virus BULLETIN Covering the global threat landscape

OBFUSCATION IN ANDROID MALWARE, AND HOW TO FIGHT BACK

Axelle Apvrille & Ruchna Nigam
Fortinet, France

Malware authors are certainly creative when it comes to hiding their payloads from analysts' eyes, using methods such as emulator detection, application icon hiding, reflection etc. This paper focuses on obfuscation techniques encountered while analysing *Android* malware. We present five off-the-shelf products (*ProGuard*, *DexGuard*, *APK Protect*, *HoseDex2Jar* and *Bangcle*) and make suggestions as to how researchers can detect when they have been used in malware, and some techniques to help with their reversing. We also list some custom obfuscation techniques we have encountered in malware: loading native libraries, hiding exploits in package assets, truncating URLs, using encryption etc. We provide examples and supply the sha256 hash in each case. Finally, we reveal a few new obfuscation techniques of which we are aware, which might be used by malware authors in the future. There are techniques for injecting malicious bytecode, manipulating the DEX file format to hide methods, and customizing the output of encryption to hide an APK. We provide the current state of play as regards ongoing research to detect and mitigate against these mechanisms.

1. INTRODUCTION

While obfuscation is not reprehensible, it has always been particularly popular with malware authors. Numerous *Windows* malware families use packers, obfuscation and anti-debugging techniques to hide their devious intentions from end-users and security researchers alike.

> 'The use of ProGuard or a similar program to obfuscate your code is strongly recommended for all applications that use Google Play Licensing.' [1]

In this paper, our aim is to assist security researchers and anti-virus analysts in their reverse engineering of *Android* malware. We provide tips to detect specific obfuscators, as well as techniques for reversing them and accessing the real payload.

2. DETECTING AND REVERSING OFF-THE-SHELF ANDROID OBFUSCATION TOOLS

ProGuard is the most well known of all the *Android* obfuscators, as it is integrated into the *Android* build framework itself. It is also often encountered in malware[1]. However, other tools, such as *DexGuard* – the extended commercial version of *ProGuard* – and *APK Protect* also exist.

2.1 ProGuard

By default, *ProGuard* renames paths, class names, methods and variables using the alphabet. Thus, spotting strings such as 'a/a/a;->a' in the smali code is a strong indication that the sample has been obfuscated using *ProGuard*. Of course, this simplistic method of detection is not infallible because *ProGuard* can be configured to use any replacement dictionary you wish using the options -obfuscationdictionary, -classobfuscationdictionary and -packageobfuscationdictionary. For instance, Android/GinMaster.L uses a custom dictionary, where the strings were probably generated randomly using something like http://www.random.org/strings.

The replacement of path names, class names, methods and variables cannot be undone. However, usually the reversing of *ProGuard*-ed samples isn't too difficult because the strings and code layout are not modified. The work is very similar to reversing an application coded by a beginner (poor choice of variable names etc.).

2.2 DexGuard

Working on *DexGuard*-ed samples is much more difficult. [2] lists the obfuscator's features. The main reason why *DexGuard*-obfuscated samples are more difficult to work with is because the class and method names are replaced with non-ASCII characters and strings are encrypted. Tools such as *JD-GUI* [3] and *Androguard* [4] are more difficult to use (e.g. difficult to get name completion). It is as if reverse engineers have had their senses dulled: text strings and even some familiar function calls and patterns no longer exist to guide the analyst to the more interesting parts of the code.

Fortunately, no obfuscator is perfect. [5] clarifies parts of how *DexGuard* works. Meanwhile, we provide a code snippet that can be used to detect it, and three different ways to help with the reversing of *DexGuard*-ed samples.

First, its detection – i.e. identifying the use of *DexGuard* on a sample – is usually fairly visual: the repetitive use of non-ASCII characters gives it away. The code snippet below lists non-ASCII smali files in smali disassembled code.

```
$ find . -type f -name "*.smali" -print | perl -ne
'print if /[$^$ [:ascii:]]/'
```

[1] In a partial database of 460,493 samples, we spotted it in 15% of samples.

