Twitter Thread by Marcus Mengs

Marcus Mengs
@mame82

Okay, doing my first baby steps with r2frida (which combines the power of @radareorg and @fridadotre).

Gonna share my progress in this thread (live, so keep calm).

The goal: Runtime inspection of data sent out by TikTok !!before!! it gets encrypted

1/many

First of all, we do not start from zero. I got some prior knowledge from past reversing attempts and want to share some important facts.

TikTok's (log data) encryption is accomplished by a native library. The Android Java code just serves as proxy function to the native function

The decompiled code for the respective native JNI function of an older TikTok version looks something like this, but in this example I use the most current TT version (no statical analysis done, yet)

```
jbyteArray jni_ttEncrypt(JNIEnv *env,jobject thiz,jbyteArray inBytes,jint inByteLen)

{
  jbyte *elems;
  jbyte *newOutBuf;
  jbyteArray array;
  size_t outBufLen;
  int local_28;

  array = (jbyteArray)0x0;
  local_28 = __stack_chk_guard;
  if ((inBytes != (jbyteArray)0x0) && (0 < inByteLen)) {
    array = (jbyteArray)0x0;
    elems = (*(*env)->GetByteArrayElements)(env,inBytes,(jboolean *)0x0);
    if (elems != (jbyte *)0x0) {
      outBufLen = inByteLen + 0x76;
      newOutBuf = (jbyte *)malloc(outBufLen);
      if (newOutBuf == (jbyte *)0x0) {
        array = (jbyteArray)0x0;
        (*(*env)->ReleaseByteArrayElements)(env,inBytes,elems,0);
      }
      else {
        ss_encrypt(elems,inByteLen,newOutBuf,&outBufLen);
        if (outBufLen == 0) {
          array = (jbyteArray)0x0;
        }
        else {
          array = (*(*env)->NewByteArray)(env,outBufLen);
          (*(*env)->SetByteArrayRegion)(env,array,0,outBufLen,newOutBuf);
        }
        (*(*env)->ReleaseByteArrayElements)(env,inBytes,elems,0);
        free(newOutBuf);
      }
    }
  }
  if (__stack_chk_guard != local_28) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return array;
}
```

In case you never reversed native libraries which were build to interface with Android Java layer via JNI, I highly suggest the entry level introduction on the topic by @maddiestone

https://t.co/T63vo3N4fw

Before we start, I want to pinpoint some important aspects (which are also covered by Maddie's videos).

1) Unlike raw C-functions, JNI functions like the one showcased above, receive pointers to complex Java objects .

F.e. a function receiving a String on the Java layer...

... would receive a pointer to a 'jstring' on the native layer (not a zero-terminated C-String).

In order to retrieve a C-String, to go on working with it in the native code, some translation functionality is required. This functionality is provided by the ...

JNI (Java Native Interface). The JNI environment is passed in to JNI functions as first parameter.

If you look at the example screenshot again, you see exactly this. Functions provided by the 'env' pointer are used to parse the Java function arguments (f.e. jByteArrays) ...

```c
jbyteArray jni_ttEncrypt(JNIEnv *env,jobject thiz,jbyteArray inBytes,jint inByteLen)

{
  jbyte *elems;
  jbyte *newOutBuf;
  jbyteArray array;
  size_t outBufLen;
  int local_28;

  array = (jbyteArray)0x0;
  local_28 = __stack_chk_guard;
  if ((inBytes != (jbyteArray)0x0) && (0 < inByteLen)) {
    array = (jbyteArray)0x0;
    elems = (*(*env)->GetByteArrayElements)(env,inBytes,(jboolean *)0x0);
    if (elems != (jbyte *)0x0) {
      outBufLen = inByteLen + 0x76;
      newOutBuf = (jbyte *)malloc(outBufLen);
      if (newOutBuf == (jbyte *)0x0) {
        array = (jbyteArray)0x0;
        (*(*env)->ReleaseByteArrayElements)(env,inBytes,elems,0);
      }
      else {
        ss_encrypt(elems,inByteLen,newOutBuf,&outBufLen);
        if (outBufLen == 0) {
          array = (jbyteArray)0x0;
        }
        else {
          array = (*(*env)->NewByteArray)(env,outBufLen);
          (*(*env)->SetByteArrayRegion)(env,array,0,outBufLen,newOutBuf);
        }
        (*(*env)->ReleaseByteArrayElements)(env,inBytes,elems,0);
        free(newOutBuf);
      }
    }
  }
  if (__stack_chk_guard != local_28) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return array;
}
```

Once the raw data is converted to a more C-ish form, it gets passed to a inner function 'ss_encrypt' in my example. The inner function, in this case, is a pure C function and thus receives only C-style parameters (also no 'env' parameter, so it would not be able to access JNI)

A 2nd important aspect on JNI libraries, covered by @maddiestone

2) There are two ways to expose JNI methods from a native library:

a) export them with proper naming convention, so that JNI could recognize same on library load
b) use the JNI functionality 'registerNatives'...

... to register the JNI functions once the library gets loaded.

The second method of registering methods is wel suited for obfuscated code, as the methods neither have to follow naming convention, nor do they have to be exported.

