Android Hooking and SSLPinning using Objection Framework

December 17, 2020 By Raj Chandel

Introduction

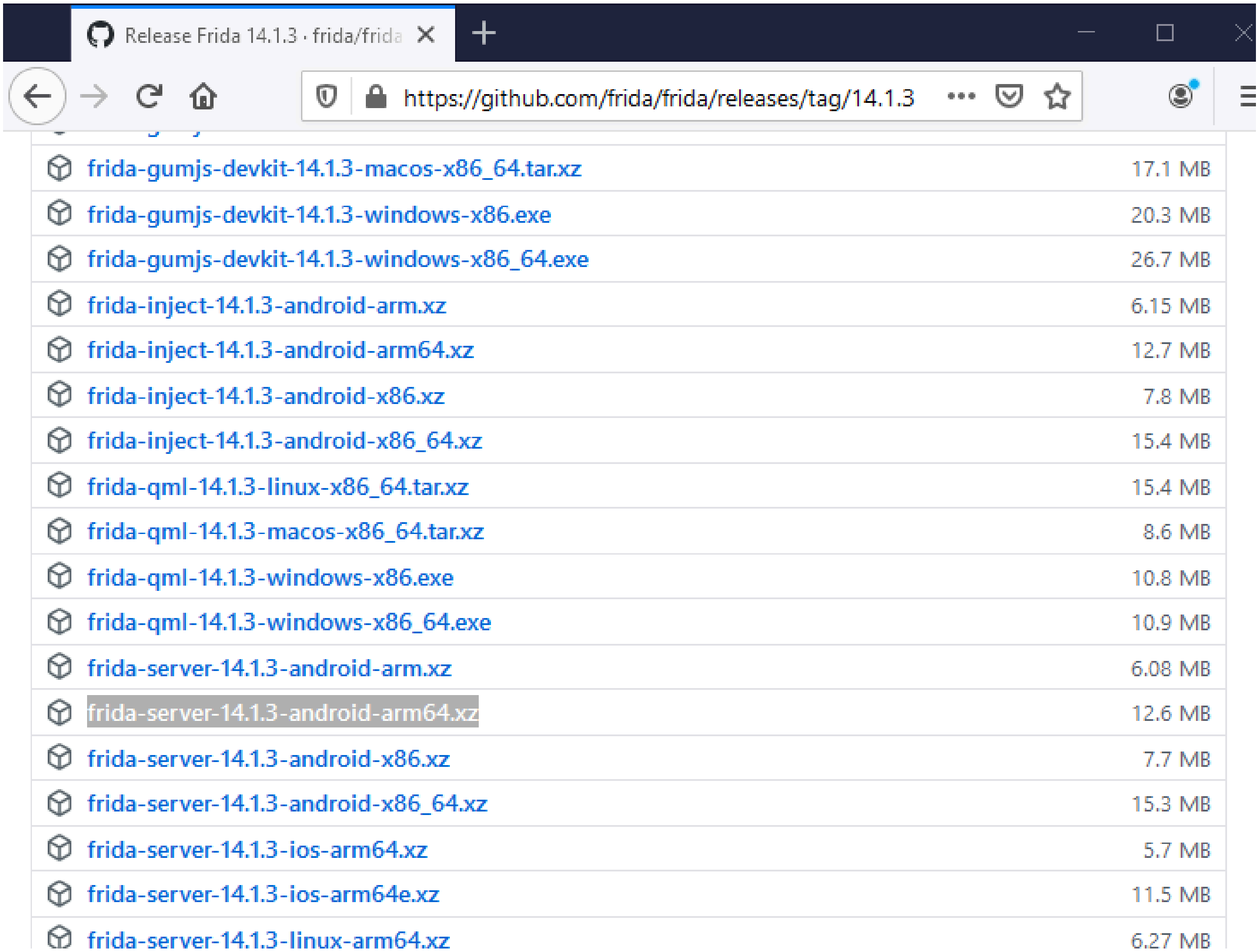
Objection is runtime mobile exploration toolkit built on top of frida which is used in Android and iOS pentesting. We can use Objection to perform numerous functions like **SSLPinning bypass, root detection bypass, performing memory tasks, heap tasks and more without even being root/ jailbroken.** However, it is to be noted that to take full advantage of all the functions it is recommended for the device to be root. In this article, some of the functions demonstrated will require root and some may not. It is recommended to test the application on a root device. Let's begin.

Table of Content

Setting up Objection

Frida is a dynamic instrumentation toolkit that is used to deploy scripts into the testing process of apk. Frida provides APIs for developers to make tools on top of it. One such tool is objection. Before setting up objection, we'll have to deploy Frida server in the android system.

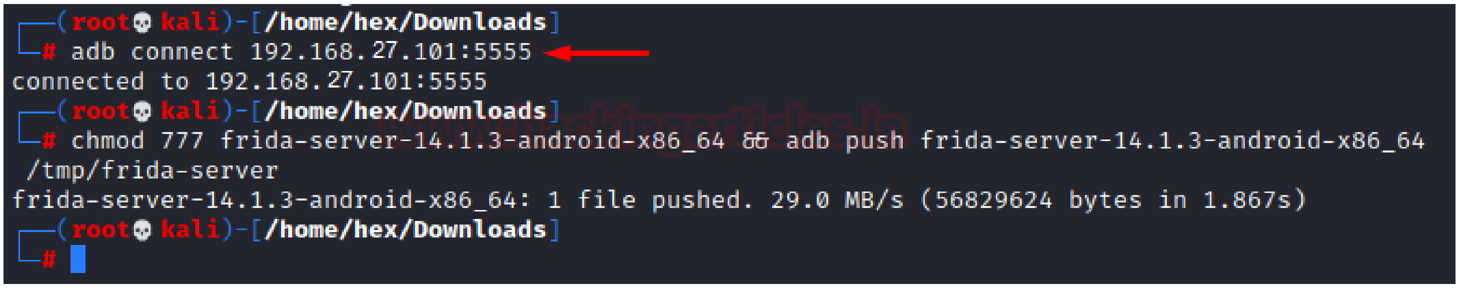
So, to do that we'd go over to **this** location and download the latest release for android available:


Release Frida 14.1.3 · frida/frida
https://github.com/frida/frida/releases/tag/14.1.3

frida-gumjs-devkit-14.1.3-macos-x86_64.tar.xz	17.1 MB
frida-gumjs-devkit-14.1.3-windows-x86.exe	20.3 MB
frida-gumjs-devkit-14.1.3-windows-x86_64.exe	26.7 MB
frida-inject-14.1.3-android-arm.xz	6.15 MB
frida-inject-14.1.3-android-arm64.xz	12.7 MB
frida-inject-14.1.3-android-x86.xz	7.8 MB
frida-inject-14.1.3-android-x86_64.xz	15.4 MB
frida-qml-14.1.3-linux-x86_64.tar.xz	15.4 MB
frida-qml-14.1.3-macos-x86_64.tar.xz	8.6 MB
frida-qml-14.1.3-windows-x86.exe	10.8 MB
frida-qml-14.1.3-windows-x86_64.exe	10.9 MB
frida-server-14.1.3-android-arm.xz	6.08 MB
frida-server-14.1.3-android-arm64.xz	12.6 MB
frida-server-14.1.3-android-x86.xz	7.7 MB
frida-server-14.1.3-android-x86_64.xz	15.3 MB
frida-server-14.1.3-ios-arm64.xz	5.7 MB
frida-server-14.1.3-ios-arm64e.xz	11.5 MB
frida-server-14.1.3-linux-arm64.xz	6.27 MB

Now we need to deploy this in android system. First, extract the file in the downloaded location and then we type in the following command:

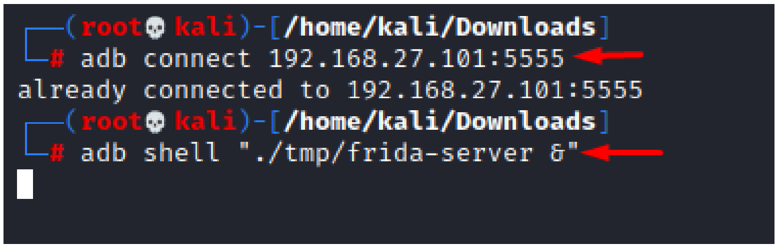
```
adb connect 192.168.27.101:5555
chmod 777 frida-server-14.1.3-android-x86_64 && adb push frida-server-14.1.3-andro
```



```
┌──(root💀 kali)-[/home/hex/Downloads]
└─# adb connect 192.168.27.101:5555
connected to 192.168.27.101:5555
┌──(root💀 kali)-[/home/hex/Downloads]
└─# chmod 777 frida-server-14.1.3-android-x86_64 && adb push frida-server-14.1.3-android-x86_64
 /tmp/frida-server
frida-server-14.1.3-android-x86_64: 1 file pushed. 29.0 MB/s (56829624 bytes in 1.867s)
┌──(root💀 kali)-[/home/hex/Downloads]
└─# 
```

To be able to run objection we'll have to launch the frida server first by doing:

