NVISO Labs

Cyber security research, straight from the lab!

Intercepting traffic from Android Flutter applications

Jeroen Beckers android, burpsuite, Mobile August 13, 2019 9 Minutes

Update: The explanation below explains the step for ARMv7. For ARMv8 (64bit), see this blogpost.

Flutter is Google's new open source mobile development framework that allows developers to write a single code base and build for Android, iOS, web and desktop. Flutter applications are written in Dart, a language created by Google more than 7 years ago.

It's often necessary to intercept traffic between a mobile application and the backend (either for a security assessment or a bounty hunt), which is typically done by adding Burp as an intercepting proxy. Flutter applications are a little bit more difficult to proxy, but it’s definitely possible.

TL;DR

- Flutter uses Dart, which doesn’t use the system CA store
- Dart uses a list of CA’s that’s compiled into the application
- Dart is not proxy aware on Android, so use ProxyDroid with iptables
- Hook the session_verify_cert_chain function in x509.cc to disable chain validation
- You might be able to use the script at the bottom of this article directly, or you can follow the steps below to get the right bytes or offset.

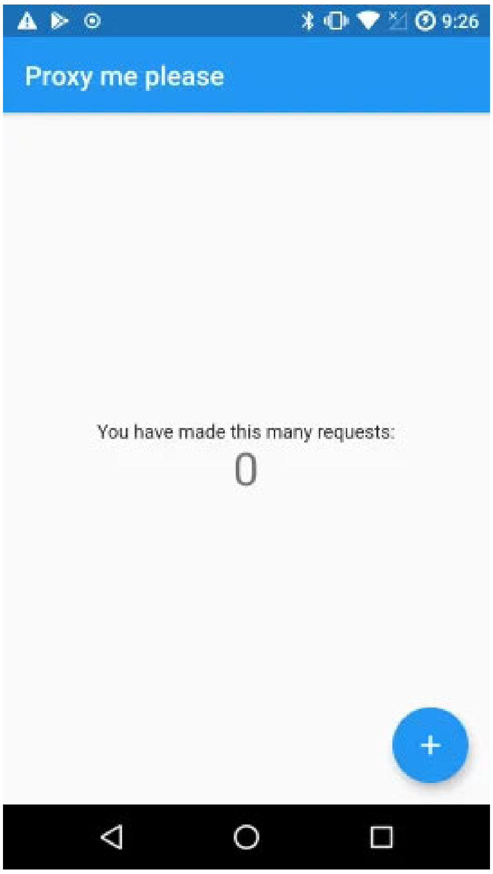
Test setup

In order to perform my tests, I installed the flutter plugin and created a Flutter application that comes with a default interactive button that increments a counter. I modified it to fetch a URL through the HttpClient class:

```
class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;
  HttpClient client;

  _MyHomePageState()
  {
      _start();
  }
  void _start() async
  {
    client = HttpClient();
  }
  void _incrementCounter() {
    setState(() {
      if(client != null)
      {
          client
              .getUrl(Uri.parse('http://www.nviso.eu')) // produces a reque
              .then((request) => request.close()) // sends the request
              .then((response) => print("SUCCESS - " + response.headers.val
          _counter++;
     }
    });
  }
```

The app can be compiled using `flutter build aot` and pushed to the device through `adb install`.



You have made this many requests:
0

Every time we press the button, a call is sent to http://www.nviso.eu and if it’s successful it is printed to the device logs.

On my device I have Frida installed through Magisk-Frida-Server and my Burp certificate is added to the system CA store with the MagiskTrustUserCerts module. Unfortunately, Burp does not see any traffic passing through, even though the app logs indicate that the request was successful.