As you might expect, TikTok uses the 'registerNative' approach. The screenshot below shows log output from a custom tool, which monitors JNI methods registered by instrumented Android apps (TikTok's encryption method in the example)

```
[log 3845 "com.zhiliaoapp.musically" info] ====>>> Init finished after 4734ms process=3845
[log 3845 "com.zhiliaoapp.musically" info] Called android_dlopen_ext /data/app/com.zhiliaoapp.musically-Q46lIu9w78FscnnBrj-F4Q==/lib/arm/libEncryptor.so
[log 3845 "com.zhiliaoapp.musically" info] Found 'ttEncrypt' in class com.bytedance.frameworks.encryptor.EncryptorUtil, hooking ...
[log 3845 "com.zhiliaoapp.musically" info] ttEncrypt: 0x7d70d1d5 libEncryptor.so!0xb1d5
[log 3845 "com.zhiliaoapp.musically" info] registerNatives(class=com.bytedance.frameworks.encryptor.EncryptorUtil, pMethods=0x7d716004, nMethods=1)
    0x7d70d1d5: ttEncrypt ([BI)[B
```

If you would decompile the Java part of the TikTok apk, the encryption functionality (on an older version) would look something like this:

```java
package com.bytedance.frameworks.encryptor;

import android.os.SystemClock;
import com.bytedance.p521l.C9011a;
import com.p546ss.android.ugc.aweme.lancet.p586a.C9496c;

public class EncryptorUtil {
    private static native byte[] ttEncrypt(byte[] bArr, int i);

    static {
        String str = "Encryptor";
        try {
            long uptimeMillis = SystemClock.uptimeMillis();
            C9011a.m18576a(str);
            C9496c.m19700a(uptimeMillis, str);
        } catch (UnsatisfiedLinkError unused) {
        }
    }

    /* renamed from: a */
    public static byte[] m18421a(byte[] bArr, int i) {
        if (bArr != null && i > 0) {
            try {
                if (bArr.length == i) {
                    return ttEncrypt(bArr, i);
                }
            } catch (Throwable unused) {
            }
        }
        return null;
    }
}
```

The Java method 'm18421a' receives a Java 'byte[]' and a Java 'int' as parameters and returns a 'byte[]', again.

Internally, this data is forwarded to the native JNI method 'ttEncrypt'.

The important aspect about this, is that the native 'ttEncrypt' JNI method has to accept those exact parameter types and thus has to "register" with a proper method signature.

We already saw this signature in a previous screenshot

```
[log 3845 "com.zhiliaoapp.musically" info] ====>>> Init finished after 4734ms process=3845
[log 3845 "com.zhiliaoapp.musically" info] Called android_dlopen_ext /data/app/com.zhiliaoapp.musically-Q46lIu9w78FscnnBrj-F4Q==/lib/arm/libEncryptor.so
[log 3845 "com.zhiliaoapp.musically" info] Found 'ttEncrypt' in class com.bytedance.frameworks.encryptor.EncryptorUtil, hooking ...
[log 3845 "com.zhiliaoapp.musically" info] ttEncrypt: 0x7d70d1d5 libEncryptor.so!0xb1d5
[log 3845 "com.zhiliaoapp.musically" info] registerNatives(class=com.bytedance.frameworks.encryptor.EncryptorUtil, pMethods=0x7d716004, nMethods=1)
    0x7d70d1d5: ttEncrypt ([BI)[B
```

The last line from the screenshot above, shows 3 things the native code has to provide for each method, when calling register natives:

1) the call address of the native function implementation (0x7d70d1d5 in example)
2) The function name (ttEncrypt)
...

3) The method signature, which is '([BI)[B' in this case and translates to:

'(' start of parameters
'[B' byte[]
'I' int
')' end of parameters
'[B' byte[] (return value)

So we keep this in mind: Even if the native library does not export the encryption method, it has to store the 1) funtion address, 2) name and 3) signature in a data structure, in order to provide it to 'registerNatives' once the library gets loaded by JNI

We now know almost everything we need, in order to head over to r2frida, except some important facts on my test setup:

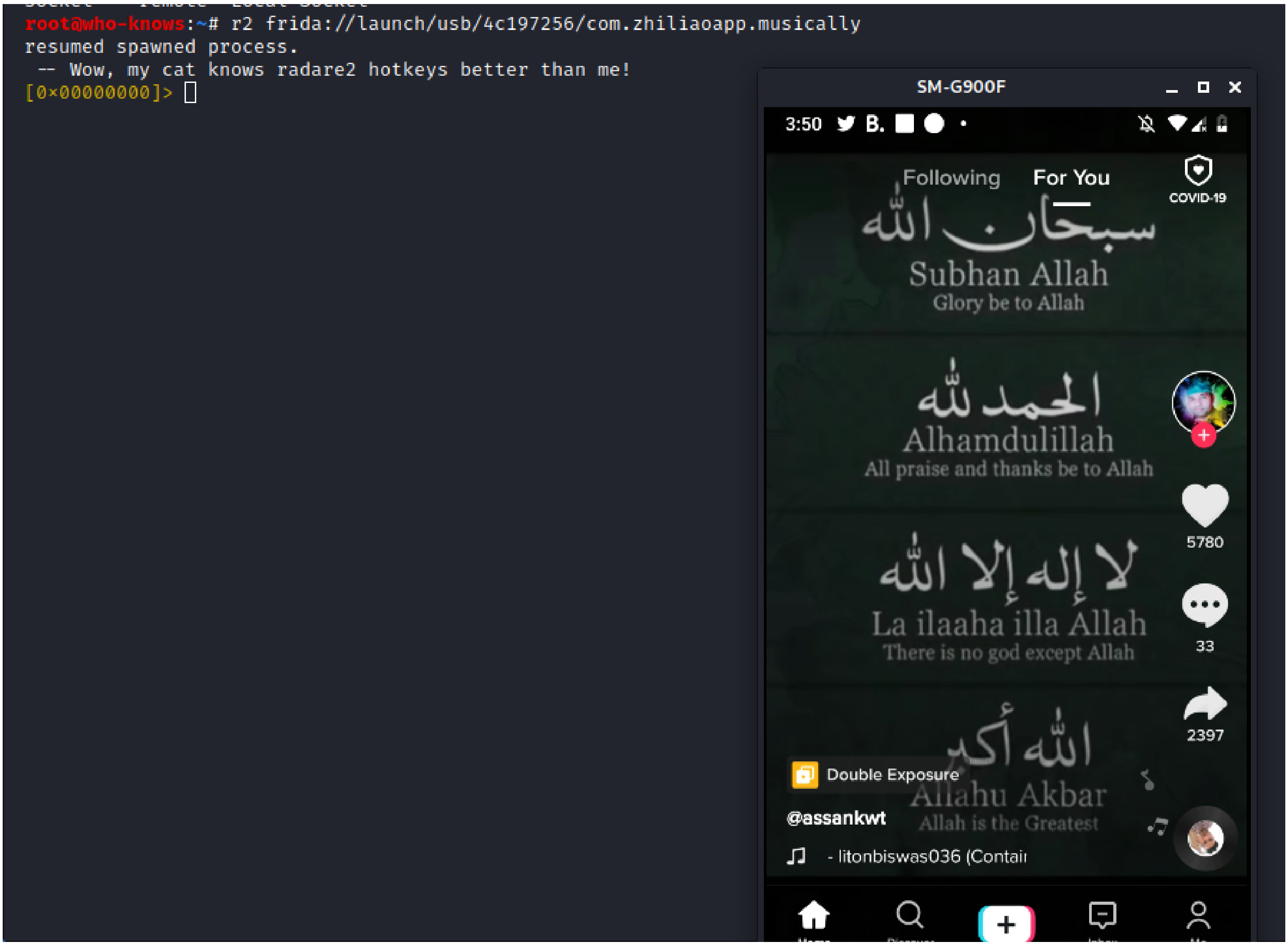
- the app is inspected on a physical device, running Android 9
- the device uses a !!32bit!! ARM application core

