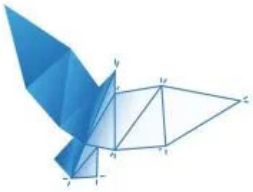


NVISO Labs

Cyber security research, straight from the lab!

Intercepting Flutter traffic on Android (ARMv8)

Jeroen Beckers android, Mobile May 20, 2020 6 Minutes

In a previous blogpost, I explained my steps for reversing the flutter.so binary to identify the correct offset/pattern to bypass certificate validation. As a very quick summary: **Flutter doesn't use the system's proxy settings, and it doesn't use the system's certificate store**, so normal approaches don't work. My previous guide only explained how to intercept Flutter on ARMv7 Android devices, but the steps don't fully transfer to ARMv8 so this blogpost quickly explains the steps for ARMv8

This blogpost is written as a guide / thought process, so you can find a **TL;DR at the bottom**.

Testing apps

First, we'll need a testing app. I've slightly updated the previous one to have two buttons: one for HTTP and one for HTTPS calls. This way, I can validate whether the proxy works, and then whether the Frida script works.

The app can be downloaded from our GitHub.

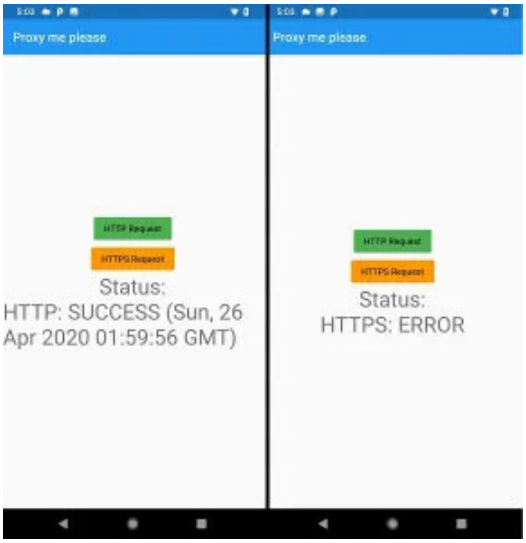
There are two functions in the app that call an HTTP and HTTPS endpoint:

```
1  void callHTTP(){
2    client = HttpClient();
3    _status = "Calling...";
```

```
  client
      .getUrl(Uri.parse('http://neverssl.com'))
      .then((request) => request.close())
      .then((response) => setState((){_status = "HTTP: SUCCESS (" + respons
      .catchError((e) =>
          setState(() {
            _status = "HTTP: ERROR";
            print(e.toString());
          })
      );
}
void callHTTPS(){
  client = HttpClient();
  _status = "Calling...";
  client
      .getUrl(Uri.parse('https://www.nviso.eu')) // produces a request obje
      .then((request) => request.close()) // sends the request
      .then((response) => setState((){
                                          _status = "HTTPS: SUCCESS (" + resp
                                      }))
      .catchError((e) =>
                      setState(() {
                        _status = "HTTPS: ERROR";
                        print(e.toString());
                      })
                );
}
```

Proxying the application

Flutter applications still don't automatically use the system's proxy, unless the developer adds this functionality by creating custom Android & iOS plugins that provide this information. Obviously, many developers won't do this, so we still need to intercept the traffic using ProxyDroid's root-based method rather than configuring the WIFI's proxy through the Android OS. After configuring ProxyDroid with the correct settings, Burp can see the requests from the app.



The HTTP requests work without any special requirement, while the HTTPS call prints an error to logcat:

```
04-26 16:59:02.758 11773 11802 E flutter : [ERROR:flutter/lib/ui/ui_dart_state.c
04-26 16:59:02.758 11773 11802 E flutter :       NO_START_LINE(pem_lib.c:622)
04-26 16:59:02.758 11773 11802 E flutter :       PEM routines(by_file.c:148)
04-26 16:59:02.758 11773 11802 E flutter :       NO_START_LINE(pem_lib.c:622)
04-26 16:59:02.758 11773 11802 E flutter :       PEM routines(by_file.c:148)
04-26 16:59:02.758 11773 11802 E flutter :       CERTIFICATE_VERIFY_FAILED: self
```

Disabling SSL verification

I initially thought the x64 version would be identical to the x86 version. It's the same source code, so why would the steps be any different... Unfortunately, when searching for 'x509.cc' in flutter.so, I found the same number of hits, but none of them were the correct function:

```
                    s_../../third_party/boringssl/src/_001c885a     XREF[4]:     FUN_005f4e1c:005f65f4(*),
                                                                                 FUN_0065a224:0065a3a0(*),
                                                                                 FUN_0065a224:0065a3b4(*),
                                                                                 FUN_0065a224:0065a400(*)
001c885a 2e 2e 2f        ds         "../../third_party/boringssl/src/ssl/ssl_x509.…
         2e 2e 2f
         74 68 69 …
```

