Challenge #6 Solution

by Moritz Raabe

The sixth challenge is an Android application package (APK) file. Figure 1 shows the app run in a virtual Android device. Entering a random string into the text field and clicking the `Validate` button displays a new screen that shows the output `No` (see Figure 2).

Figure 1: Running the app

Figure 2: Result after submitting a random string

Android Analysis

Our goal is to obtain the next email address by understanding the app's internals. Using `apktool` (https://ibotpeaches.github.io/Apktool/), we are able to decode the app's configuration files and resources and can disassemble the application's source code to almost original form (Figure 3).

```
C:\apktool>apktool d android.apk
I: Using Apktool 2.0.0 on android.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: C:\Users\limited_user\apktool\framework\1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
testI: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
```

Figure 3: Decoding the APK file using `apktool`

In the file `AndroidManifest.xml` we can see that the app has two user interface screens (called `Activities` in Android): `MainActivity` and `ValidateActivity` (Figure 4, line 11 and 18).

```xml
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.flare_on.flare"
          platformBuildVersionCode="22"
          platformBuildVersionName="5.1.1-1819727">
    <application android:allowBackup="true"
                 android:icon="@drawable/icon"
                 android:label="@string/app_name"
                 android:theme="@style/AppTheme">
        <activity android:label="@string/app_name"
                  android:name="com.flareon.flare.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
        <activity android:label="@string/title_activity_validate_email"
                  android:name="com.flareon.flare.ValidateActivity"
                  android:parentActivityName="com.flareon.flare.MainActivity">
            <meta-data android:name="android.support.PARENT_ACTIVITY"
                       android:value="com.flareon.flare.MainActivity"/>
        </activity>
    </application>
</manifest>
```

Figure 4: `AndroidManifest.xml`

During packaging, an application developer compiles Android application code files into Dalvik Executable (DEX) files. `apktool` disassembles DEX files into easily readable smali files (the syntax is borrowed from the Jasmin project: http://jasmin.sourceforge.net/about.html). A smali file contains disassembled Java opcodes in a textual format. These files are located in the `smali/` directory.

In `MainActivity.smali` we can see a `validateEmail` method that passes the input string from the text field to the `ValidateActivity` activity (Figure 5). Layout files define the visual structure for activities. In the layout file `res/layout/activity_main.xml` we discover that `validateEmail` is executed after clicking the `Validate` button.

```
.method public validateEmail(Landroid/view/View;)V
    .locals 4
    .param p1, "view"    # Landroid/view/View;

    .prologue
    .line 21
    new-instance v2, Landroid/content/Intent;

    const-class v3, Lcom/flareon/flare/ValidateActivity;

    invoke-direct {v2, p0, v3}, Landroid/content/Intent;-><init>(Landroid/content/Context;Ljava/lang/Class;)V

    .line 22
    .local v2, "intent":Landroid/content/Intent;
    const v3, 0x7f0c004f

    invoke-virtual {p0, v3}, Lcom/flareon/flare/MainActivity;->findViewById(I)Landroid/view/View;

    move-result-object v1

    check-cast v1, Landroid/widget/EditText;

    .line 23
    .local v1, "emailAddress":Landroid/widget/EditText;
    invoke-virtual {v1}, Landroid/widget/EditText;->getText()Landroid/text/Editable;

    move-result-object v3

    invoke-virtual {v3}, Ljava/lang/Object;->toString()Ljava/lang/String;

    move-result-object v0

    .line 24
    .local v0, "email":Ljava/lang/String;
    const-string v3, "com.flare_on.flare.MESSAGE"

    invoke-virtual {v2, v3, v0}, Landroid/content/Intent;->putExtra(Ljava/lang/String;Ljava/lang/String;)Landroid/content/Intent;

    .line 25
    invoke-virtual {p0, v2}, Lcom/flareon/flare/MainActivity;->startActivity(Landroid/content/Intent;)V

    .line 26
    return-void
.end method
```

