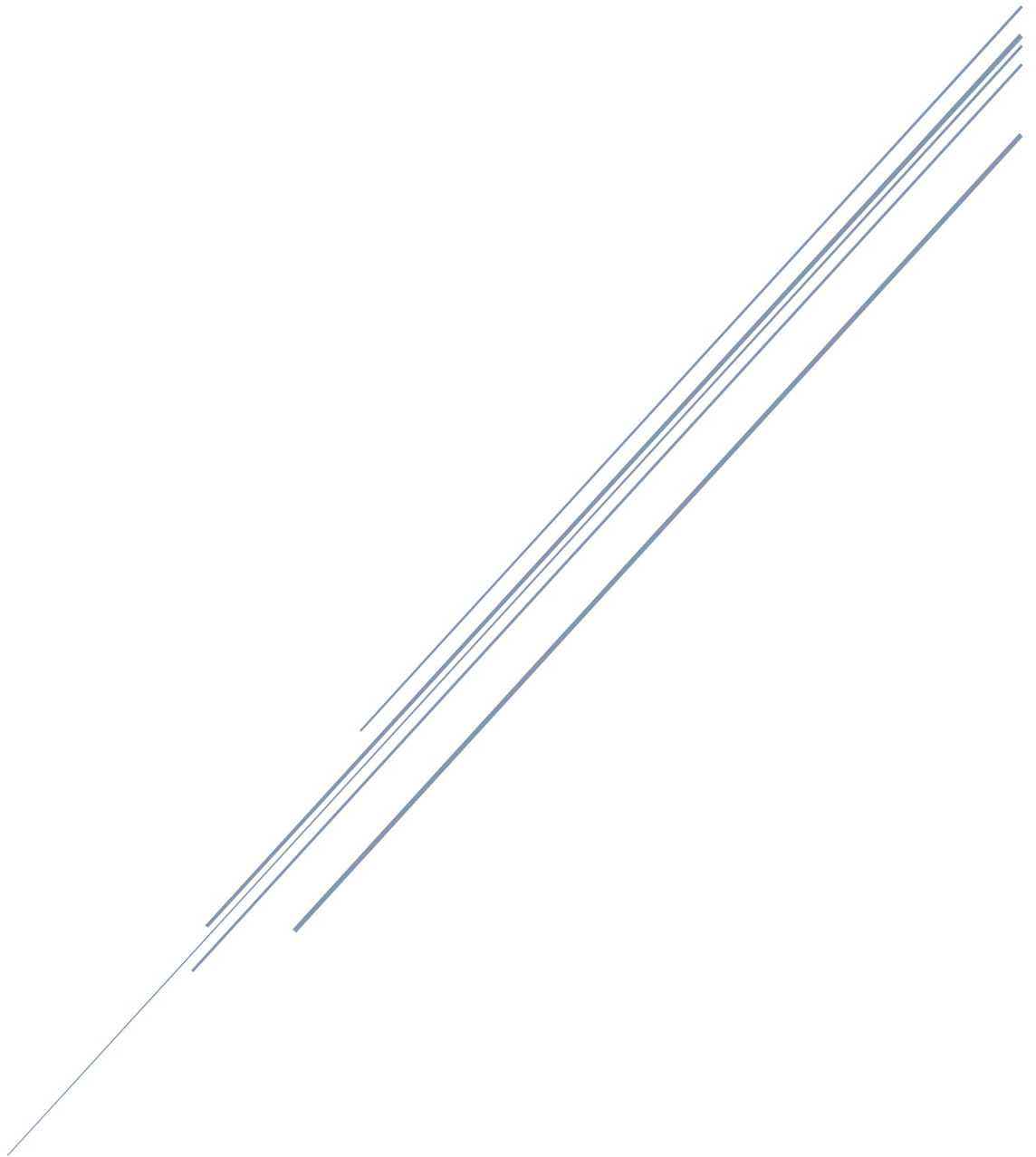


ANDROID MALWARE ANALYSIS

Decrypting two stages



Max 'Libra' Kersten

Table of Contents

Technical information.....	2
Permissions.....	2
Automated decryption.....	2
Notes.....	3
Architecture	3
Stage 1.....	3
Stage 2.....	4
Anti-hooking functionality	4
Decrypting the reflection calls.....	5
Decrypting the asset	6
Conclusion	9
About the author.....	10
Appendix A.....	11
Bibliography	13