As I am new to r2frida, chances are high that things could be achieved in an easier ways.

Now to get started, I already have the latest @fridadotre server running on my USB connected android device and 'frida-ls-device' shows it being ready-for-action

```
root@who-knows:~# frida-ls-devices
Id         Type    Name
--------   -----   ------------
local      local   Local System
4c197256   usb     SM-G900F
socket     remote  Local Socket
```

So let's spawn a fresh TikTok instance with r2frida, using the following command



The command 'launch'es the process, on the attached frida 'usb' device with the device id '4c197256' and the process's package name, of course is 'com.zhiliaoapp.musically'

Instead of 'launch', two other options could be used:

- 'attach' (would attach to an already running process, given by name or PID)
- 'spawn' (like 'launch', but the process would not be resumed automatically after attaching)

So for the warm up, let us use the Frida functionality, which alllows enumerating loaded Java classes. This nicely combines with the r2 syntax (concatenation of single-letter commands, '~' for grep)

Important: commands targeting the r2frida plugin have to be prefixed with '\'

The r2frida command to list classes is '\ic' (note the backslash prefix). The unfiltered result would be a bit overwhelming ...

```
[0x00000000]> \ic
Do you want to print 25073 lines? (y/N) n
```

... so we grep for classes including the term "crypt"

```
[0x00000000]> \ic~cryptor
com.bytedance.frameworks.encryptor.EncryptorUtil
```

The '\ic ' command lists the !Java! methods of the respective class
The signature of the static method 'EncryptorUtil.a' should look familiar to us (if you read the first tweets). It represents the Java layer of the encryption method and is called 'a' in this version

```
[0x00000000]> \ic com.bytedance.frameworks.encryptor.EncryptorUtil
public static byte[] com.bytedance.frameworks.encryptor.EncryptorUtil.a(byte[],int)
public boolean java.lang.Object.equals(java.lang.Object)
public final java.lang.Class java.lang.Object.getClass()
public int java.lang.Object.hashCode()
public final native void java.lang.Object.notify()
public final native void java.lang.Object.notifyAll()
public java.lang.String java.lang.Object.toString()
public final native void java.lang.Object.wait() throws java.lang.InterruptedException
public final void java.lang.Object.wait(long) throws java.lang.InterruptedException
public final native void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
public com.bytedance.frameworks.encryptor.EncryptorUtil()
```

Note: The information above would be enough, to Intercept the method from the Java layer (f.e. with @fridadotre or Xposed), in order to inspect the call arguments (the byte[] parameter represents the plain data before encryption)

... but we are here for the native layer and to inspect data at runtime, right?

So lets search the whole address space for our native method name 'ttEncrypt'

Note: If you'd use r2's ascii search nothing would happen, you have to use the '\' prefix to search with r2frida

```
[0x00000000]> / ttEncrypt
[0x00000000]> \/ ttEncrypt
Searching 9 bytes: 74 74 45 6e 63 72 79 70 74
Searching 9 bytes in [0x12c00000-0x13980000]
Searching 9 bytes in [0x13f00000-0x13f40000]
Searching 9 bytes in [0x13fc0000-0x14180000]
Searching 9 bytes in [0x141c0000-0x14200000]
Searching 9 bytes in [0x14300000-0x14340000]
Searching 9 bytes in [0x143c0000-0x14700000]
Searching 9 bytes in [0x14740000-0x14780000]
Searching 9 bytes in [0x147c0000-0x14940000]
Searching 9 bytes in [0x149c0000-0x14b00000]
Searching 9 bytes in [0x14b40000-0x14b80000]
Searching 9 bytes in [0x14c80000-0x14cc0000]
Searching 9 bytes in [0x14d80000-0x14dc0000]
Searching 9 bytes in [0x14e40000-0x14e80000]
```

The search ends with two hits:

```
Searching 9 bytes in [0xbe024000-0xbe823000]
Searching 9 bytes in [0xffff0000-0xffff1000]
hits: 2
0x8448b74c hit0_0
0x947ede70 hit0_1 ttEncrypt

[0x00000000]>
```

The screenshot below shows, that the attempt to print a hexdump from the address of the first hit fails with r2, while r2frida (backslash prefix) works.