Second, its reversing can be made easier by using the following tools or techniques:

- ***DexGuard* decryption python script.** [6] provides a script template that can be applied to each *DexGuard*-ed sample. The script decrypts encrypted strings, which makes reversing easier. However, this tool only works with samples that use old versions of *DexGuard*, not the more recent ones.
- **Logging.** A reverse engineer can disassemble the sample with baksmali [7], insert calls to *Android* logging functions (see below), recompile the application (smali), and run it.

```
invoke-static {v1, v2}, Landroid/util/Log;->e( Ljava/
lang/String;Ljava/lang/String;)I
```

  This displays corresponding strings in *Android* logs. It is an archaic, but simple and useful debugging technique. Nevertheless, this technique requires modification of the malicious sample – a practice anti-virus analysts are usually not authorized (or willing) to perform for ethical and security reasons.

- **String renaming.** To work around the problems caused by non-ASCII characters, all strings can automatically be renamed to a dummy ASCII string. To do this, we enhanced *Hidex* [8]. Originally, this tool was created to demonstrate the feasibility of hiding methods in a DEX file (see Section 4 and [9]). However, progressively, it has evolved into a small DEX utility tool that can be used for the following:
  - To list strings (option --show-strings).
  - To automatically rename non-ASCII strings (option --rename-strings). This is what we use, for instance, in the case of *DexGuard*. Each string that contains non-ASCII characters is replaced automatically by a unique string generated only with ASCII characters and which is the same size as the original string[2]. The replacement string must meet the aforementioned requirements of uniqueness and size, to conform to the DEX file format. For proper replacement, note that string size (UTF16 size field of string data item) is in UTF16 code units, not in bytes. Please refer to [10].

    There is one constraint that *Hidex* does not currently handle: the ordering of strings. In DEX files, strings must be ordered alphabetically. Renaming the strings usually breaks the correct ordering. Consequently, *Android* will refuse to load the modified classes.dex file. In the case of reverse engineering malware, this is not a real problem (perhaps it is even more secure/ ethically correct) because *Android* reversing tools such as *baksmali*, *apktool*, *dex2jar* and *Androguard* do not enforce correct ordering of strings either. Thus, they are able to disassemble the modified classes.dex without any problem.
  - To parse DEX headers and detect headers hiding additional information (see Section 2.4).
  - To detect potential hidden methods (option --detect).

[2] In theory, there are cases where we should fall short of replacement strings and thus fail to do the renaming. For example, if a sample has more strings of a single character than possible ASCII characters, the replacement is impossible. In practice, we have never encountered this limitation.

2.3 APK Protect

APK Protect [11] is another advanced off-the-shelf obfuscation product. The first time we spotted it being used in *Android* malware was in Android/SmsSend.ND!tr in March 2014. It is easy to identify its use in malware, because the string 'APKProtected' is present in the DEX. Like *DexGuard*, its reversing is difficult. In particular, we worked out its string encryption process, which is illustrated in Figure 1.

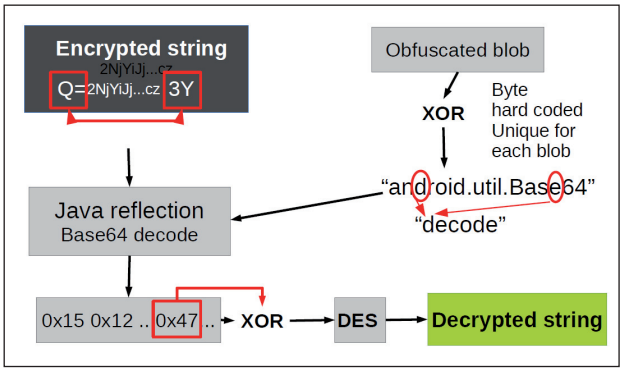

Figure 1: String encryption process used in APK Protect-ed malware.

To decrypt an encrypted string, one must:

1. Swap the first and last two bytes.
2. Base64 decode the string. Actually, the code of the *APK Protect*-ed sample hides the call to Base64 decoding methods. It does not call the method directly but via Java reflection. The path for Base64 (android.util.Base64) is decoded from a XOR-encrypted string, and the method name (decode) is created by picking up the appropriate characters in the path name.
3. XOR the decoded string.
4. Decrypt the result using the hard-coded key '#safeguar'.

Knowing this, it is possible to implement one's own string decryptor. The implementation must be adapted to each sample as XOR keys change.

```
$ java SmsDecrypt
Processing string: ==aFgIDU0oPWgoK...
d64xor: 96500db3f2242a4b2ac920e4...
Decrypting: ybbc[CENSORED]icp.cc
```

An alternative to this labour-intensive method (which has to be repeated for every single sample) is to send the sample for analysis by *Andrubis* [12]. As shown in Figure 2, *Andrubis* does the work for us, showing the URLs the malware contacts and the decryption key.

