goingseriousnative write-up

First, decompile the app APK file with jadx and open the *MainActivity* of the app under the path *goingseriousnative/sources/com/mobiotsec/goingseriousnative/MainActivity.java*. We can immediately see the value of the FLAG this time. However, the form online also expects a PIN value. Thus, further analysis is required.

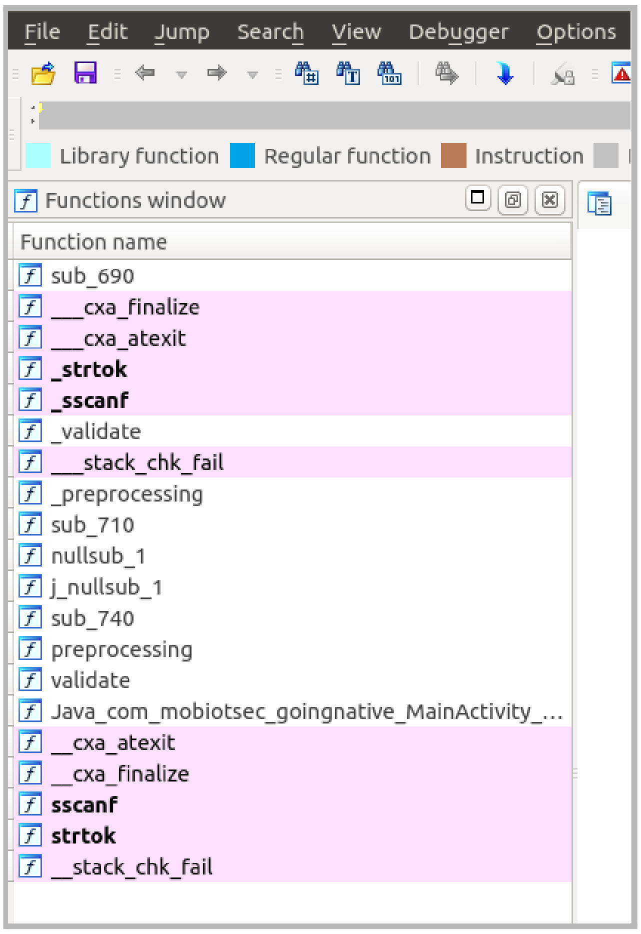
```
((Button) findViewById(R.id.checkflag)).setOnClickListener(new
View.OnClickListener() {
 int output = 0;
 public void onClick(View v) {
  int checkFlag =
MainActivity.this.checkFlag(flagWidget.getText().toString());
  this.output = checkFlag;
  if (checkFlag == 0) {
resultWidget.setText("FLAG{omnia_prius_experiri_quam_armis_sapientem_decet}")
;
  } else {
   resultWidget.setText("Invalid flag");
  }
 }
});
```