The previous approach seems inefective

It's pretty obvious that the ssl_x509.cc class has been compiled somewhere in the 0x650000 region, but that's still a lot of functions to try to find the correct one. If searching for the filename doesn't work, maybe searching for the line number would work. If we take a look at the ssl_crypto_x509_session_verify_cert_chain function again, we can see that the OPENSSL_PUT_ERROR macro is called at line 390. Searching for the number 390 (or 0x186) gives us some results (Search > For Scalars...):

libflutter.so

Location	Preview	Hex	Decimal (Si...	Function Name
00103440	ddw 186h (dword[437][324])	186	390	
001aa6ea	ushort 186h	186	390	
001ab348	ushort 186h	186	390	
0037d124	add x8,x8,#0x186	186	390	FUN_0037c188
003b90a8	mov w3,#0x186	186	390	FUN_003b8f6c
0053609c	add x1,x1,#0x186	186	390	FUN_00533184
0065a604	mov w3,#0x186	186	390	FUN_0065a4ec
006e1574	add x8,x8,#0x186	186	390	FUN_006e1384
006fd854	add x9,x8,#0x186	186	390	FUN_006fd5bc
0075a5b8	cmp w0,#0x186	186	390	FUN_0075a5b8

Filter:

Min: -2147483647 Dec Max: 2147483647 Dec

Searching for magic numbers

A few of the results are around the 0x650000 region. The highlighted function (FUN_0065a4ec) looks like a good candidate, as the constant is loaded in w3 (the lower 32bit part of the x3 register), which is one of the argument registers on ARMv8. The function FUN_0065a4ec also has the correct signature, and it generally looks the same as the ARMv7 version:

Decompiled method in x86 vs x64

My normal approach would be to copy the first bytes of FUN_0065a4ec and search for them in-memory while the application is running, as I did in the previous blogpost, so I don't need to find the offset each time. Unfortunately, Frida's Memory.scan seems to crash on my test app, so for now we'll have to use the offset. (*Edit: An alternative approach was posted in the comments of this post, using Process.enumerateRangesSync*)

Ghidra uses 0x100000 as the base address of the module, so we have to subtract that from the Ghidra offset, resulting in an offset of **0x55a4ec**.

Opening Ghidra every time works, but it's not that convenient. We can also use binwalk to find the correct offset based on those first bytes of the function:

```
# The first bytes of the FUN_0065a4ec function
ff 03 05 d1 fc 6b 0f a9 f9 63 10 a9 f7 5b 11 a9 f5 53 12 a9 f3 7b 13 a9 08 0
# Find it using binwalk
binwalk -R "\xff\x03\x05\xd1\xfc\x6b\x0f\xa9\xf9\x63\x10\xa9\xf7\x5b\x11\xa9
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
5612780       0x55A4EC        Raw signature (\xff\x03\x05\xd1\xfc\x6b\x0f\xa
```

Let's throw this in a Frida script and test it!

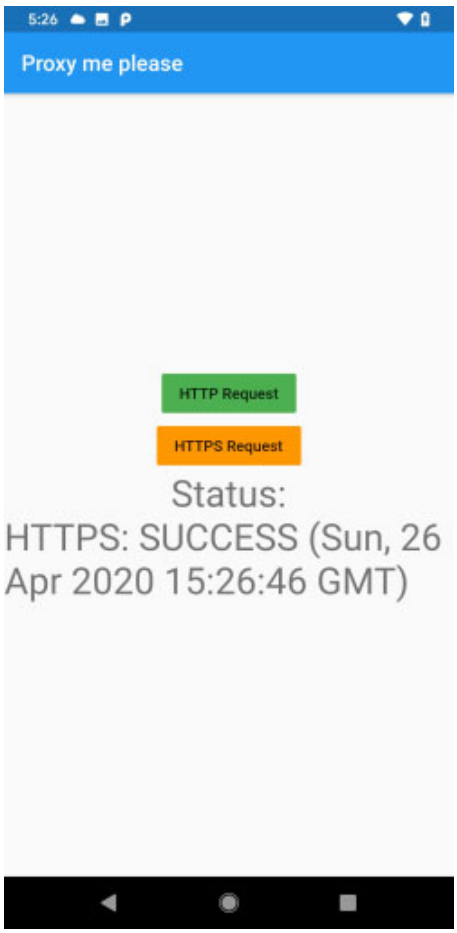
```
function hook_ssl_verify_result(address)
{
  Interceptor.attach(address, {
    onEnter: function(args) {
      console.log("Disabling SSL validation")
    },
    onLeave: function(retval)
    {
      console.log("Retval: " + retval)
      retval.replace(0x1);

    }
  });
}
function disablePinning(){
    // Change the offset on the line below with the binwalk result
    // If you are on 32 bit, add 1 to the offset to indicate it is a THUMB
    // Otherwise, you will get  'Error: unable to intercept function at ...
    var address = Module.findBaseAddress('libflutter.so').add(0x55a4ec)
    hook_ssl_verify_result(address);
}
setTimeout(disablePinning, 1000)
```