Figure 2: Andrubis analysis results showing the decryption key and output.

2.4 HoseDex2Jar

HoseDex2Jar is a packer that was released a year ago. It is quite simple, and thus easy to circumvent. It is based on the premise that, normally, DEX headers are exactly 0x70 bytes long. However, it was found that *Android* does not strictly enforce the header size, so one can add data at the end of the header.

This is precisely what *HoseDex2Jar* does:

1. Encrypts the DEX.
2. Creates a new DEX for the packed app.
3. Puts the encrypted DEX into the new DEX header (e.g. end).
4. Sets the DEX header size.

This is easy to spot: look for DEX files with header size greater than 0x70 (= 112). This can be done using *Hidex*, which displays a warning:

```
$ ~/dev/hideandseek/hidex/hidex.pl --input classes.
dex-hosed
WARNING: strange header size: 136080
DEX Header of file:
Magic : 6465780a30333500
```

To reverse hosed applications, Tim Strazzere released a de-hoser [13]. We have not encountered any hosed malware yet.

2.5 Bangcle

Bangcle [14] is an online service for packing *Android* executables. The process is the following:

1. Register on *Bangcle* to get a user account.
2. Download the *Bangcle Assistant* tool.
3. Use the tool to upload your package. At this point, *Bangcle* servers do check that the package is not malicious, but they can be fooled.
4. Retrieve the protected app (for a signed version of the protected app, a keystore must be uploaded by the user).

The packing process modifies the structure of the original APK quite extensively:

- The name of the application is changed (always) to com.secapk.wrapper.ApplicationWrapper.
- There are new assets and new native libraries.
- The manifest is modified.
- The classes.dex file is completely modified. The original activity no longer exists and is replaced by a generic placeholder.

There are several ways to detect the use of *Bangcle*: the application's name 'com.secapk.wrapper.ApplicationWrapper', the presence of an asset named 'bangcle classes.jar', the presence of native libraries named 'libsecexe' and 'libsecmain', and class names such as 'FirstApplication' or 'ACall'.

The difficulty lies in reversing samples that are protected with *Bangcle*. Though this has yet to be confirmed, [15] claims that 'a growing percentage of malware, such as bank Zeus, SMS Sender, and re-packaged applications, are packed by [the *Bangcle*] service'. We spotted *Bangcle* in Android/Feejar.B.

Bangcle is particularly resistant to reverse engineering because:

- Functions exported by native libraries have obfuscated names.
- Several libc functions, like mmap2, munmap, open, read, write, close and msync, are hooked. It is likely that ptrace is hooked too, as debuggers have difficulty attaching to certain *Bangcle* processes.
- The libraries are compiled with stack protection enabled (stack chk guard).
- The real application is encrypted, and only decrypted in memory at runtime. In particular, the RC4 algorithm is used [16].

Interesting analyses can be found in [17, 18] (in Chinese).

The solution we used in order to gain a better understanding of packed malware consists of using *IDA Pro*'s ARM remote debugger. The remote debugger server is on the *Android* platform, while it communicates with *IDA Pro* on a remote host. We attach to the thread of a process that loads libsecmain and dump the memory when it is decrypted (see Figure 3).

3. CUSTOM OBFUSCATION

Malware authors have been very active in designing their own obfuscation techniques. Some of the techniques are basic, and others are more complicated:

Figure 3: Decrypted memory of a protected application.

Android malware name	Year of discovery	Obfuscation
SmsSend.N 66699d5c55f442203d5b933e87339d3c2f7f256037b45d6ad3ba9e00a6500851	2012	*ProGuard*-ed
Plankton.B!tr 6600fdf4e758bfab3b73ab26270dd9f4c02847f144e28c255919aee7d91a0f11	2011	*ProGuard*-ed parts
DroidKungFu.D!tr 938efb5bdc96d353b28af57da2021b6a3c5a64452067059bf50d7fb7c7a66426	2011	*ProGuard*-ed parts
Dendroid.A!tr 0b8ba0c6cebe5695639bf1b282b52f126dba733f3c204e37615a3ba5f7dd6fe8	2014	*DexGuard*-ed
Rmspy.A!tr 57e37d4cfc9e0ea9287ba72185c12bb4ccf4e1a56041f3c3d12c31be1aaf5506	2013	*DexGuard*-ed
0bad.A b65c352d44fa1c73841c929757b3ae808522aa2ee3fd0a3591d4ab67598d17	2013	*DexGuard*-ed
SmsSend.ND 3aee81db24540fb6b3666a38683259fd32713187ec6e0b421da9b91bd216205f	2014	*APK Protect*-ed
Feejar.B 0000350c0792f61ee513f40bd9a42d09144cc6a3c4f2171f812ef415a9a51640	2014	*Bangcle*

Table 1: Examples of malware using off-the-shelf obfuscation tools.