Technical information

SHA-512	a94180f000c5519345c8c33fdcf84d966f0ca7cd22cea0d21ff98c098b366df2be66b59b87bf19397404204f6748439e07c173ec46ee0ed07873ab7f39182dc1
SHA-256	c52f628a6718c01d2131a3b6ef676bbde46504f2cab7f207dda95e3dba85f254
SHA-1	a0577222e464611390e418fe56179834bab96abe
MD-5	ca0a60770d6db6aa9fbb7ce2907bf9b9
CRC32	DE036837
SSDeep	12288:uuwZJF6T3j4i0/8CjCHmIKq2ZCxjoYgooxJ7BJc:s96XW8fKKBUtlTonBJc

APK size: 459.5 kB (470526 bytes)

Package name: com.wbgsyas.gmhayacl

Application name: ebony_fuck

Permissions

android.permission.CHANGE_NETWORK_STATE
android.permission.REAL_GET_TASKS
android.permission.RECEIVE_BOOT_COMPLETED
com.android.launcher.permission.UNINSTALL_SHORTCUT
android.permission.READ_CONTACTS
android.permission.WRITE_SMS
android.permission.ACCESS_WIFI_STATE
android.permission.ACCESS_NETWORK_STATE
android.permission.WAKE_LOCK
android.permission.GET_TASKS
android.permission.CHANGE_WIFI_STATE
android.permission.RECEIVE_SMS
android.permission.READ_PHONE_STATE
android.permission.INTERNET
android.permission.READ_PROFILE
android.permission.WRITE_EXTERNAL_STORAGE
android.permission.GET_ACCOUNTS
android.permission.READ_SMS

Automated decryption

For those interested in a proof-of-concept automated decryption tool for this sample, I advise you to check out [my Github repository](#).

Notes

The code itself is kept as close as possible to the decompiled code from the malware itself. Refactoring is done due to garbage names, which results in understandable names. These are created by the analyst, which'd be me, and not by the malware author.

Architecture

Stage 1

The malware itself contains two assets. The first asset is saved as a base64-encoded string in the original application. Firstly, the base64 string is decoded using the default base64 package in Android. The decoded value, a `byte[]`, is passed to the decrypt-function as `byteArray1`. The second argument that has to be passed to the decrypt function, is the key. This is a hardcoded key, which equals `"pgtqOqBeLbrMXbWPfPzUMEuF".getBytes()`. The function itself swaps bytes around and eventually returns the decrypted `byte[]`.

```
private static byte[] decrypt(byte[] byteArray1, byte[] byteArray2) {
    byte[] output = new byte[byteArray1.length];
    Long l = new Long(0);
    for (int i = 0; i < byteArray1.length; ++i) {
        Long l2;
        output[i] = (byte) (byteArray1[i] ^ byteArray2[(int) l.longValue()]);
        l = l2 = Long.valueOf(l + 1);
        if (l2 < (long) byteArray2.length) {
            continue;
        }
        l = new Long(0);
    }
    return output;
}
```

The output is then written to the files directory of the context: `new File(context.getFilesDir(), "pzJDrFMdboF.zip");`

The written file is then loaded using the `DexClassLoader`, after which the constructor of the class `"ntnhrhp/fggxp"` is invoked. Directly afterwards, the function `attachBaseContext` is called with the current context provided as argument. Any error will simply return, after which the `"onCreate"` method will return due to the given try-catch structure, resulting in the application to exit. Starting the application again will initiate the process described above.

