UNPACKING THE PACKED UNPACKER: REVERSING AN ANDROID ANTI-ANALYSIS NATIVE LIBRARY

Maddie Stone
Google, USA
maddiestone@google.com

ABSTRACT

Malware authors implement many different techniques to frustrate analysis and make reverse engineering malware more difficult. Many of these anti-analysis and anti-reverse engineering techniques attempt to send a reverse engineer down a different investigation path or require them to invest large amounts of time reversing simple code. This talk analyses one of the most interesting anti-analysis native libraries we’ve seen in the Android ecosystem. No previous references to this library have been found. We’ve named this anti-analysis library ‘WeddingCake’ because it has lots of layers.

This paper covers four techniques the malware authors used in the WeddingCake anti-analysis library to prevent reverse engineering. These include: manipulating the Java Native Interface, writing complex algorithms for simple functionality, encryption, and run-time environment checks. This paper discusses the steps and the process required to proceed through the anti-analysis traps and expose what the developers are trying to hide.

INTRODUCTION

To protect their code, authors may implement obfuscation, encryption, and anti-analysis techniques. There are both legitimate and malicious reasons why developers may want to prevent analysis and reverse engineering of their code. Legitimate developers may want to protect their intellectual property, while malicious developers may want to prevent detection. This paper details an Android anti-analysis native library used by multiple malware families to prevent analysis and detection of their malicious behaviours. Some variants of the Chamois malware family [1] use this anti-analysis library, which has been seen in over 5,000 unique Android APKs. The APK with SHA256 hash e8e1bc048ef123a9757a9b27d1bf53c092352a26bdbf9bdc1010915b5cadac is used as the sample for this paper.

Introduction to the Java Native Interface (JNI)

The sample Android application includes a native library to hide the contents and functionality of native code. The Java Native Interface (JNI) allows developers to define Java native methods that run in other languages, such as C or C++, in the application. This allows bytecode and native code to interface with each other. In Android, the Native Development Kit (NDK) is a toolset that permits developers to write C and C++ code for their Android apps [2]. Using the NDK, Android developers can include native shared libraries in their Android applications. These native shared libraries are .so files, a shared object library in the ELF format.

In this paper, the terms ‘native library’, ‘.ELF’, and ‘.so file’ are used interchangeably to refer to the anti-analysis library. The anti-analysis library that is detailed in this paper is one of these Android native shared libraries.

The bytecode in the .dex file of the Android application defines the native methods [3]. These native method definitions pair with a subroutine in the shared library. Before the native method can be run from the Java code, the Java code must call System.loadLibrary or System.load on the shared library (.so file). When the Java code calls one of the two load methods, the JNI_OnLoad() function is called from the shared library. The shared library needs to export the JNI_OnLoad() function.

In order to run a native method from Java, the native method must be ‘registered’, meaning that the JNI knows how to pair the Java method definition with the correct function in the native library. This can be done either by leveraging the RegisterNatives JNI function or through ‘discovery’ based on the function names and function signatures matching in both Java and the .so file [4]. For either method, a string of the Java method name is required for the JNI to know which native function to call.

CHARACTERISTICS OF THE ANTI-ANALYSIS LIBRARY

WeddingCake, the anti-analysis library discussed in this paper, is an Android native library, an ELF file, included in the APK. In the sample, the anti-analysis library is named lib/arm64-v7a.so.

Naming

Within the classes.dex of the APK, there is a package of classes whose whole name is random characters. For the sample described in this paper, the class name is ses.fdkxxcr.udayjfrgxp.ojoyqmos.xien.xmdowmbkdqfgk. This class declares three native methods: quaqrq, lxxjkwu, and vxeg.

The native library discussed in this paper is usually named lib[3-8 random lowercase characters].so. However, we’ve encountered a few samples whose name does not match this convention. All APK samples that include WeddingCake use different random characters for their class and function names. It is likely that WeddingCake provides tooling that generates new random names each time it is compiled.

Variants

The most common version of the library is a 32-bit ‘generic’ ARM (armeabi) ELF, but I’ve also identified 32-bit ARMv7 (armeabi-v7a), ARM64 (arm64-v8a), and x86 (x86) versions of the library. All of the variants include the same functionality. If not otherwise specified, this paper focuses on the 32-bit ‘generic’ ARM implementation of WeddingCake because this is the most common variant.