- **Using very long class names to defeat tools.** This technique has been mentioned in [19] and seen in the wild in Android/Mseg.A!tr.spy (sha256 hash: cc42f8a1fc6805a9deeaae198fb4580b304b51489dec4209929a09b9c3868aee).
- **Using nops to modify the bytecode flow.** This was mentioned in [20], and is extremely common.
- **Path obfuscation.** For example, in an Android/Plankton sample, the normal Airpush SDK path is replaced by com/OajgOKqg/FYmaEVCV92392.
- **Path phishing.** This consists of using a well known (legitimate) path and hijacking it for illegitimate purposes. For example, in Android/RuSMS.AO, com.adobe.air (normally used by *Adobe AIR*) is used to

hide the malicious functionality. Path phishing is very common too.

- **Hiding packages, JARs etc. in raw resources or assets.** Table 2 lists some examples of malware samples that hide malicious packages in resource files. For example, Android/SmsZombie.A!tr hides a malicious package in a JPG named 'a33.jpg' in the assets directory. Android/Gamex.A!tr hides an encrypted malicious package in an asset named 'logos.png'. This is close to what is referred to as a polyglot file [21], i.e. a file which is valid and meaningful for different formats. In Gamex, the asset 'logos.png' is not a valid PNG (thus not really a polyglot), but a ZIP. However, it has the peculiarity of being a valid ZIP file as such, and also another valid ZIP file when XOR'ed with the right key (18).
- **Hiding bytecode.** (For instance, abusing linear sweep disassemblers [22].) According to [16], this is encountered in up to 30% of obfuscated samples. For example, we find it in Android/Agent.SZ!tr. This technique can be detected by looking for Dalvik bytecode that does a goto followed by fill-array-data opcode (see Figure 4). Reverse engineers can use the script androdis.py released with *Androguard* [4].
- **String table.** Android/GinMaster.L (sha256 hash: e86467622b8faf903edcebe0a57b85c036aa59b1820694ef326b50062dfdc910) builds its own string table as a char array (see below array named 'OGqHAYq8N6Y6tswt8g').

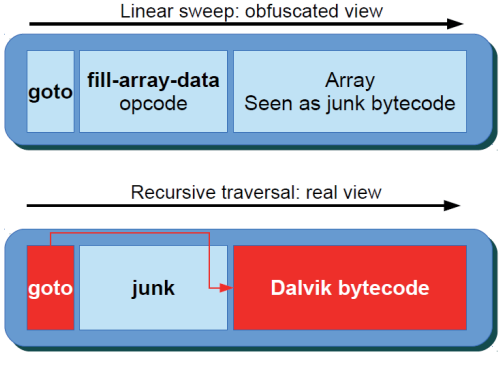

Figure 4: Bytecode is hidden in the array of fill-array-data and invisible to Dalvik disassemblers, which use linear sweep.

```
package Eg9Vk5Jan;
  class x18nAzukp {
    final private static char[][] OGqHAYq8N6Y6tswt8g;
    static x18nAzukp()
    {
      v0 = new char[][48];
      v1 = new char[49];
      v1 = {97, 0, 110, 0, 100, 0, 114, 0, 111, ...
      v0[0] = v1;
      v2 = new char[56];
      v2 = {... 110, 0, 97, 0, 103, 0, 101, 0, 114, 0};
      v0[1] = v2;
      ...
    }
protected static String rLGAEh9JeCgGn73A(int p2) {
      return new String(
Eg9Vk5Jan.x18nAzukp.OGqHAYq8N6Y6tswt8g[p2]);
}
...
new StringBuilder(x18nAzukp.rLGAEh9JeCgGn73A(43))
```

The rest of the code references the strings in that char array. So you never see the strings directly, but instead indirect calls like rLGAEh9JeCgGn73A(43) etc.