Stage 2

Anti-hooking functionality

The reason the malware did not run in my virtual environment was because of a function that threw an exception. In the stacktrace of this exception, multiple checks for hooking frameworks were done, as can be seen in the figure below.

```
private boolean checkforHooks()
{
    try
    {
        throw new Exception("blah");
    }
    catch (Exception localException)
    {
        int k = 0;
        StackTraceElement[] arrayOfStackTraceElement = localException.getStackTrace();
        int m = arrayOfStackTraceElement.length;
        int i = 0;
        while (i < m)
        {
            StackTraceElement localStackTraceElement = arrayOfStackTraceElement[i];
            int j = k;
            if (localStackTraceElement.getClassName().equals("com.android.internal.os.ZygoteInit"))
            {
                k += 1;
                j = k;
                if (k != 2) {}
            }
            while (((localStackTraceElement.getClassName().equals("com.saurik.substrate.MS$2")) && (localStackTraceElement.getMethodName().equals("invoked"))) ||
                return true;
            i += 1;
            k = j;
        }
    }
    return new File("/system/framework/XposedBridge.jar").exists();
}
```

The result of this function is a `boolean`, which is `true` if any of the frameworks is detected. Based on the result of the `boolean`, another action is performed, as is shown in the figure below.

```
public void attachBaseContext(Context paramContext)
{
    super.attachBaseContext(paramContext);
    if (checkforHooks())
    {
        Process.killProcess(Process.myPid());
        return;
    }
    //Executes the function named 'HwKdwuyZa' in this class
    try
    {
        //
        HwKdwuyZa
        getClass().getDeclaredMethod(decryptString("LhAsHAcUH2OG"), new Class[0]).invoke(this, new Object[0]);
        return;
    }
    catch (Exception paramContext) {}
}
```

If the `checkForHooks` method returns `true`, the application terminates itself. Otherwise, it simply continues.

Decrypting the reflection calls

The second stage was also encrypted in a similar way, yet this one was much harder to decrypt. Both the stages used Java reflection to call functions. This time, the code was slightly different. Each of the strings used in the reflection calls, was encrypted. The method used to decrypt the strings was included in the class, as can be seen in the figure below.

```
public String decryptString(String paramString) {  
    new StringBuilder().append("MjUhOyY4ISUyMDkvKS0yISQmMCElOw==").append("AxAVEAIEEQ8CDwIJAxAVEA==").toString();  
    return "" + new String(xorFunction(decodeBase64(paramString), getClass().getSimpleName().getBytes()));  
}
```

The creation of the new `StringBuilder` is irrelevant, since it does nothing. The value that the `toString()` method returns is not saved nor used anywhere. This is garbage code. The next line is the important part of the function.

At first, the “`decodeBase64`” method is called. This method is merely a wrapper for the default `base64` package, which contains a `decode` function. In Android applications, a second argument is required upon decoding `base64`. A ‘0’ or `DEFAULT` as enum equals normal decoding. This wrapper returns the value of the normal `base64` decode function with a “0” as second argument. The value that it returns, is a `byte[]`. The next function that is called, is the `xorFunction`. This function can be seen in the figure below.

```
private byte[] xorFunction(byte[] paramArrayOfByte1, byte[] paramArrayOfByte2)  
{  
    byte[] arrayOfByte = new byte[paramArrayOfByte1.length];  
    int i = 0;  
    while (i < paramArrayOfByte1.length)  
    {  
        arrayOfByte[i] = ((byte) (paramArrayOfByte1[i] ^ paramArrayOfByte2[(i % paramArrayOfByte2.length)]));  
        i += 1;  
    }  
    return arrayOfByte;  
}
```

The first argument, called `paramArrayOfByte1`, is equal to the decoded `base64` string. The second `byte[]` is the key used in the decryption process. This is the `getSimpleName` of the current class, which equals `"fggxpa".getBytes()`. The class name was both visible in the Java code of this stage and in the code to load stage 2. The returned value of this is another `byte[]`. Upon creating a new string with said `byte[]`, the decrypted text is retrieved.

