DARRYN CAMPBELL

Mobile computing and enterprise software development

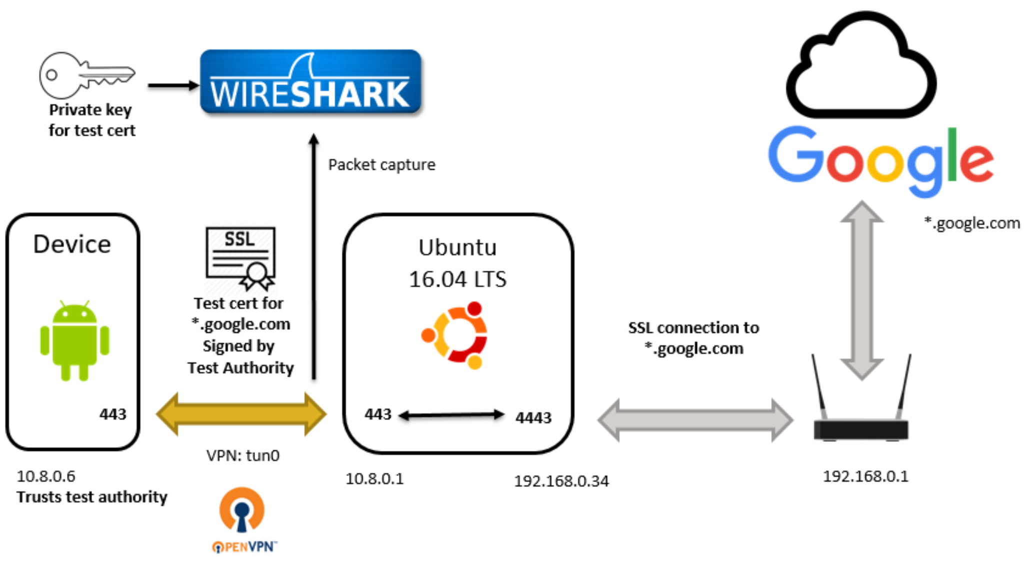
Android traffic analysis to Google servers – methodology

29th December 2017 0 By DARRYNCAMPBELL

I am working on a separate project to understand what sort of traffic is going back and forth between Android devices and Google's servers. That is still a work in progress but I thought it worth documenting the test methodology here, though I won't be including any results just yet.

I am drawing heavily on this blog written about 5 years ago: http://www.myhowto.org/java/81-intercepting-and-decrypting-ssl-communications-between-android-phone-and-3rd-party-server/ that goes into detail on how to intercept and decrypt SSL communications between an Android phone and a 3rd party server in general, it's a great blog and I would second the author's insistence that the main requirement to achieve this is patience!

What we are looking to do is something like the following:



So, running down the various component pre-requisites:

Linux computer:

- I used Ubuntu 16.04 LTS because it's what I happened to have laying around on a spare laptop
- OpenSSL (1.0.2g) (https://www.openssl.org/)
- SoCat (http://www.dest-unreach.org/socat/)
- Java JDK version 1.8.0_72-b15 (just what happened to already be on the machine)
- The blog I was following also said Bouncy Castle was required "*Bouncy Castle library (http://bouncycastle.org/download/bcprov-jdk16-141.jar) – place this JAR file into your …/jre/lib/ext directory*", I don't know if this is still required with the move away from Bouncy Castle on the client but I followed the step anyway.
- The following two commands were run on the Linux box to enable IP forwarding and configure NAT:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
sudo /sbin/iptables -t nat -A POSTROUTING -o wlan0 -j MASQUERADE
```

Note that the iptables command must be executed following each boot

Note that wlan0 is the name of my Linux host's wireless card.

Device traffic:

- We want all the traffic from our Android device to be routed through our Linux host. There are a number of ways to achieve this, particularly if you are just testing WiFi traffic as you could specify the default gateway to be the Linux box or you could expose a wireless hotspot on the host and share the external connection.
- I chose to establish a VPN connection between the Android device and the Linux host, primarily because this methodology would also take account of WAN traffic but also because it is a more thoroughly documented setup if I ran into trouble later.
- I chose OpenVPN as my VPN provider as it was free, appeared legitimate and supported Ubuntu 16.04 with a great step by step guide to set the thing up.
    - You can download OpenVPN from the Play Store or apkmirror.com.
    - In retrospect I wish I had chosen a VPN provider whose Android client was open source and I could build myself. I wanted to test on a device ***without*** a Play account registered but obviously you can only download applications from the Play Store after you have signed into Google. Downloading the apk file from apkmirror worked perfectly fine for this purpose but personally I just have some hangups about the site.
- I can't add anything to OpenVPN's step by step guide (I followed all the steps, including the optional steps). Once you get success with the following command you are all set:

```
sudo systemctl enable openvpn@server
```

Once configured the OpenVPN server will persist across a host reboot.

Wireshark:

- I used Wireshark because I am most familiar with it and it is capable of decrypting SSL traffic (as long as it knows the private key)
- I'm sure the version does not matter but I am using version 2.2.6 on the Linux box to capture the traffic on tun0 interface (the tunnel created when I configured the VPN)
- The following options are handy to determine Google bound traffic (along with the filter "ip.host contains google"):
    - View –> Name Resolution –> Resolve Network Addresses
    - View –> Name Resolution –> Resolve Transport Addresses
    - Remember however that not all Google domains contain the word 'google'.

It is worth checking at this stage that your test device can communicate with an external server, although the traffic on the wire will still be encrypted at this stage you should at least see something in Wireshark and you should be able to use Chrome to browse to https://www.google.com.

Certificates:

In short, the idea here is to:

- Define our own custom certificate authority (CA)
- Get our test device to trust that CA
- Create a test certificate for *.google.com
    - That is the CN for the certificate, it also covers *.google.co.uk and many others, see the SAN section below.
- Sign that test certificate with our custom CA
- When the device tries to connect to a Google owned server we present the device with our own certificate and because the device trusts our CA it trusts our test certificate for Google.com. Since we know the private key for the test certificate (we created it) we can give that key to Wireshark to see the data being transmitted.

Define a custom certificate authority (CA)

```
//  Generate a new key for the CA
openssl genrsa -des3 -out ca.key 4096
<Enter passphrase>:  0000
<Verify passphrase>: 0000
//  Generate the CA certificate
openssl req -new -x509 -days 365 -key ca.key -out ca.crt
<Enter passphrase>: 0000
<Country Name>:     US
<State>:            .
<Locality>:         .
<Organization>:     Darryn Organization
<Unit Name>:
<Common Name>:      Darryn Authority
<Email Address>:
```

This gives us a ca.crt file which we need to instruct our test device to trust. The afore mentioned blog goes into detail of how this can be done on a rooted device but for the purposes of my testing I am using a device where you can define your own trusted CAs so this is not an issue. Just remember to reboot the device after installation.

Generate the test certificate for Google.com

Note this is mimics the certificate served for the *.google.com common name (CN). There is a different certificate for *.googleapis.com and whilst I have not yet tried it, the process for generating a test certificate for that domain would be very similar.

```
//  Generate a server key
openssl genrsa -des3 -out server.key 4096
<Enter passphrase>:  0000
<Verify passphrase>: 0000
//  Generate a certificate signing request
openssl req -new -key server.key -out server.csr
<Enter passphrase>:  0000
<Country Name>:     US
<State>:            California
<Locality>:         Mountain View
```

```
<Organization>:     Google Inc
<Unit Name>:
<Common Name>:      *.google.com
<Email Address>:
<Challenge pw>:
<Opt Company Name>:
```

To add the subject alternative name field we need to first create a configuration file as follows (called google.cnf). Presumably the line endings need to be \n but I'm not sure:

```
[ req ]
req_extensions     = v3_req
distinguished_name = req_distinguished_name

[ req_distinguished_name ]
C = US
S = California
L = Mountain View
O = Google Inc
CN = *.google.com

[ v3_req ]
subjectAltName = @alt_names
keyUsage = keyEncipherment, dataEncipherment
extendedKeyUsage = serverAuth