Figure 5: `validateEmail` method in `MainActivity.smali`

`ValidateActivity` encodes the string in ASCII and passes it to a native function named `validate` (Figure 6, line 90 and 115). This function is implemented in a shared library called `validate` and loaded using the `loadLibrary` function. The app displays the string returned by the native function to the user (Figure 6, line 96 and 101). We deduce that the library is responsible for the user input validation.

```
    .line 30
    invoke-virtual {p0, v4}, Lcom/flareon/flare/ValidateActivity;->validate(Ljava/lang/String;)Ljava/lang/String;

    move-result-object v2

    .line 31
    .local v2, "js":Ljava/lang/String;
    invoke-virtual {v3, v2}, Landroid/widget/TextView;->setText(Ljava/lang/CharSequence;)V

    .line 35
    .end local v2    # "js":Ljava/lang/String;
    :goto_0
    invoke-virtual {p0, v3}, Lcom/flareon/flare/ValidateActivity;->setContentView(Landroid/view/View;)V

    .line 36
    return-void

    .line 33
    :cond_0
    const-string v5, "No"

    invoke-virtual {v3, v5}, Landroid/widget/TextView;->setText(Ljava/lang/CharSequence;)V

    goto :goto_0
.end method

.method public native validate(Ljava/lang/String;)Ljava/lang/String;
.end method
```

Figure 6: Disassembled `ValidateActivity`

`apktool` extracted the associated shared library file to `lib/armeabi/libvalidate.so`. We run the `file` command on it and see that it is a stripped 32-bit ARM shared object file. Subsequently, we will focus on statically analyzing the library file in IDA Pro.

Basic Introduction to ARM

Since many contestants were stuck on this challenge because they did not know ARM assembly language, we will start with a short introduction to this processor architecture.

ARM is a Reduced Instruction Set Computer (RISC) architecture with data processing operating only on register contents. The available registers include 13 general-purpose registers (`R0`-`R12`), stack pointer (called `R13` or `SP`), link register (called `R14` or `LR`), program counter (called `R15` or `PC`), and an Application Program Status Register (`APSR`).

ARM instructions are made of a mnemonic and between zero and three operands. Immediate operands are identified by a leading `#` character. Load (`LDR`) and store (`STR`) instructions work on immediate, register, and `SP` relative offsets.

Arguments are passed to subroutines in `R0-R3` and on the stack. The return value is stored in `R0`. Subroutines are called with the `BL` and `BLX` instructions, which perform a branch after storing the return address to `LR`.

ARM Analysis

In IDA Pro's Exports tab, we find the validation function `Java_com_flareon_flare_ValidateActivity_validate`. We work our way backwards through the disassembly, to see how we end up at a successful output. Before the function ends at offset 0xFA4, a reference to one of the strings `That's it` or `No` is used (see Figure 7, after converting offset 0x3D3C to a reference to a string).

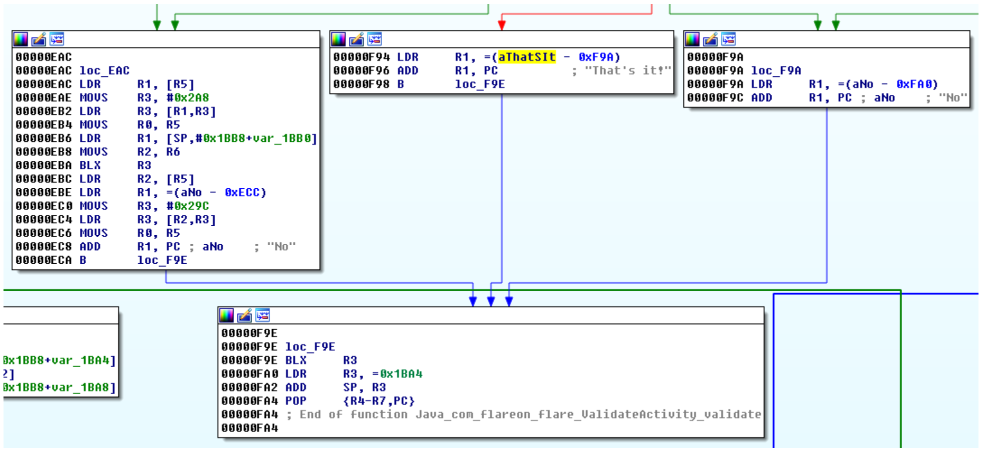

Figure 7: Use of string references before function end

Tracing the desired program flow we can see that the local variable `var_1BAC` must not be zero and that `var_1BB4` must be 23. We rename the variables accordingly.

