goingnative write-up

First, decompile the app APK file with jadx and open the *MainActivity* of the app under the path *goingnative/sources/com/mobiotsec/goingnative/MainActivity.java*.

We can start focusing on the *splitFlag* method, where we find important information about the flag:

- The format of the flag is FLAG{XXXXXXXXX}
- XXXXXXX has to be 15-character long

```
public String splitFlag(String flag) {
  if (!flag.startsWith("FLAG{") || !flag.endsWith("}")) {
   return "Invalid flag";
  }
  String flag2 = flag.replace("FLAG{", "").replace("}", "");
  if (flag2.length() != 15 || checkFlag(flag2) == -1) {
   return "Invalid flag";
  }
  return "Correct flag!";
}
```

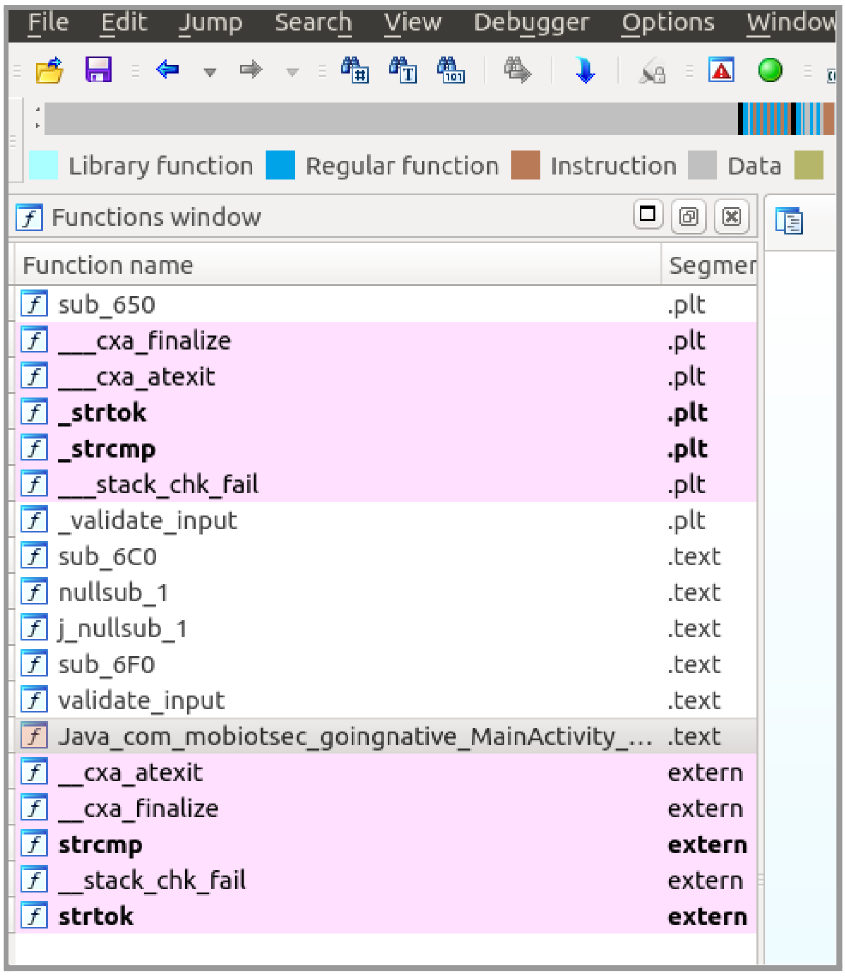
Further inspecting the code of the *MainActivity* class, we find that a native library (i.e., *goingnative*) is loaded and that the app Java code calls a native function (i.e., *checkFlag*) from the native library.

```
public native int checkFlag(String str);

static {
      System.loadLibrary("goingnative");
}
```

To inspect the logic of the native library, we rely on the IDA tool. First, we open the file *goingnative/resources/lib/x86_64/libgoingnative.so* in IDA.