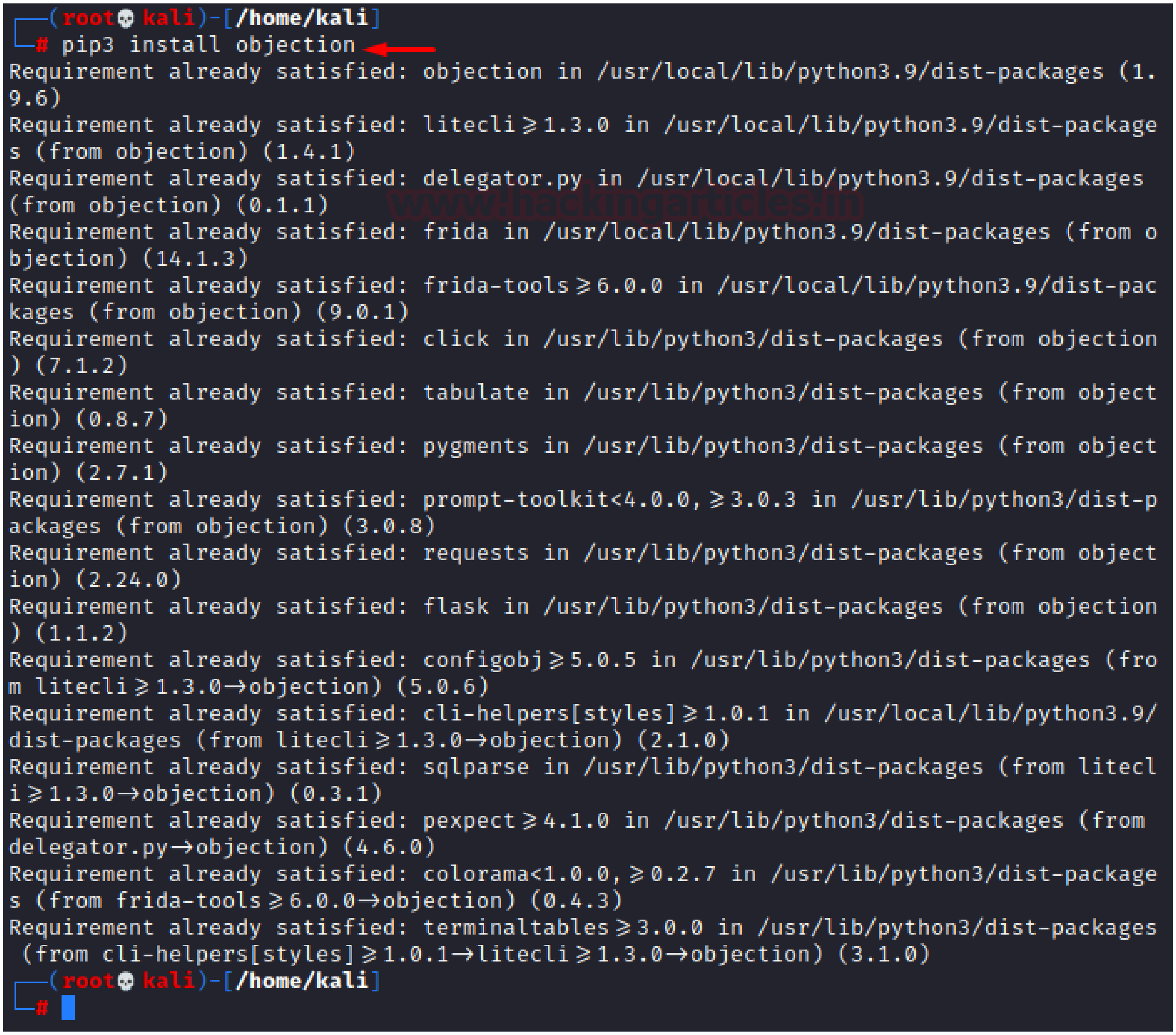
```
adb shell "./tmp/frida-server &"
```



```
┌──(root💀kali)-[/home/kali/Downloads]
└─# adb connect 192.168.27.101:5555
already connected to 192.168.27.101:5555
┌──(root💀kali)-[/home/kali/Downloads]
└─# adb shell "./tmp/frida-server &"
```

And finally, we need to install objection using pip3:

```
pip3 install objection
```

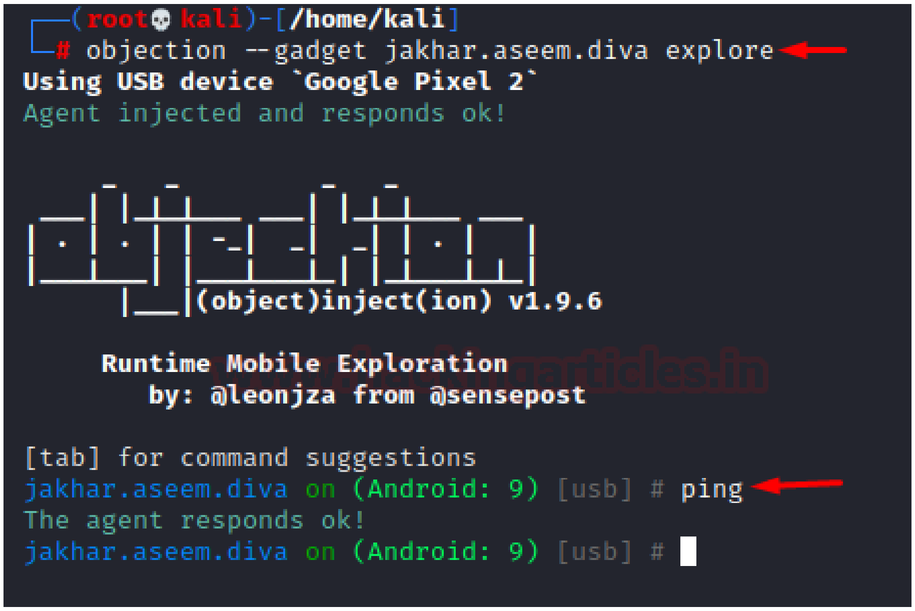


```
┌──(root💀kali)-[/home/kali]
└─# pip3 install objection
Requirement already satisfied: objection in /usr/local/lib/python3.9/dist-packages (1.
9.6)
Requirement already satisfied: litecli≥1.3.0 in /usr/local/lib/python3.9/dist-package
s (from objection) (1.4.1)
Requirement already satisfied: delegator.py in /usr/local/lib/python3.9/dist-packages
(from objection) (0.1.1)
Requirement already satisfied: frida in /usr/local/lib/python3.9/dist-packages (from o
bjection) (14.1.3)
Requirement already satisfied: frida-tools≥6.0.0 in /usr/local/lib/python3.9/dist-pac
kages (from objection) (9.0.1)
Requirement already satisfied: click in /usr/lib/python3/dist-packages (from objection
) (7.1.2)
Requirement already satisfied: tabulate in /usr/lib/python3/dist-packages (from object
ion) (0.8.7)
Requirement already satisfied: pygments in /usr/lib/python3/dist-packages (from object
ion) (2.7.1)
Requirement already satisfied: prompt-toolkit<4.0.0,≥3.0.3 in /usr/lib/python3/dist-p
ackages (from objection) (3.0.8)
Requirement already satisfied: requests in /usr/lib/python3/dist-packages (from object
ion) (2.24.0)
Requirement already satisfied: flask in /usr/lib/python3/dist-packages (from objection
) (1.1.2)
Requirement already satisfied: configobj≥5.0.5 in /usr/lib/python3/dist-packages (fro
m litecli≥1.3.0→objection) (5.0.6)
Requirement already satisfied: cli-helpers[styles]≥1.0.1 in /usr/local/lib/python3.9/
dist-packages (from litecli≥1.3.0→objection) (2.1.0)
Requirement already satisfied: sqlparse in /usr/lib/python3/dist-packages (from litecl
i≥1.3.0→objection) (0.3.1)
Requirement already satisfied: pexpect≥4.1.0 in /usr/lib/python3/dist-packages (from
delegator.py→objection) (4.6.0)
Requirement already satisfied: colorama<1.0.0,≥0.2.7 in /usr/lib/python3/dist-package
s (from frida-tools≥6.0.0→objection) (0.4.3)
Requirement already satisfied: terminaltables≥3.0.0 in /usr/lib/python3/dist-packages
 (from cli-helpers[styles]≥1.0.1→litecli≥1.3.0→objection) (3.1.0)
┌──(root💀kali)-[/home/kali]
└─#
```

And now we are good to go. Now, the packages (apps) installed in an android device are referred to as **gadget** in objection and to interact with an application to test, we'll have to inject our objection agent in the app. To do so:

```
objection –gadget jakhar.aseem.diva explore
ping
```

Pinging rechecks if the agent is successfully attached to the app or not.

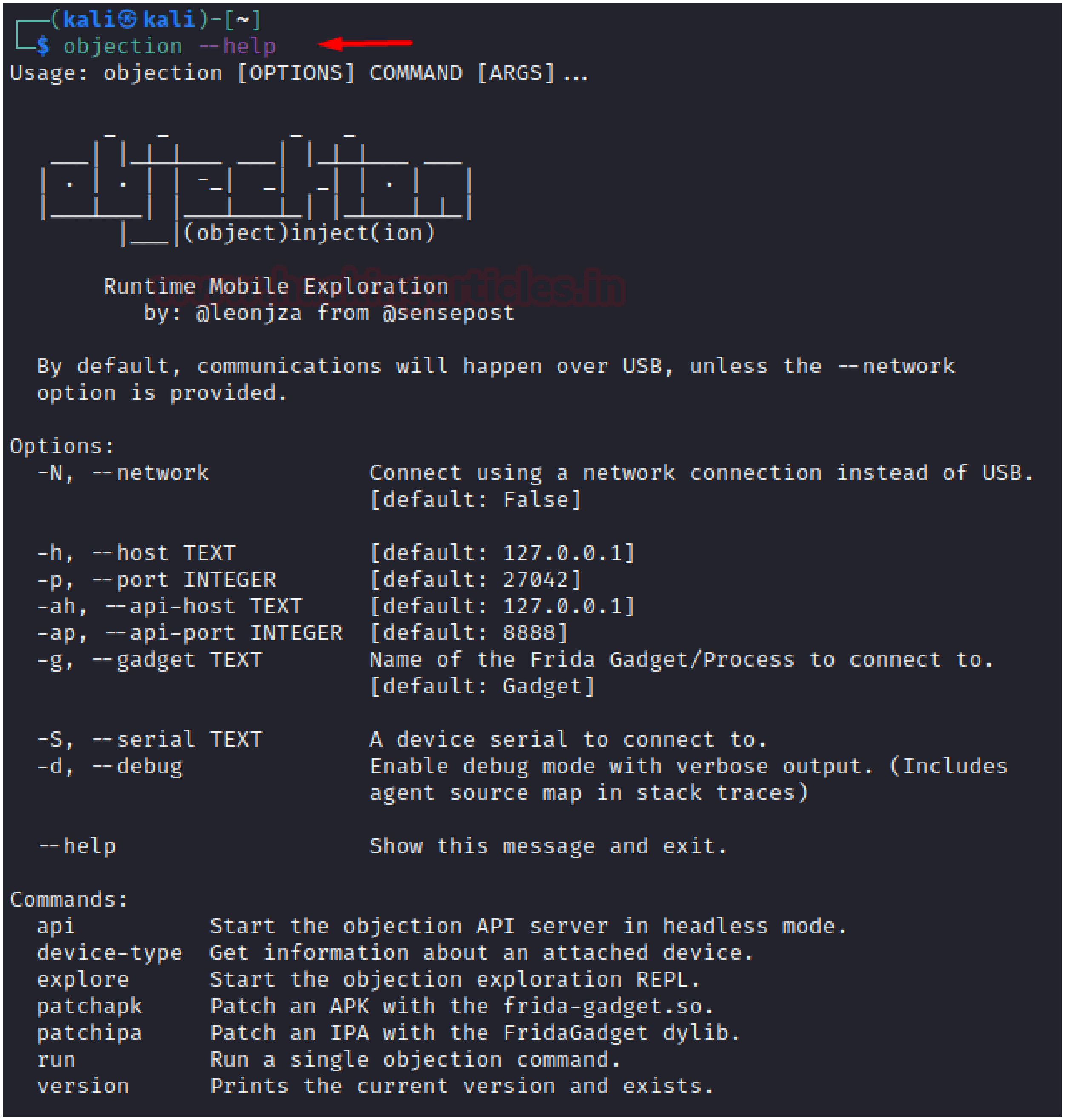


```
┌──(root💀 kali)-[/home/kali]
└─# objection --gadget jakhar.aseem.diva explore
Using USB device `Google Pixel 2`
Agent injected and responds ok!

     _   _         _   _
 ___| |_|_|___ ___| |_|_|___ ___
| . | . | | -_|  _|  _| | . |   |
|___|___| |___|___|_| |_|___|_|_|
      |___|(object)inject(ion) v1.9.6

     Runtime Mobile Exploration
        by: @leonjza from @sensepost

[tab] for command suggestions
jakhar.aseem.diva on (Android: 9) [usb] # ping
The agent responds ok!
jakhar.aseem.diva on (Android: 9) [usb] #
```