Further inspecting the code of the *MainActivity* class, we find that a native library (i.e., *goingseriousnative*) 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("goingseriousnative");
}
```

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

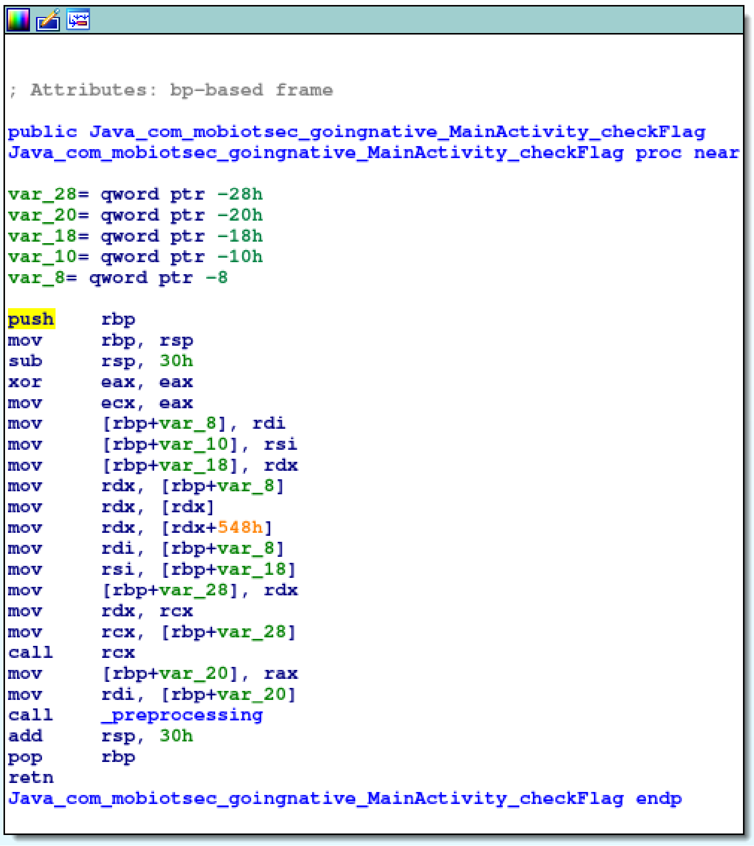
In the window on the left of the tool, shown in Fig.1, we can see all the functions called in the native library. The interesting ones are *Java_com_mobiotsec_goingnative_MainActivity_checkFlag*, *preprocessing* and *validate*.



Function name
sub_690
___cxa_finalize
___cxa_atexit
_strtok
_sscanf
_validate
___stack_chk_fail
_preprocessing
sub_710
nullsub_1
j_nullsub_1
sub_740
preprocessing
validate
Java_com_mobiotsec_goingnative_MainActivity_...
__cxa_atexit
__cxa_finalize
sscanf
strtok
__stack_chk_fail

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

Starting from the *Java_com_mobiotsec_goingnative_MainActivity_checkFlag* function, shown in Fig.2, the only relevant information is the call towards the *preprocessing* function.

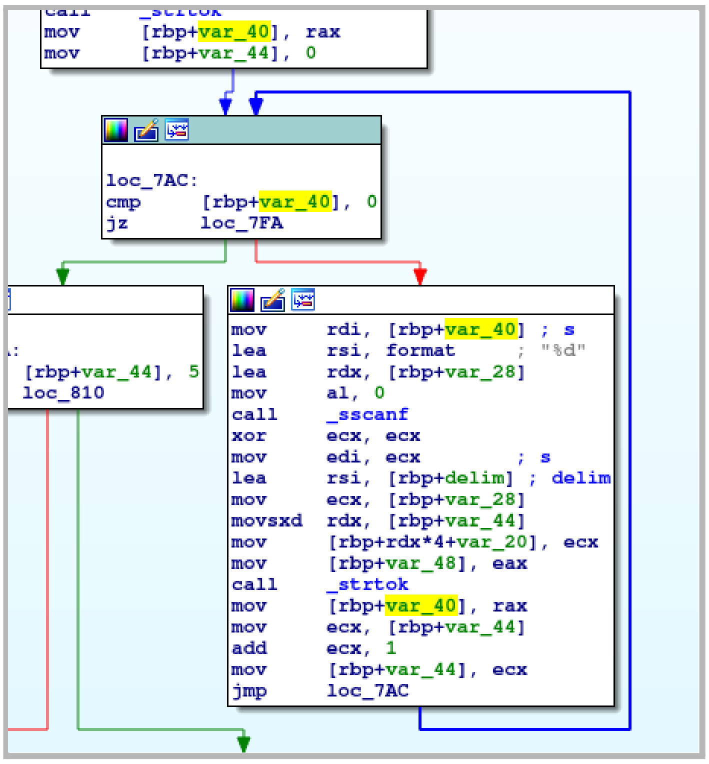


```
; Attributes: bp-based frame

public Java_com_mobiotsec_goingnative_MainActivity_checkFlag
Java_com_mobiotsec_goingnative_MainActivity_checkFlag proc near

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_8], rdi
mov     [rbp+var_10], rsi
mov     [rbp+var_18], rdx
mov     rdx, [rbp+var_8]
mov     rdx, [rdx]
mov     rdx, [rdx+548h]
mov     rdi, [rbp+var_8]
mov     rsi, [rbp+var_18]
mov     [rbp+var_28], rdx
mov     rdx, rcx
mov     rcx, [rbp+var_28]
call    rcx
mov     [rbp+var_20], rax
mov     rdi, [rbp+var_20]
call    _preprocessing
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.