Decrypting the asset

After reading and refactoring the code, I found that an asset that was present in the APK from the start was loaded and used. The function, named “HwKdwuyZa” couldn’t be decompiled at first. Using another online decompiler, I retrieved the Java code. Due to efficient optimising and good usage of a few variables of the type `Object`, some variables were used as multiple types throughout the method, which made it harder to understand the code.

Using the asset manager, an asset named “jzRNvE” was read and stored into a `byte[]`. This was confusing at first, because the “Read” method of an `InputStream` returned an integer, which was neglected. Only after reading the Javadoc, I found out that the bytes that were read, were put in the given `byte[]`: the one that was used as an argument in the read method. The `byte[]` with the content is then passed to the decrypt function, which can be seen in the figure below.

```
private static byte[] decrypt(byte[] paramArrayOfByte) {
    try {
        InputStream inputSignature = new FileInputStream("726793.sig");
        int sigLength = inputSignature.available();
        byte[] signatureBytes = new byte[sigLength];
        inputSignature.read(signatureBytes);
        inputSignature.close();

        byte[] value = paramArrayOfByte;
        byte[] key = byteArrayToStringMutator(signatureBytes).getBytes();

        byte[] output = xorFunction(value, key);
        return output;
    } catch (IOException ex) {
        System.out.println("IOException during decryption");
        System.exit(0);
        return null;
    }
}
```

This method is refactored to use comprehensible names and contains no Java reflection methods. At first, the Java-class `Signature` of the application is obtained. Because of the absence of a test phone, the only way to obtain the signature was to recreate it based on the given certificate in the APK. The RSA-certificate is located in the APK at “/META-INF/726793.RSA”. Then, one needs to extract the file and save it with the same name and extension. Assuming you’re using a Linux distribution, you can use the terminal to use “openssl”. The complete command is “openssl pkcs7 -in 726793.RSA -inform DER -print_certs -out 726793.RSA.raw”. Then, one needs to open the newly created “726793.RSA.raw” file with a text editor and remove everything but the base64 encoded parts. Save this file as “726793.RSA.encoded”. Then, decode the file using the command “base64 -d ./726793.RSA.encoded > 726793.sig”.

The content in the file "726793.sig", is the one that is used in the malware. The standard Java base64 decoder is not able to decode the given base64, even though the Android version is able to do so. I'd like to thank Khaled Nassar for helping me out when I was stuck on this part.

Another method, one which requires a test phone, is to write another Android application which is also installed on the infected device. Requesting the signature of another application is possible with a given package name. The package name of the malware is known in the `AndroidManifest.xml`, which means that we can read the signature and write it to the internal memory of the phone. This can be done using the proof-of-concept provided in Appendix A.

The signature is mutated to create the key, using the function "byteArrayToStringMutator" function, which can be found in the figure below.

```
private static String byteArrayToStringMutator(byte[] paramArrayOfByte) {
    try {
        // MD5
        paramArrayOfByte = MessageDigest.getInstance("MD5").digest(paramArrayOfByte);
        StringBuffer localStringBuffer = new StringBuffer();
        int i = 0;
        while (i < paramArrayOfByte.length) {
            localStringBuffer.append(Integer.toHexString(paramArrayOfByte[i] & 0xFF | 0x100).substring(1, 3));
            i += 1;
        }
        return localStringBuffer.toString();
    } catch (Exception ex) {
    }
    return "";
}
```

The provided `byte[]`, the key, is decrypted here using the MD-5 hash and some other security through obscurity. Understanding the concept was all that was needed to copy and paste this method, even though it is always better to have a more in-depth insight in what method does.