Help Menu

To open the help menu in objection:

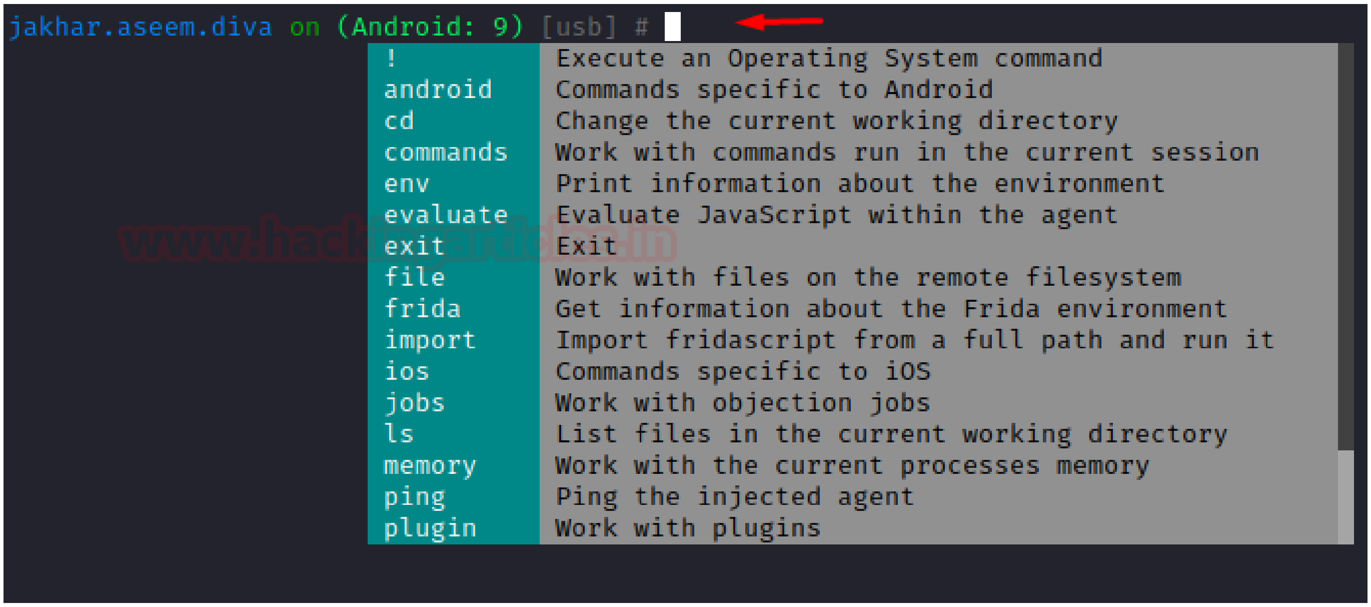
```
objection --help
```



```
┌──(kali㉿kali)-[~]
└─$ objection --help
Usage: objection [OPTIONS] COMMAND [ARGS]...

       _   _         _   _
   ___| |_|_|___ ___| |_|_|___ ___
  | . | . | | -_|  _|  _| | . |   |
  |___|___| |___|___|_| |_|___|_|_|
        |___|(object)inject(ion)

       Runtime Mobile Exploration
          by: @leonjza from @sensepost

  By default, communications will happen over USB, unless the --network
  option is provided.

Options:
  -N, --network             Connect using a network connection instead of USB.
                            [default: False]

  -h, --host TEXT           [default: 127.0.0.1]
  -p, --port INTEGER        [default: 27042]
  -ah, --api-host TEXT      [default: 127.0.0.1]
  -ap, --api-port INTEGER   [default: 8888]
  -g, --gadget TEXT         Name of the Frida Gadget/Process to connect to.
                            [default: Gadget]

  -S, --serial TEXT         A device serial to connect to.
  -d, --debug               Enable debug mode with verbose output. (Includes
                            agent source map in stack traces)

  --help                    Show this message and exit.

Commands:
  api          Start the objection API server in headless mode.
  device-type  Get information about an attached device.
  explore      Start the objection exploration REPL.
  patchapk     Patch an APK with the frida-gadget.so.
  patchipa     Patch an IPA with the FridaGadget dylib.
  run          Run a single objection command.
  version      Prints the current version and exists.
```

To launch a help menu in a gadget in Objection, we have two commands:

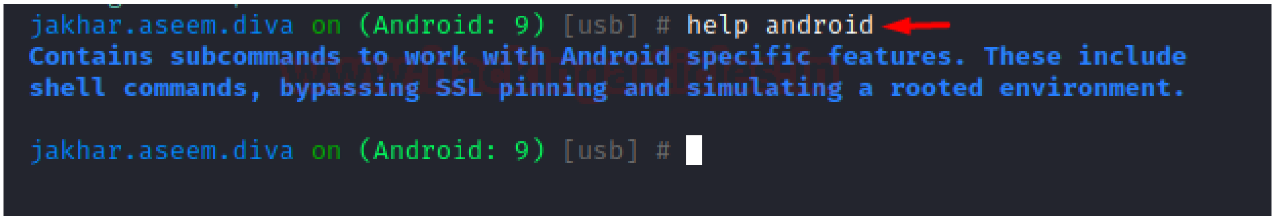
```
<tab>
```



```
jakhar.aseem.diva on (Android: 9) [usb] # 
                  !           Execute an Operating System command
                  android     Commands specific to Android
                  cd          Change the current working directory
                  commands    Work with commands run in the current session
                  env         Print information about the environment
                  evaluate    Evaluate JavaScript within the agent
                  exit        Exit
                  file        Work with files on the remote filesystem
                  frida       Get information about the Frida environment
                  import      Import fridascript from a full path and run it
                  ios         Commands specific to iOS
                  jobs        Work with objection jobs
                  ls          List files in the current working directory
                  memory      Work with the current processes memory
                  ping        Ping the injected agent
                  plugin      Work with plugins
```

Then further, to elaborate about a specific module:

```
help android
```

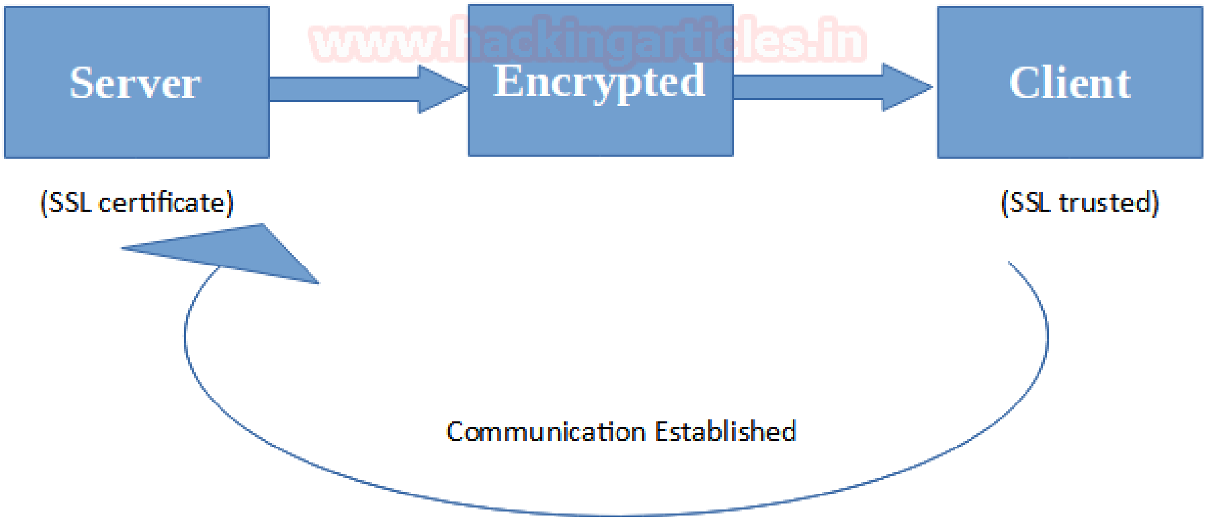


```
jakhar.aseem.diva on (Android: 9) [usb] # help android
Contains subcommands to work with Android specific features. These include
shell commands, bypassing SSL pinning and simulating a rooted environment.

jakhar.aseem.diva on (Android: 9) [usb] # 
```

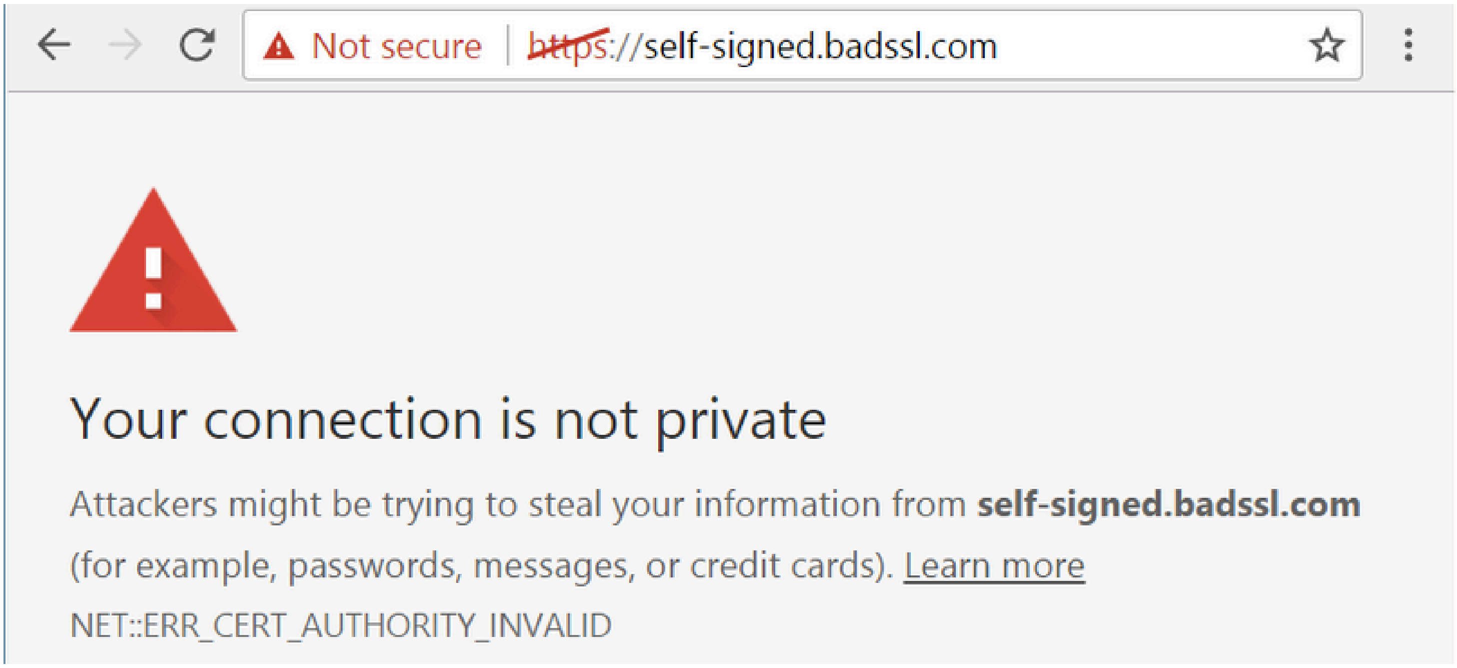
SSL Pinning Bypass

To understand pinning bypass, we’ll first look at what SSL pinning is.