Jumping into the *preprocessing* function, we can see a loop, shown in Fig.3, involving the following variables: *rbp+var_40*, *rbp+var_44*, *rbp+var_28* and *rbp+var_20*. *Rbp+var_40* contains the token extracted by the *strtok* function with a specific delimiter, saved into *rax* and then moved into the register. *Rbp+var_44* is the counter incremented for every new token. *rbp+var_28* is the third argument of the *sscanf* function (*lea rdx, [rbp+var_28]*) together with the current token (*mov rdi, [rbp+var_40] ; s*) and the "%d" format (*lea rsi, format; "%d"*). Thus, *rbp+var_28* contains the current token converted from the string format into the decimal one. *Rbp+var_20* is an array, saving all the decimal values converted from the string format at each round. In fact, *mov ecx, [rbp+var_28]* moves the current decimal value from *rbp+var_28* to the *ecx* register. Moreover, the instruction *mov [rbp+rdx*4+var_20], ecx* moves the value saved in the *ecx* register into a specific position of the *rbp+var_20* array. The position is specified as *rbp+rdx*4+var_20: rpb+var_20* is the baseline; *rdx*4* is the shift given by the value of *rdx*, which contains the current counter value (*movsxd rdx, [rbp+var_44]*) multiplied by 4 since it is an integer.

We can, thus, conclude that the *preprocessing* function identifies the tokens in the original string, transforms them from string into decimal format and saves them into an array of integers.



```
call    _strtok
mov     [rbp+var_40], rax
mov     [rbp+var_44], 0
```

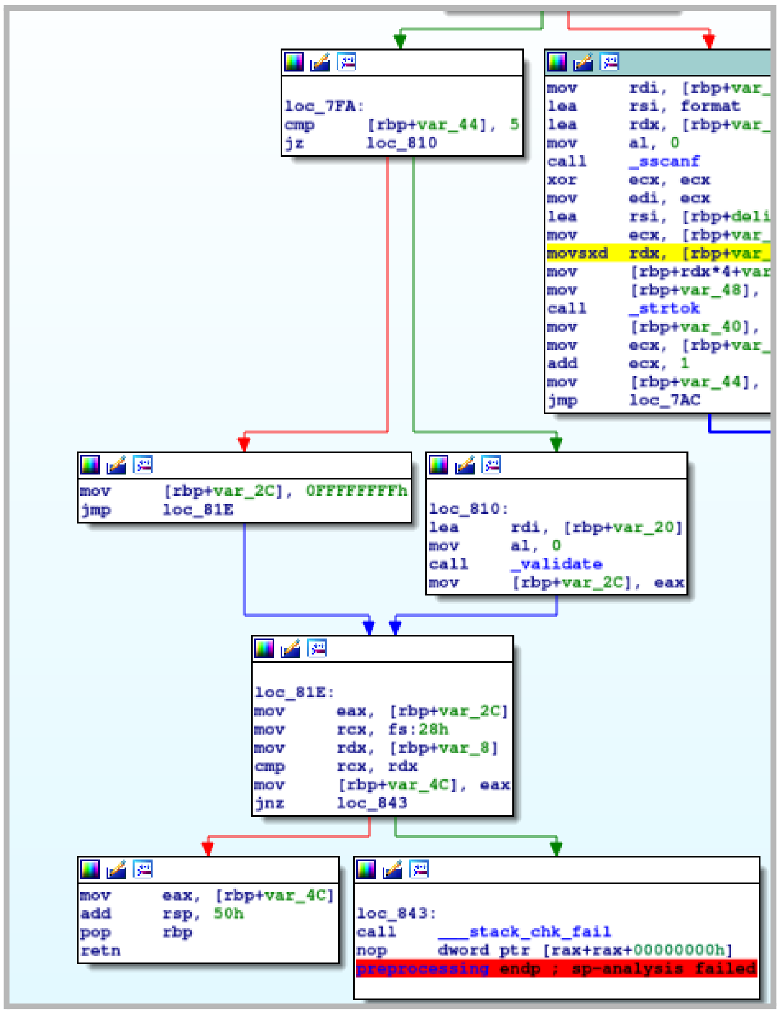
```
loc_7AC:
cmp     [rbp+var_40], 0
jz      loc_7FA
```

```
:
        [rbp+var_44], 5
        loc_810
```

```
mov     rdi, [rbp+var_40] ; s
lea     rsi, format     ; "%d"
lea     rdx, [rbp+var_28]
mov     al, 0
call    _sscanf
xor     ecx, ecx
mov     edi, ecx        ; s
lea     rsi, [rbp+delim] ; delim
mov     ecx, [rbp+var_28]
movsxd  rdx, [rbp+var_44]
mov     [rbp+rdx*4+var_20], ecx
mov     [rbp+var_48], eax
call    _strtok
mov     [rbp+var_40], rax
mov     ecx, [rbp+var_44]
add     ecx, 1
mov     [rbp+var_44], ecx
jmp     loc_7AC
```