Sending traffic to the proxy through ProxyDroid/iptables

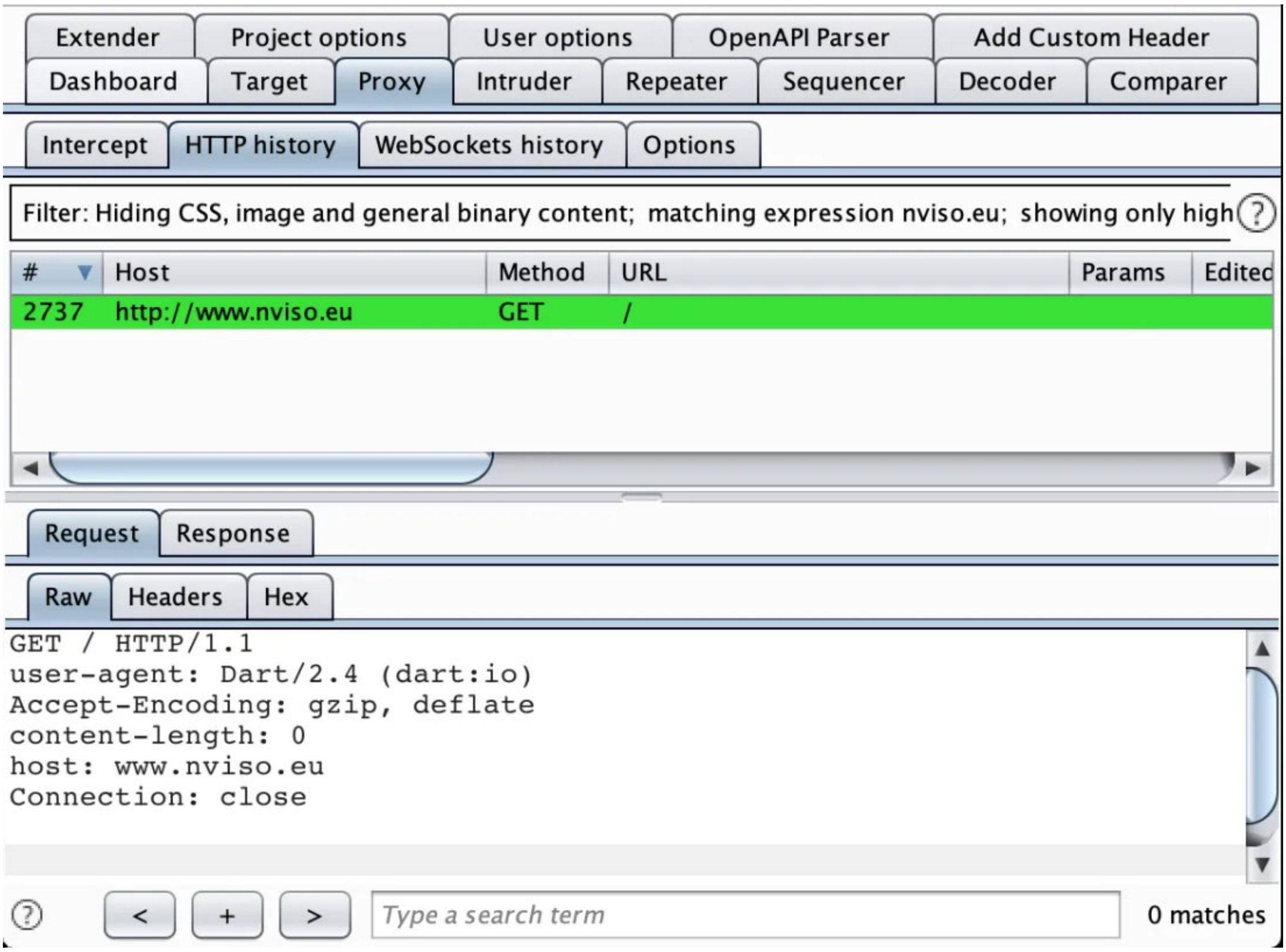
The HttpClient has a findProxy method and its documentation is pretty clear on this: By default all traffic is sent directly to the target server, without taking any proxy settings into account:

> *Sets the function used to resolve the proxy server to be used for opening a HTTP connection to the specified* `url` *. If this function is not set, direct connections will always be used.*
>
> — **findProxy documentation**

The application can set this property to `HttpClient.findProxyFromEnvironment` which searches for specific environment variables such as `http_proxy` and `https_proxy` . Even if the application would be compiled with this implementation, it would be pretty useless on Android since all applications are children of the initial zygote process which does not have these environment variables.