1. **Traditional Certificates and self-signed certificate:** In traditional server-client architecture, a client validates a connection using a certificate presented by a server during handshake. **Certificate is the proof of the identity of a server**. This certificate is proved to be valid only if it is signed by a trusted Certificate Authority (eg: RapidSSL, Symantec, digicert). After the certificate is validated, the data transmission begins which is encrypted based on the cipher suite the certificate has been configured to provide.



This can be abused by installing a **self-created root CA** certificate on the client system. A common example is Burp Suite's Portswigger CA. Burp is sort of like a Man in the Middle in this situation and traffic coming from server passes through burp, it intercepts it and tries to forward it to browser but the browser doesn't trust the burp's certificate, so it throws an error (ERR_CERT_AUTHORITY_INVALID). Some message is also shown like this:



Your connection is not private

Attackers might be trying to steal your information from **self-signed.badssl.com** (for example, passwords, messages, or credit cards). Learn more

NET::ERR_CERT_AUTHORITY_INVALID

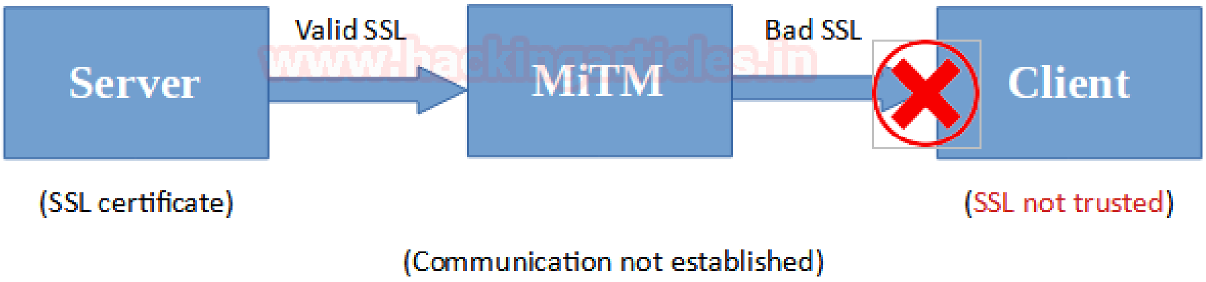
So, to make browser trust burp's server certificate, a tester installs **cacert.der** in his own system and adds Burp's certificate as **trusted** so that browser starts trusting Burp's server certificate and the website accessed through burp's proxy. This way, the attacker is able to read, modify and send requests originating from his system.

2. **SSL Pinning:** Most commonly used in Android APK and iOS IPA, it is a technique deployed by developers in their code at client side to prevent MiTM attacks. It validates server certificates again even after SSL handshaking. **The developers adds or pins a list of trusted certificates to the client application's code itself and uses it to compare against the server certificates during run time.**

Hence, there is a problem for an attacker now as even if there is a self-generated cert installed on the client device, the communication is still not being intercepted due to the developer embedded certificate pinning code that is within the application. These are generally an **x509 cert** or a public key.

So, now if there is a mismatch, the connection will be disrupted, and no further communication will take place.

This can be understood as diagrammatically as follows:



More about it can be read on OWASP's website **here.**

3. **<u>SSL Pinning Bypass</u>:** To bypass SSL pinning there are further two methods

<u>Method 1:</u>

In Android applications, code having some strings like "**checkClientTrusted**" or "**checkServerTrusted**" is generally the code with pinning. It could be some other as well. So, one way to bypass SSL pinning is to decompile the source code, search for this, remove these lines of code, recompile and sign using apktool. More on this would be covered in a further article on reverse engineering Android applications.

<u>Method 2:</u>

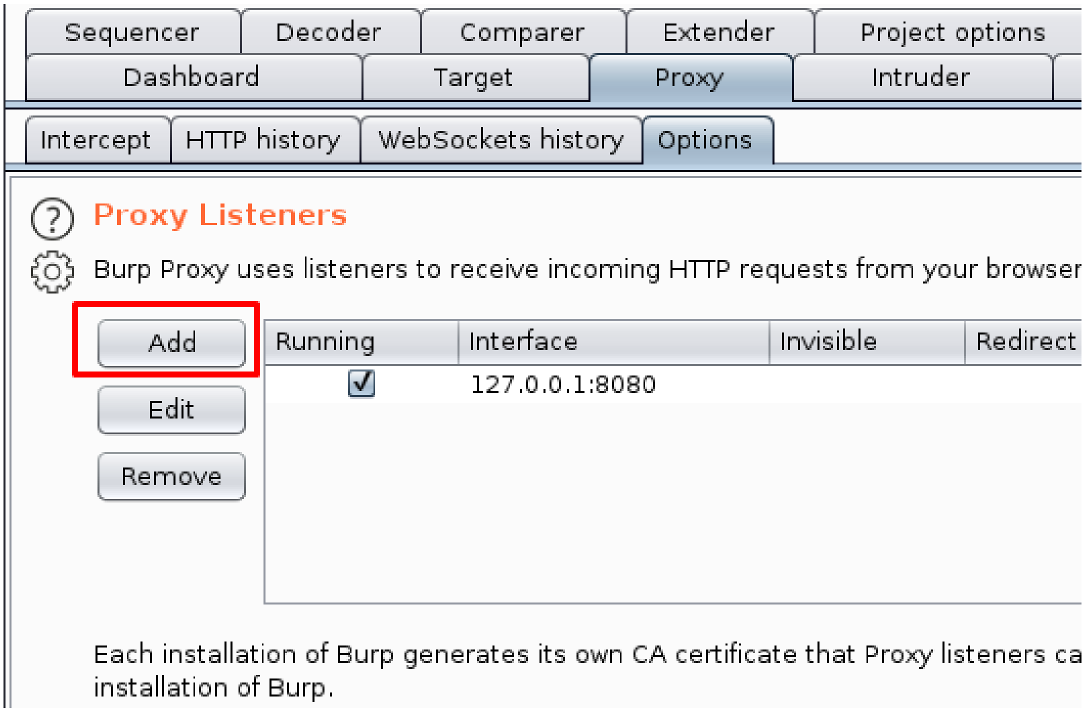
To bypass SSL pinning we will **inject a custom code** into the application, **while it is running**, so that it adds our self-generated certificate as a trusted certificate for the application. The process of injecting a code while an application is running is called **hooking**. Frida allows a user to create hooks and change an application's behavior while run time. Hooking also allows a person to intercept software to hardware calls and modify it.

Frida hooking: Now, Frida alone is very much capable of hooking into android apps using javascript. But the knowledge of javascript is essential to coding this manually. There are various scripts available to do this on github like **here** but to avoid this hassle altogether with just a few clicks, is where objection comes in the picture.

4. **SSLPinning bypass using Objection**: As stated above, Objection is based over Frida's API. It also has handsome tools to bypass SSL pinning by generating javascript code and hooking it into the application.

Let’s see how it is done but before that, we’ll set up burpsuite on our android to intercept the communication from an SSLpinned application.

Step 1: Open burp and add an interface.



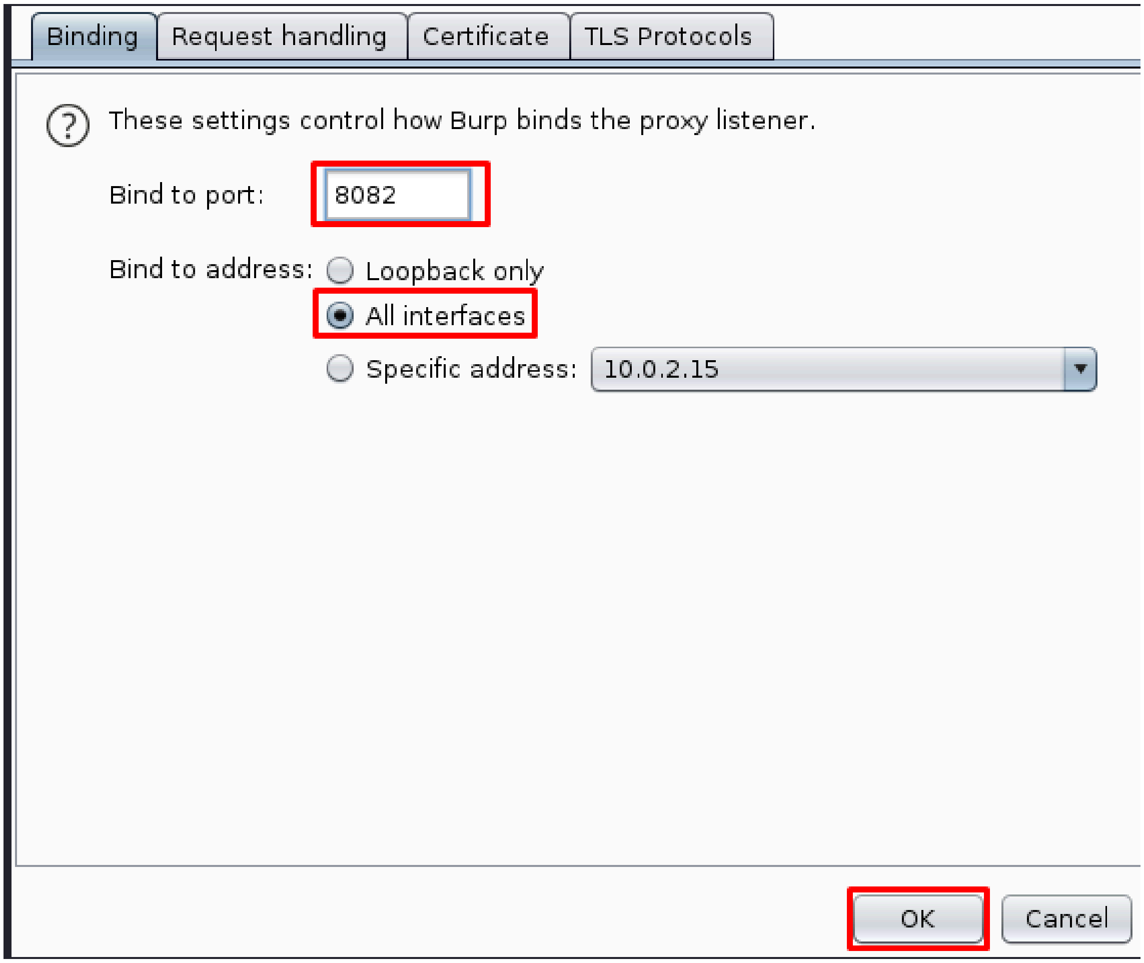
Proxy Listeners

Burp Proxy uses listeners to receive incoming HTTP requests from your browser

Running	Interface	Invisible	Redirect
☑	127.0.0.1:8080		

Each installation of Burp generates its own CA certificate that Proxy listeners ca
installation of Burp.