Fig.3 Loop in the *preprocessing* function.

Moving on with the left part of the graph, shown in Fig. 4, we see that *rbp+var_44* is compared to value 5. If this is the case, the program jumps to the *loc_810* block, where the *validate* function is called receiving the array of integers as an argument (*lea rdi, [rbp+var_20]*). Thus, we can conclude that the original string contains five tokens, each one being a decimal value and separated from the others by a delimiter.



```
loc_7FA:
cmp     [rbp+var_44], 5
jz      loc_810
```

```
mov     rdi, [rbp+var_
lea     rsi, format
lea     rdx, [rbp+var_
mov     al, 0
call    _sscanf
xor     ecx, ecx
mov     edi, ecx
lea     rsi, [rbp+deli
mov     ecx, [rbp+var_
movsxd  rdx, [rbp+var_
mov     [rbp+rdx*4+var
mov     [rbp+var_48],
call    _strtok
mov     [rbp+var_40],
mov     ecx, [rbp+var_
add     ecx, 1
mov     [rbp+var_44],
jmp     loc_7AC
```

```
mov     [rbp+var_2C], 0FFFFFFFFh
jmp     loc_81E
```

```
loc_810:
lea     rdi, [rbp+var_20]
mov     al, 0
call    _validate
mov     [rbp+var_2C], eax
```

```
loc_81E:
mov     eax, [rbp+var_2C]
mov     rcx, fs:28h
mov     rdx, [rbp+var_8]
cmp     rcx, rdx
mov     [rbp+var_4C], eax
jnz     loc_843
```