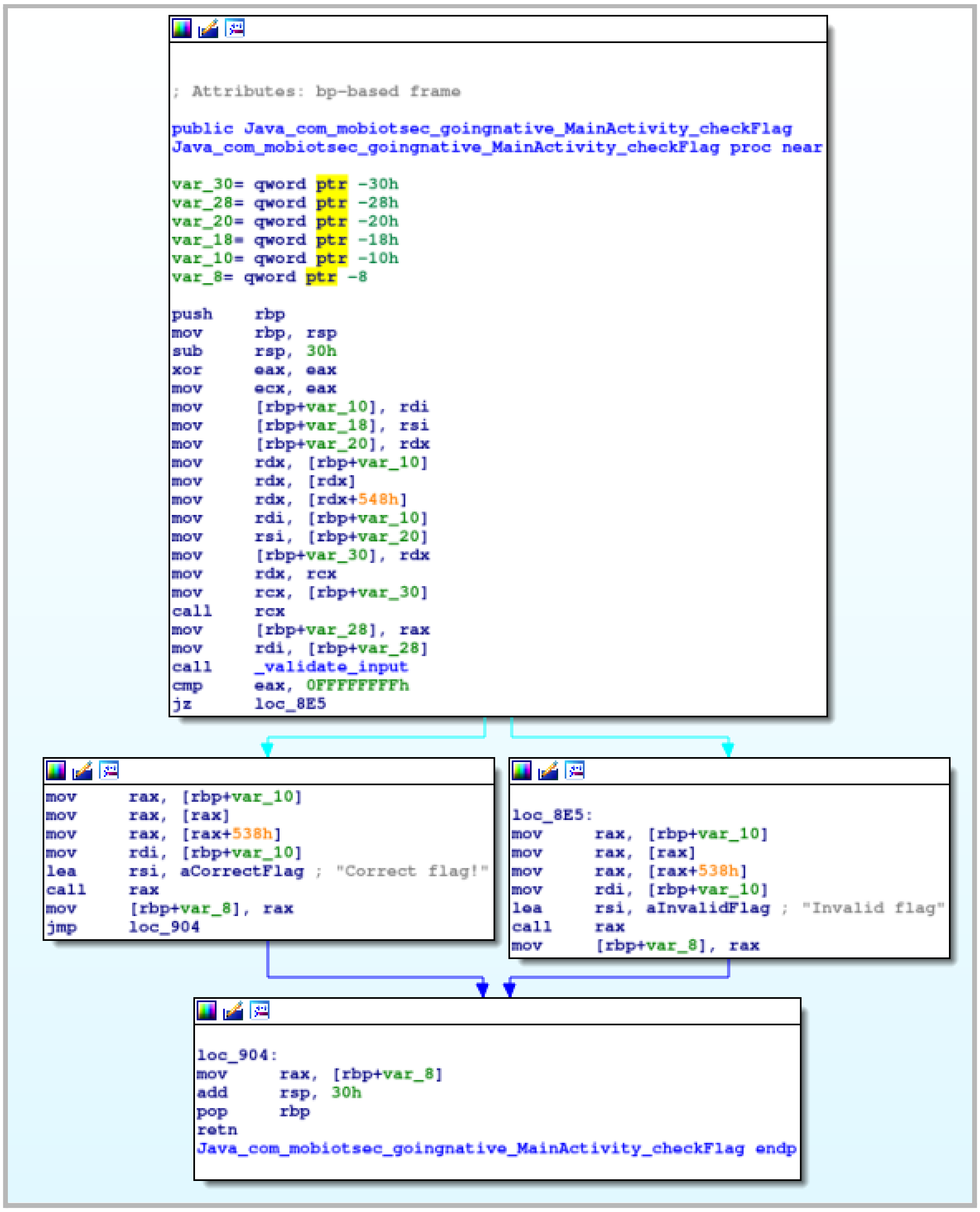
In the window on the left of the tool, shown in Fig.1, we can see all the functions called in the native library. The interesting ones are *Java_com_mobiotsec_goingnative_MainActivity_checkFlag* and *validate_input*.



Function name	Segment
sub_650	.plt
___cxa_finalize	.plt
___cxa_atexit	.plt
_strtok	**.plt**
_strcmp	**.plt**
___stack_chk_fail	.plt
_validate_input	.plt
sub_6C0	.text
nullsub_1	.text
j_nullsub_1	.text
sub_6F0	.text
validate_input	.text
Java_com_mobiotsec_goingnative_MainActivity_...	.text
__cxa_atexit	extern
__cxa_finalize	extern
strcmp	**extern**
__stack_chk_fail	extern
strtok	**extern**

Fig.1 Functions called in the *libgoingnative.so* library.