At offset 0xF7C we see a function call to register `R3`. At this point `R3` holds `R1+0x2A8` with `R1` pointing to the data referenced in `R5`. `R5` holds the first argument passed to the validate function which is the `JNIEnv` (a pointer to pointers to function tables). We can resolve the JNI function by looking at the vtable definition at [https://svn.apache.org/repos/asf/harmony/enhanced/java/trunk/drlvm/vm/vmcore/src/jni/jni.cpp.](https://svn.apache.org/repos/asf/harmony/enhanced/java/trunk/drlvm/vm/vmcore/src/jni/jni.cpp) Because we are dealing with 32-bit ARM, we obtain the correct index into the JNI vtable by first dividing the given offset by 4 and then subtracting 4. The offset 0x2A8 translates to the JNI function `ReleaseStringUTFChars`. **Error! Reference source not found.** Figure 8 shows our changes to the disassembly so far.

```
00000F6E
00000F6E loc_F6E
00000F6E LDR     R1, [R5]
00000F70 MOVS    R3, #0x2A8
00000F74 LDR     R3, [R1,R3]
00000F76 MOVS    R0, R5
00000F78 LDR     R1, [SP,#0x1BB8+ptr_user_input_string]
00000F7A MOVS    R2, R6
00000F7C BLX     R3                 ; ReleaseStringUTFChars
00000F7E LDR     R2, [R5]
00000F80 LDR     R1, [SP,#0x1BB8+must_be_23]
00000F82 MOVS    R3, #0x29C
00000F86 LDR     R3, [R2,R3]
00000F88 MOVS    R0, R5
00000F8A CMP     R1, #23
00000F8C BNE     loc_F9A
```

```
00000F8E LDR     R1, [SP,#0x1BB8+must_not_be_0]
00000F90 CMP     R1, #0
00000F92 BEQ     loc_F9A
```

Figure 8: Renamed variables and resolved JNI call

Repeating this procedure for the other offsets reveals calls to `NewStringUTF` (offset 0x9E) and `GetStringUTFChars` (offset 0xE92). `GetStringUTFChars` argument's include the `JNIEnv` and the third argument passed to the validate function (stored in `var_1BB0`). This argument is a pointer to the Java string received from the text input field (see Figure 9).

```
00000E84 LDR     R1, [R5]
00000E86 MOVS    R3, #0x2A4
00000E8A LDR     R3, [R1,R3]
00000E8C MOVS    R0, R5
00000E8E LDR     R1, [SP,#0x1BB8+ptr_user_input_string]
00000E90 MOVS    R2, #0
00000E92 BLX     R3                 ; GetStringUTFChars
00000E94 SUBS    R6, R0, #0         ; R6 holds user input
00000E96 BEQ     loc_EAC
```

Figure 9: Retrieving the user input

We continue our analysis at the loop starting at offset 0xF32. A local buffer of 0x1B28 bytes is set to zero. The value that is compared to the string length of the user input is used as an offset into the user input string. Starting at offset 0xF40 the byte at `user_input_string[offset+1]` is loaded. If it is not zero, `user_input_string[offset]` is shifted to the left by 8 bits and a bitwise OR is applied to both values. Logically, this means that the characters of the user input are pairwise translated into 16-bit numerical representations. For example, an input of `ab` would be represented as 0x6162 in hex or 24,930 in decimal.

At offset 0xEE6 we see a memory location being stored in a local variable. On closer examination we discover that the memory holds 3,476 16-bit numbers. By recognizing the sequence of the first numbers, we hypothesize that the values are the prime numbers between 2 and 32,381 (see Figure 10).

```
.rodata:00002214 primes          DCW 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43
.rodata:00002214                                         ; DATA XREF: Java_com_flareon_flar
.rodata:00002214                                         ; .text:off_FB8↑o ...
.rodata:00002214                 DCW 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103
.rodata:00002214                 DCW 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163
.rodata:00002214                 DCW 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227
.rodata:00002214                 DCW 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281
.rodata:00002214                 DCW 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353
```

Figure 10: An array holding the prime numbers between 2 and 32,381

Starting at offset 0xEE8, we discover another loop which divides the numerical representation of the user input by a prime from the primes array. If the prime divides the number, the value at the field in the local variable is incremented by one. The number is then divided by the current prime (`primes[index]`, with `index` being the offset into the primes array). The code loops back if the quotient is greater than one. If the prime does not divide the number, the loop repeats with the next prime in the array. In summary, this code generates a number's prime factorization and stores the factors in an array (see Figure 11).