[alt_names]
DNS.1   = *.google.com
DNS.2   = *.android.com
DNS.3   = *.appengine.google.com
DNS.4   = *.cloud.google.com
DNS.5   = *.db833953.google.cn
DNS.6   = *.g.co
DNS.7   = *.gcp.gvt2.com
DNS.8   = *.google-analytics.com
DNS.9   = *.google.ca
DNS.10   = *.google.cl
DNS.11   = *.google.co.in
DNS.12   = *.google.co.jp
DNS.13   = *.google.co.uk
DNS.14   = *.google.com.ar
DNS.15   = *.google.com.au
DNS.16   = *.google.com.br
DNS.17   = *.google.com.co
DNS.18   = *.google.com.mx
DNS.19   = *.google.com.tr
DNS.20   = *.google.com.vn
DNS.21   = *.google.de
DNS.22   = *.google.es
DNS.23   = *.google.fr
DNS.24   = *.google.hu
DNS.25   = *.google.it
DNS.26   = *.google.nl
DNS.27   = *.google.pl
DNS.28   = *.google.pt
DNS.29   = *.googleadapis.com
DNS.30   = *.googleapis.cn
DNS.31   = *.googlecommerce.com
DNS.32   = *.googlevideo.com
DNS.33   = *.gstatic.cn
DNS.34   = *.gstatic.com
DNS.35   = *.gvt1.com
DNS.36   = *.gvt2.com
DNS.37   = *.metric.gstatic.com
DNS.38   = *.urchin.com
DNS.39   = *.url.google.com
DNS.40   = *.youtube-nocookie.com
DNS.41   = *.youtube.com
DNS.42   = *.youtubeeducation.com
DNS.43   = *.yt.be
DNS.44   = *.ytimg.com
DNS.45   = android.clients.google.com
DNS.46   = android.com
DNS.47   = developer.android.google.cn
DNS.48   = developers.android.google.cn
DNS.49   = g.co
DNS.50   = goo.gl
DNS.51   = google-analytics.com
DNS.52   = google.com
DNS.53   = googlecommerce.com
DNS.54   = source.android.google.cn
DNS.55   = urchin.com
DNS.56   = www.goo.gl
DNS.57   = youtu.be
DNS.58   = youtube.com
DNS.59   = youtubeeducation.com
DNS.60   = yt.be
```

Now we can sign our test server certificate with our own certificate authority

```
openssl x509 -req -days 365 -in server.csr -CA ca.crt -CAkey ca.key -set_serial 01 -out server.crt -extfile google.cnf -extensions v3_req
<Enter passphrase>:  0000
//  Remove the password from the key, this enables WireShark to use the key (I think, at any rate it stops you being asked for the password elsewhere)
openssl rsa -in server.key -out server.key.insecure
<Enter passphrase>:  0000
mv server.key server.key.secure
mv server.key.insecure server.key
```

Conducting the Tests

All the pre-requisite steps should now be complete; in order to test the setup redirect all SSL traffic (443) to port 4443:

```
sudo iptables -t nat -A PREROUTING -p tcp --dport 443 -j REDIRECT --to-port 4443
```

This iptables command must be repeated following each reboot of the Linux host

And use Socat to:

- Listen for traffic on port 4443
- Serve our test certificate back to the test device
- Forward requests on to the requested Google server