Selecting the *Java_com_mobiotsec_goingnative_MainActivity_checkFlag* function, we can inspect its logic through the flow shown in Fig. 2. At the end of the first block, we can see that the value returned by the *validate_input* function and saved in the *eax* register is compared with -1 (0FFFFFFFFh hex value). According to the result of the comparison, the native code prints the “Correct flag!” or the “Invalid flag” strings. Thus, we go further with the inspection of the *validate_input* function.



```
; Attributes: bp-based frame

public Java_com_mobiotsec_goingnative_MainActivity_checkFlag
Java_com_mobiotsec_goingnative_MainActivity_checkFlag proc near

var_30= qword ptr -30h
var_28= qword ptr -28h
var_20= qword ptr -20h
var_18= qword ptr -18h
var_10= qword ptr -10h
var_8= qword ptr -8

push    rbp
mov     rbp, rsp
sub     rsp, 30h
xor     eax, eax
mov     ecx, eax
mov     [rbp+var_10], rdi
mov     [rbp+var_18], rsi
mov     [rbp+var_20], rdx
mov     rdx, [rbp+var_10]
mov     rdx, [rdx]
mov     rdx, [rdx+548h]
mov     rdi, [rbp+var_10]
mov     rsi, [rbp+var_20]
mov     [rbp+var_30], rdx
mov     rdx, rcx
mov     rcx, [rbp+var_30]
call    rcx
mov     [rbp+var_28], rax
mov     rdi, [rbp+var_28]
call    _validate_input
cmp     eax, 0FFFFFFFFh
jz      loc_8E5
```

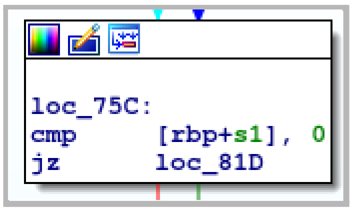
```
mov     rax, [rbp+var_10]
mov     rax, [rax]
mov     rax, [rax+538h]
mov     rdi, [rbp+var_10]
lea     rsi, aCorrectFlag ; "Correct flag!"
call    rax
mov     [rbp+var_8], rax
jmp     loc_904
```

```
loc_8E5:
mov     rax, [rbp+var_10]
mov     rax, [rax]
mov     rax, [rax+538h]
mov     rdi, [rbp+var_10]
lea     rsi, aInvalidFlag ; "Invalid flag"
call    rax
mov     [rbp+var_8], rax
```

```
loc_904:
mov     rax, [rbp+var_8]
add     rsp, 30h
pop     rbp
retn
Java_com_mobiotsec_goingnative_MainActivity_checkFlag endp
```

Fig.2 Flow of the *Java_com_mobiotsec_goingnative_MainActivity_checkFlag* function.

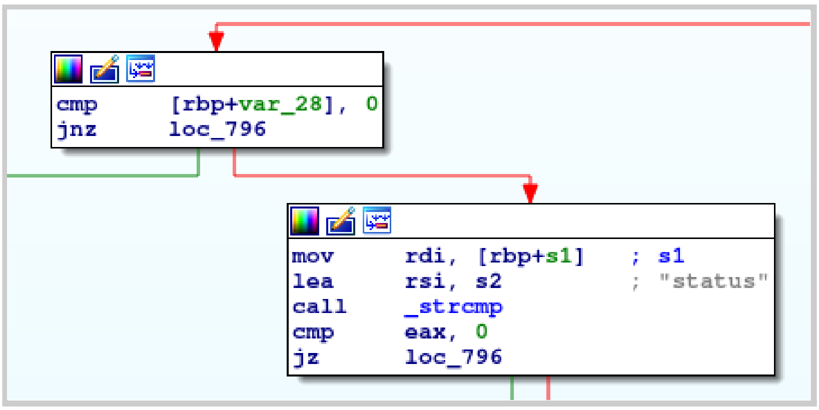
Looking at the high level structure, we immediately identify a loop. Also, we notice that in the first block there is a call to the *strtok* function. The two most important variables of the loop are *rbp+s1* and *rbp+var_28*. *Rbp+s1* is involved in the condition which determines the end of the loop, as shown in Fig. 3. The value contained in the *rbp+s1* variable is compared with 0 (or NULL). If this is the case, the program goes towards another block where the *rbp+var_28* variable is evaluated before exiting the *validate_input* function. Otherwise, the value in *rbp+s1* undergoes additional checks. Since in the first block the return value of the *strtok* function is saved into *rax* and further saved into *rbp+s1* (*mov [rbp+s1], rax*), we can conclude *rbp+s1* contains the current token, evaluated at each cycle.



```
loc_75C:
cmp     [rbp+s1], 0
jz      loc_81D
```

Fig.3 Conditional block of the loop in the *validate_input* function.

