even decompile dex files, make some modifications and then compile it back, and even sign an Apk. With the help of this tool, we

are able to do the recovery within an Android device. In the next a few sections, we will show the experiment steps and details.

Environment:

The experiment is running on an Android emulator (Nox emulator) with MT Manager installed, a packed app (we used a simple log app that we need to perform other dynamic experiments) packed with Qihoo360, and dex files that are unpacked by FDex2 (the last approach of Chapter 7).

Steps:

1. Launch MT Manager
2. Move dex files to /mnt/shared/Other/
3. Search each dex file by the target package name until finding the right dex file
4. Rename the found dex file to “classes.dex”
5. Open the original packed Apk and view contents
6. Replace the “classes.dex” in original packed Apk with the one we renamed
7. Open “AndroidManifest.xml” with decompiler
    a. Replace the <application … name = “{package_name}”/>
    b. Remove <meta-data … /> block
    c. Compile
8. Sign the Apk by going to function -> Apk sign
9. Install the signed Apk and run it

Evaluation:

First of all, the recovered and the signed app is running successfully as the original one. Now we have got two Apks, the original, and the packed then unpacked and recovered

one. In order to verify the identity, we performed a comparison between these two Apks. After decompiling these two Apks using JEB, we found the classes of these are the same, no added or resected classed found due to packing or unpacking. Furthermore, we performed a line-by-line code comparison using “file diff”, and the result shows that all the code files between two Apks are identical.

Conclusion

Overall, this recovery approach is successful and could be applied to our team’s other security experiment’s workflow if it involves a packed app. Also, this approach completes the entire unpacking process, from a packed app to a complete unpacked and re-signed app that is runnable, which is a great step forward for our future Android security research.

CHAPTER 9. SUMMARY AND FUTURE WORK

In this paper, three typical Android ransomware applications are analyzed and got resolved using reverse engineering approach and both statistic and dynamic methods. Although we cannot crack all the ransomware spreading in the Android market, it is time to stand out and fight back. We hope my work can improve the Android market environment a little bit, and we encourage everyone to refuse to pay any money for unlocking or decryption. Only in this way, the ransomware packaging business can be stopped.

In addition to ransomware, we have performed a few experiments on the Android unpacking process, with principles, experiments steps explained in detail, and also discussed the pros and cons for each unpacker we tested. We also included recovery in the unpacking process to best support our team's other security experiments. As a result, some of the tools that we have tested could be applied to our other researches workflow.

In the future, we are going to analyze more complex Android ransomware, for those using remote unlocking, and SMS control unlocking. We are also planning to implement a detection application that can identify the latest ransomware applications.

REFERENCES

Blogs.360.cn. (2019). *Android Ransomware Research Report*. [online] Available at: http://blogs.360.cn/360mobile/2016/04/12/analysis of mobile ransomware/ [Accessed 15 Feb. 2019].

Bluestacks - The Best Android Emulator on PC as Rated by You. (2019). *BlueStacks - Play Mobile Games on PC 6x Faster Than Any Phone*. [online] Available at: https://www.bluestacks.com [Accessed 15 Feb. 2019].

Bobao.360.cn. (2019). *Analysis on the Recovery Possibility from WanaCrypt0r*. [online] Available at: http://bobao.360.cn/learning/detail/3874.html?fromsinglemessage&isappinstalled0 [Accessed 15 Feb. 2019].

Ducklin, P. (2019). *Android malware in pictures – a blow-by-blow account of mobile scareware*. [online] Naked Security. Available at: https://nakedsecurity.sophos.com/2013/05/31/android-malware-in-pictures-a-blow-by-blow-account-of-mobile-scareware/ [Accessed 15 Feb. 2019].

DrizzleRisk. (2017, April 12). DrizzleRisk/drizzleDumper. Retrieved from https://github.com/DrizzleRisk/drizzleDumper

En.wikipedia.org. (2019). *Mobile security*. [online] Available at: https://en.wikipedia.org/wiki/Mobile security#Ransomware [Accessed 15 Feb. 2019].

GitHub. (2019). *Sable/soot*. [online] Available at: https://github.com/Sable/soot [Accessed 15 Feb. 2019].

IDC: The premier global market intelligence company. (2019). *IDC - Smartphone Market Share - OS*. [online] Available at: https://www.idc.com/promo/smartphone-market-share/os [Accessed 15 Feb. 2019].

Lipovsk, R., Tefanko, L. and Brania, G. (2017). Trends in Android Ransomware. *ESET*.

Lookout.com. (2019). *Lookout | The Leader in securing the Post-Perimeter World*. [online] Available at: https://www.lookout.com/resources/reports/mobile-threat-report [Accessed 15 Feb. 2019].

MT Manager (2019) - Apps on Google Play. Retrieved from https://play.google.com/store/apps/details?id=bin.mt.plus&hl=en_US

Pnfsoftware.com. (2019). *JEB Decompiler by PNF Software*. [online] Available at: https://www.pnfsoftware.com [Accessed 15 Feb. 2019].

Schmeelk, S. (2014). Static Analysis Techniques Used in Android Application Security Analysis.

Secure Software Engineering. (2019). *FlowDroid Taint Analysis*. [online] Available at: http://sseblog.ec-spride.de/tools/flowdroid/ [Accessed 15 Feb. 2019].

Secure Software Engineering. (2019). *FlowDroid Taint Analysis*. [online] Available at: http://sseblog.ec-spride.de/tools/flowdroid/ [Accessed 15 Feb. 2019].

Securehoney.net. (2019). *Creating An Antidote For Android Simplelocker Ransomware | SSH honeypot written in C*. [online] Available at: http://securehoney.net/blog/creating-an-antidote-for-android-simplelocker-ransomware.html#.Wi2oeraZMnU [Accessed 15 Feb. 2019].

Securehoney.net. (2019). *How To Dissect Android Simplelocker Ransomware — SSH honeypot written in C*. [online] Available at: http://securehoney.net/blog/how-to-dissect-android-simplelocker- ransomware.html#.Wi2nvraZMnU [Accessed 15 Feb. 2019].

Secureworks.com. (2019). *WCry (WannaCry) Ransomware Analysis*. [online] Available at: https://www.secureworks.com/research/wcry-ransomware-analysis [Accessed 15 Feb. 2019].

Song, S., Kim, B. and Lee, S. (2016). The Effective Ransomware Prevention Technique Using Process Monitoring on Android Platform. *Mobile Information Systems*, 2016, pp.1-9.

Spriteviki. (2016, June 4). spriteviki/Dex2oatHunter. Retrieved from https://github.com/spriteviki/Dex2oatHunter

Symantec Security Response. (2019). *FakeAV holds Android Phones for Ransom*. [online] Available at: https://www.symantec.com/connect/blogs/fakeav-holds-android-phones-ransom [Accessed 15 Feb. 2019].

Tencent Games. (2019). *Ransomware: King of Glory - Beta Announcement (Solution included)*. [online] Available at: https://wx.gamesafe.qq.com/temporary_virus [Accessed 15 Feb. 2019].

Yang, T. (2015). Automated Detection and Analysis for An- droid Ransomware. *2015 IEEE 17th International Conference on High Performance Computing and Communication*