

NVISO Labs

Cyber security research, straight from the lab! 🌩️



Intercepting Flutter traffic on iOS

👤 Jeroen Beckers 📁 Uncategorized ⌚ June 12, 2020 📖 11 Minutes

My previous blogposts explained [how to intercept Flutter traffic on Android ARMv8](#), with a [detailed follow along guide for ARMv7](#). This blogpost does the same for iOS.

Testing apps

The beauty of a cross-platform application is of course that I can use my previous Android test app for iOS so it has the same functionality. You can find [an IPA version of the test file on our github](#), and you can install the app by copying it over to your jailbroken device and using appinst:

```
1 | appinst proxyme.ipa
2 | 2020-06-12 10:54:57.722 appinst[2454:755332] appinst (App Installer)
3 | 2020-06-12 10:54:57.724 appinst[2454:755332] Copyright (C) 2014-2019 Linus Y
4 | 2020-06-12 10:54:57.724 appinst[2454:755332] ** PLEASE DO NOT USE APPINST FO
5 | 2020-06-12 10:54:57.731 appinst[2454:755332] appinst: main:58 Cleaning up te
6 | 2020-06-12 10:54:57.751 appinst[2454:755332] appinst: main:133 Installing be
7 | 2020-06-12 10:55:02.500 appinst[2454:755332] appinst: main:183 Successfully
```

The app contains two buttons, one to send an HTTP request and one to send an HTTPS one:

```
1 | void callHTTP(){
2 |     client = HttpClient();
3 |     _status = "Calling...";
4 |     client
5 |     .getUrl(Uri.parse('http://neverssl.com'))
```

```

6      .then((request) => request.close())
7      .then((response) => setState((){_status = "HTTP: SUCCESS (" + respons
8      .catchError((e) =>
9          setState(() {
10              _status = "HTTP: ERROR";
11              print(e.toString());
12          }))
13      );
14  }
15  void callHTTPS(){
16      client = HttpClient();
17      _status = "Calling...";
18      client
19          .getUrl(Uri.parse('https://www.nviso.eu')) // produces a request obje
20          .then((request) => request.close()) // sends the request
21          .then((response) => setState((){
22              _status = "HTTPS: SUCCESS (" + resp
23          })))
24          .catchError((e) =>
25              setState(() {
26                  _status = "HTTPS: ERROR";
27                  print(e.toString());
28              }))
29      );
30  }

```

Let's get started

On iOS, the story is exactly the same as on Android. The app is proxy unaware and uses its own certificate store. Setting a proxy in your WIFI settings won't have any effect, and trusting your certificate in the system settings won't validate any HTTPS certificates. The first idea to fix the proxy issue would be to SSH into your iOS device and use iptables to redirect the traffic, just like ProxyDroid does on Android. Unfortunately, iptables requires kernel support, and the iOS kernel does not have any support for it. The next obvious step is to recompile the iOS kernel to implement this support. So download your copy of the iOS kernel and let's get started!

All jokes aside, adding kernel support is not an option, so we will have to look elsewhere. The easiest approach is to create a WIFI hotspot using a second WIFI adapter and use iptables to redirect all traffic to Burp. However, if you don't have an extra WIFI adapter, you can also set up an OpenVPN server and have your device connect to it. Both possibilities are explained below.

Setting up a WIFI hotspot

The steps are rather straightforward, though depending on your OS and network setup it might require a bit of troubleshooting. I'll run through the steps I took, starting from a clean Kali image, but if something goes wrong, you'll have to troubleshoot yourself ;). Note that you can also set one up through the Kali 'Advanced Network Configuration' panel (type: hotspot), but where's the fun in that?

Setting up kali

Download the latest Kali (2020.1b in my case) and spin up a VM instance. First, we need `hostapd` for a wireless network, and `dnsmasq` for the DHCP server:

```
1 | sudo apt-get update && sudo apt-get install hostapd dnsmasq
```

I'm using a small WIFI dongle with a Ralink 5370 chipset, so I have two adapters: `etho` and `wlan0`.