Step 2: Add some other port than already being used, say, port 8082. Change the bind to address option to All interfaces and click ok.



These settings control how Burp binds the proxy listener.

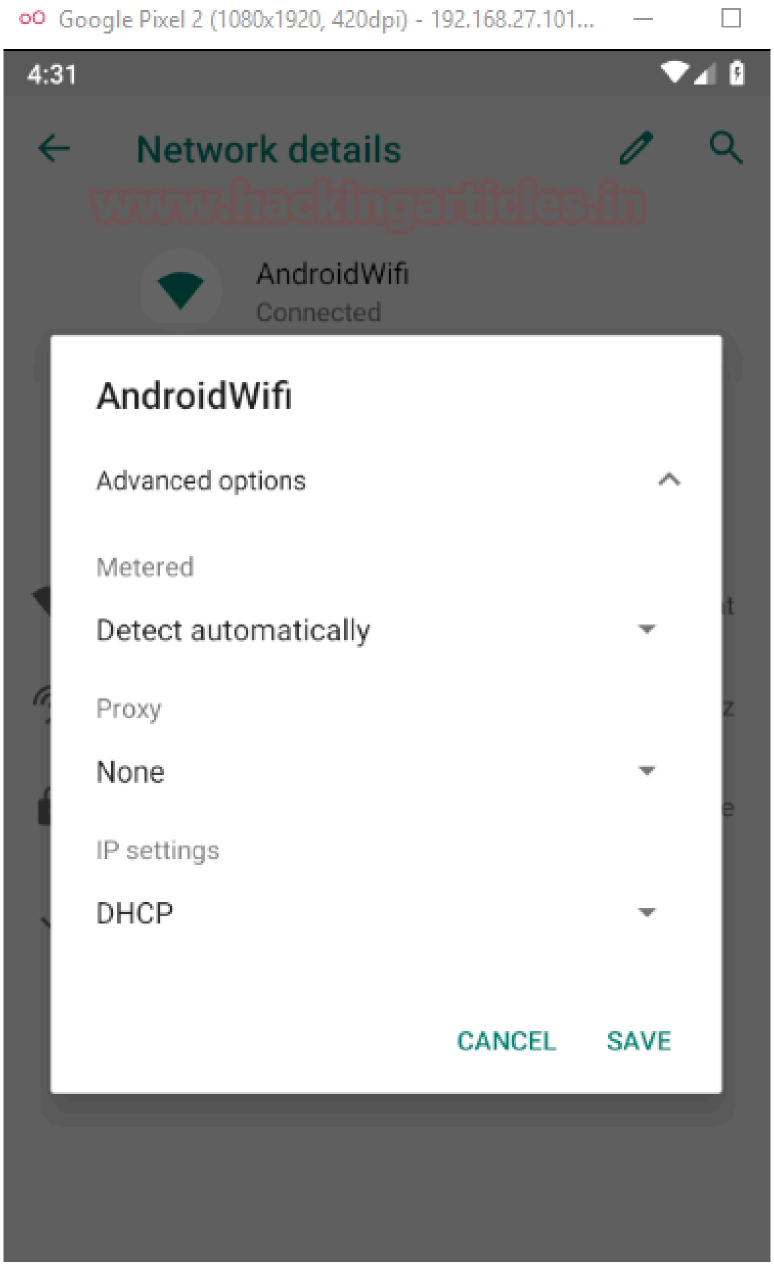
Bind to port: 8082

Bind to address: ○ Loopback only

● All interfaces

○ Specific address: 10.0.2.15

Step 3: Head over to your android Wi-Fi and click on edit connection (pencil icon)



Google Pixel 2 (1080x1920, 420dpi) - 192.168.27.101...

4:31

Network details

AndroidWifi
Connected

AndroidWifi

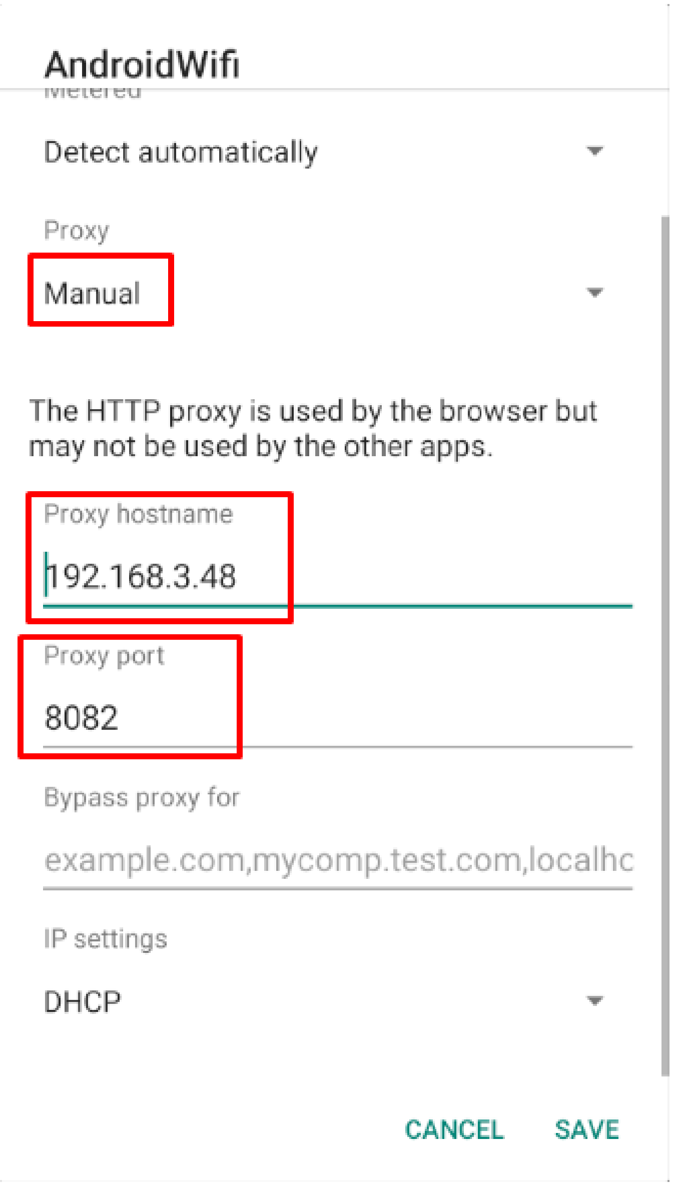
Advanced options

Metered: Detect automatically

Proxy: None

IP settings: DHCP

Step 4: Change proxy to manual and add the hostname as your IP on which burp is being run. Also, change your port to 8082 and click save.



AndroidWifi

Metered

Detect automatically

Proxy

Manual

The HTTP proxy is used by the browser but may not be used by the other apps.