It's also possible to define a custom findProxy implementation that returns the preferred proxy. A quick modification on my test application indeed shows that this configuration sends all HTTP data to my proxy:

```
client.findProxy = (uri) {
    return "PROXY 10.153.103.222:8888";
};
```

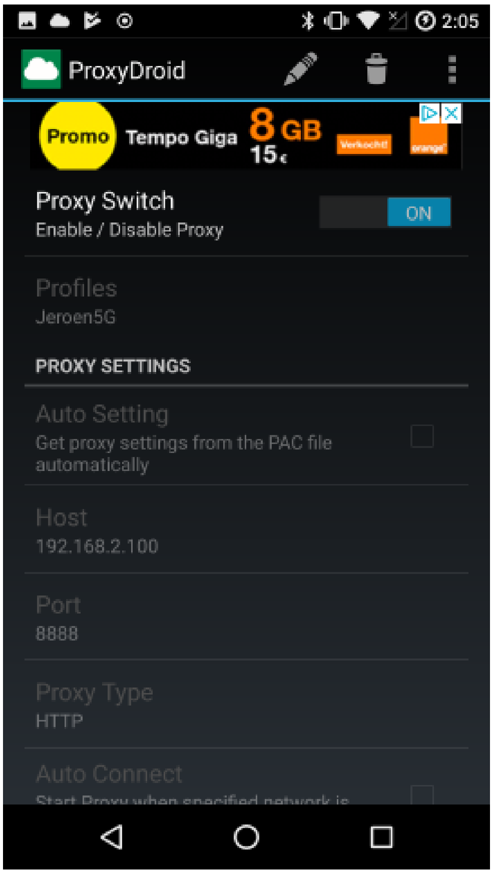


Filter: Hiding CSS, image and general binary content; matching expression nviso.eu; showing only high

#	Host	Method	URL	Params	Edited
2737	http://www.nviso.eu	GET	/		

```
GET / HTTP/1.1
user-agent: Dart/2.4 (dart:io)
Accept-Encoding: gzip, deflate
content-length: 0
host: www.nviso.eu
Connection: close
```

Of course, we can't modify the application during a black-box assessment, so another approach is needed. Luckily, we always have the iptables fallback to route all traffic from the device to our proxy. On a rooted device, ProxyDroid handles this pretty well and we can see all HTTP traffic flowing through Burp.



Proxy Switch
Enable / Disable Proxy
ON

Profiles
Jeroen5G

PROXY SETTINGS

Auto Setting
Get proxy settings from the PAC file automatically
☐

Host
192.168.2.100

Port
8888

Proxy Type
HTTP

Auto Connect
Start Proxy when specified network is
☐

ProxyDroid with root access using iptables

Intercepting HTTPS traffic

This is where it gets more tricky. If I change the URL to HTTPS, Burp complains that the SSL handshake fails. This is weird since my device is set up to include my Burp certificate as a trusted root CA.

After some research, I ended up on a GitHub issue that explains the issue for Windows, but the same is applicable to Android: Dart generates and compiles its own Keystore using Mozilla's NSS library.

This means that we can't bypass SSL validation by adding our proxy CA to the system CA store. To solve this we have to dig into libflutter.so and figure out what we need to patch or hook in order to validate our certificate. Dart uses Google's BoringSSL to handle everything SSL related, and luckily both Dart and BoringSSL are open source.

When sending HTTPS traffic to Burp, the Flutter application actually throws an error, which we can take as a starting point:

```
E/flutter (10371): [ERROR:flutter/runtime/dart_isolate.cc(805)]
Unhandled exception:
 E/flutter (10371): HandshakeException: Handshake error in client
(OS Error:
 E/flutter (10371):  NO_START_LINE(pem_lib.c:631)
 E/flutter (10371):  PEM routines(by_file.c:146)
 E/flutter (10371):  NO_START_LINE(pem_lib.c:631)
 E/flutter (10371):  PEM routines(by_file.c:146)
 E/flutter (10371):  CERTIFICATE_VERIFY_FAILED: self signed
certificate in certificate chain(handshake.cc:352))
 E/flutter (10371): #0      _rootHandleUncaughtError.
(dart:async/zone.dart:1112:29)
 E/flutter (10371): #1      _microtaskLoop
(dart:async/schedule_microtask.dart:41:21)
 E/flutter (10371): #2      _startMicrotaskLoop
(dart:async/schedule_microtask.dart:50:5)
 E/flutter (10371): #3      _runPendingImmediateCallback
(dart:isolate-patch/isolate_patch.dart:116:13)
 E/flutter (10371): #4      _RawReceivePortImpl._handleMessage
(dart:isolate-patch/isolate_patch.dart:173:5)
```