Setting up the WIFI network

We need to create a `hostapd` configuration for our network. Create the `mitmwifi.conf` file and add the data as seen below. This will create a WIFI network with SSID `MobileTestbed` and `Password123` as a password.

```
1 | sudo nano /etc/hostapd/mitmwifi.conf
2 | # Enter the following configuration:
3 | interface=wlan0
4 | driver=nl80211
5 | ssid=MobileTestbed
6 | hw_mode=g
7 | channel=6
8 | macaddr_acl=0
9 | ignore_broadcast_ssid=0
10 | auth_algs=1
11 | wpa=2
12 | wpa_key_mgmt=WPA-PSK
13 | rsn_pairwise=TKIP
14 | wpa_passphrase>Password123
```

Next, we update the `hostapd` init script to reference to the `mitmwifi.conf` file:

```
1 | sudo nano /etc/default/hostapd
2 | # Update the DAEMON_CONF line:
3 | DAEMON_CONF="/etc/hostapd/mitmwifi.conf"
```

Set up `dnsmasq` by modifying `/etc/dnsmasq.conf`

```
1 | sudo nano /etc/dnsmasq.conf
2 | # Add the following configuration to the end of the file:
3 | # The interface to listen on
4 | interface=wlan0
5 | # The range to distribute (192.168.10.100-250)
6 | dhcp-range=192.168.10.100,192.168.10.250,255.255.255.0,12h
7 | # The gateway (this ip)
8 | dhcp-option=3,192.168.10.1
9 | # The DNS server
10 | dhcp-option=6,1.1.1.1
11 | # Another DNS server
12 | server=1.1.1.1
13 | # Some logging
14 | log-queries
15 | log-dhcp
16 | # Listen on the localhost address
17 | listen-address=127.0.0.1
```

Assign the correct IP address to the wlan0 interface, enable IP forwarding and route the traffic to eth0 so that the subnet has internet access.

```
1 | sudo ifconfig wlan0 up 192.168.10.1 netmask 255.255.255.0
2 | sudo sysctl -w net.ipv4.ip_forward=1
3 | sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

Finally, start dnsmasq and restart hostapd. The WIFI network should show up on your device:

```
1 | sudo systemctl unmask hostapd
2 | sudo service hostapd start
3 | sudo service dnsmasq start
```

At this point, you have a working WIFI hotspot with outgoing internet access. Skip to ‘Setting up the MITM’ in case these steps were successful. Otherwise, you can try the OpenVPN approach explained below.

Setting up OpenVPN

In case you don’t have access to a WIFI adapter, or there are other reasons why the hotspot approach doesn’t work, we need a second option. Fortunately, there is a good way to have all the traffic from an iOS device go over an intermediate node: Through a VPN. This functionality has [already](#) been [used](#) by some corporations to spy on iOS users, so if they can do it, we can too!

Installing OpenVPN is pretty straightforward with the help of this [setup script](#). The script detects our Kali as a Debian OS, but fails to determine the version and will therefore exit. That’s why I added a step to delete the `exit` statement in the Debian version check. I used a bit of bash hacking, but if it doesn’t work anymore, just remove the line manually.

```
1 | wget https://git.io/vpn -O openvpn-install.sh
2 | sed -i "$((${grep -ni "debian is too old" openvpn-install.sh} | cut -d : -f 1))s/exit//" openvpn-install.sh
3 | chmod +x openvpn-install.sh
4 | sudo ./openvpn-install.sh
```

Choose the following options:

```
1 | # Choose the following options:
2 | Public IPv4 address / hostname [xx.xx.xx.xx]: 192.168.10.1      <<< Change wi
3 | Protocol [1]: 1          (UDP)
4 | Port [1194]: 1194
5 | DNS server [1]: 3          (1.1.1.1)
6 | Name [client]: nviso
```

The script will set everything up and create an OpenVPN configuration located in the `/root/` home directory. If you run `sudo ifconfig` now, you can see that a `tun0` interface has been

added.

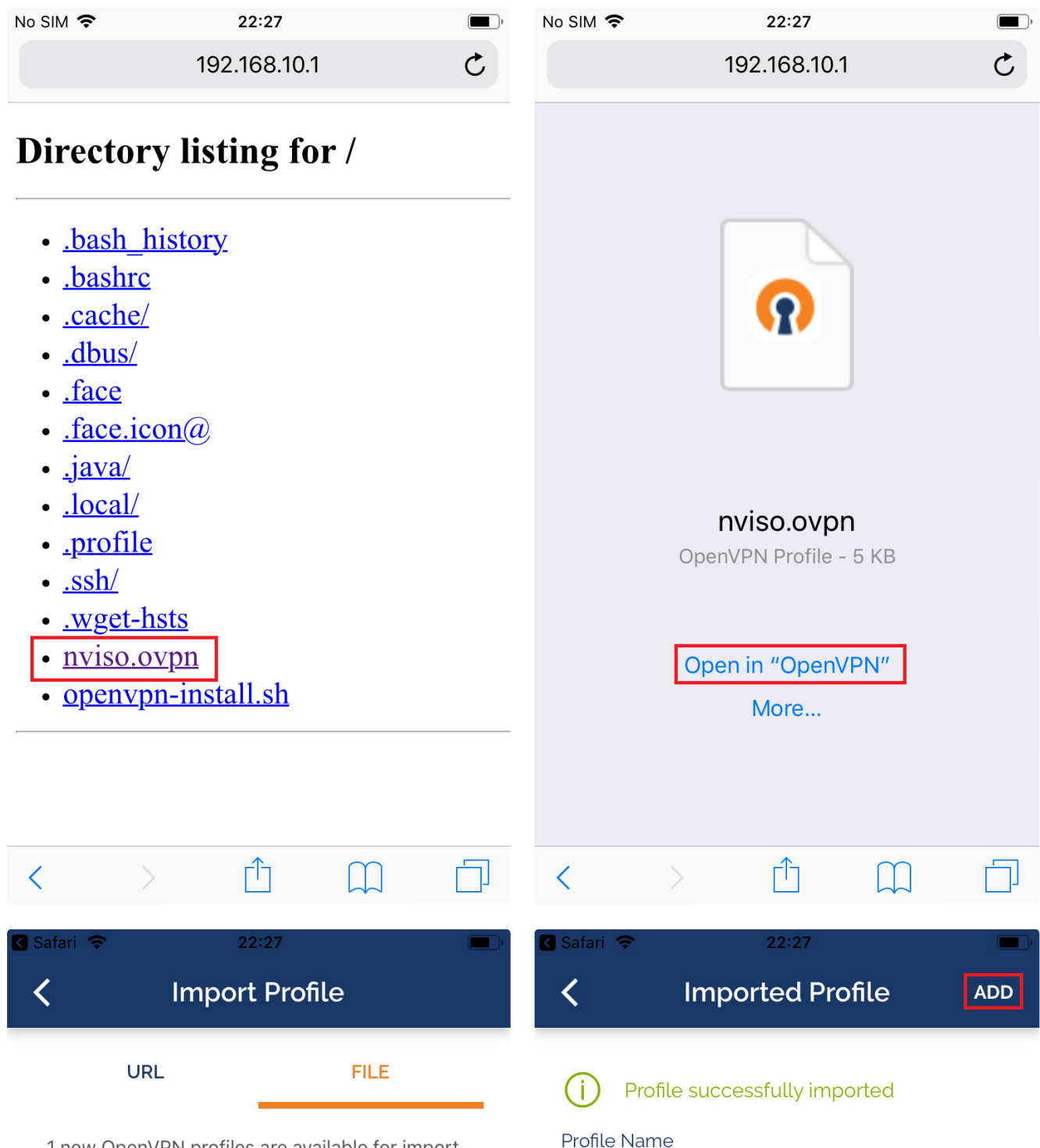
Finally, start the OpenVPN service:

```
1 | sudo service openvpn start
```

Install the OpenVPN client on your iPhone and start a python HTTP server to host the OpenVPN configuration:

```
1 | sudo python3 -m http.server 8080 --directory /root/
```

Navigate to <yourip>:8080 on your iPhone in Safari and download the ovpn file. Open the file and follow the steps to add it to the OpenVPN app.



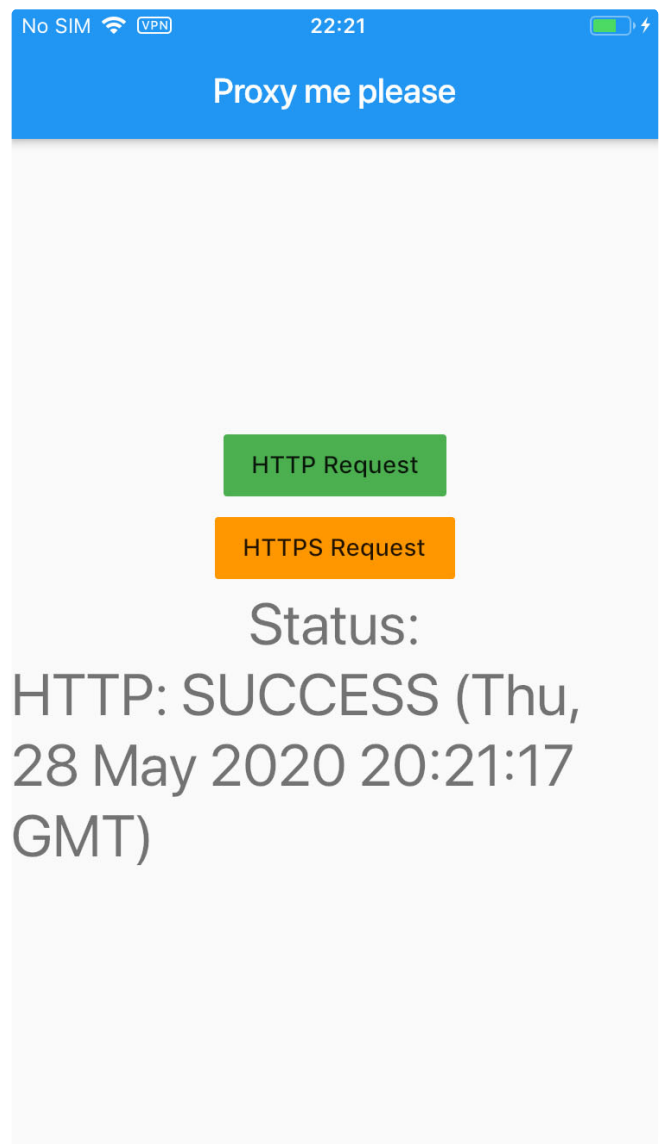
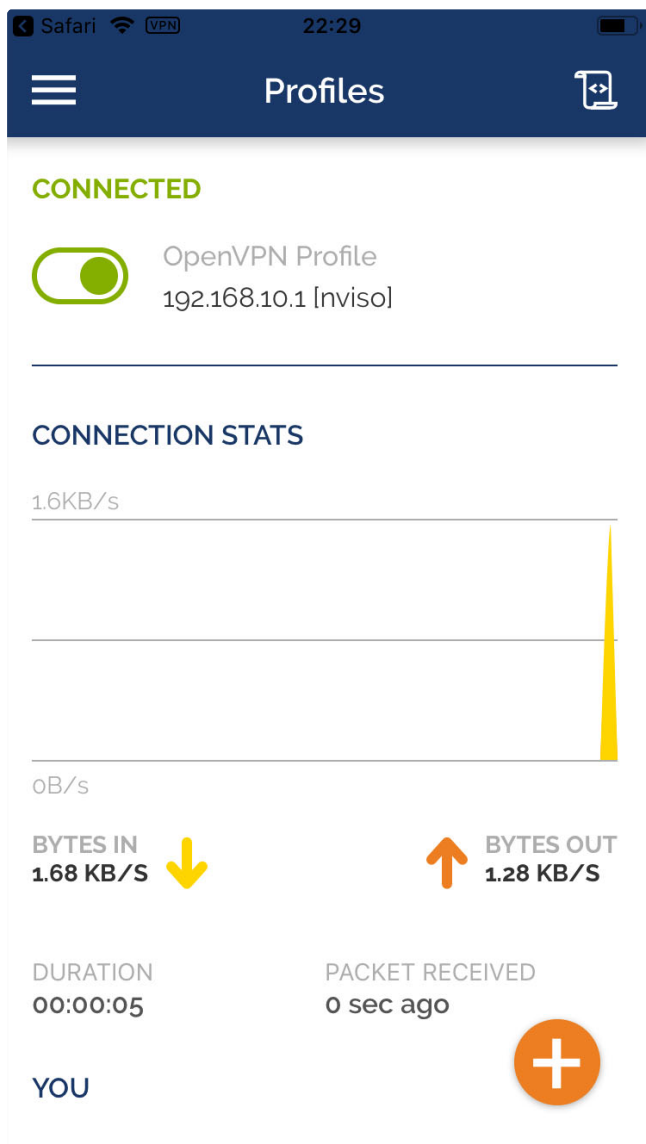
New OpenVPN profiles are available for import

192.168.10.1 [nviso]

Standard Profile

ADD**DELETE**

192.168.10.1 [nviso]

☐ Connect after import

Installing the OpenVPN profile

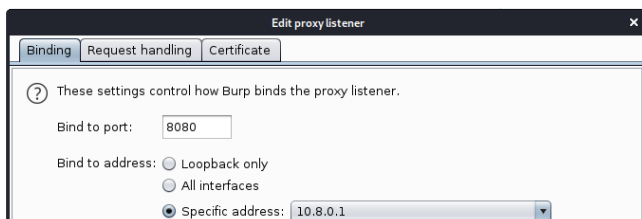
At this point, you should have internet access on the device and see a VPN icon on the top of your screen.

Setting up the MITM

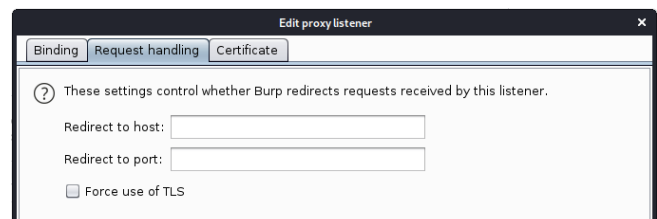
Finally, we need to intercept the traffic when it leaves either the WIFI interface or the OpenVPN interface and before it goes to the eth0 interface. We can do this by using iptables. Modify `192.168.10.0` with the actual IP address where your traffic enters the network.