As an example, the APK with SHA256 hash 92e80872cf49f33c53993d52290af2e87cbe55db4adff1ba97297340f23e0,
which is different from the one analysed in this paper, includes three variants of the anti-analysis library: generic ARM, ARMv7, and x86.

<table>
<thead>
<tr>
<th>Anti-analysis lib file paths</th>
<th>Anti-analysis library ‘type’</th>
</tr>
</thead>
<tbody>
<tr>
<td>lib/armeabi/librxovdx.so</td>
<td>32-bit ‘generic’ ARM</td>
</tr>
<tr>
<td>lib/armeabi-v7a/librxovdx.so</td>
<td>32-bit ARMv7</td>
</tr>
<tr>
<td>lib/x86/libaojp.so</td>
<td>x86</td>
</tr>
</tbody>
</table>

Table 1: Anti-analysis lib paths in

<table>
<thead>
<tr>
<th>Key signatures of the ELF</th>
</tr>
</thead>
<tbody>
<tr>
<td>There are some signatures that help identify ELF files as a WeddingCake anti-analysis library:</td>
</tr>
<tr>
<td>• Two strings under the .comment section in the ELF:</td>
</tr>
<tr>
<td>- Android clang version 3.8.275480 (based on LLVM 3.8.275480)</td>
</tr>
<tr>
<td>- GCC: (GNU) 4.9.x 20150123 (prerelease)</td>
</tr>
<tr>
<td>• The native function names defined in the APK do not exist in the shared library</td>
</tr>
</tbody>
</table>

For the 32-bit generic ARM version of the library, when loaded into IDA Pro, 

\[
\text{JNI\_OnLoad (Figure 1)} \text{ is an exported function name, but does not exist in ‘functions’ because there are 12 bytes (three words) that are defined as data, which inhibit IDA’s ability to identify the function. The bytes defined as data are always at offsets +0x24, +0x28, and +0x44 from the beginning of the JNI\_OnLoad function.}
\]

ANALYSING THE LIBRARY

The JNI\_OnLoad is the starting point for analysis because there are no references to the native methods that were defined in the APK. For this sample, the following three methods were defined as native methods in 

\[
\text{ses.fdkxxcr.UDayjfrgxp.ojoyqmosj.xien.xmdowmbdkdmgk}.
\]

There are no instances of these strings existing in the native library being analysed. As described in the ‘Introduction to JNI’ section, in order to call a native function from the Java code in the APK, the ELF must know how to match a Java method (as listed
previously) to the native function in the ELF file. This is done by registering the native function using `RegisterNatives()` and the `JNINativeMethod` struct [5]. We would normally expect to see the Java native method name and its associated function signature (`(Ljava/lang/Object;)I`) as strings in the ELF file. Since we do not, the ELF file is probably using an anti-analysis technique.

Because `JNI_OnLoad` must be executed prior to the application calling one of its defined native methods, I began analysis in the `JNI_OnLoad` function.

In the sample, the `JNI_OnLoad()` function ends with many calls to the same function. This is shown in Figure 2. Each call takes a different block of memory as its argument, which is often a signal of decryption. In this sample, the subroutine at 0x2F30 (sub_2F30) is the in-place decryption function.

### In-place decryption

To obscure its functionality, this library’s contents are decrypted dynamically when the library is loaded. The decryption algorithm used in this library was not matched to a known encryption/decryption algorithm. The decryption function, found at sub_2F30 in this sample, takes the following arguments:

- `encrypted_array`: Pointer to the encrypted byte array (bytes to be decrypted)
- `length`: Length of the encrypted byte array
- `word_seed_array`: Word (each value in array is 4 bytes) seed array
- `byte_seed_array`: Byte (each value in array is 1 byte) seed array

```
sub_2F30(Byte[] encrypted_array, int length, Word[] word_seed_array, Byte[] byte_seed_array)
```

### Generating the seed arrays

The decryption function takes two seed arrays as arguments each time it is called: the word seed array and the byte seed array. These two arrays are generated once, beginning at 0x1B58 in this sample, prior to the first call to the decryption function. The IDA decompiled code for the generation of the two arrays, `byte_seed_array` and `word_seed_array`, is shown in Listing 1.

```assembly
Listing 1: The IDA decompiled code for the generation of the two arrays, byte_seed_array and word_seed_array.
```
function. The byte array is created first; in this sample, it's generated at 0x1B58. The word array is created immediately after the byte array initialization at 0x1BD0. The word seed array and byte seed array are the same for every call to the decryption function within the ELF and are never modified.

