

Deep down in my heart

- Link: <https://ash.firebird.sh/challenges?id=35>
- Author: stanley0010
- Category: Reverse
- Points: 909
- Solves: 4

What! It is obfuscated?! Legend has it that finding the original code is impossible!

If you feel dizzy, you can listen to [this](#)

[File](#)

solution

Extract `lib/*/libnative-lib.so`. `*` is a wildcard standing for a folder of any name. There are 4 such files, and any one of them will work. Then open the library file in a text editor of your choice and search for `driberif`, the reverse of `firebird`. One should find

```
}3v0l_07_5r3gn4r75_0n_3r4_3w{driberif . Reverse the text to get  
firebird{w3_4r3_n0_57r4ng3r5_70_l0v3} .
```

process

First, one should know that `apk` is just another format using the ubiquitous `zip`. This is the case for quite a lot of file formats as no one is gonna reinvent the wheel and make their own archive format. Even if you do not know that, trying to extract the file as a `zip` first is a good idea in general. So unzip the file into a folder.

After the unzipping the file, you will find that the files inside are binary. Of course, that is because it is compiled. This should be easy if you know Java or Android development. Decompile the `apk` file with a decompiler (online decompilers are a Google search away).

Now where is the flag? The idea of searching for it directly is quickly killed by the fake flag `firebird{0f_c0z_1_w1ll_n0t_put_7h3_fl4g_h3r3}`. You will also see this fake flag if you install the `apk` file. The fake flag does have one use though: Helping you to find the actual application code out of all those obfuscated Java files. The other obfuscated Java files are simply library files and you can just ignore them.

After finding the application code (or installing the `apk` file), you will find that it is a calculator app. One thing of interest is that the calculator uses [Java Native Interface](#) (JNI). JNI is basically a way to call native code from Java. The 4 JNI methods are named `jniADD`, `jniMUL`, `jniDIV`, `jniSUB`. If you review the name of this challenge, "Deep down in my

heart", one may conjecture that the native code contains the flag. After all, native code is more "deep down" than Java, right?

The native code is located under `lib`. There are 4 folders: `arm64-v8a`, `armeabi-v7a`, `x86`, `x86_64`. It is somewhat obvious from the name that each folder contains the exact same native code but compiled for different architectures, so you can choose any one of them. You will find a `so` file inside. A bit of Linux knowledge would tell you that the `so` file is a native library file, so that is where the native code is located.

Disassemble the `so` file. In this case, we used [IDA Free](#). Now if you know a bit of JNI, you should know that the JNI methods have names formatted in the following way:

`Java_(package name with dots replaced by underscores)_(Java method name)`. So search for symbols starting with `Java_`. One would find 5 functions instead of the expected 4, so the extra unused function is what we are interested in:

`Java_com_x64m_xsfmnative_MainActivity_jniMSG`. Looking at the disassembly of the starting part of the function:

```
public Java_com_x64m_xsfmnative_MainActivity_jniMSG
Java_com_x64m_xsfmnative_MainActivity_jniMSG proc near

var_30= byte ptr -30h
var_2F= byte ptr -2Fh
ptr= qword ptr -20h
var_18= qword ptr -18h

; __unwind { // __gxx_personality_v0
push    r14
push    rbx
sub     rsp, 28h
mov     r14, rdi
mov     rax, fs:28h
mov     [rsp+38h+var_18], rax
lea     rsi, unk_37FE0
lea     rbx, [rsp+38h+var_30]
mov     rdi, rbx
call
__ZNSt6__ndk112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEEEC2IDnEEPK
c ;
std::__ndk1::basic_string<char,std::__ndk1::char_traits<char>,std::__ndk1::a
llocator<char>>::basic_string<decltype(nullptr)>(char const*)
; try {
lea     rsi, unk_37FE1
mov     edx, 0ADh
mov     rdi, rbx
call
__ZNSt6__ndk112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEEE6assignEP
Kcm ;
std::__ndk1::basic_string<char,std::__ndk1::char_traits<char>,std::__ndk1::a
```

```

llocator<char>>::assign(char const*,ulong)
lea     rsi, unk_3808F
lea     rdi, [rsp+38h+var_30]
mov     edx, 0FCh
call
__ZNSt6__ndk112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEE6assignEP
Kcm ;
std::__ndk1::basic_string<char,std::__ndk1::char_traits<char>,std::__ndk1::a
llocator<char>>::assign(char const*,ulong)
lea     rsi, unk_3818C
lea     rdi, [rsp+38h+var_30]
mov     edx, 0E2h
call
__ZNSt6__ndk112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEE6assignEP
Kcm ;
std::__ndk1::basic_string<char,std::__ndk1::char_traits<char>,std::__ndk1::a
llocator<char>>::assign(char const*,ulong)
lea     rsi, aNeverGonnaGive ; "Never gonna give you up. Never gonna le" ...
lea     rdi, [rsp+38h+var_30]
mov     edx, 0B0h
call
__ZNSt6__ndk112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEE6assignEP
Kcm ;
std::__ndk1::basic_string<char,std::__ndk1::char_traits<char>,std::__ndk1::a
llocator<char>>::assign(char const*,ulong)
test    [rsp+38h+var_30], 1
jz      short loc_12860
// ...