Reason: The memory region was not populated when r2 was started (encryption library was loaded after process launch)

```
[0x00000000]> px @ 0x8448b74c
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x8448b74c  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x8448b75c  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x8448b76c  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x8448b77c  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x8448b78c  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x8448b79c  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x8448b7ac  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x8448b7bc  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x8448b7cc  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x8448b7dc  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x8448b7ec  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x8448b7fc  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x8448b80c  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x8448b81c  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x8448b82c  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x8448b83c  ffff ffff ffff ffff ffff ffff ffff ffff  ................
[0x00000000]> \px @ 0x8448b74c
            0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
8448b74c  74 74 45 6e 63 72 79 70 74 00 28 5b 42 49 29 5b  ttEncrypt.([BI)[
8448b75c  42 00 00 00 a5 63 63 c6 84 7c 7c f8 99 77 77 ee  B....cc..||..ww.
8448b76c  8d 7b 7b f6 0d f2 f2 ff bd 6b 6b d6 b1 6f 6f de  .{{......kk..oo.
8448b77c  54 c5 c5 91 50 30 30 60 03 01 01 02 a9 67 67 ce  T...P00`.....gg.
8448b78c  7d 2b 2b 56 19 fe fe e7 62 d7 d7 b5 e6 ab ab 4d  }++V....b......M
8448b79c  9a 76 76 ec 45 ca ca 8f 9d 82 82 1f 40 c9 c9 89  .vv.E.......@...
8448b7ac  87 7d 7d fa 15 fa fa ef eb 59 59 b2 c9 47 47 8e  .}}......YY..GG.
8448b7bc  0b f0 f0 fb ec ad ad 41 67 d4 d4 b3 fd a2 a2 5f  .......Ag......_
8448b7cc  ea af af 45 bf 9c 9c 23 f7 a4 a4 53 96 72 72 e4  ...E...#...S.rr.
8448b7dc  5b c0 c0 9b c2 b7 b7 75 1c fd fd e1 ae 93 93 3d  [......u.......=
8448b7ec  6a 26 26 4c 5a 36 36 6c 41 3f 3f 7e 02 f7 f7 f5  j&&LZ66lA??~....
8448b7fc  4f cc cc 83 5c 34 34 68 f4 a5 a5 51 34 e5 e5 d1  O...\44h...Q4...
8448b80c  08 f1 f1 f9 93 71 71 e2 73 d8 d8 ab 53 31 31 62  .....qq.s...S11b
8448b81c  3f 15 15 2a 0c 04 04 08 52 c7 c7 95 65 23 23 46  ?..*....R...e##F
8448b82c  5e c3 c3 9d 28 18 18 30 a1 96 96 37 0f 05 05 0a  ^...(..0...7....
8448b83c  b5 9a 9a 2f 09 07 07 0e 36 12 12 24 9b 80 80 1b  .../....6..$....
```

I solved this issue like this:

1) Quit r2

2) Open r2 with r2frida, again, but this time **attach** to the already running process

et voila ... the memory offset is mapped and dumpable with 'px' (without backslash prefix)

```
[0x00000000]> q
root@who-knows:~# r2 frida://attach/usb/4c197256/com.zhiliaoapp.musically
 -- Too old to crash
[0x00000000]> px @ 0x8448b74c
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x8448b74c  7474 456e 6372 7970 7400 285b 4249 295b  ttEncrypt.([BI)[
0x8448b75c  4200 0000 a563 63c6 847c 7cf8 9977 77ee  B....cc..||..ww.
0x8448b76c  8d7b 7bf6 0df2 f2ff bd6b 6bd6 b16f 6fde  .{{......kk..oo.
0x8448b77c  54c5 c591 5030 3060 0301 0102 a967 67ce  T...P00`.....gg.
0x8448b78c  7d2b 2b56 19fe fee7 62d7 d7b5 e6ab ab4d  }++V....b......M
0x8448b79c  9a76 76ec 45ca ca8f 9d82 821f 40c9 c989  .vv.E.......@...
0x8448b7ac  877d 7dfa 15fa faef eb59 59b2 c947 478e  .}}......YY..GG.
0x8448b7bc  0bf0 f0fb ecad ad41 67d4 d4b3 fda2 a25f  .......Ag......_
0x8448b7cc  eaaf af45 bf9c 9c23 f7a4 a453 9672 72e4  ...E...#...S.rr.
0x8448b7dc  5bc0 c09b c2b7 b775 1cfd fde1 ae93 933d  [......u.......=
0x8448b7ec  6a26 264c 5a36 366c 413f 3f7e 02f7 f7f5  j&&LZ66lA??~....
0x8448b7fc  4fcc cc83 5c34 3468 f4a5 a551 34e5 e5d1  O...\44h...Q4...
0x8448b80c  08f1 f1f9 9371 71e2 73d8 d8ab 5331 3162  .....qq.s...S11b
0x8448b81c  3f15 152a 0c04 0408 52c7 c795 6523 2346  ?..*....R...e##F
0x8448b82c  5ec3 c39d 2818 1830 a196 9637 0f05 050a  ^...(..0...7....
0x8448b83c  b59a 9a2f 0907 070e 3612 1224 9b80 801b  .../....6..$....
[0x00000000]>
```

Note: The last step is not necessary for a data hexdump, as you could still use '\px', but it turned out to be useful when it comes to printing the disassembly of "late loaded" code regions. This is because I sometimes struggled with '\pd', but 'pd' worked (+ various r2 views)

Having a closer look at the first hit of out string search for 'ttEncrypt', we notice that it is directly followed by a C-string with our method signature.

So chances are high, that this data is part of the structure which gets handed in to 'registerNatives'

```
0x8448b74c  7474 456e 6372 7970 7400 285b 4249 295b  ttEncrypt.([BI)[
0x8448b75c  4200 0000 a563 63c6 847c 7cf8 9977 77ee  B....cc..||..ww.
```

Reminder: in order to register the 'ttEncrypt' method to JNI, the 'registerNatives' method requires a structure containing
- method name (C-string)
- method signature (C-string)
- method pointer (native pointer)