```
1 # For WIFI: -i wlan0
2 sudo iptables -t nat -A PREROUTING -i wlan0 -p tcp --dport 80 -j REDIRECT --
3 sudo iptables -t nat -A PREROUTING -i wlan0 -p tcp --dport 443 -j REDIRECT -
4 sudo iptables -t nat -A POSTROUTING -s 192.168.10.0/24 -o eth0 -j MASQUERADE
5 # For OpenVPN: -i tun0
6 sudo iptables -t nat -A PREROUTING -i tun0 -p tcp --dport 80 -j REDIRECT --t
7 sudo iptables -t nat -A PREROUTING -i tun0 -p tcp --dport 443 -j REDIRECT --
8 sudo iptables -t nat -A POSTROUTING -s 192.168.10.0/24 -o eth0 -j MASQUERADE
```

Next, start up Burp, enable a listener on port 8080 on either `10.8.0.1` or `192.168.10.1` (or ‘all interfaces’) and enable ‘Invisible proxy’ mode:



Listen on the OpenVPN interface



Enable invisible proxying

At this point, the HTTP traffic is intercepted, from both Safari and the Flutter test app.

Disable SSL verification and intercept HTTPS traffic

Now that we have a MITM on the HTTP traffic, it's time to do the same for HTTPS.

Unfortunately, Flutter doesn't use any of iOS's default libraries so the standard approach of Objection or SSLKillSwitch won't work. Flutter apps use the BoringSSL library to create TLS connections, and those are the methods we need to hook or modify in order to modify the certificate validation logic. Which method we want to change is explained in more detail in a [previous blogpost](#), so be sure to read that for background information. To be able to intercept HTTPS we will need to:

- Get the Flutter binary in a decrypted form
- Find the correct method to hook

- Write a Frida script to modify behavior.

Let's get started!

Acquire the Flutter binary

First, we need the Flutter framework file from the target app. Depending on how the IPA is installed, you will need to take a different approach, as the IPA may or may not be encrypted. The test app in this case is installed through appintst with a development certificate and is not encrypted. We can therefore extract it using [ipainstaller](#):

```
1 | ipainstaller -b be.nviso.flutterApp
```

Alternatively, if the app was downloaded from the App Store, you should use [Clutch](#). Clutch needs to be built on MacOS and pushed to the device through SCP. The exact instructions can be found on the GitHub page. Once it has been installed, you can use it to create a decrypted IPA file:

```
1 | ./Clutch -d <packagename>
```

After that, you end up with an IPA file that you can copy to your host with SCP and get to the Flutter binary which is located at

<app>/Payload/Runner.app/Frameworks/Flutter.framework/Flutter :