The first thing we need to do is find this error in the BoringSSL library. The error actually shows us where the error is triggered: `handshake.cc:352`. Handshake.cc is indeed part of the BoringSSL library and does contain logic to perform certificate validation. The code at line 352 is shown below, and this is most likely the error we are seeing. The line numbers don't match exactly, but this is most likely the result of a version difference.

```
352 if (ret == ssl_verify_invalid) {
353     OPENSSL_PUT_ERROR(SSL, SSL_R_CERTIFICATE_VERIFY_FAILED);
354     ssl_send_alert(ssl, SSL3_AL_FATAL, alert);
355   }
```

This is part of the ssl_verify_peer_cert function which returns the ssl_verify_result_t enum which is defined in ssl.h at line 2290:

```
2290 enum ssl_verify_result_t BORINGSSL_ENUM_INT {
2291   ssl_verify_ok,
2292   ssl_verify_invalid,
2293   ssl_verify_retry,
2294 };
```

If we can change the return value of ssl_verify_peer_cert to ssl_verify_ok (=0), we should be good to go. However, a lot of stuff is going on in this method, and Frida can only (easily) change the return value of a function. If we change this value, it would still fail due to the ssl_send_alert() function call above (trust me, I tried 🙂).

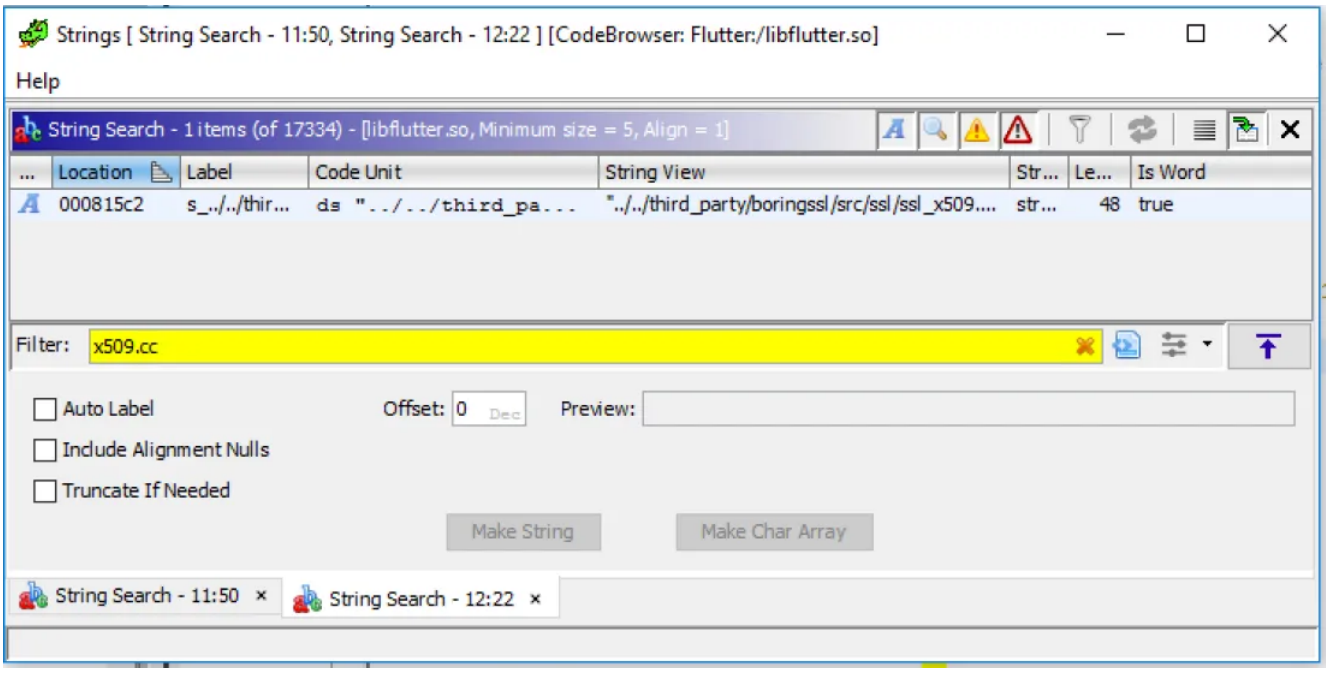
Let’s find a better method to hook. Right above the snippet from handshake.cc is the following code, which is the actual part of the method that is validating the chain:

```
ret = ssl->ctx->x509_method->session_verify_cert_chain(
              hs->new_session.get(), hs, &alert)
              ? ssl_verify_ok
              : ssl_verify_invalid;
```