- **Naïve encoding or encryption.** Many samples use Base64 (e.g. Android/Stels), XOR (Android/FakeInst), Caesar (Android/Pincer), or simply chop the data into several chunks (e.g. Android/RuSMS.AO below).

```
String.valueOf("http") + "://" + "ap" + "iad" + "ver"
+ "t.ru");
```

Android malware name	Year of discovery	Obfuscation
Gamex.A!tr ae7a20692250f85d7a2ed205994f2d26f2d695aef15a9356938454bccbbbd069	2013	Assets contain a file named 'logos.png'. This is not a PNG, but a ZIP, and it unzips to different valid outputs depending on whether XOR'ed with key (18) or not.
SmsZombie.A!tr 45099416acd51a4517bd8f6fb994ee0bb9408bdd80dd906183a3cdb4b39c4791	2012	Hides malicious package in 'a33.jpg'.
DroidCoupon.A!tr 94112b350d0fece0a788fb042706cb623a55b559ab4697cb10ca6200ea7714	2011	The Rage Against the Cage exploit is hidden in a PNG file in raw resources.

Table 2: Examples of samples hiding malicious packages in resource files.

Android malware name	Year of discovery	Obfuscation
Agent.SZ!tr 1673f18d7f5778dc4875f10dc507fc9d59be82eaf5060dfc4bfa7a7d6007f7df	2014	Hides bytecode using [22].
RuSMS.AO 768cfe8f5ca52c13508b113875f04a68174387e44321d68c132e2a7b6e0cbe0a	2014	Strings are cut into several parts so as not to be spotted. Uses *Adobe*'s *AIR* namespace so as not to look suspicious.
Stels.A!tr 03c1b44c94c86c3137862c20f9f745e0f89ce2cdb778dc6466a06a65b7a591ae	2013	Custom base64 to decode the URL.
Pincer.A!tr.spy fee013fcbbd30ef37c99eab56aa550d27e00e69150f342b80b08d689a98ccefe	2013	Caesar shift to read C&C URL and phone number.
Tascudap.A!tr 0be2a4b3a0e68769fa5b3c9cd737e0e87abd6cddb29a7e1fdf326f407a658b54	2013	*ProGuard*-ed. URL is generated from custom encryption. Malware also uses AES with a key which is built from a hard-coded seed.
SaurFtp.A!tr.spy e769fdf8f2e1a5311ef089c422a7c0cb360d77082d7d1ef139a95c9321ec40	2013	C&C URL is XOR encrypted.
FakeInst.A!tr.dial ac118892190417c39a9ccbc81ce740cf4777fde1	2012	SMS text bodies and phone numbers are hidden in a text chunk inside a PNG and 'encrypted' using XOR.
Vdloader.A!tr c17ca0937891974d852f619d3b7be5defc79c6d7bf6f3beeebb991e684563902	2012	Custom encryption: decrypted = char - pos.
Temai.A!tr 14354ddd2a9d63b3b5c5db94fd717953572f1293f291e26bc7a4725be4b0b3b8	2012	Downloads another password-protected ZIP file. This ZIP file is decrypted with a hard-coded password, and is a script that opens a backdoor on the phone.
LuckyCat.A!tr 5d2b0d143f09f31bf52f0a0810c66f94660490945a4ee679ea80f709aae3bd	2012	XOR encryption of traffic sent to attacker.
Pjapps.A!tr 02329dc3aa91b5175461b3c298b411fe9d35c8425a5fa485c3a3c4daa12c7d2a	2011	URL to contact is 'encrypted' with a simple algorithm where you only keep one character in every two.

Table 3: A non-exhaustive list of malicious Android samples using custom obfuscation techniques.

Some other samples are more creative: Android/Vdloader encrypts characters by subtracting their position in the string (first character minus 0, second character minus 1, etc.), while Android/Tascudap uses its own algorithm. Table 3 lists a few examples of samples that use their own custom techniques.

- **Encryption.** Malware authors use encryption for various reasons [23]: to conceal strings and exploits, to encrypt communication with the C&C server, to send encrypted emails, and so on. Recent statistical analysis of our *Android* malware database showed that 27% of malware samples use encryption[3]. For example, Android/Geinimi uses DES, Android/SmsSpy.HW!tr uses Blowfish, and Android/RootSmart uses AES. Also note that *Android*'s License Verification Library (LVL) uses AES-based obfuscation:

[3] This percentage should be understood as an approximate maximum, as some pieces of malware use encryption but in the 'legitimate' parts of their code, not for malicious intent. This has been computed over a set of 460,493 *Android* samples.