```
1 | # Copy the ipa to the host
2 | scp root@192.168.2.4:~/private/var/mobile/Documents/flutter_app\ \(be.nviso.
3 | # Unzip the ipa file
4 | unzip flutterapp.ipa
5 | # Find the Flutter binary
6 | file Payload/Runner.app/Frameworks/Flutter.framework/Flutter
7 | # Result: Payload/Runner.app/Frameworks/Flutter.framework/Flutter: Mach-O un
```

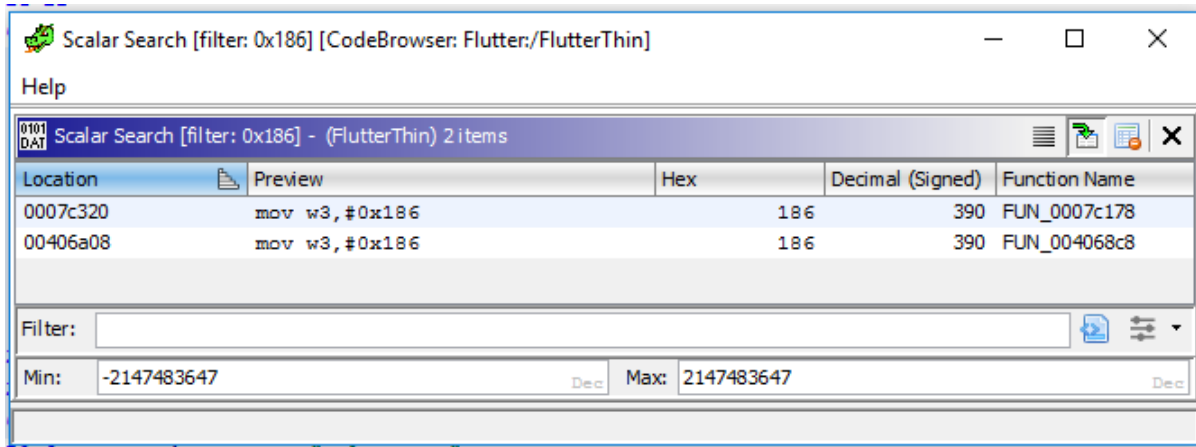
For the test app, you end up with a fat binary, as it still contains 2 architectures. You can extract the arm64 version with lipo on MacOS:

```
1 | lipo -thin arm64 Flutter -output FlutterThin
2 | file FlutterThin
3 | # Result: FlutterThin: Mach-O 64-bit dynamically linked shared library arm64
```

Find the session_verify_cert_chain method

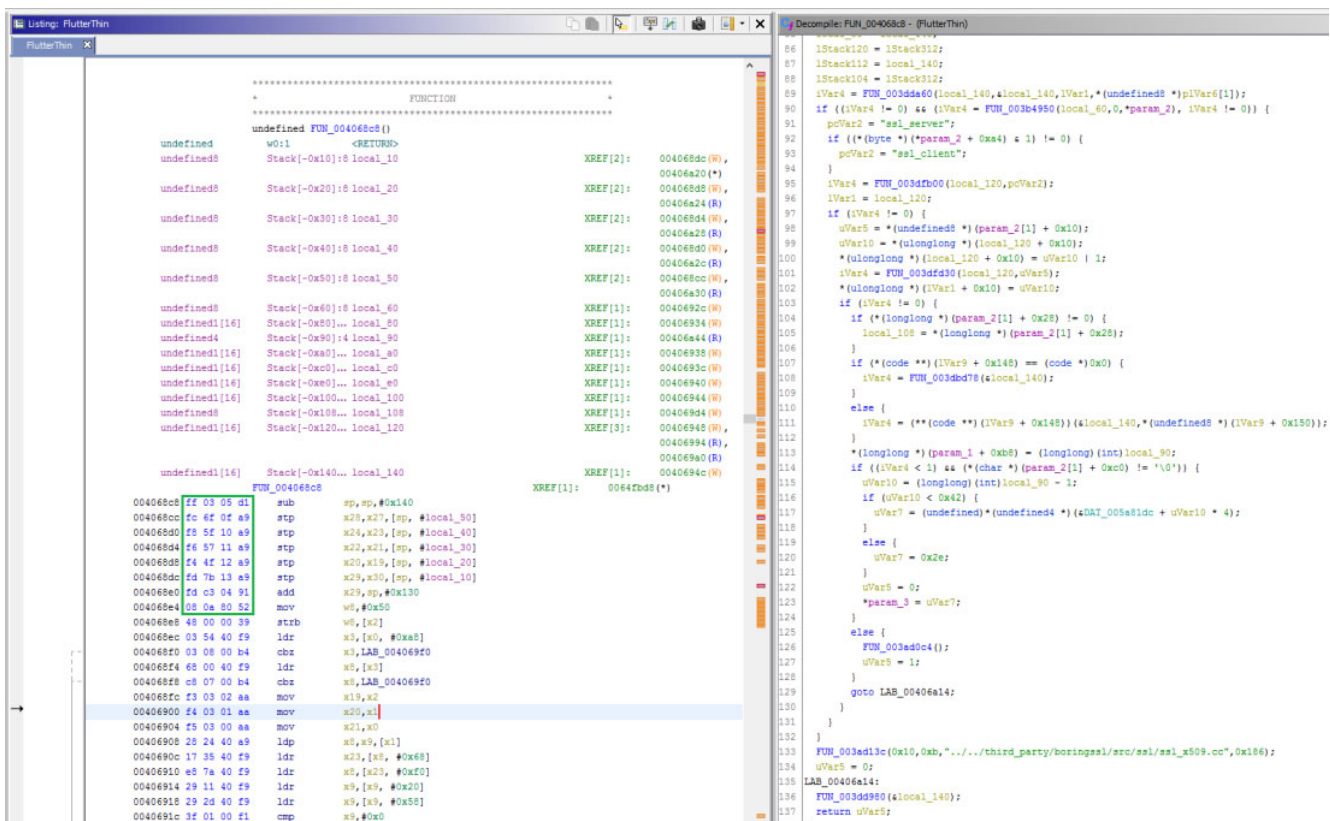
Now that we have the binary, we can identify and patch the method performing the SSL verification in order to for the binary to accept our certificate. As explained in-depth in [my previous blogpost](#), the 'session_verify_cert_chain method' is the one we are looking for. There are two approaches to locate that method in the binary: Search for the [magic number of 0x186](#), or for the x509.cc string. Since there are less references to the magic number than there are to the

x509.cc string, let's take the first approach. Select *Search > For Scalars* and enter 0x186 in the *Specific Scalar* field.



Searching for the magic number 0x186

The correct reference is in *FUN_004068C8*. The decompiled version of this function is very similar to the one identified on Android ARM64, and the x509.cc string is also referenced from here, so we can be pretty sure this is the right function. If you've read the other blog posts, you know I'm a fan of searching for the correct method using a bunch of bytes rather than take the offset directly, so that's what we'll do.



The session_verify_cert_chain method and the method signature

The first bytes of this method are ff 03 05 d1 fc 6f 0f a9 f8 5f 10 a9 f6 57 11 a9 f4 4f 12 a9 fd 7b 13 a9 fd c3 04 91 08 0a 80 52 and we can use binwalk to get the correct offset:

```

1 binwalk -R "\xff\x03\x05\xd1\xfc\x6f\x0f\xa9\xf8\x5f\x10\xa9\xf6\x57\x11\xa9
2 DECIMAL      HEXADECIMAL      DESCRIPTION
3 -----
4 4221128      0x4068C8          Raw signature (\xff\x03\x05\xd1\xfc\x6f\x0f\xa

```

To make sure this is a repeatable process, I farmed some Flutter apps and ran the signature over all of them:

```

1 binwalk -R "\xff\x03\x05\xd1\xfc\x6f\x0f\xa9\xf8\x5f\x10\xa9\xf6\x57\x11\xa
2 Target File: /home/flutter/testapps/anon/Flutter1
3 DECIMAL      HEXADECIMAL      DESCRIPTION
4 -----
5 4221128      0x4068C8          Raw signature (\xff\x03\x05\xd1\xfc\x6f\x0f\x
6 Target File: /home/flutter/testapps/anon/Flutter2
7 DECIMAL      HEXADECIMAL      DESCRIPTION
8 -----
9 Target File: /home/flutter/testapps/anon/Flutter3
10 DECIMAL      HEXADECIMAL      DESCRIPTION
11 -----
12 4247284      0x40CEF4          Raw signature (\xff\x03\x05\xd1\xfc\x6f\x0f\x
13 Target File: /home/flutter/testapps/anon/Flutter4
14 DECIMAL      HEXADECIMAL      DESCRIPTION
15 -----
16 4370908      0x42B1DC          Raw signature (\xff\x03\x05\xd1\xfc\x6f\x0f\x
17 Target File: /home/flutter/testapps/anon/Flutter5
18 DECIMAL      HEXADECIMAL      DESCRIPTION
19 -----
20 4221128      0x4068C8          Raw signature (\xff\x03\x05\xd1\xfc\x6f\x0f\x

```

Four versions are playing nice, but one version doesn't have a match. For this version, most likely an older one, the last four bytes don't match. In this case, you can shorten the signature and use trial & error, or open up Ghidra and find the correct offset manually.

Hook it with Frida

With the correct signature, you can now let Frida search for the correct function to hook. The script is very similar to the one for Android:

```

1 function hook_ssl_verify_result(address)
2 {
3     Interceptor.attach(address, {
4         onEnter: function(args) {
5             console.log("Disabling SSL validation")
6         },
7         onLeave: function(retval)
8         {
9             retval.replace(0x1);
10        }
11    });
12 }
13 function disablePinning()
14 {
15     var pattern = "ff 03 05 d1 fc 6f 0f a9 f8 5f 10 a9 f6 57 11 a9 f4 4f 12
16     Process.enumerateRangesSync('r-x').filter(function (m)

```