The session_verify_cert_chain function is defined in ssl_x509.cc at line 362. This function also returns a primitive datatype (boolean) and is a better candidate to hook. If a check fails in this function, it only reports the issue via OPENSSL_PUT_ERROR, but it doesn’t have side effects like the ssl_verify_peer_cert function. The OPENSSL_PUT_ERROR is a macro defined in err.h at line 418 that includes the source filename. This is the same macro that was used for the error that made it to the Flutter app.

```
418 #define OPENSSL_PUT_ERROR(library, reason) \
419   ERR_put_error(ERR_LIB_##library, 0, reason, __FILE__, __LINE__)
```

Now that we know which function we want to hook, we need to find it in libflutter.so. The OPENSSL_PUT_ERROR macro is called a few times in the session_verify_cert_chain function, which makes it easy to find the correct method using Ghidra. So import the library into Ghidra, use Search -> Find Strings and search for `x509.cc` .



Strings [String Search - 11:50, String Search - 12:22] [CodeBrowser: Flutter:/libflutter.so]

String Search - 1 items (of 17334) - [libflutter.so, Minimum size = 5, Align = 1]

...	Location	Label	Code Unit	String View	Str...	Le...	Is Word
	000815c2	s_../../thir...	ds "../../third_pa...	"../../third_party/boringssl/src/ssl/ssl_x509....	str...	48	true