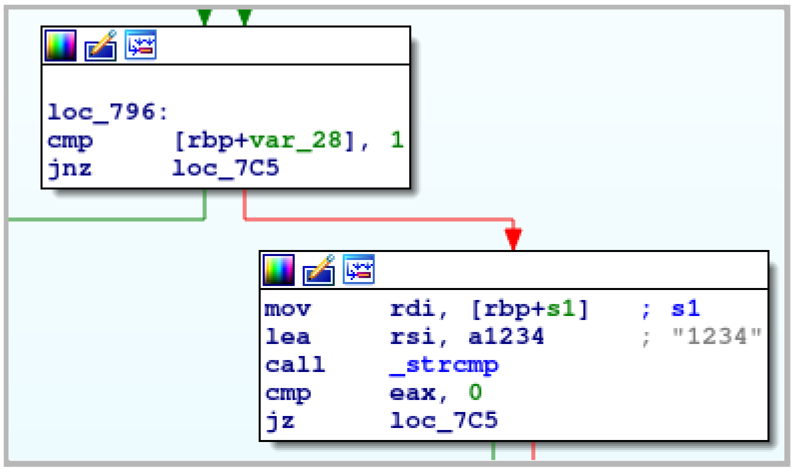
Moving on, we see that the left part of the flow has a very repetitive structure. In particular, we have first an evaluation of the value in *rbp+var_28* and then an evaluation of the value in *rbp+s1,* as shown in Fig. 4, Fig.5 and Fig.6.



```
cmp     [rbp+var_28], 0
jnz     loc_796
```

```
mov     rdi, [rbp+s1]    ; s1
lea     rsi, s2          ; "status"
call    _strcmp
cmp     eax, 0
jz      loc_796
```

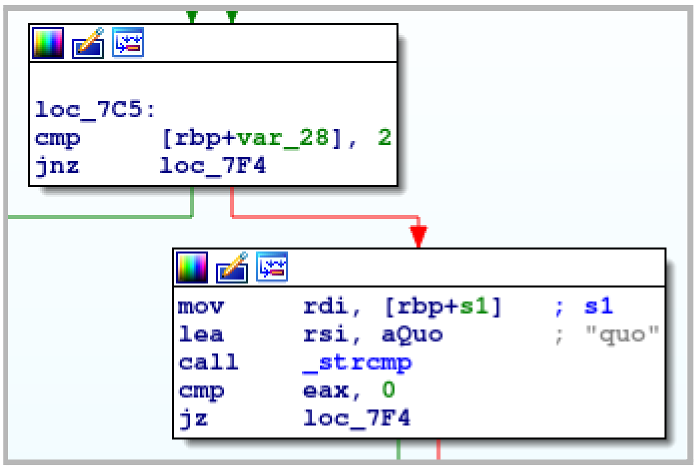
Fig.4 First evaluation of the values in *rbp+var_28* and *rbp+s1*.



```
loc_796:
cmp     [rbp+var_28], 1
jnz     loc_7C5
```

```
mov     rdi, [rbp+s1]    ; s1
lea     rsi, a1234       ; "1234"
call    _strcmp
cmp     eax, 0
jz      loc_7C5
```

Fig.5 Second evaluation of the values in *rbp+var_28* and *rbp+s1*.



```
loc_7C5:
cmp     [rbp+var_28], 2
jnz     loc_7F4
```

```
mov     rdi, [rbp+s1]     ; s1
lea     rsi, aQuo         ; "quo"
call    _strcmp
cmp     eax, 0
jz      loc_7F4
```

Fig.6 Third evaluation of the values in *rbp+var_28* and *rbp+s1*.

From the different checks presented before, we can assume that the first token should be “status”, the second one should be “1234” and the third one should be “quo”. Thus, the flag should be something like FLAG{status*delimiter*1234*delimiter*quo}. To identify the right delimiter, we can either try out all the possible delimiters through the app or remember that MOBIOTSEC flags usually have the “_” character as a delimiter.