The author of this code obfuscated the generation of the seed arrays. The IDA decompiled code for the generation of the two arrays, byte_seed_array and word_seed_array, is shown in Listing 1.

These algorithms output the byte_seed_array and word_seed_array shown in Listing 2. The author of this code intended to frustrate the reverse engineering process of this library by writing complex algorithms which would require more investment of effort, time and skill to reverse engineer. Using a complex algorithm to accomplish a simple task is a common anti-reverse engineering technique.

Knowing that these arrays are static, an analyst could dump the arrays any time post-initialization, thus bypassing this anti-reversing technique.

### Decryption algorithm

The overall framework of the in-place decryption process is:

1. Decryption function is called on an array of encrypted bytes.
2. Decryption is performed.
3. Encrypted bytes are overwritten by the decryption bytes.

This process is repeated in JNI_OnLoad() for each encrypted array. I did not identify the decryption algorithm used in the library as being a variation of a known encryption algorithm. The Python code I wrote to implement the decryption algorithm is shown in Listing 3.

I wrote an IDA Python script to statically decrypt the contents of the ELF so that reverse engineering could continue. This script and description is provided in the Appendix.

### Decrypted contents

Each of the encrypted arrays decrypts to a string. Before-and-after samples of the encrypted bytes and the decrypted bytes at

```python
byte_seed_array = [0x00, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f, 0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f, 0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f, 0x40, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49, 0x4a, 0x4b, 0x4c, 0x4d, 0x4e, 0xf0, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59, 0x5a, 0x5b, 0x5c, 0x5d, 0x5e, 0x5f, 0x60, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79, 0x7a, 0x7b, 0x7c, 0x7d, 0x7e, 0x7f, 0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89, 0x8a, 0x8b, 0x8c, 0x8d, 0x8e, 0x8f, 0x90, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98, 0x99, 0x9a, 0x9b, 0x9c, 0x9d, 0x9e, 0x9f, 0xa0, 0xa1, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7, 0xa8, 0xa9, 0xaa, 0xab, 0xac, 0xad, 0xae, 0xaf, 0xb0, 0xb1, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6, 0xb7, 0xb8, 0xb9, 0xba, 0xbb, 0xbc, 0xbd, 0xbe, 0xbf, 0xc0, 0xc1, 0xc2, 0xc3, 0xc4, 0xc5, 0xc6, 0xc7, 0xc8, 0xc9, 0xca, 0xcb, 0xcc, 0xd0, 0xd1, 0xd2, 0xd3, 0xd4, 0xd5, 0xd6, 0xd7, 0xdb, 0xda, 0xdb, 0xdc, 0xde, 0xdf, 0xe0, 0xe1, 0xe2, 0xe3, 0xe4, 0xe5, 0xe6, 0xe7, 0xe8, 0xe9, 0xea, 0xeb, 0xec, 0xed, 0xee, 0xef, 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff]

Listing 2: The byte_seed_array and word_seed_array.
```python
def decrypt(encrypted_bytes, length, byte_seed_array, word_seed_array):
    if (encrypted_bytes is None):
        print ( "encrypted_bytes is null. -- Exiting ")
        return
    if (length < 1):
        print ( "encrypted_bytes len < 1 -- Exiting ")
        return
    reg_4 = ~(0x00000004)
    reg_0 = 4
    reg_2 = 0
    reg_5 = 0
    do_loop = True
    # Address 0x2F58 in Sample e8e1bc048ef123a9757a9b27d1bf53c092352a26dbf9fbdc10109415b5cadac
    while (do_loop):
        reg_6 = length + reg_0
        reg_6 = encrypted_bytes[(reg_6 + reg_4)]
        if (reg_6 & 0x80):
            if (reg_5 > 3):
                return
            reg_6 = reg_6 & 0x7F
            reg_2 = reg_2 & 0xFF
            reg_2 = reg_2 << 7
            reg_2 = reg_2 | reg_6
            reg_0 = reg_0 + reg_4 + 4
            reg_3 = length + reg_0 + reg_4 + 2
            reg_5 += 1
            if (reg_3 & 0x80000000 or reg_3 <= 1):
                return
        else:
            do_loop = False
            reg_5 = 0xF0 & reg_6
            reg_3 = length + reg_0 + reg_4
            reg_1 = reg_3 + 1
            if (reg_0 == 0 and reg_5 != 0):
                return
            # Address 0x2F9A in Sample e8e1bc048ef123a9757a9b27d1bf53c092352a26dbf9fbdc10109415b5cadac
            reg_5 = reg_1
            reg_1 = (reg_2 << 7) + reg_6
            byte_FF = 0xFF
            reg_1 = reg_1 & byte_FF
            last_byte = reg_1
            if (reg_5 == 0 or reg_5 & 0x80000000 or last_byte == 0 or signed_ble(reg_3, last_byte)):
                return
            reg_1 = (reg_4 + 4)
            reg_1 = (reg_1 * last_byte)
            reg_1 += length
            crazy_num = reg_1 + reg_0 + reg_4
            if (crazy_num < 1):
                return
            new_index = reg_1 + reg_0
            reg_5 = 0
    # Address 0x2FD8 in Sample e8e1bc048ef123a9757a9b27d1bf53c092352a26dbf9fbdc10109415b5cadac
    while (1):
        byte = encrypted_bytes[reg_5]
        reg_0 = byte << 2
        reg_6 = word_seed_array[byte]