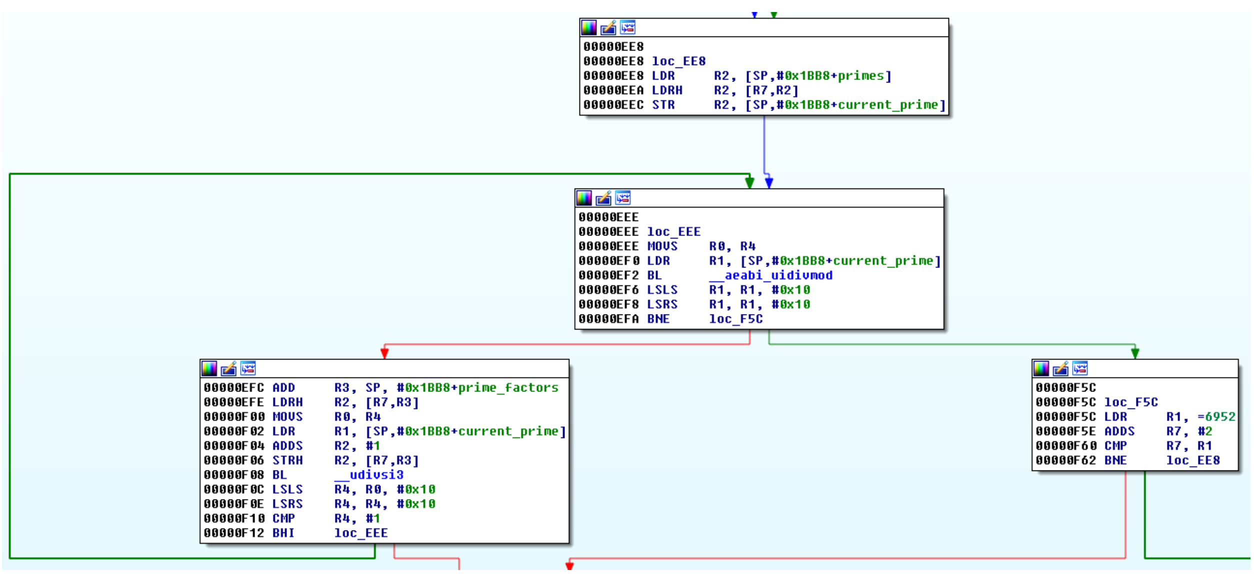

Figure 11: Code calculating the prime factorization and storing it in an array

Starting at offset 0xF14, the prime factorization gets compared to an offset into a local variable. This variable is loaded with 23 memory locations at the beginning of the validate function. Looking at the offsets, we deduce that each memory location represents a prime factorization array. Figure 12 shows the disassembly after renaming the offsets and creating arrays for each factorization.

```
.data:00005004 offsets_array   DCD factorization_01    ; DATA XREF: Java_com_flareon_f
.data:00005004                                         ; .text:off_FB0↑o
.data:00005008                 DCD factorization_02
.data:0000500C                 DCD factorization_03
.data:00005010                 DCD factorization_04
.data:00005014                 DCD factorization_05
.data:00005018                 DCD factorization_06
.data:0000501C                 DCD factorization_07
.data:00005020                 DCD factorization_08
.data:00005024                 DCD factorization_09
.data:00005028                 DCD factorization_10
.data:0000502C                 DCD factorization_11
.data:00005030                 DCD factorization_12
.data:00005034                 DCD factorization_13
.data:00005038                 DCD factorization_14
.data:0000503C                 DCD factorization_15
.data:00005040                 DCD factorization_16
.data:00005044                 DCD factorization_17
.data:00005048                 DCD factorization_18
.data:0000504C                 DCD factorization_19
.data:00005050                 DCD factorization_20
.data:00005054                 DCD factorization_21
.data:00005058                 DCD factorization_22
.data:0000505C                 DCD factorization_23
.data:00005060 factorization_23 DCW 0, 0, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
.data:00005060                                         ; DATA XREF: .data:0000505C↑o
.data:00005060                 DCW 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
.data:00005060                 DCW 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
.data:00005060                 DCW 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
.data:00005060                 DCW 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
.data:00005060                 DCW 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
.data:00005060                 DCW 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
.data:00005060                 DCW 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Figure 12: Offsets and factorization array

We can see that we obtain the correct string only if the comparison is equal for every iteration. Figure 13 shows the disassembly after we identified the local variable `var_1BB8` to be the index into the user input string.

Figure 13: Comparison of calculated prime factorization and stored values

Now that we know the algorithm we can translate the stored factorization arrays back to their numerical value and then to ASCII characters. For example, array `factorization_01` has field 0 set to 3, field 6 set to 1, and field 17 set to 1 (Figure 14). This translates to:

$$\text{primes}[0]^3 * \text{primes}[6]^1 * \text{primes}[36]^1 = 2^3 * 17^1 * 157^1 = 8 * 17 * 157 = 21352 = \text{0x5368}$$

```
.data:0002A5D0 factorization_01 DCW 3, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
.data:0002A5D0                                       ; DATA XREF: .data:offsets_array↑o
.data:0002A5D0                  DCW 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
.data:0002A5D0                  DCW 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Figure 14: The first stored prime factorization

0x53 translates to ASCII character `S` and 0x68 translates to ASCII character `h`.