1. A hard-coded prefix ('com.android.vending.licensing.AESObfuscator-1|') is added to the string to be obfuscated.
2. The string is encrypted using AES in CBC mode and PKCS5 padding. The key and IV are hard coded.
3. The encrypted result is encoded with Base64.

```
package com.android.vending.licensing;
...
public class AESObfuscator implements Obfuscator {
...
  private static final String CIPHER_ALGORITHM =
    "AES/CBC/PKCS5Padding";
  private static final byte[] IV = { 16, 74, 71, -80...
  private static final String header =
    "com.android.vending.licensing.AESObfuscator-1|";
```

LVL's obfuscation is used in some samples of Android/Plankton.

In most cases, the encryption is hard coded. However, some malware do not actually hard code it, but regenerate the key from a random number generator seeded with a hard-coded seed. For instance, this technique is used by Android/RootSmart and Android/Fjcon.

Table 4 lists a few examples of samples that use encryption as an obfuscation technique.

The reversing of samples using cryptography usually means copy-pasting the decompiled Java code that handles the decryption (perhaps with slight adaptation) and running it independently on the data to decrypt. Python comes in handy for writing quick decryption code as there are many decryption libraries. For example, we decrypt an encrypted XML configuration file of Android/SmsSpy.HW!tr using the following code:

```
import Crypto
from Crypto.Cipher import Blowfish

def PKCS5Padding(string):
        byteNum = len(string)
        packingLength = 8 - byteNum % 8
        appendage = chr(packingLength) * packingLength
        return string + appendage

def DoDecrypt(string):
        key = 'tisWsx2xivgQXRxq'
        c1 = Blowfish.new(key, Blowfish.MODE_ECB)
        packedString = PKCS5Padding(string)
        return c1.decrypt(packedString)
```

- **Loading non-Dalvik code.** For instance, Android/DroidKungFu.G loads an ELF executable which holds the payload. Android/FakePlay.B!tr holds a malicious JavaScript that implements click fraud. On *Windows Mobile*, we have seen WinCE/Redoc loading Basic via Basic4PPC. Basic4Android exists, but we haven't seen any malicious samples using it yet. Flash code could hold malicious payloads too.

4. OBFUSCATION IN THE FUTURE

As we have seen in the previous sections, malware authors are interested in obfuscating their code, and if *Android*'s crime scene continues to follow the evolution of *Windows* malware (as it has done until now), then we are only at the beginning of the story. In particular, packers are likely to normalize as UPX (and others) did for *Windows*. In this section, we prepare for techniques malware authors might use in the near future.

In [24], Bremer demonstrates that it is possible to inject bytecode into nearly any class, with only minor modification. The class needs to have at least a virtual function, and the injection code must read the bytecode to inject as a string and replace the address of that virtual method with the address of the string. An attacker could use this technique for evil:

- Create a genuine application which acts as a bytecode loader.
- Read (possibly decrypt) bytecode to inject from a resource, or a remote host.
- Inject that bytecode into the genuine application and have it perform a malicious action.

Fortunately, for now, Bremer's technique is limited to returning integers (see Figure 5). However, there is no doubt that it can (and perhaps will) be extended in the future. Anti-virus analysts may try to detect the bytecode loading code, which is based on the iput-quick and invoke-virtual opcodes, however a generic signature will be difficult to design as there are several possible variations and potential false positives.

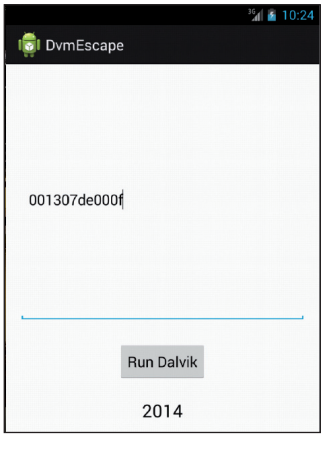

Figure 5: Injecting constant 0x07de = 2014 bytecode in Bremer's proof of concept.