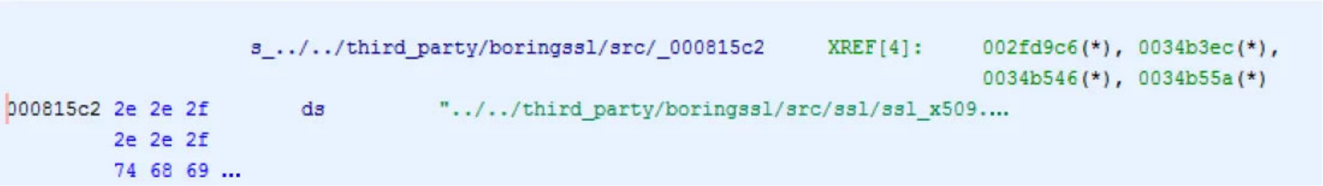
Filter: x509.cc

☐ Auto Label
☐ Include Alignment Nulls
☐ Truncate If Needed

Offset: 0

Searching for the x509.cc string

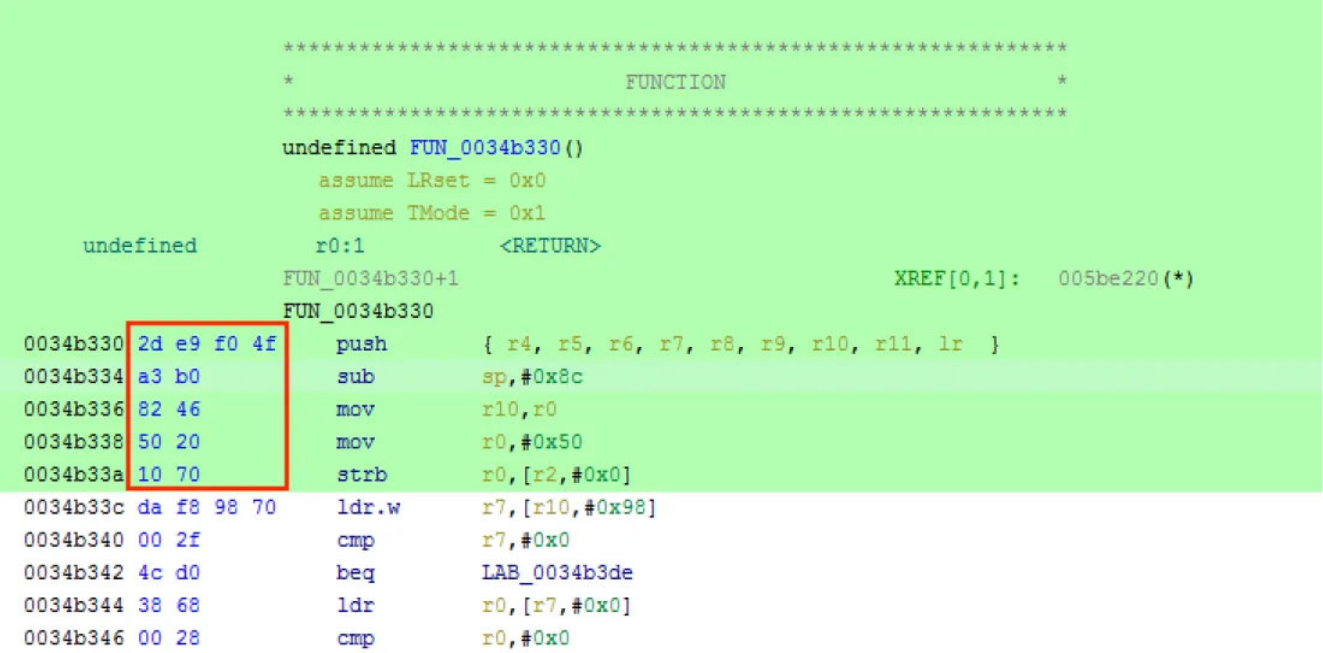
There are only 4 XREFs so it’s easy to go over them and find one that looks like the session_verify_cert_chain function:



```
                 s_../../third_party/boringssl/src/_000815c2     XREF[4]:     002fd9c6(*), 0034b3ec(*),
                                                                              0034b546(*), 0034b55a(*)
000815c2 2e 2e 2f        ds          "../../third_party/boringssl/src/ssl/ssl_x509....
         2e 2e 2f
         74 68 69 ...
```

Only 4 xrefs

One of the functions takes 2 ints, 1 'undefined' and contains a single call to OPENSSL_PUT_ERROR (`FUN_00316500`). In my version of libflutter.so, this is `FUN_0034b330`. What you typically do now is calculate the offset of this function from one of the exported functions and hook it. I usually take a lazy approach where I copy the first 10 or so bytes of the function and check how often that pattern occurs. If it only occurs once, I know I found the function and I can hook it. This is useful because I can often use the same script for different versions of the library. With an offset based approach, this is more difficult.



```
                      **************************************************************
                      *                          FUNCTION                          *
                      **************************************************************
                      undefined FUN_0034b330()
                         assume LRset = 0x0
                         assume TMode = 0x1
     undefined           r0:1          <RETURN>
                      FUN_0034b330+1                                  XREF[0,1]:   005be220(*)
                      FUN_0034b330
0034b330 2d e9 f0 4f     push        { r4, r5, r6, r7, r8, r9, r10, r11, lr  }
0034b334 a3 b0           sub         sp,#0x8c
0034b336 82 46           mov         r10,r0
0034b338 50 20           mov         r0,#0x50
0034b33a 10 70           strb        r0,[r2,#0x0]
0034b33c da f8 98 70     ldr.w       r7,[r10,#0x98]
0034b340 00 2f           cmp         r7,#0x0
0034b342 4c d0           beq         LAB_0034b3de
0034b344 38 68           ldr         r0,[r7,#0x0]
0034b346 00 28           cmp         r0,#0x0
```

So now we let Frida search the `libflutter.so` library for this pattern:

```javascript
var m = Process.findModuleByName("libflutter.so");
var pattern = "2d e9 f0 4f a3 b0 82 46 50 20 10 70"
var res = Memory.scan(m.base, m.size, pattern, {
  onMatch: function(address, size){
      console.log('[+] ssl_verify_result found at: ' + address.toString());
    },
  onError: function(reason){
      console.log('[!] There was an error scanning memory');
    },
    onComplete: function()
    {
      console.log("All done")
    }
  });
```