Running this file using Frida gives the expected outcome:

```
(secenv) ➜  flutter frida -Uf be.nviso.flutter_app -l hook.js --no-pause

     ____
    / _  |   Frida 12.8.20 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ | |       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://www.frida.re/docs/home/
Spawned `be.nviso.flutter_app`. Resuming main thread!
[SM-G950F::be.nviso.flutter_app]-> disablePinning()
[SM-G950F::be.nviso.flutter_app]-> Disabling SSL validation
Retval: 0x0
[SM-G950F::be.nviso.flutter_app]->
```



SSL Verification successfully disabled

Tangent: Why can this app perform cleartext HTTP calls?

My flutter app is making HTTP connections. This is forbidden by default since Android P, and you have to add a Network Security Policy that explicitly allows cleartext request if you still want to do so on Android 9+. My test app does not have a Network Security Policy, so what's going on?

The reason for this is the same reason why these blog posts are necessary: Flutter doesn't use default Android libraries. Because Flutter creates low level sockets and implements the HTTP stack on top of that, the requests never pass by the Android security controls that should prevent cleartext traffic from being used. This is an important thing to keep in mind when auditing the security of Flutter apps, as you might miss things if you're not careful.

TL;DR (ARMv7 and ARMv8)

1. Redirect with ProxyDroid on rooted device since Flutter apps are still proxy-unaware
2. Find the offset using binwalk
3. Use the Frida script to hook the method at that offset

Since the last blogpost, the signature for 32bit also changed, so I've included both signatures.

```
# Method signatures for ARMv7 (32bit)
2d e9 f0 4f a3 b0 81 46 50 20 10 70
```

```
2d e9 f0 4f a3 b0 82 46 50 20 10 70
# Get the offset
binwalk -R "\x2d\xe9\xf0\x4f\xa3\xb0\x81\x46\x50\x20\x10\x70" -R "\x2d\xe9\
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
3831160       0x3A7578        Raw signature (\x2d\xe9\xf0\x4f\xa3\xb0\x81\x
# Method signature for ARMv8 (64bit)
ff 03 05 d1 fc 6b 0f a9 f9 63 10 a9 f7 5b 11 a9 f5 53 12 a9 f3 7b 13 a9 08
# Get the offset
binwalk -R "\xff\x03\x05\xd1\xfc\x6b\x0f\xa9\xf9\x63\x10\xa9\xf7\x5b\x11\xa
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
5612780       0x55A4EC        Raw signature (\xff\x03\x05\xd1\xfc\x6b\x0f\x
```

Frida script to use the offset:

```
function hook_ssl_verify_result(address)
{
  Interceptor.attach(address, {
    onEnter: function(args) {
      console.log("Disabling SSL validation")
    },
    onLeave: function(retval)
    {
      console.log("Retval: " + retval)
      retval.replace(0x1);

    }
  });
}
function disablePinning(){
    // Change the offset on the line below with the binwalk result
    // If you are on 32 bit, add 1 to the offset to indicate it is a THUMB
    // Otherwise, you will get  'Error: unable to intercept function at ...
    var address = Module.findBaseAddress('libflutter.so').add(0x55a4ec)
    hook_ssl_verify_result(address);
}
setTimeout(disablePinning, 1000)
```

And launch it using Frida:

```
frida -Uf hook.js -f be.nviso.flutter_app --no-pause
```

If it still doesn't work, you'll have to figure out the correct method to hook yourself. You can try following the steps for ARMv7 as described on this blog.

About the author

Jeroen Beckers is a mobile security expert working in the NVISO Cyber Resilience team and co-author of the OWASP Mobile Security Testing Guide (MSTG). He also loves to program, both on high and low level stuff, and deep diving into the Android internals doesn't scare him. You can find Jeroen on LinkedIn.

Like this:

Like

Be the first to like this.

Tagged: android, flutter, Mobile

Published by Jeroen Beckers

Jeroen Beckers is a mobile security expert working in the NVISO Software and Security assessment team. He is a SANS instructor and SANS lead author of the SEC575 course. Jeroen is also a co-author of OWASP Mobile Security Testing Guide (MSTG) and the OWASP Mobile Application Security Verification Standard (MASVS). He loves to both program and reverse engineer stuff. View all posts by Jeroen Beckers

‹ Three tips for a better IT Acceptable Use Policy

A checklist to populate your Acceptable Use Policy ›