Proxy hostname

192.168.3.48

Proxy port

8082

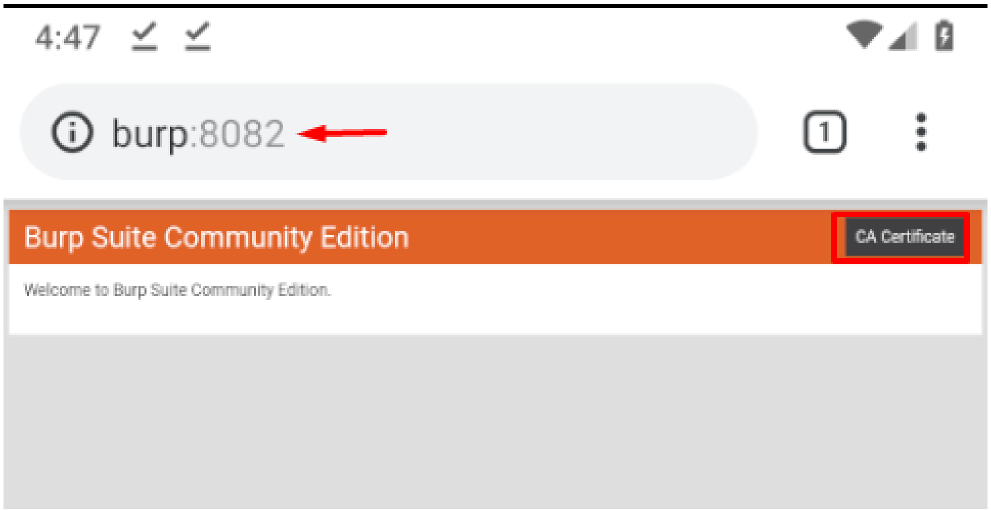
Bypass proxy for

example.com,mycomp.test.com,localhc

IP settings

DHCP

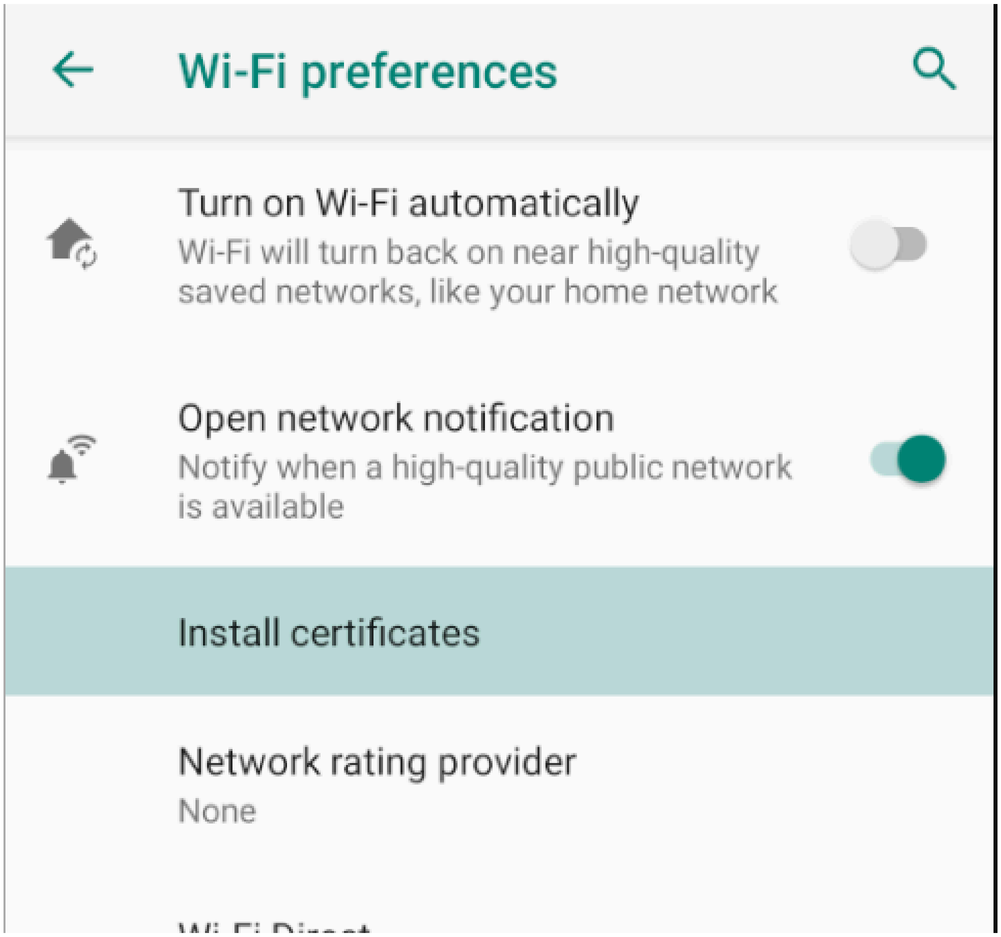
Step 5: Move over to your android device and type “http://burp:8082” and download the burp’s cacert.



Burp Suite Community Edition

Welcome to Burp Suite Community Edition.

Step 6: Once the certificate is downloaded, head over to Wi-Fi preferences and down at the bottom you’ll find an option “**install certificates**.” Click on it and choose the certificate you just downloaded. But you’ll note that the downloaded certificate is **in *.der format** and android doesn't recognise it as a valid certificate. So, we'll pull the certificate and rename it as "**cert-der.crt**"


← Wi-Fi preferences

Turn on Wi-Fi automatically
Wi-Fi will turn back on near high-quality saved networks, like your home network

Open network notification
Notify when a high-quality public network is available

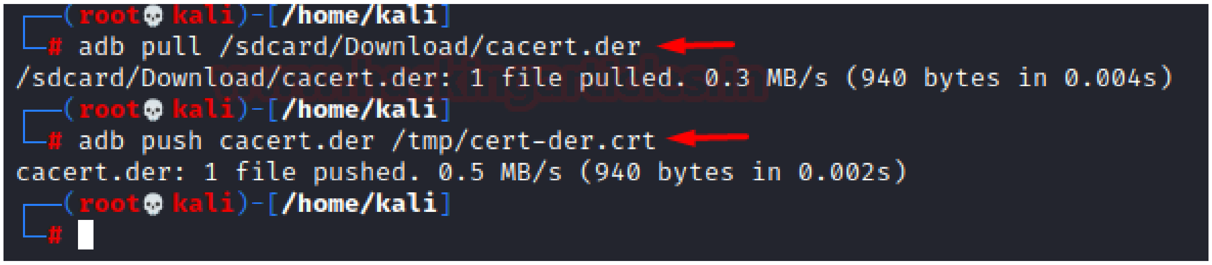
Install certificates

Network rating provider
None

Wi-Fi Direct

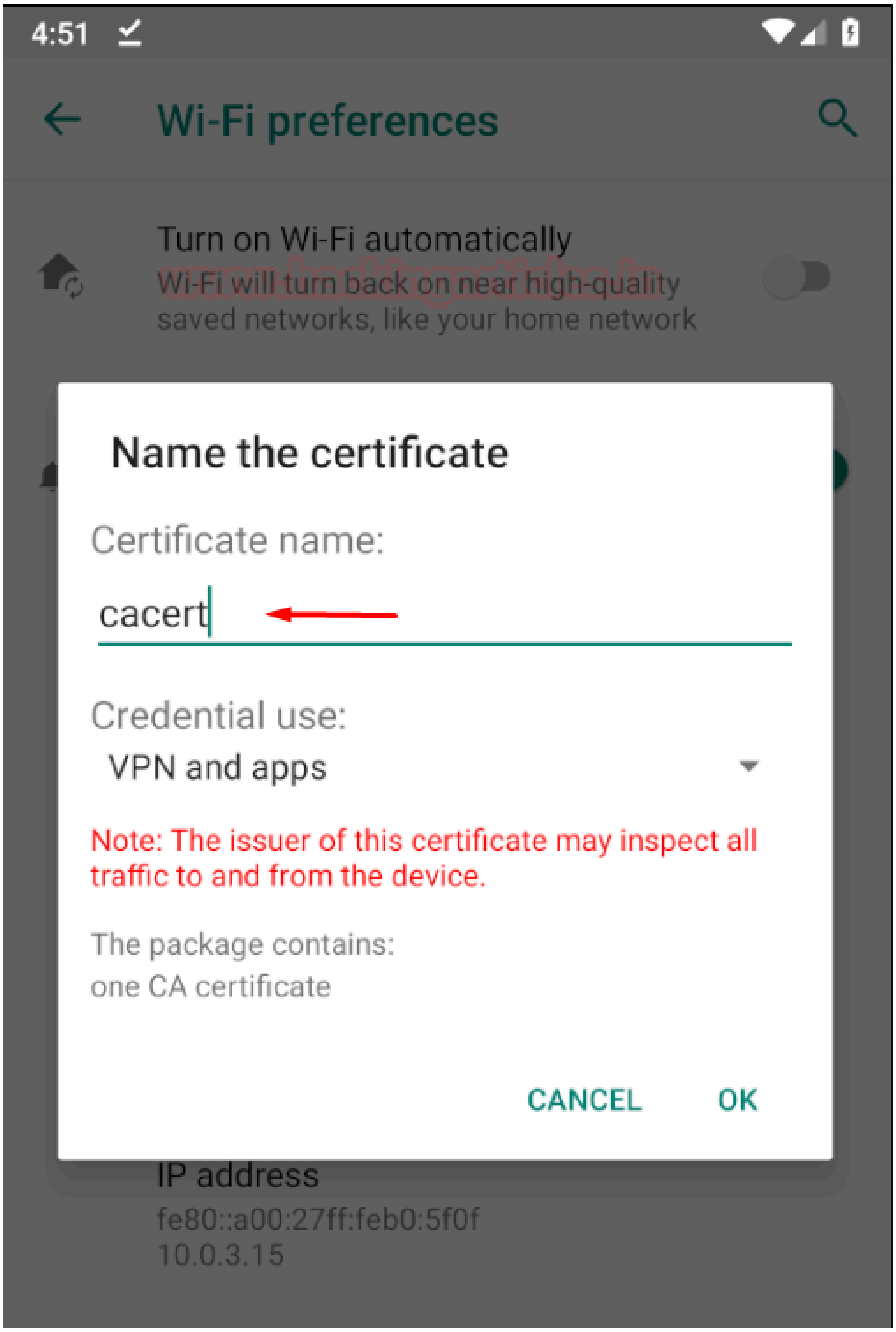
To rename this, we type in the following command:

```
adb pull /sdcard/Download/cacert.der
adb push cacert.der /tmp/cert-der.crt
```



```
┌──(root💀kali)-[/home/kali]
└─# adb pull /sdcard/Download/cacert.der
/sdcard/Download/cacert.der: 1 file pulled. 0.3 MB/s (940 bytes in 0.004s)
┌──(root💀kali)-[/home/kali]
└─# adb push cacert.der /tmp/cert-der.crt
cacert.der: 1 file pushed. 0.5 MB/s (940 bytes in 0.002s)
┌──(root💀kali)-[/home/kali]
└─# 
```

Step 7: Once you have downloaded, you'll find an option to rename the cert. Just type in "cacert" and make sure in credential use apps is selected.



Turn on Wi-Fi automatically

Wi-Fi will turn back on near high-quality saved networks, like your home network

Name the certificate

Certificate name:

cacert

Credential use:

VPN and apps

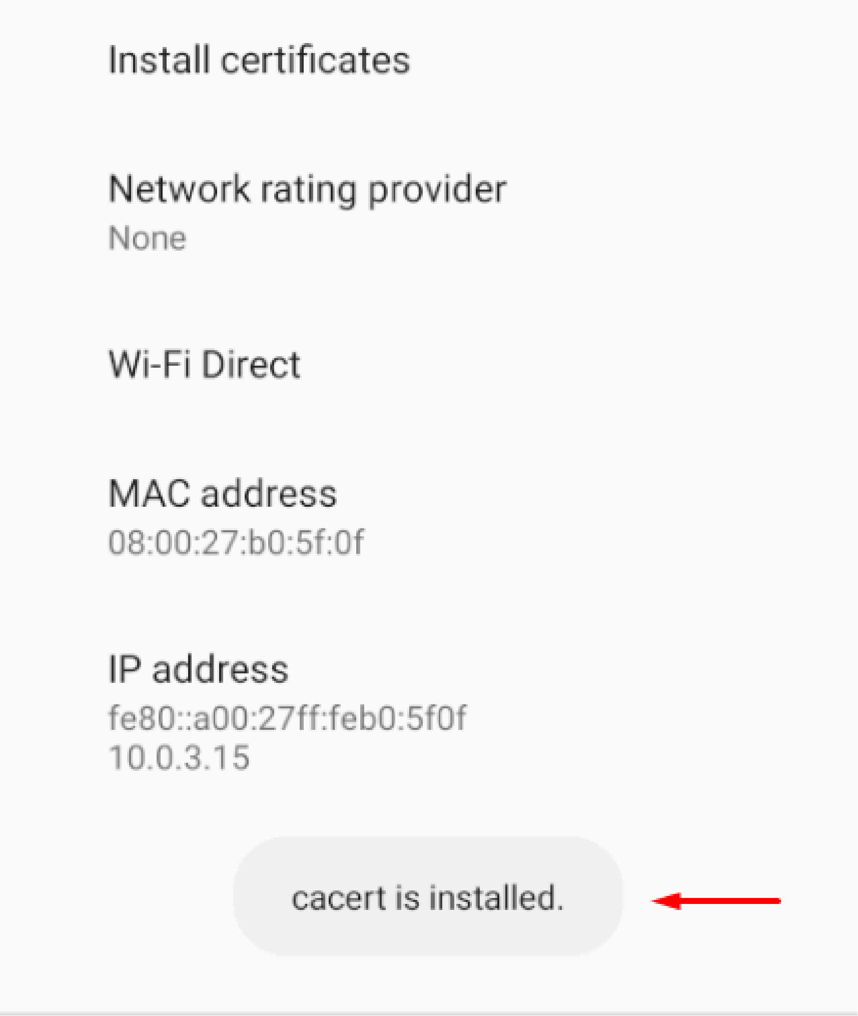
Note: The issuer of this certificate may inspect all traffic to and from the device.