The return value is a `string`, of which the bytes are saved in a `byte[]`.

The value and the key are then used in another "xorFunction", which can be seen in the figure below.

```
private static byte[] xorFunction(byte[] paramArrayOfByte1, byte[] paramArrayOfByte2) {
    byte[] arrayOfByte = new byte[paramArrayOfByte1.length];
    int i = 0;
    while (i < paramArrayOfByte1.length) {
        arrayOfByte[i] = ((byte) (paramArrayOfByte1[i] ^ paramArrayOfByte2[(i % paramArrayOfByte2.length)]));
        i += 1;
    }
    return arrayOfByte;
}
```

The value that is returned after the decryption process is complete, is the decrypted "classes.dex" file. This file is then put in a ZIP-folder and is written to "getFilesDir() + Separator + "cls.dex"".

Using dex2jar (Pan, 2017), one can transform a dex-file to a JAR-file. Using JD-GUI (Dupuy, 2017), one can decompile a JAR-file to Java source code. These two stages were the decryption of both protection mechanisms that were put in place by the malicious author to evade antivirus suites and reverse engineers.

Conclusion

The two stages evade common dynamic analysis tools and might confuse reverse engineers. Even though some clever tricks are used, the original source code is recovered in the end. The techniques that were used from time to time were new to me, such as the exception that is thrown to check for a hooking frameworks. This gives me new insights for future usage.

About the author

My name is Max 'Libra' Kersten and I have an interest in offensive security. To stop current and future attacks, one should think like an attacker. The anti-virus and anti-malware branches will always be too late to stop malware from infecting users. By anticipating what the malicious coders might do, the delay can be minimised.

Appendix A

Note that the application itself should have the permission to both READ and WRITE on the external store (SD-card) of the device. This is a prerequisite for the code to work. Additionally, the string named 'packageName' should contain the name of the package which signature you want to extract. Note that every signature is written with the same name on the SD-card, so any previous versions of a signature are overwritten. The way to request the read and write permission might not work on newer versions of the Android platform, due to changes every so often, the newest way is not present in the code.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    //Default code
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    //Initialise the array with a length of 10, which we wont
use
    Signature[] signatures = new Signature[10];
    String packageName = "com.example.class";
    try {
        signatures =
getPackageManager().getPackageInfo(packageName,
PackageManager.GET_SIGNATURES).signatures;
    } catch (PackageManager.NameNotFoundException e) {
        e.printStackTrace();
    }
    if (signatures.length > 0) {
        Signature signature = signatures[0];
        byte[] byteArray = signature.toByteArray();
        InputStream inStream = new
ByteArrayInputStream(byteArray);
        FileOutputStream fos = null;
        String path = Environment.getExternalStorageDirectory()
+ File.separator + "signature.sig";
        try {
            fos = new FileOutputStream(path);
        } catch (FileNotFoundException e) {
            Toast.makeText(this, "FileNotFoundException",
Toast.LENGTH_LONG).show();
            e.printStackTrace();
        }
        int b;
        byte[] d = new byte[4096];
        try {
            while ((b = inStream.read(d)) != -1) {
                fos.write(d, 0, b);
            }
        }
    }
}
```

```
        Toast.makeText(this, "Writing finished to :" + path,
Toast.LENGTH_LONG).show();
    } catch (IOException e) {
        Toast.makeText(this, "Error writing",
Toast.LENGTH_LONG).show();
        e.printStackTrace();
    }
    } else {
        Toast.makeText(this, "No signatures found!",
Toast.LENGTH_SHORT).show();
    }
}
```

Bibliography

Dupuy, E. (2017, 10 30). *Java Decompiler*. Retrieved from jd.benow.ca:
<http://jd.benow.ca/>

Pan, B. (2017, 10 30). *Tools to work with android .dex and java .class files*.
Retrieved from Github: <https://github.com/pxb1988/dex2jar>