Listing 3: Python code to implement the decryption algorithm (continues on next page).
```
reg_0 = 0xFF - reg_6
if (not reg_6 & 0x80000000):
    reg_6 = reg_0
reg_0 = reg_0 & last_byte
reg_0 = new_index + reg_1
reg_0 = encrypted_bytes[reg_0 + reg_4] & 0xFF
reg_1 = word_seed_array[reg_0]
reg_2 = reg_1 | reg_6
index_reg_0 = reg_5
if (reg_2 & 0x80000000):
    break
# Address 0x3012 in Sample e8e1bc048ef123a9757a9b27d1bf53c092352a26edbf9fbd1c10109415b5cadac
reg_1 = arith_shift_rt(reg_1, 0x1F)
reg_2 = reg_2 >> 0x18
reg_2 = reg_2 & ~0x000000FF
reg_1 -= reg_2
reg_1 = 0x000000FF - reg_1
reg_1 = byte_seed_array[reg_1 & 0xFF]
encrypted_bytes[index_reg_0] = reg_1 & 0xFF
reg_5 += 1
if (reg_5 >= crazy_num):
    break
print "*********** FINISHED DECRYPT *************** 
Listing 3: Python code to implement the decryption algorithm (continued from previous page)."
0x9480 are shown in Figures 3 and 4. The bytes were decrypted using the IDAPython decryption script described in the Appendix.

Within the decrypted strings of the ELF, we see the names of the native functions defined in the Java code at the following locations in the ELF file:

- `quaqrd` (0xA107)
- `vxeg` (0x936E)
- `ixkjwu` (0x9330)

Figure 4: Decrypted bytes in ELF beginning at 0x9480.

Now that these strings are decrypted, we can see which subroutines in the ELF are called when the native function is called from the APK. Table 2 shows the native functions defined for this sample in the anti-analysis ELF.

The Java-declared native method that has the same signature as `vxeg` has in this sample `([Ljava/lang/Object;)I`, is responsible for doing all of the run-time environment checks described in the next section. In each sample, this function is named differently due to the automatic obfuscator run on the Java code, but it always has this signature. For clarity, the rest of

<table>
<thead>
<tr>
<th>Native function name</th>
<th>Native subroutine address</th>
<th>Signature</th>
<th>Human-readable signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>vxeg</td>
<td>0x30D4</td>
<td><code>([Ljava/lang/Object;)I</code></td>
<td>public native int vxeg(Object[] p0);</td>
</tr>
<tr>
<td>guaqrq</td>
<td>0x4814</td>
<td><code>(I)Ljava/lang/String;</code></td>
<td>public static native String guaqrq(int p0);</td>
</tr>
<tr>
<td>ixkjwu</td>
<td>---</td>
<td><code>([Ljava/lang/Object;)Ljava/lang/Object;</code></td>
<td>public native Object ixkjwu(Object[] p0);</td>
</tr>
</tbody>
</table>

Table 2: Native functions in the anti-analysis library.
this paper will refer to the native subroutine that performs all of
the run-time checks as vxeg().

The Java-declared native method that has the same signature as
quarqrd has in this sample ((I)Ljava/lang/String;) returns a string from an array. The argument to the method is the
index into the array and the address of the array is hard coded
into the native subroutine. The strings in this array are decrypted
by the decryption function described above.

Via static reverse engineering, I did not determine the native
subroutine corresponding to the ixkjwu method. In the Java
code, the ixkjwu method is only called in one place and is only
called based on the value of a variable. It is possible that this
method is never called based on the value of that variable and
thus the ixkjwu native subroutine does not exist.

vxeg and quarqrd are registered with the RegisterNatives
JNI method at 0x2B60 in this sample. The array at 0x9048 is
used for this call to RegisterNatives. It includes the native
method name, signature, and pointer to the native subroutine as
shown below. The code at 0x2B42, prior to the call to
RegisterNatives, shows that this subroutine can support the