So the next step would be to search the process memory space for cross references to the address of this method name string (0x8448b74c). As I haven't applied any auto analysis, I use a simple hex search for this (in my case the byte order of the address has to be reversed ...

```
[0x00000000]> px @ 0x8448b74c
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x8448b74c  7474 456e 6372 7970 7400 285b 4249 295b  ttEncrypt.([BI)[
0x8448b75c  4200 0000 a563 63c6 847c 7cf8 9977 77ee  B....cc..||..ww.
0x8448b76c  8d7b 7bf6 0df2 f2ff bd6b 6bd6 b16f 6fde  .{{......kk..oo.
0x8448b77c  54c5 c591 5030 3060 0301 0102 a967 67ce  T...P00`.....gg.
0x8448b78c  7d2b 2b56 19fe fee7 62d7 d7b5 e6ab ab4d  }++V....b......M
0x8448b79c  9a76 76ec 45ca ca8f 9d82 821f 40c9 c989  .vv.E.......@...
0x8448b7ac  877d 7dfa 15fa faef eb59 59b2 c947 478e  .}}......YY..GG.
0x8448b7bc  0bf0 f0fb ecad ad41 67d4 d4b3 fda2 a25f  .......Ag......_
0x8448b7cc  eaaf af45 bf9c 9c23 f7a4 a453 9672 72e4  ...E...#...S.rr.
0x8448b7dc  5bc0 c09b c2b7 b775 1cfd fde1 ae93 933d  [......u.......=
0x8448b7ec  6a26 264c 5a36 366c 413f 3f7e 02f7 f7f5  j&&LZ66lA??~....
0x8448b7fc  4fcc cc83 5c34 3468 f4a5 a551 34e5 e5d1  O...\44h...Q4...
0x8448b80c  08f1 f1f9 9371 71e2 73d8 d8ab 5331 3162  .....qq.s...S11b
0x8448b81c  3f15 152a 0c04 0408 52c7 c795 6523 2346  ?..*....R...e##F
0x8448b82c  5ec3 c39d 2818 1830 a196 9637 0f05 050a  ^...(..0...7....
0x8448b83c  b59a 9a2f 0907 070e 3612 1224 9b80 801b  .../....6..$....
[0x00000000]> \/x 4cb74884
Searching 4 bytes: 4c b7 48 84
Searching 4 bytes in [0x12c00000-0x136c0000]
Searching 4 bytes in [0x13740000-0x13780000]
Searching 4 bytes in [0x13880000-0x138c0000]
Searching 4 bytes in [0x13980000-0x139c0000]
Searching 4 bytes in [0x13a40000-0x13a80000]
Searching 4 bytes in [0x13c00000-0x13c40000]
```

... to account for the architecture endianess).

The result is promising: Only one hit, for a search across the whole address space:

```
Searching 4 bytes in [0xb434b000-0xb434c000]
Searching 4 bytes in [0xbe024000-0xbe823000]
Searching 4 bytes in [0xffff0000-0xffff1000]
hits: 1
0x84494004 hit0_0 4cb74884
```

Printing the first 12 bytes from this XREF offset, reveals 3 pointers again (reversed endianess):

- 0x8448b74c (expected, method name pointer)
- 0x8448b756 (ptr to signature string, yay)
- 0x8448b1d5 (likely pointer to JNI method implementation)

```
hits: 1
0x84494004 hit0_0 4cb74884

[0x00000000]> px 12 @ 0x84494004
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x84494004  4cb7 4884 56b7 4884 d5b1 4884            L.H.V.H...H.
[0x00000000]>
```

So the layout of the 3 pointers from above speaks a clear language. Very likely this is the data struct passed in to 'registerNatives' and thud 0x8448b1d5 points to the native implementation of 'ttEncrypt'

Sorry, before going on I have to insert a small excurse on adressing/instruction sets on arm 32 (specific to my test setup). Anyways it is crucial:

Arm 32 supports two instruction sets "ARM mode" (32bit) and "Thumb mode" (16bit) which could be used interchangebly

In order to distinguish if a function call target (branch) should be interpreted as ARM or THUMB, the least significant bit (LSB) of the function address is taken into account

For ARM the LSB is 0 (even address)
For THUMB the LSB is 1 (odd address)

The actual instruction ALWAYS resides on an odd address.

This means the function address 0x8448b1d5 homes code in THUMB mode (16bit), while the first instruction resides at 0x8448b1d4

(sorry if it gets a bit complicated, will be clear in a second)

So if we print the disassembly at the (assumed) 'ttEncrypt' address, things look a bit weird

```
[0x00000000]> pd @ 0x8448b1d5
            0x8448b1d5                      unaligned
            0x8448b1d6                      unaligned
            0x8448b1d7                      unaligned
            0x8448b1d8      2de9000f        svceq 0xe92d
            0x8448b1dc      83b00546        strmi fp, [r5], -r3, lsl 1
            0x8448b1e0      2f489246        ldrmi r4, [r2], pc, lsr 16
            0x8448b1e4      1c467844        ldrbtmi r4, [r8], -0x61c
            0x8448b1e8      0026baf1        invalid
            0x8448b1ec      000fd0f8        invalid
            0x8448b1f0      00b0dbf8        invalid
            0x8448b1f4      00000290        andls r0, r2, r0
            0x8448b1f8      45d0012c        invalid
```

... this is because instructions are interpreted in ARM mode (32bit).

Lets fix this:

```
[0x00000000]> e asm.bits=16
[0x00000000]> pd @ 0x8448b1d5
            0x8448b1d5                       unaligned
            0x8448b1d6      03af           add r7, sp, 0xc
            0x8448b1d8      2de9000f       push.w {r8, sb, sl, fp}
            0x8448b1dc      83b0           sub sp, 0xc
            0x8448b1de      0546           mov r5, r0
            0x8448b1e0      2f48           ldr r0, [0x8448b2a0]         ; [0x8448b2a0:4]=0x8d46
            0x8448b1e2      9246           mov sl, r2
            0x8448b1e4      1c46           mov r4, r3
            0x8448b1e6      7844           add r0, pc
            0x8448b1e8      0026           movs r6, 0
            0x8448b1ea      baf1000f       cmp.w sl, 0
            0x8448b1ee      d0f800b0       ldr.w fp, [r0]
            0x8448b1f2      dbf80000       ldr.w r0, [fp]
            0x8448b1f6      0290           str r0, [sp, 8]
        ┌─< 0x8448b1f8      45d0           beq 0x8448b286
        │   0x8448b1fa      012c           cmp r4, 1                    ; 1
       ┌──< 0x8448b1fc      43db           blt 0x8448b286
       ││   0x8448b1fe      2868           ldr r0, [r5]
       ││   0x8448b200      5146           mov r1, sl
       ││   0x8448b202      0022           movs r2, 0
       ││   0x8448b204      0026           movs r6, 0
```