In [9], we demonstrated that it is possible to hide methods from disassemblers. This is potentially interesting to attackers

Android malware name	Year of discovery	Obfuscation
SmsSpy.HW!tr.spy 69cb8163e959e60b0e024457449c4c8d2586ed3cf2e46351fdedec8ef64a7a79	2014	Contains an asset, 'data.xml', which is encrypted using Blowfish ECB and a hard-coded key.
Agent.BH!tr.spy 5c89b1b008efee0c3a6294d0a02c77845cd91d1faad5df6bf7b6d54a5f3cd0d3	2014	Sends emails using SMTP with TLS authentication.
GMaster.B 18ad4064750a0e4733a828794f76e6d5b4e60b0fc79c54ba1d8955db82e489d2	2013	Uses Triple DES EDE, CBC with PKCS7 padding to send JSON object containing IMSI, IMEI and various OS parameters.
FakeDefend.A!tr 5ad411cdcbf68f8f449c470b514ed4ee31cafdf2997c3cd0e6af032750edca58	2013	List of fake infections to display on the device is encrypted with AES.
NotCompatible.A!tr.bdr 2c5e656af90044cf5cc694519a42477cb18bd4b2722b1474cdead4a8748d3f70	2012	C&C URLs located in a raw resources file are encrypted using AES in ECB mode. The encryption key is the sha256 hash of a hard-coded value.
Fjcon.A!tr 39f64285207b8184c4940252e2fadf7e903ea0a611bc1bebc84d33a8b692bada	2012	URLs are encrypted using AES. The encryption key is generated using a SHA-1-based PRNG, seeded with value 125.
RootSmart.A!tr.dldr ccdfe44762c1c3492f0ca4135afdc258fa7b39ecb9c156a6f0f15e9d05a3ac7e	2012	Domain name is encrypted using AES. The encryption key is generated using a SHA-1-based PRNG.
BaseBridge.A!tr 07e1349dfc31e9e6251a2920521e453f71ce296352861902b99734a8a7b7f554	2011	Uses variable and string obfuscation. Uses AES encryption.
Hongtoutou.A!tr 4ae1c0faa06ee4dfb6c96b6537d027e90c870d7d3ddcfd5fcde680be9dc51c69	2011	Encrypts phone info sent to attacker, using DES.
SndApp.A!tr.spy 7e057d3133639374195da6c9805fd7f0edb818047d49955c3f5291f01b94	2011	Uses AES in CBC mode.
JSmsHider.A!tr 0ea2d931ebb55668ecb101304f316725f6fa1574dbb191dc2d647c65b3aebf	2011	Encrypts its communication with the C&C using DES.
Geinimi.A!tr 2e998614b17adbafeb55b5fb9820f63aec5ce8b4	2011	Communication with the C&C is encrypted, so are commands and strings inside the binary. The algorithm is DES, and the key is hard coded.

Table 4: Examples of malicious samples using cryptography as an obfuscation technique.

Android malware name	Year of discovery	Obfuscation
FakePlay.B!tr 4bde46accfeb2c85fe75c6dd57bba898fbb3316f7c4be788bc18676451b54561	2013	The malicious payload is in the JavaScript.
DroidKungFu.G!tr b03a8fc6d508e16652b07fb0c3418ce04bd9a3c8e47a3b134615c339e6e66bf7	2012	Asset named 'mylogo.jpg' is a valid JPG file, but it also contains an ELF.

Table 5: Examples of samples loading non-Dalvik malicious code.

if they locate their malicious code in those hidden parts. Fortunately, the technique was published along with the *Hidex* detection tool [8]. (For more information, please see slides from Insomni'hack 2014 [9].)

Ange Albertini has released a Python script [25, 26] that is able to manipulate the encrypted output of AES or DES so that it looks like a customizable PNG, JPG or sound file. A malware author might be interested in using this technique to hide an APK in assets or resources. He/she would create an application which looks fairly genuine, with a seemingly innocent PNG as an asset. The code would load the asset and decrypt it with a hard-coded key to reveal the real, evil APK. The malicious APK would then be installed on the device. The attack is feasible, and such an APK can be created using AngeCryption. However, a few hacks are necessary: the End Of Central Directory (EOCD), which marks the end of the ZIP file, must be duplicated and padded to 16 bytes (for encryption with AES). We are currently working on a proof of concept and detection tool.

5. CONCLUSION

We have seen *Android* malware authors use plenty of different techniques to obfuscate their code. With new tools like *Bangcle*, *APK Protect* and *DexGuard*, we fear that mobile malware will become increasingly difficult to reverse in the near future – not to mention techniques such as bytecode injection, method hiding or AngeCryption which haven't been seen on the malware scene, yet.