following array entries for three native methods instead of the
two that exist in this instance.

The rest of this paper will focus on the functionality found in
vxeg() because it contains the anti-analysis run-time
environment checks.

### Run-time environment checks

The Java classes associated with WeddingCake in the APK
define three native functions in the Java code. In this sample
vxeg() performs all of the run-time environment checks prior to

<table>
<thead>
<tr>
<th>System property checked</th>
<th>Value(s) that trigger exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>init.svc.gce_fs_monitor</td>
<td>running</td>
</tr>
<tr>
<td>init.svc.dumpeventlog</td>
<td>running</td>
</tr>
<tr>
<td>init.svc.dumpiplog</td>
<td>running</td>
</tr>
<tr>
<td>init.svc.dumplogcat</td>
<td>running</td>
</tr>
<tr>
<td>init.svc.dumplogcat-efs</td>
<td>running</td>
</tr>
<tr>
<td>init.svc.filemon</td>
<td>running</td>
</tr>
<tr>
<td>ro.hardware.virtual_device</td>
<td>gce_x86</td>
</tr>
<tr>
<td>ro.kernel.androidboot.hardware</td>
<td>gce_x86</td>
</tr>
<tr>
<td>ro.hardware.virtual_device</td>
<td>gce_x86</td>
</tr>
<tr>
<td>ro.boot.hardware</td>
<td>gce_x86</td>
</tr>
<tr>
<td>ro.boot.selinux</td>
<td>disable</td>
</tr>
<tr>
<td>ro.factorytest</td>
<td>true, i, y</td>
</tr>
<tr>
<td>ro.kernel.android.checkjni</td>
<td>true, i, y</td>
</tr>
<tr>
<td>ro.hardware.virtual_device</td>
<td>vbox86</td>
</tr>
<tr>
<td>ro.kernel.androidboot.hardware</td>
<td>vbox86</td>
</tr>
<tr>
<td>ro.hardware</td>
<td>vbox86</td>
</tr>
<tr>
<td>ro.boot.hardware</td>
<td>vbox86</td>
</tr>
<tr>
<td>ro.build.product</td>
<td>google_sdk</td>
</tr>
<tr>
<td>ro.build.product</td>
<td>Droid4x</td>
</tr>
<tr>
<td>ro.build.product</td>
<td>sdk x86</td>
</tr>
<tr>
<td>ro.build.product</td>
<td>sdk google</td>
</tr>
<tr>
<td>ro.build.product</td>
<td>vbox86p</td>
</tr>
<tr>
<td>ro.product.manufacturer</td>
<td>Genymotion</td>
</tr>
<tr>
<td>ro.product.brand</td>
<td>generic</td>
</tr>
<tr>
<td>ro.product.brand</td>
<td>generic_x86</td>
</tr>
<tr>
<td>ro.product.device</td>
<td>generic</td>
</tr>
<tr>
<td>ro.product.device</td>
<td>generic_x86</td>
</tr>
<tr>
<td>ro.device</td>
<td>Droid4x</td>
</tr>
<tr>
<td>ro.product.device</td>
<td>vbox86p</td>
</tr>
<tr>
<td>ro.kernel.androidboot.hardware</td>
<td>goldfish</td>
</tr>
<tr>
<td>ro.hardware</td>
<td>goldfish</td>
</tr>
<tr>
<td>ro.boot.hardware</td>
<td>goldfish</td>
</tr>
<tr>
<td>ro.hardware.audio.primary</td>
<td>goldfish</td>
</tr>
<tr>
<td>ro.kernel.androidboot.hardware</td>
<td>ranchu</td>
</tr>
<tr>
<td>ro.hardware</td>
<td>ranchu</td>
</tr>
<tr>
<td>ro.boot.hardware</td>
<td>ranchu</td>
</tr>
</tbody>
</table>

Table 3: System properties checked and the values that trigger exit.
performing the hidden behaviour. This function performs more than 45 different run-time checks. They can be grouped as follows:

- Checking system properties
- Verifying CPU architecture by reading the /system/lib/libc.so ELF header
- Looking for Monkey [6] by iterating through all PIDs in /proc/
- Ensuring the Xposed Framework [7] is not mapped to the application process memory

If the library detects any of the conditions outlined in this section, the Linux \texttt{exit(0)} function is called, which terminates the Android application [8]. The application stops running if any of the 45+ environment checks fail.

### System properties checks

The \texttt{vxeg()} subroutine begins by checking the values of the listed system properties. The \texttt{system_property_get()} function is used to get the value of each system property checked. The code checks if the value matches the listed value for each property. If any one of the system properties matches the listed value, the Android application exits. Table 3 lists each of the system properties that is checked and the value which will trigger an exit.