... looks much better, still the first instruction is off-by-one ■

No seriously, as explained, on arm32 we have to disassemble at [THUMB mode address - 1] = 0x8448b1d4

```
[0x00000000]> pd @ 0x8448b1d4
            0x8448b1d4      f0b5           push {r4, r5, r6, r7, lr}
            0x8448b1d6      03af           add r7, sp, 0xc
            0x8448b1d8      2de9000f       push.w {r8, sb, sl, fp}
            0x8448b1dc      83b0           sub sp, 0xc
            0x8448b1de      0546           mov r5, r0
            0x8448b1e0      2f48           ldr r0, [0x8448b2a0]         ; [0x8448b2a0:4]=0x8d46
            0x8448b1e2      9246           mov sl, r2
            0x8448b1e4      1c46           mov r4, r3
            0x8448b1e6      7844           add r0, pc
            0x8448b1e8      0026           movs r6, 0
            0x8448b1ea      baf1000f       cmp.w sl, 0
            0x8448b1ee      d0f800b0       ldr.w fp, [r0]
            0x8448b1f2      dbf80000       ldr.w r0, [fp]
            0x8448b1f6      0290           str r0, [sp, 8]
        ┌─< 0x8448b1f8      45d0           beq 0x8448b286
        │   0x8448b1fa      012c           cmp r4, 1                    ; 1
       ┌──< 0x8448b1fc      43db           blt 0x8448b286
       ││   0x8448b1fe      2868           ldr r0, [r5]
       ││   0x8448b200      5146           mov r1, sl
       ││   0x8448b202      0022           movs r2, 0
       ││   0x8448b204      0026           movs r6, 0
       ││   0x8448b206      d0f8e032       ldr.w r3, [r0, 0x2e0]
       ││   0x8448b20a      2846           mov r0, r5
       ││   0x8448b20c      9847           blx r3
```

Nice, this looks like a proper function stub (note how the callee stores reg values on the stack, before moving on).

Now to get a feeling on how often this function is called, lets use 'r2frida' power to trace it.

Important: The thumb address has to be used here!!!

```
[0x00000000]> \dt @ 0x8448b1d5
0       0       0x8448b1d5      dt      libEncryptor.so 0x8448b1d5
```

The resulting output of the '\dt' command, which places the trace hook also indicates that the function address maps to an offset in 'https://t.co/fxhDnQke2o' ... let us call this a "nice confirmation"

Some actions in the TikTok app ... trace logs for ttEncrypt-calls arrive

```
[0x00000000]> [dt][Tue Jan 12 2021 16:56:28 GMT+0100] 0x8448b1d5 - args: []
[dt][Tue Jan 12 2021 16:58:30 GMT+0100] 0x8448b1d5 - args: []
[dt][Tue Jan 12 2021 16:58:31 GMT+0100] 0x8448b1d5 - args: []
[dt][Tue Jan 12 2021 16:58:32 GMT+0100] 0x8448b1d5 - args: []
[dt][Tue Jan 12 2021 16:58:33 GMT+0100] 0x8448b1d5 - args: []
[dt][Tue Jan 12 2021 16:58:33 GMT+0100] 0x8448b1d5 - args: []
```

... cigarette break ... stay tuned (if the app crashes meanwhile, I'll start from scratch)

Let's remove the trace hook for now, with '\dt-*'

```
[0x00000000]>
[0x00000000]> \dt-*
```

Remember my screenshot of a decompiled 'ttEncrypt' function from an older TT version. We traced the corresponding functions.

Trying to runtime-parse the function parameters, which represent Java object instances would be insane (maybe impossible)

```
jbyteArray jni_ttEncrypt(JNIEnv *env,jobject thiz,jbyteArray inBytes,jint inByteLen)

{
  jbyte *elems;
  jbyte *newOutBuf;
  jbyteArray array;
  size_t outBufLen;
  int local_28;

  array = (jbyteArray)0x0;
  local_28 = __stack_chk_guard;
  if ((inBytes != (jbyteArray)0x0) && (0 < inByteLen)) {
    array = (jbyteArray)0x0;
    elems = (*(*env)->GetByteArrayElements)(env,inBytes,(jboolean *)0x0);
    if (elems != (jbyte *)0x0) {
      outBufLen = inByteLen + 0x76;
      newOutBuf = (jbyte *)malloc(outBufLen);
      if (newOutBuf == (jbyte *)0x0) {
        array = (jbyteArray)0x0;
        (*(*env)->ReleaseByteArrayElements)(env,inBytes,elems,0);
      }
      else {
        ss_encrypt(elems,inByteLen,newOutBuf,&outBufLen);
        if (outBufLen == 0) {
          array = (jbyteArray)0x0;
        }
        else {
          array = (*(*env)->NewByteArray)(env,outBufLen);
          (*(*env)->SetByteArrayRegion)(env,array,0,outBufLen,newOutBuf);
        }
        (*(*env)->ReleaseByteArrayElements)(env,inBytes,elems,0);
        free(newOutBuf);
      }
    }
  }
  if (__stack_chk_guard != local_28) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return array;
}
```

... luckily, at least the old implementation, internally called a method 'ss_encrypt' which received a c-style byte array pointer and an integer representing the length as first two parameters.
It would be way easier to runtime-inspect these