Running this script on my Flutter application gives just a single result:

```
(env) ~/D/Temp » frida -U -f be.nviso.flutter_app -l frida.js --no-pause
[LGE Nexus 5::be.nviso.flutter_app]-> [+] ssl_verify_result found at: 0x9a7f7040
All done
```

Now we just need to use the Interceptor to change the return value to 1 (true):

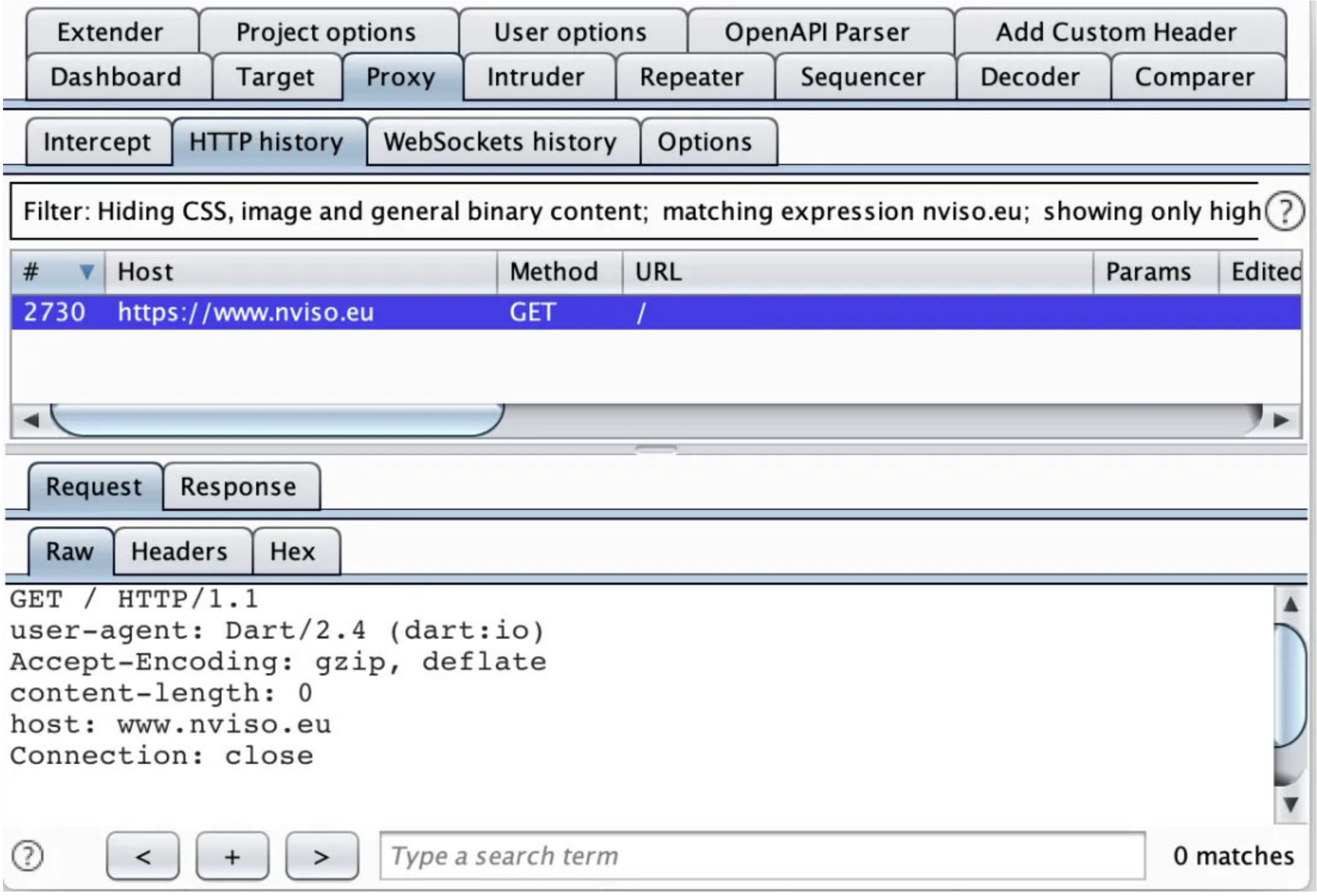
```
function hook_ssl_verify_result(address)
{
  Interceptor.attach(address, {
    onEnter: function(args) {
      console.log("Disabling SSL validation")
    },
    onLeave: function(retval)
    {
      console.log("Retval: " + retval)
      retval.replace(0x1);

    }
  });
}
function disablePinning()
{
 var m = Process.findModuleByName("libflutter.so");
 var pattern = "2d e9 f0 4f a3 b0 82 46 50 20 10 70"
 var res = Memory.scan(m.base, m.size, pattern, {
  onMatch: function(address, size){
      console.log('[+] ssl_verify_result found at: ' + address.toString());

      // Add 0x01 because it's a THUMB function
      // Otherwise, we would get 'Error: unable to intercept function at 0x
      hook_ssl_verify_result(address.add(0x01));

    },
  onError: function(reason){
      console.log('[!] There was an error scanning memory');
    },
    onComplete: function()
    {
      console.log("All done")
    }
  });
}
setTimeout(disablePinning, 1000)
```