```

One would notice that the assembly references some data containing interesting song lyrics. The most obvious one above is "Never gonna give you up", but the `unk_*` ones are also song lyrics references. The last step is to inspect the referenced text. In IDA, you can do so by `ctrl`-clicking the `aNeverGonnaGive`. Then start looking at the surrounding text. If you look long enough, you will eventually find something that looks like a flag:

```

.rodata:00000000000038141      db  7Dh ; }
.rodata:00000000000038142      db  33h ; 3
.rodata:00000000000038143      db  76h ; v
.rodata:00000000000038144      db  30h ; 0
.rodata:00000000000038145      db  6Ch ; l
.rodata:00000000000038146      db  5Fh ; _
.rodata:00000000000038147      db  30h ; 0
.rodata:00000000000038148      db  37h ; 7
.rodata:00000000000038149      db  5Fh ; _
.rodata:0000000000003814A      db  35h ; 5
.rodata:0000000000003814B      db  72h ; r
.rodata:0000000000003814C      db  33h ; 3
.rodata:0000000000003814D      db  67h ; g

```

.rodata:0000000000003814E	db 6Eh ; n
.rodata:0000000000003814F	db 34h ; 4
.rodata:00000000000038150	db 72h ; r
.rodata:00000000000038151	db 37h ; 7
.rodata:00000000000038152	db 35h ; 5
.rodata:00000000000038153	db 5Fh ; _
.rodata:00000000000038154	db 30h ; 0
.rodata:00000000000038155	db 6Eh ; n
.rodata:00000000000038156	db 5Fh ; _
.rodata:00000000000038157	db 33h ; 3
.rodata:00000000000038158	db 72h ; r
.rodata:00000000000038159	db 34h ; 4
.rodata:0000000000003815A	db 5Fh ; _
.rodata:0000000000003815B	db 33h ; 3
.rodata:0000000000003815C	db 77h ; w
.rodata:0000000000003815D	db 7Bh ; {
.rodata:0000000000003815E	db 64h ; d
.rodata:0000000000003815F	db 72h ; r
.rodata:00000000000038160	db 69h ; i
.rodata:00000000000038161	db 62h ; b
.rodata:00000000000038162	db 65h ; e
.rodata:00000000000038163	db 72h ; r
.rodata:00000000000038164	db 69h ; i
.rodata:00000000000038165	db 66h ; f

Reading it gives }3v0l_07_5r3gn4r75_0n_3r4_3w{driberif. It should not be too difficult to notice that reversing it gives the flag. Reverse the text to get
 firebird{w3_4r3_n0_57r4ng3r5_70_l0v3}.

Unfortunately for us, we are a bit "too blind" as in the song lyrics. After some struggling, we opened the library file in a text editor and search for "Never gonna give you up" to find the reversed flag next to it. 🐼