Lets take a closer look on the disassembly of our assumed 'ttEncrypt' function, by seeking to its offset with 's 0x8448b1d5' and switching to a more suitable r2 view with uppercase 'V' command (press 'p' till the view changes to disassembly)

```
[0x8448b1d4 [xAdvc]0 240 frida://attach/usb/4c197256/com.zhiliaoapp.musically]> pd $r
            0x8448b1d4      f0b5           push {r4, r5, r6, r7, lr}
            0x8448b1d6      03af           add r7, sp, 0xc
            0x8448b1d8      2de9000f       push.w {r8, sb, sl, fp}
            0x8448b1dc      83b0           sub sp, 0xc
            0x8448b1de      0546           mov r5, r0
            0x8448b1e0      2f48           ldr r0, [0x8448b2a0]          ; [0x8448b2a0:4]=0x8d46
            0x8448b1e2      9246           mov sl, r2
            0x8448b1e4      1c46           mov r4, r3
            0x8448b1e6      7844           add r0, pc
            0x8448b1e8      0026           movs r6, 0
            0x8448b1ea      baf1000f       cmp.w sl, 0
            0x8448b1ee      d0f800b0       ldr.w fp, [r0]
            0x8448b1f2      dbf80000       ldr.w r0, [fp]
            0x8448b1f6      0290           str r0, [sp, 8]
        ┌─< 0x8448b1f8      45d0           beq 0x8448b286
        │   0x8448b1fa      012c           cmp r4, 1                     ; 1
       ┌──< 0x8448b1fc      43db           blt 0x8448b286
       ││   0x8448b1fe      2868           ldr r0, [r5]
       ││   0x8448b200      5146           mov r1, sl
       ││   0x8448b202      0022           movs r2, 0
       ││   0x8448b204      0026           movs r6, 0
       ││   0x8448b206      d0f8e032       ldr.w r3, [r0, 0x2e0]
       ││   0x8448b20a      2846           mov r0, r5
       ││   0x8448b20c      9847           blx r3
       ││   0x8448b20e      8046           mov r8, r0
       ││   0x8448b210      b8f1000f       cmp.w r8, 0
      ┌───< 0x8448b214      37d0           beq 0x8448b286
      │││   0x8448b216      04f17600       add.w r0, r4, 0x76
      │││   0x8448b21a      0190           str r0, [sp, 4]
      │││   0x8448b21c      f5f7c4ef       blx 0x844811a8                ;[1]
      │││   0x8448b220      8146           mov sb, r0
      │││   0x8448b222      b9f1000f       cmp.w sb, 0
     ┌────< 0x8448b226      18d0           beq 0x8448b25a
     ││││   0x8448b228      01ab           add r3, sp, 4
     ││││   0x8448b22a      4046           mov r0, r8
     ││││   0x8448b22c      2146           mov r1, r4
     ││││   0x8448b22e      4a46           mov r2, sb
     ││││   0x8448b230      f8f738fc       bl 0x84483aa4                 ;[2]
     ││││   0x8448b234      0199           ldr r1, [sp, 4]
    ┌─────< 0x8448b236      d1b1           cbz r1, 0x8448b26e
    │││││   0x8448b238      2868           ldr r0, [r5]
    │││││   0x8448b23a      d0f8c022       ldr.w r2, [r0, 0x2c0]
    │││││   0x8448b23e      2846           mov r0, r5
    │││││   0x8448b240      9047           blx r2
    │││││   0x8448b242      0646           mov r6, r0
    │││││   0x8448b244      2868           ldr r0, [r5]
    │││││   0x8448b246      019b           ldr r3, [sp, 4]
```

The view from above allows scrolling through the functions code with cursor keys. Most important: calls to other code parts (branches) are printed bold and suffixed with [1], [2] ...

Hitting [alt+1] moves us straight to the marked branch offset:

```
[0x844811a8 [xAdvc]0 16384 frida://attach/usb/4c197256/com.zhiliaoapp.musically]> pd $r
          0x844811a8      00c68fe2       invalid
          0x844811ac      12ca           ldm r2!, {r1, r4}
       ,< 0x844811ae      8ce2           b 0x844816ca
          0x844811b0      b8fdbce5       ldc2 p5, c14, [r8, 0x2f0]!
          0x844811b4      00c68fe2       invalid
          0x844811b8      12ca           ldm r2!, {r1, r4}
       ,< 0x844811ba      8ce2           b 0x844816d6
          0x844811bc      b0fdbce5       ldc2 p5, c14, [r0, 0x2f0]!
          0x844811c0      00c68fe2       invalid
          0x844811c4      12ca           ldm r2!, {r1, r4}
       ,< 0x844811c6      8ce2           b 0x844816e2
          0x844811c8      a8fdbce5       stc2 p5, c14, [r8, 0x2f0]!
          0x844811cc      00c68fe2       invalid
          0x844811d0      12ca           ldm r2!, {r1, r4}
       ,< 0x844811d2      8ce2           b 0x844816ee
          0x844811d4      a0fdbce5       stc2 p5, c14, [r0, 0x2f0]!
          0x844811d8      00c68fe2       invalid
          0x844811dc      12ca           ldm r2!, {r1, r4}
       ,< 0x844811de      8ce2           b 0x844816fa
          0x844811e0      98fdbce5       ldc2 p5, c14, [r8, 0x2f0]
          0x844811e4      00c68fe2       invalid
          0x844811e8      12ca           ldm r2!, {r1, r4}
       ,< 0x844811ea      8ce2           b 0x84481706
          0x844811ec      90fdbce5       ldc2 p5, c14, [r0, 0x2f0]
          0x844811f0      00c68fe2       invalid
          0x844811f4      12ca           ldm r2!, {r1, r4}
       ,< 0x844811f6      8ce2           b 0x84481712
          0x844811f8      88fdbce5       stc2 p5, c14, [r8, 0x2f0]
          0x844811fc      00c68fe2       invalid
          0x84481200      12ca           ldm r2!, {r1, r4}
       ,< 0x84481202      8ce2           b 0x8448171e
          0x84481204      80fdbce5       stc2 p5, c14, [r0, 0x2f0]
          0x84481208      00c68fe2       invalid
          0x8448120c      12ca           ldm r2!, {r1, r4}
       ,< 0x8448120e      8ce2           b 0x8448172a
          0x84481210      78fdbce5       ldc2l p5, c14, [r8, -0x2f0]!
          0x84481214      00c68fe2       invalid
          0x84481218      12ca           ldm r2!, {r1, r4}
          0x8448121a      8ce2           b 0x84481736
          0x8448121c      70fdbce5       ldc2l p5, c14, [r0, -0x2f0]!
          0x84481220      00c68fe2       invalid
          0x84481224      12ca           ldm r2!, {r1, r4}
          0x84481226      8ce2           b 0x84481742
```

The code above looks not like a legit inner function (we do not care for alignment and inspect the next branch).