After setting up ProxyDroid and launching the application with this script, we can now finally see HTTPs traffic:



Filter: Hiding CSS, image and general binary content; matching expression nviso.eu; showing only high

#	Host	Method	URL	Params	Edited
2730	https://www.nviso.eu	GET	/		

```
GET / HTTP/1.1
user-agent: Dart/2.4 (dart:io)
Accept-Encoding: gzip, deflate
content-length: 0
host: www.nviso.eu
Connection: close
```

I've tested this on a few Flutter apps and this approach worked on all of them. As the BoringSSL library will most likely stay rather stable, this approach might work for some time to come.

Disable SSL Pinning (SecurityContext)

Finally, let's see how we can get around SSL Pinning. One way of doing this is by defining a new SecurityContext that contains specific certificates. While this is not technically SSL pinning (you don't protect against a compromised private key), it's often implemented to prevent against easy eavesdropping of the communication channel.

For my app, I added the following code to have it accept only my burp certificate. The SecurityContext constructor takes one argument, `withTrustedRoots`, which defaults to false.

```
ByteData data = await rootBundle.load('certs/burp.crt');
    SecurityContext context = new SecurityContext();
    context.setTrustedCertificatesBytes(data.buffer.asUint8List());
    client = HttpClient(context: context);
```

The application will now automatically accept our Burp proxy as the certificate for any website, which shows that this method can be used to specify a specific certificate that the application must comply to. If we now switch this to the nviso.eu certificate, we can no longer intercept the connection.

Fortunately, the Frida script listed above already bypasses this kind of root-ca-pinning implementation, as the underlying logic still depends on the same methods of the BoringSSL library.

Disable SSL Pinning (ssl_pinning_plugin)

One of the ways Flutter developers might want to perform ssl pinning is through the ssl_pinning_plugin flutter plugin. This plugin is actually designed to send one HTTPS connection and verify the certificate, after which the developer will trust the channel and perform non-pinned HTTPS requests:

With correct timing of ProxyDroid, this can already be circumvented, but let's just disable it anyway.

```
void testPin() async
  {
    List<String> hashes = new List<String>();
    hashes.add("randomhash");
    try
    {
      await SslPinningPlugin.check(serverURL: "https://www.nviso.eu", heade

      doImportanStuff()
    }catch(e)
    {
      abortWithError(e);
    }
  }
```

The plugin is a bridge to a Java implementation which we can easily hook with Frida:

```
function disablePinning()
{
    var SslPinningPlugin = Java.use("com.macif.plugin.sslpinningplugin.SslP
    SslPinningPlugin.checkConnexion.implementation = function()
    {
        console.log("Disabled SslPinningPlugin");
        return true;
    }
}

Java.perform(disablePinning)
```

Conclusion

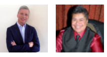
This was a pretty fun ride, and it went quite smoothly since both Dart and BoringSSL are open source. Due to just a few interesting strings, it's pretty easy to find the correct place to disable the ssl verification logic, even without any symbols. My approach with scanning for the function prologue might not always work, but since BoringSSL is pretty stable, it should work for some time to come.

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



2 bloggers like this.

Tagged: andoid, 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

‹ Solving Flaggy Bird (Google CTF 2019)

Extracting Certificates From the Windows Registry ›