In this paper, we have shown that we are not totally helpless in the face of obfuscation. A few simple, but well chosen Unix find/grep commands are useful for understanding what is happening. And in most cases, we have managed to reverse samples with known existing tools such as *baksmali*, *apktool* and *Androguard* – these tools usually work adequately (or nearly), it is more a matter of looking at the right location. Moreover, encryption, which sounds frightening at first, does not turn out to be so difficult to reverse in practice: we just have to write a few lines of code to decrypt the ciphertext. For situations in which reversing remains difficult, we have provided a few enhancements to *Hidex*, a Perl script which assists reverse engineers in detecting some situations, and helps with the renaming of non-ASCII strings used by some obfuscators.

So we are not helpless, but if we want to keep pace with the techniques malware authors are likely to use in the near future, we had better focus on tools and research in this area as soon as possible.

ACKNOWLEDGEMENTS

We thank Ange Albertini, Jurriaan Bremer, Anthony Desnos, Robert Lipovsky and Miroslav Legen for their help.

REFERENCES

[1] Implementing an Obfuscator. https://developer.android.com/google/play/licensing/adding-licensing.html#impl-Obfuscator.

[2] DexGuard. http://www.saikoa.com/dexguard/.

[3] JD-GUI. http://jd.benow.ca/.

[4] Androguard. https://code.google.com/p/androguard/.

[5] Nihilus. Reversing DexGuard 5.x. version 1.

[6] Fallière, N. A look inside DexGuard. http://www.android-decompiler.com/blog/2013/04/02/a-look-inside-dexguard/.

[7] Smali. https://code.google.com/p/smali/.

[8] Hidex. https://github.com/cryptax/dextools/tree/master/hidex.

[9] Apvrille, A. Playing Hide and Seek with Dalvik Executables. In Hack.Lu, October 2013. http://www.fortiguard.com/uploads/general/hidex_insomni.pdf.

[10] Android. Dalvik Executable Format. http://source.android.com/devices/tech/dalvik/dex-format.html.

[11] APK Protect. http://www.apkprotect.com/.

[12] Andrubis. http://anubis.iseclab.org/.

[13] Dehoser. https://github.com/strazzere/dehoser/.

[14] Bangcle. http://www.bangcle.com/.

[15] Yu, R. Android packer: facing the challenges, building solutions. In Proceedings of the 24th Virus Bulletin International Conference (VB2014). (To be published.)

[16] Lipovsky, R. Obfuzzcation issues. In CARO Workshop, May 2014.

[17] Jia, J. Android APK. May 2013. http://blog.csdn.net/androidsecurity/ (in Chinese).

[18] Pan, B. Bangcle and crack the encryption method. December 2013. http: //pandazheng.blog.163.com/blog/static/176817209201311931705/ (in Chinese).

[19] Strazzere, T. Dex Education: Practicing Safe Dex. BlackHat USA, July 2012. http://www.strazzere.com/papers/DexEducation-PracticingSafeDex.pdf.

[20] Mody, S. 'I am not the D'r.0,1d you are looking for': an Analysis of Android Malware Obfuscation. In Proceedings of the 23rd Virus Bulletin International Conference, pp.105–113, October 2013.

[21] Albertini, A. This PDF is a JPEG; or This Proof of Concept is a Picture of Cats. Journal of PoC – GTFO, 3, 2014.

[22] Schulz, P. Dalvik Bytecode Obfuscation on Android, July 2012.

[23] Apvrille, A. Cryptography for Mobile Malware Obfuscation. In RSA Europe Conference, 2011. http://www.fortiguard.com/files/NMS-305-Apvrille-Revised.pdf.

[24] Bremer, J. Abusing Dalvik Beyond Recognition, October 2013. Hack.lu.

[25] Albertini, A. When AES(*)=*, April 2014. https://corkami.googlecode.com/svn/trunk/ src/angecryption/slides/AngeCryption.pdf.

[26] Angecrypt.py. http://corkami.googlecode.com/svn/trunk/src/angecryption/angecrypt.py.

Editor: Martijn Grooten
Chief of Operations: John Hawes
Security Test Engineers: Scott James, Tony Oliveira
Sales Executive: Allison Sketchley
Editorial Assistant: Helen Martin
Perl Developer: Tom Gracey
Consultant Technical Editors: Dr Morton Swimmer, Ian Whalley

Tel: +44 (0)1235 555139. Fax: +44 (0)1865 543153
Email: editorial@virusbtn.com
Web: http://www.virusbtn.com/