The package contains:
one CA certificate

IP address

fe80::a00:27ff:feb0:5f0f
10.0.3.15

Step 8: cacert would now have been successfully installed and you should note that burp has now started capturing all the traffic from the browser and some apps as well. Note that these are only the apps which don't have SSL pinning enabled.



Install certificates

Network rating provider
None

Wi-Fi Direct

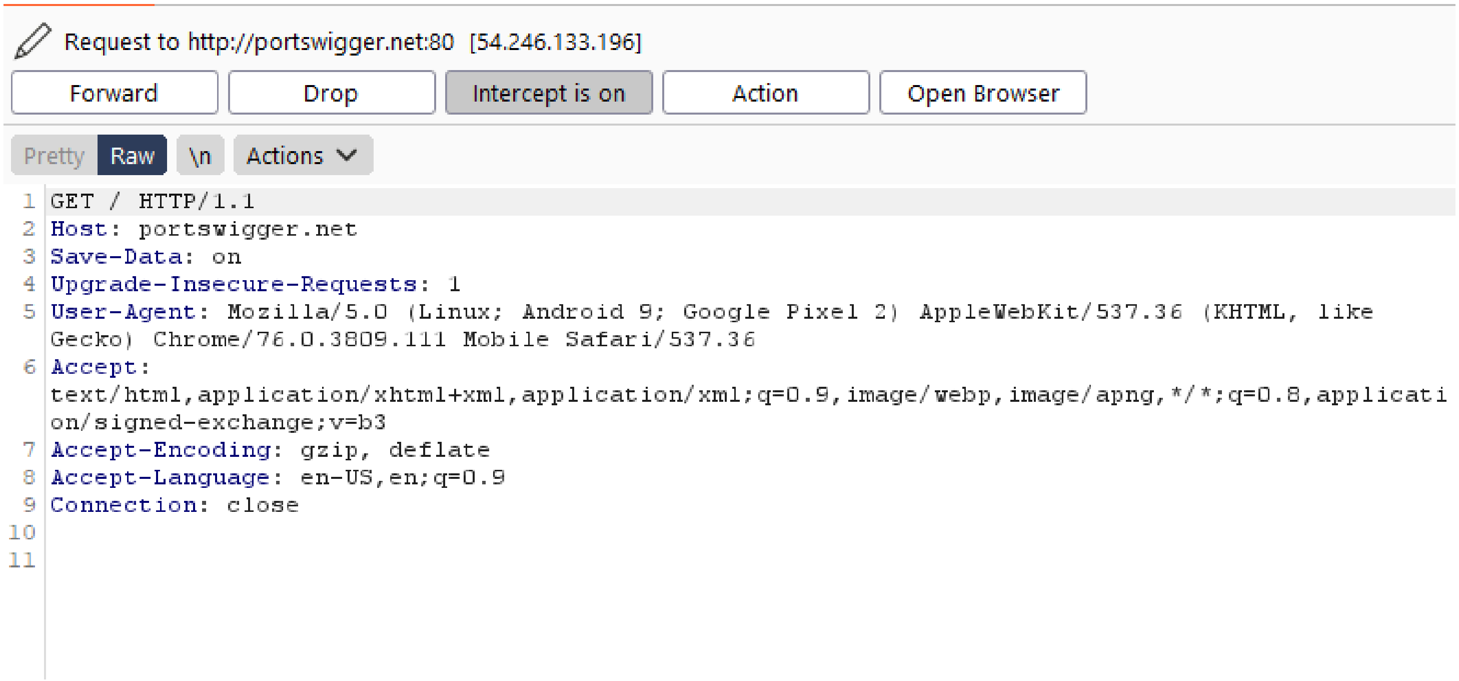
MAC address
08:00:27:b0:5f:0f

IP address
fe80::a00:27ff:feb0:5f0f
10.0.3.15

cacert is installed.

Head over to an HTTPS website and you'll note the requests are now getting captured. You can verify installed certificate by going in **Settings->trusted credentials->users**.

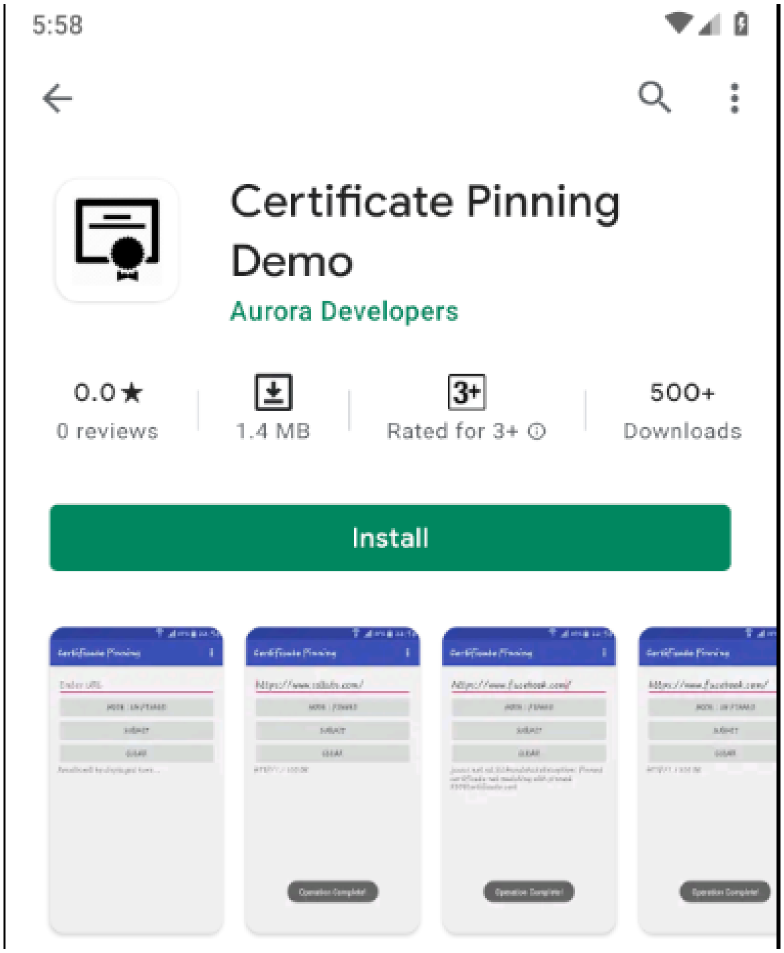
After installing it, SSL connections are also getting captured now. For example, portswigger.net



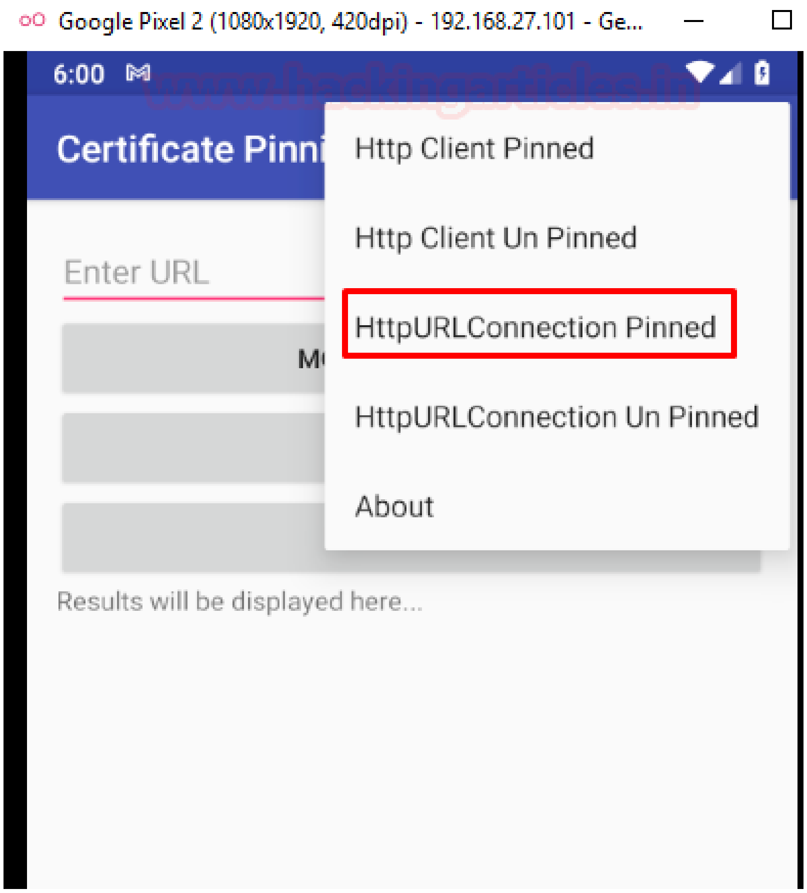
Request to http://portswigger.net:80 [54.246.133.196]

```
GET / HTTP/1.1
Host: portswigger.net
Save-Data: on
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Linux; Android 9; Google Pixel 2) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/76.0.3809.111 Mobile Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,applicati
on/signed-exchange;v=b3
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close


```

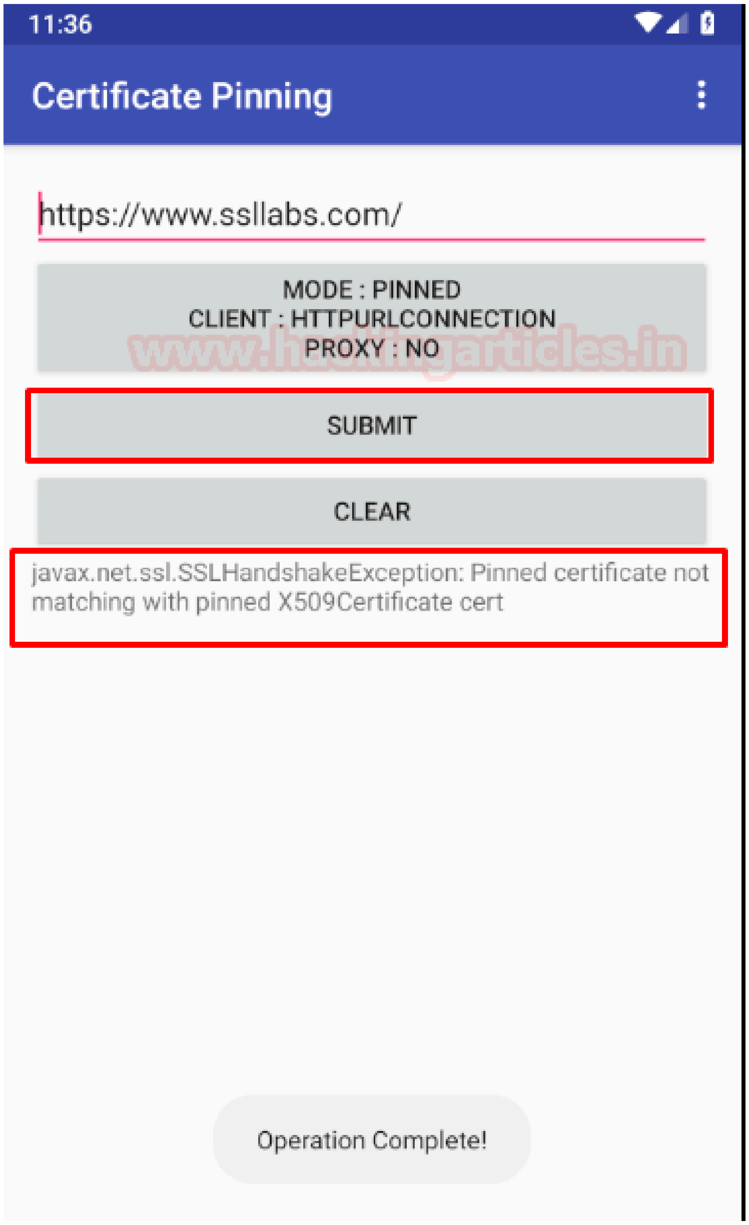
Step 9: Now that everything is set up, we'll need an application to test SSL Pinning. There is an application in playstore by the name "**Certificate Pinning Demo**." This app would help us demonstrate SSL pinning bypass using objection. So, install this on the device.