The anti-analysis library also checks if any of five system properties exist on the device using the \texttt{system_property_find()} function. If any of these five system properties exist, the Android application exits. The properties that the library searches for are listed in Table 4. The presence of any of these properties usually indicates that the application is running on an emulator.

<table>
<thead>
<tr>
<th>If any of these system properties exist, the application exits</th>
</tr>
</thead>
<tbody>
<tr>
<td>init.svc.vbox86-setup</td>
</tr>
<tr>
<td>qemu.sf.fake_camera</td>
</tr>
<tr>
<td>init.svc.goldfish-logcat</td>
</tr>
<tr>
<td>init.svc.goldfish-setup</td>
</tr>
<tr>
<td>init.svc.qemud</td>
</tr>
</tbody>
</table>

**Table 4:** System properties checked for using \texttt{system_property_find}.

### Verifying CPU architecture

If the library has passed all of the system property checks, it (still in \texttt{vxeg()}) then verifies the CPU architecture of the phone on which the application is running. In order to verify the CPU architecture, the code reads 0x14 bytes from the beginning of the /system/lib/libc.so file on the device. If the read is successful, the code looks at the bytes corresponding to the \texttt{e_ident[EI_CLASS]} and \texttt{e_machine} fields of the ELF header. \texttt{e_ident[EI_CLASS]} is set to 1 to signal a 32-bit architecture and set to 2 to signal a 64-bit architecture. \texttt{e_machine} is a 2-byte value identifying the instruction set architecture. The code will only continue if one of the following statements is true. Otherwise, the application exits:

- \texttt{e_ident[EI_CLASS]} == 0x01 (32-bit) AND \texttt{e_machine} == 0x0028 (ARM)
- \texttt{e_ident[EI_CLASS]} == 0x02 (64-bit) AND \texttt{e_machine} == 0x00B7 (AArch64)
- Unable to read 0x14 bytes from /system/lib/libc.so

The anti-analysis library is verifying that it is only running on a 32-bit ARM or 64-bit AArch64 CPU. Even when the library is running its x86 variant, it still checks whether the CPU is ARM and will exit if the detected CPU is not ARM or AArch64.

### Identifying if Monkey is running

After the CPU architecture check, the library attempts to iterate through every PID directory under /proc/ to determine if \texttt{com.android.commands.monkey} is running [6]. The code does this by opening the /proc/ directory and iterating through each entry in the directory, completing the following steps. If any step fails, execution moves to the next entry in the directory.

1. Verifies \texttt{d_type} from the dirent struct == DT_DIR
2. Verifies that \texttt{d_name} from the dirent struct is an integer
3. Constructs path strings: \texttt{/proc/[pid]/comm} and \texttt{/proc/[pid]/cmdline} where \texttt{[pid]} is the directory entry name that has been verified to be an integer
4. Attempts to read 0x7F bytes from both \texttt{comm} and \texttt{cmdline} constructed path strings
5. Stores the data from whichever attempt (\texttt{comm} or \texttt{cmdline}) reads more data

**Figure 5:** Check for Monkey.
This method of iterating through each directory in `/proc/` doesn’t work in Android N and above [9]. If the library is not able to iterate through the directories in `/proc/` it will continue executing.

**Current process not hooked with Xposed Framework**

The Xposed Framework allows hooking and modifying of the system code running on an Android device. This library ensures that the Xposed Framework is not currently mapped to the application process. If Xposed is running the process, it could allow for some of the anti-analysis techniques to be bypassed. If the library did not check for Xposed and allowed the application to continue running when Xposed was hooked to the process, an analyst could instrument the application to bypass the anti-analysis hurdles and uncover the functionality that the application author is trying to hide.