We use an IDAPython script (Figure 15) to translate the remaining 22 arrays into ASCII code. The script creates a list of all the primes stored in the binary. It then creates a list of all addresses storing prime factorizations. For every factorization array, we read the factors and return the original number. This number is then converted to ASCII characters and appended to the solution string.

```
from struct import unpack
from idc import GetManyBytes

NUMBER_OF_PRIMES = 3476
SIZE_OF_NUM = 2
SIZE_OF_ADDR = 4
PRIMES_OFFSET = 0x2214
OFFSETS_ARRAY_OFFSET = 0x5004


def get_primes():
    """ Return list of primes. """
    primes = []
    for i in xrange(0, NUMBER_OF_PRIMES * SIZE_OF_NUM, SIZE_OF_NUM):
        bytes = GetManyBytes(PRIMES_OFFSET + i, SIZE_OF_NUM)
        primes.append(unpack("h", bytes)[0])
    return primes


def get_array_offsets():
    """ Return list of array offsets. """
    addresses = []
    for i in xrange(23):
        offset = i * SIZE_OF_ADDR
        bytes = GetManyBytes(0x5004 + offset, SIZE_OF_ADDR)
        addresses.append(unpack("l", bytes)[0])
    return addresses


def get_number(address, primes):
    """ Return original number based on prime factorization. """
    n = 1
    for i in xrange(0, NUMBER_OF_PRIMES * SIZE_OF_NUM, SIZE_OF_NUM):
        bytes = GetManyBytes(address + i, SIZE_OF_NUM)
```

```
        factor = unpack("h", bytes)[0]
        if factor != 0:
            n = n * (primes[i//2] ** factor)
    return n


def short_to_char(n):
    """ Return a number's character representation. """
    char1 = chr(n & 0xFF)
    if n > 0xFF:
        char2 = chr((n & 0xFF00) >> 8)
        return "%c%c" % (char2, char1)
    return "%c" % char1


def main():
    primes = get_primes()
    addresses = get_array_offsets()
    string = ""
    for array_address in addresses:
        n = get_number(array_address, primes)
        string = "%s%s" % (string, short_to_char(n))
    print string


main()
```
Figure 15: IDAPython script to obtain the email address

Running the script in IDA Pro presents us with the email address “Should_have_g0ne_to_tashi_$tation@flare-on.com”. When we enter this string into the app, we get the expected output `That's it!` (Figure 16).

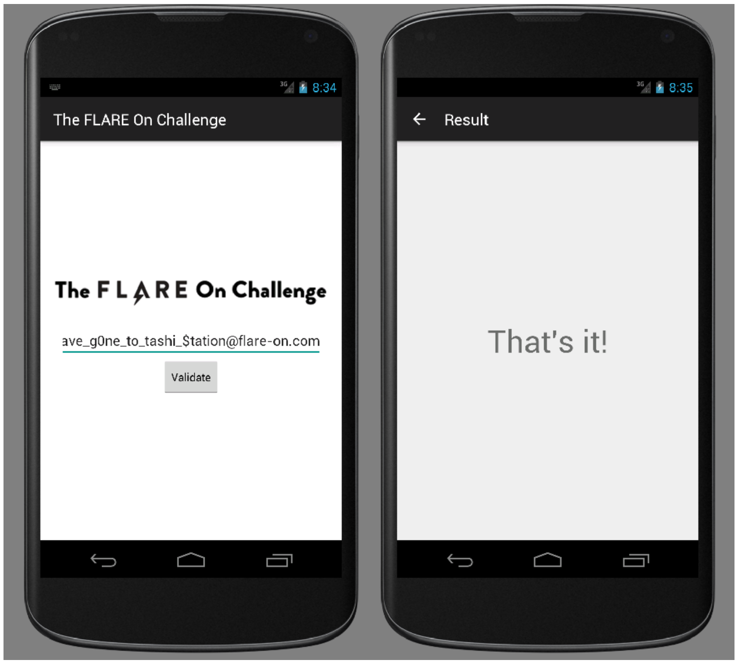

Figure 16: Successful output after entering the correct email address