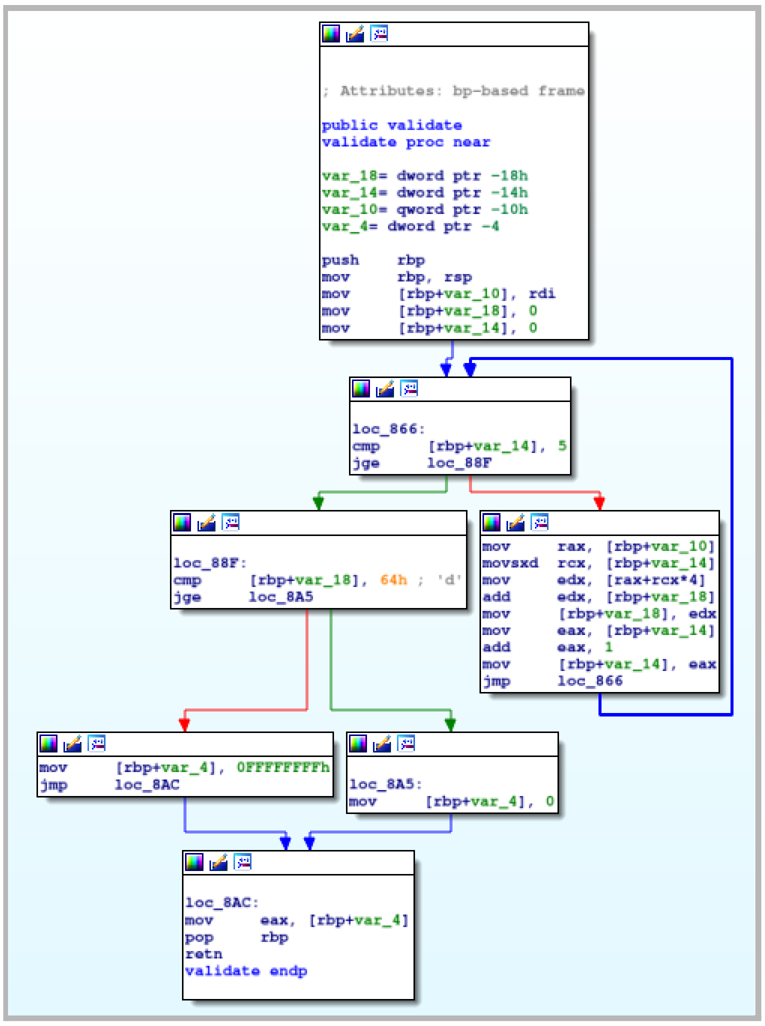
```
mov     eax, [rbp+var_4C]
add     rsp, 50h
pop     rbp
retn
```

```
loc_843:
call    ___stack_chk_fail
nop     dword ptr [rax+rax+00000000h]
preprocessing endp ; sp-analysis failed
```

Fig.4 Remaining logic of the *preprocessing* function.

We thus jump into the *validate* function, shown in Fig. 5. We see three relevant variables here: *rbp+var_14, rbp+var_10* and *rbp+var_18. Rbp+var_14* is the counter of the loop, incremented at each round until the value equal to 5 is reached. *Rbp+var_10* is initialized with the array of integers received as an argument of the function. At each cycle, an element of the array is retrieved and added to *rbp+var_18.*
Finally, in the *loc_88F* block, *rbp+var_18* is compared to the value 64h (100 in decimal representation).

To summarize, the PIN expected by the app should contain 5 digits, which sum should be greater than 100.



```
; Attributes: bp-based frame

public validate
validate proc near

var_18= dword ptr -18h
var_14= dword ptr -14h
var_10= qword ptr -10h
var_4= dword ptr -4

push    rbp
mov     rbp, rsp
mov     [rbp+var_10], rdi
mov     [rbp+var_18], 0
mov     [rbp+var_14], 0
```

```
loc_866:
cmp     [rbp+var_14], 5
jge     loc_88F
```

```
loc_88F:
cmp     [rbp+var_18], 64h ; 'd'
jge     loc_8A5
```

```
mov     rax, [rbp+var_10]
movsxd  rcx, [rbp+var_14]
mov     edx, [rax+rcx*4]
add     edx, [rbp+var_18]
mov     [rbp+var_18], edx
mov     eax, [rbp+var_14]
add     eax, 1
mov     [rbp+var_14], eax
jmp     loc_866
```

```
mov     [rbp+var_4], 0FFFFFFFFh
jmp     loc_8AC
```

```
loc_8A5:
mov     [rbp+var_4], 0
```

```
loc_8AC:
mov     eax, [rbp+var_4]
pop     rbp
retn
validate endp
```

Fig.5 Logic of the *validate* function.