Hitting 'u' returns us to the parent function, followed by [alt+2] which brings us into the 2nd branch

```
[0x84483aa4 [xAdvc]0 16384 frida://attach/usb/4c197256/com.zhiliaoapp.musically]> pd $r
            0x84483aa4      b0b5           push {r4, r5, r7, lr}
            0x84483aa6      02af           add r7, sp, 8
            0x84483aa8      adf5387d       sub.w sp, sp, 0x2e0
            0x84483aac      124c           ldr r4, [0x84483af8]          ; [0x84483af8:4]=0x10478
            0x84483aae      8646           mov lr, r0
            0x84483ab0      1248           ldr r0, [0x84483afc]          ; [0x84483afc:4]=0x9ca2
            0x84483ab2      9c46           mov ip, r3
            0x84483ab4      7c44           add r4, pc
            0x84483ab6      124b           ldr r3, [0x84483b00]          ; [0x84483b00:4]=0x102fe
            0x84483ab8      124d           ldr r5, [0x84483b04]          ; [0x84483b04:4]=227
            0x84483aba      7844           add r0, pc
            0x84483abc      cde901e1       strd lr, r1, [sp, 4]
            0x84483ac0      07a9           add r1, sp, 0x1c
            0x84483ac2      2468           ldr r4, [r4]
            0x84483ac4      01f52e71       add.w r1, r1, 0x2b8
            0x84483ac8      cde9032c       strd r2, ip, [sp, 0xc]
            0x84483acc      7d44           add r5, pc
            0x84483ace      7b44           add r3, pc
            0x84483ad0      2268           ldr r2, [r4]
            0x84483ad2      cde90551       strd r5, r1, [sp, 0x14]
            0x84483ad6      05a9           add r1, sp, 0x14
            0x84483ad8      b792           str r2, [sp, 0x2dc]
            0x84483ada      0022           movs r2, 0
            0x84483adc      0091           str r1, [sp]
            0x84483ade      01a9           add r1, sp, 4
            0x84483ae0      00f06ae8       blx 0x84483bb8                ;[1]
            0x84483ae4      b798           ldr r0, [sp, 0x2dc]
            0x84483ae6      2168           ldr r1, [r4]
            0x84483ae8      081a           subs r0, r1, r0
            0x84483aea      04bf           itt eq
            0x84483aec      0df5387d       addeq sp, sp, 0x2e0
            0x84483af0      b0bd           pop {r4, r5, r7, pc}
            0x84483af2      fdf754eb       blx 0x8448119c                ;[2]
```

The 2nd branch at 0x84483aa4 looks better (proper function stub). We could easily drift back to the static analysis world, to find further evidence for it being the inner 'ss_encrypt' function. But hey, we are working with instrumentation, so let us just inspect the calls

Remember: While we disassembled at 0x84483aa4, the code is THUMB mode. Thus the proper tracing address would be 0x84483aa5 (LSB set to 1), unlike you like crashes (restarting here would not be funny, 'cause thanks to ASLR all function offsets would differ)

```
[0x84483aa4 [xAdvc]0 16384 frida://attach/usb/4c197256/com.zhiliaoapp.musically]> pd $r
            0x84483aa4      b0b5           push {r4, r5, r7, lr}
            0x84483aa6      02af           add r7, sp, 8
            0x84483aa8      adf5387d       sub.w sp, sp, 0x2e0
            0x84483aac      124c           ldr r4, [0x84483af8]          ; [0x84483af8:4]=0x10478
            0x84483aae      8646           mov lr, r0
            0x84483ab0      1248           ldr r0, [0x84483afc]          ; [0x84483afc:4]=0x9ca2
            0x84483ab2      9c46           mov ip, r3
```

In contrast to our first tracing attempt, we use the beautiful command for formatted tracing, which allows us to print out function parameters for each call in a predefined format.

Command syntax:

```
[0×84483ab8]> \dtf?
Usage: dtf [format] || dtf [addr] [fmt]
   ^  = trace onEnter instead of onExit
   +  = show backtrace on trace
  p/x = show pointer in hexadecimal
   c  = show value as a string (char)
   i  = show decimal argument
   z  = show pointer to string
   s  = show string in place
   O  = show pointer to ObjC object
Undocumented: Z, S
 dtf     trace format
```

The screenshot below shows how I placed my trace hook. The 'pppp' means that the first 4 function parameters should be printed as hex values (pointers) for each call.

Ultimately 2 calls get logged

```
[0×84483aa4]> \dtf 0×84483aa5 pppp
true
[0×84483aa4]> [dtf onLeave][Tue Jan 12 2021 17:25:25 GMT+0100] 0×84483aa5@0×84483aa5 - args: 0×799d2c00, 0×1819, 0×799d4800, 0×84e6f3a4. Retval: 0×0 backtrace: 0×84483cb4 libEncryptor.so!0×3cb4,
0×93314677 base.odex!0×390677
[dtf onLeave][Tue Jan 12 2021 17:26:25 GMT+0100] 0×84483aa5@0×84483aa5 - args: 0×847cc300, 0×682, 0×85ca5000, 0×84e6f3a4. Retval: 0×0 backtrace: 0×84483cb4 libEncryptor.so!0×3cb4,0×93314677 base
.odex!0×390677
```