```
sudo socat -v OPENSSL-
LISTEN:4443,reuseaddr,verify=0,cert=server.crt,key=server.key,cafile=ca.crt,debug,f
OPENSSL:google.com:443
```

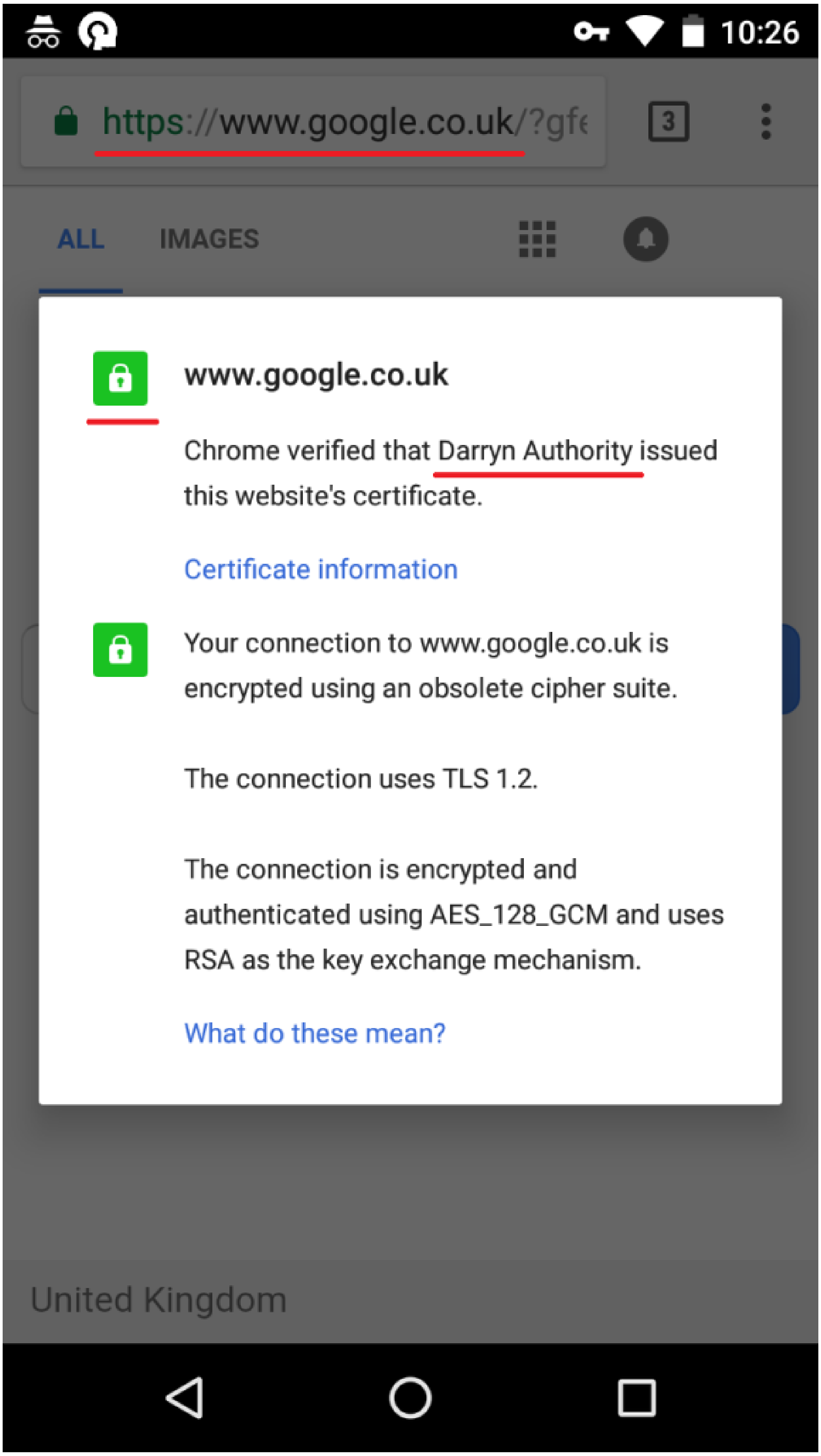
In order for the above line to work you need to be in the same directory as you placed the certificate files generated in the previous step.

That Socat command achieves everything in a single line but you may see other blogs split it into two separate lines for:

- Encrypted –> Plain text

- Plain text –> Encrypted text

As a final test that your setup is correct you should be able to browse to https://www.google.com and be served with a page showing the green lock symbol but on inspection the site certificate has been trusted with our custom certificate authority.



www.google.co.uk

Chrome verified that Darryn Authority issued this website's certificate.

Certificate information

Your connection to www.google.co.uk is encrypted using an obsolete cipher suite.

The connection uses TLS 1.2.

The connection is encrypted and authenticated using AES_128_GCM and uses RSA as the key exchange mechanism.

What do these mean?

In Wireshark:

- Edit –> Preferences –> Protocols –> SSL
- RSA keys list –> Edit…
- Specify an option for SSL Decrypt as follows:
    - IP address: 216.58.*.* (Google use **many** different IP addresses, your mileage may vary but change this value if you see encrypted traffic to a Google server on another IP address)
    - Port: 443
    - Protocol: http
    - Key file: The server.key you created earlier, that goes with your test certificate for *.google.com

QUIC

To read the data going back and forth from Chrome you will need to disable the QUIC protocol (there is a Chrome://flags option for that called 'Experimental QUIC protocol'). QUIC is encrypted and cannot be decrypted using Wireshark so it is easier to just turn it off to verify the transmission contents. If you are seeing a lot of encrypted UDP packets then the most likely culprit is that the device is communicating with the QUIC protocol.

Related

Push Messaging on AOSP Devices - Pushy.me
23rd February 2016
In "Zebra Technologies"

Keeping your application running when the device wants to sleep – updated for Android 10
11th October 2019
In "Software"

EVENT CANCELLED: Droidcon Italy 2020: Indoor Locationing. It should be easy, right?
11th February 2020
In "Speaking"

Category *Software*

Tags *Android* *traffic analysis* *vpn* *wireshark*

Leave a Reply