In order to determine if Xposed is running, the library checks if ‘LIBXPOSED_ART.SO’ or ‘XPOSEDBRIDGE.JAR’ exist in `/proc/self/maps`. If either of them exist, then the application exits. `/proc/self/maps` lists all of the memory pages mapped into the process memory. Therefore, you can see any libraries loaded by the process by reading its contents.

To further verify that the Xposed Framework is not running, the code will check if either of the following two classes can be found using the JNI `FindClass()` function [10]. If either class can be found, the application exits:

- `XC_MethodHook`: `/de/robv/android/xposed/XC_MethodHook`
- `XposedBridge`: `/de/robv/android/xposed/XposedBridge`

If the Xposed library is not found, the execution continues to the behaviour that the anti-analysis techniques were trying to protect. This behaviour continues in `vxeg()`. In the case of this sample, it was another unpacker that previously had not been protected by the anti-reversing and analysis techniques described in this paper.

**CONCLUSION**

This paper detailed the operation of WeddingCake, an Android native library using extensive anti-analysis techniques. Unlike previous packers’ anti-emulation techniques, this library is written in C/C++ and runs as a native shared library in the application. Once an analyst understands the anti-reversing and anti-analysis techniques utilized by an application, they can more effectively understand its logic and analyse and detect potentially malicious behaviours.

**REFERENCES**


https://docs.oracle.com/javase/6/docs/technotes/guides/jni/spec/functions.html#FindClass.


https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/functions.html#FindClass.

**APPENDIX: IDAPYTHON DECRYPTION SCRIPT**

In order to decrypt the encrypted portions of the ELF library that the decryption function (for this sample, `sub_2F30`) decrypts during execution, I created an IDAPython script to decrypt the ELF. This script is available at http://www.github.com/maddiestone/IDAPythonEmbeddedToolkit/Android/WeddingCake_decrypt.py. By decrypting the ELF with the IDAPython script, it’s possible to statically reverse engineer the behaviour that is hidden under the anti-analysis techniques. This section describes how the script works.

The IDAPython decryption script runs the following steps:

1. Identifies the `JNI_OnLoad` function
2. Identifies the decryption function
3. Generates the two seed arrays
4. Identifies memory addresses of arrays to be decrypted and their lengths from the ELF loaded into the IDA Pro database
5. Encrypts each array and writes the decrypted bytes back to the IDA database, defining the decrypted bytes as strings.

The script was written to dynamically identify each of the encrypted arrays and their lengths from an IDA Pro database. This allows it to be run on many different samples without an analyst having to define the encrypted byte arrays. Therefore, the IDAPython script is dependent on the library’s architecture. This script will run on the 32-bit ‘generic’ ARM versions of the IDA Pro database.
library. For the other variants of the library mentioned in the ‘Variants’ section (ARMv7, ARM64, and x86), the same decryption algorithm in the script can be used, but the code to find the encrypted arrays and lengths will not run.

Once the script has finished running, the analyst can reverse engineer the native code as it lives when executing with the decrypted string.