```

17     {
18         if (m.file) return m.file.path.indexOf('Flutter') > -1;
19         return false;
20     }).forEach(function (r)
21     {
22         Memory.scanSync(r.base, r.size, pattern).forEach(function (match) {
23             console.log('[+] ssl_verify_result found at: ' + match.address.toSt
24             hook_ssl_verify_result(match.address);
25             });
26     });
27 }
28 setTimeout(disablePinning, 1000)

```

Alternatively, if you know the offset, the following script can be used:

```

1  function hook_ssl_verify_result(address)
2  {
3      Interceptor.attach(address, {
4          onEnter: function(args) {
5              console.log("Disabling SSL validation")
6          },
7          onLeave: function(retval)
8          {
9              retval.replace(0x1);
10         }
11     });
12 }
13 function disablePinning()
14 {
15     var m = Process.findModuleByName("Flutter");
16     hook_ssl_verify_result(m.base.add(0x4068C8))
17 }
18 setTimeout(disablePinning, 1000)

```

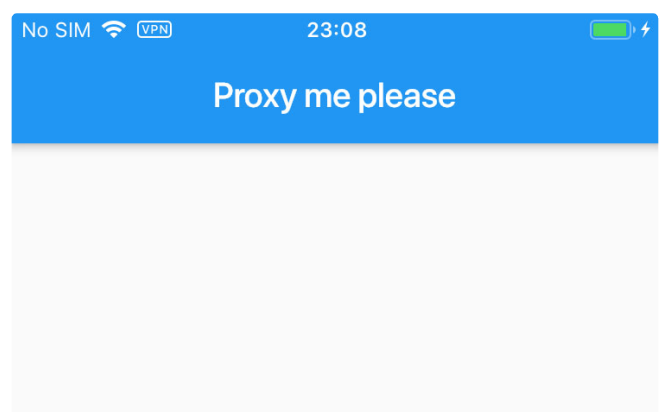
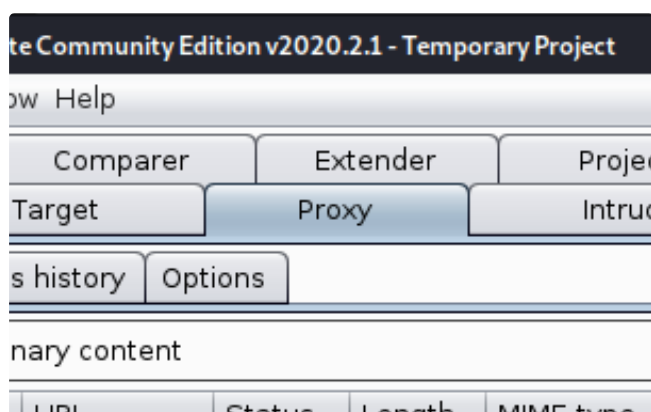
Run it with Frida:

```

1  frida -Uf be.nviso.flutterApp -l disable.js --no-pause
2  ...
3  . . . . . More info at https://www.frida.re/docs/home/
4  Spawned `be.nviso.flutterApp`. Resuming main thread!
5  [iOS Device::be.nviso.flutterApp]-> [+] ssl_verify_result found at: 0x101392
6  Disabling SSL validation

```

And finally, even the HTTPS traffic is intercepted.




```
graph TD; A[HTTP Request] --> B[HTTPS Request]; B --> C[Status: HTTPS: SUCCESS (Mon, 08 Jun 2020 21:08:47 GMT)]
```

The diagram illustrates a request flow. It begins with a green box labeled "HTTP Request". An arrow points from this box to an orange box labeled "HTTPS Request". Another arrow points from the orange box to a large text area that displays the status: "Status: HTTPS: SUCCESS (Mon, 08 Jun 2020 21:08:47 GMT)".

Because much of the reverse-engineering work was already done in my Android blogposts, it was fairly easy to find the correct method in Ghidra. This is one of the rare cases where having a cross-platform framework is actually beneficial to the reverse-engineering process, which is usually not the case. Usually, it's not possible to reuse techniques between platforms; take for example Xamarin, which is interpreted code on Android but native code on iOS, or hybrid applications where the webview communicates with a native layer in either Java/Kotlin or ObjectiveC/Swift.

Like this:

Like



One blogger likes this.

Published by Jeroen Beckers

<https://blog.nviso.eu/2020/06/12/intercepting-flutter-traffic-on-ios/>

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](#)

< Reviewing an ISO 27001 certificate: a checklist

Burp, OAuth2.0 and tons of coding: a testimony of my internship in the penetration testing team at NVISO! >