Step 10: Head over to the app and pin the HttpURLConnection.



Now, when you submit the request for the sample URL already provided by default, i.e, ssllabs.com, we'd note that request has failed due to certificate error even after we have installed our cacert in the device.



https://www.ssllabs.com/

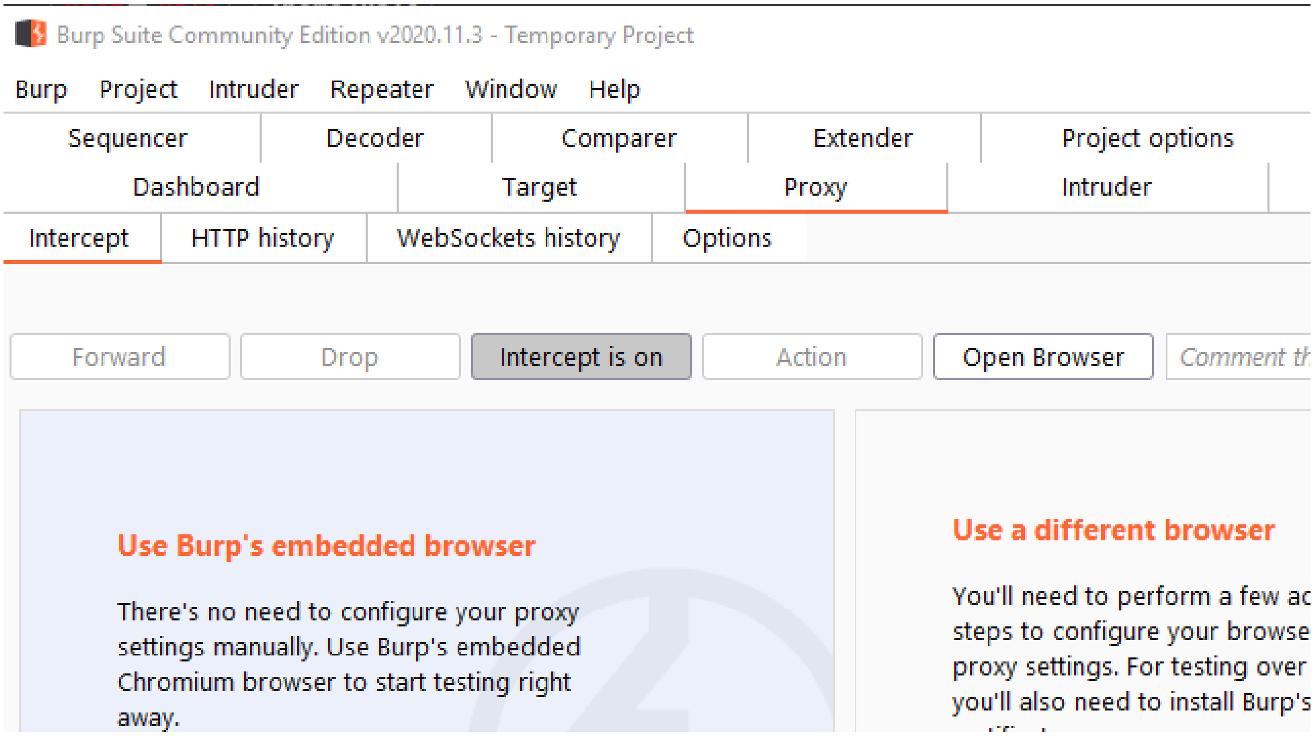
MODE : PINNED
CLIENT : HTTPURLCONNECTION
PROXY : NO

javax.net.ssl.SSLHandshakeException: Pinned certificate not matching with pinned X509Certificate cert

Operation Complete!

Heading over to burp, we'd see that no request has been captured. Now, here comes the **interesting part.**

Bypassing SSLPinning using objection.


Burp Suite Community Edition v2020.11.3 - Temporary Project
Burp Project Intruder Repeater Window Help
Sequencer Decoder Comparer Extender Project options
Dashboard Target Proxy Intruder
Intercept HTTP history WebSockets history Options
Forward Drop Intercept is on Action Open Browser Comment th

Use Burp's embedded browser

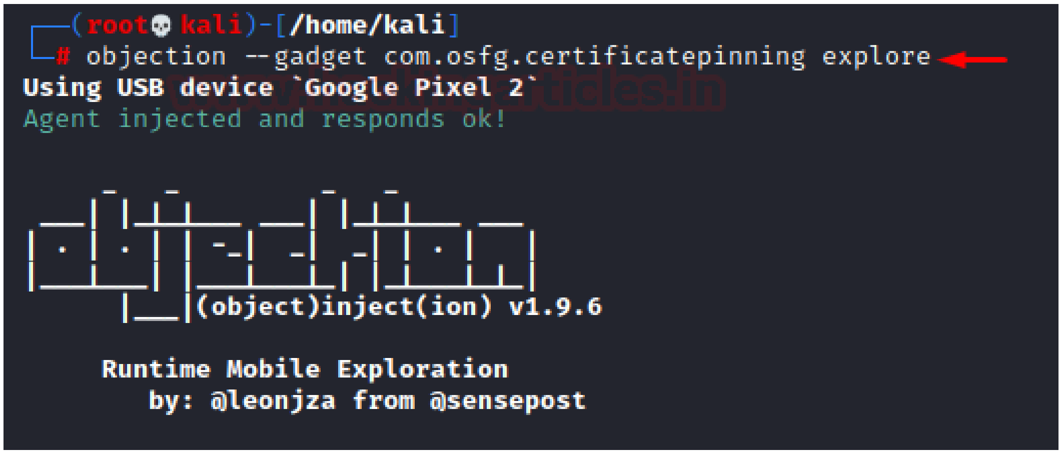
There's no need to configure your proxy settings manually. Use Burp's embedded Chromium browser to start testing right away.

Use a different browser

You'll need to perform a few ac steps to configure your browse proxy settings. For testing over you'll also need to install Burp's

Step 11: Attach the package to objection and run explore command:

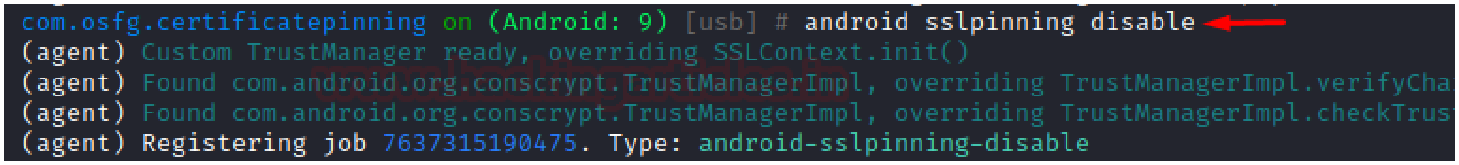
```
objection -gadget com.osfg.certificatepinning explore
```



```
┌──(root💀kali)-[/home/kali]
└─# objection --gadget com.osfg.certificatepinning explore
Using USB device `Google Pixel 2`
Agent injected and responds ok!

     _   _         _   _
 ___| |_|_|___ ___| |_|_|___ ___
| . | . | | -_|  _|  _| | . |   |
|___|___| |___|___|_| |_|___|_|_|
      |___|(object)inject(ion) v1.9.6

     Runtime Mobile Exploration
        by: @leonjza from @sensepost
```

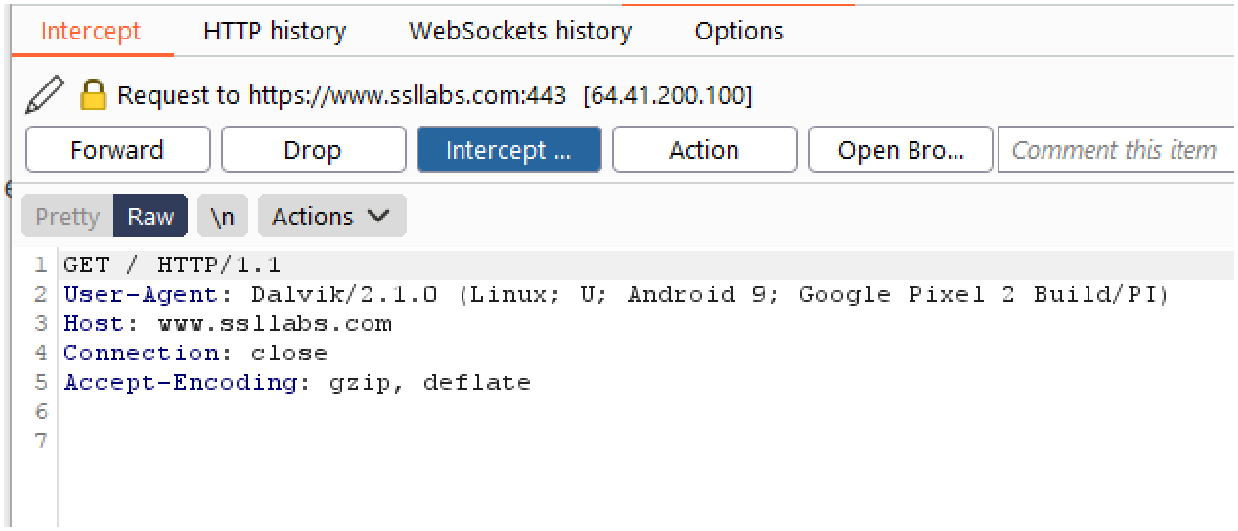
Step 12: we'll use the module android sslpinning to disable SSLPinning while the package is running. Objection is injecting a JavaScript code or "hooking" it using Frida's agent to do so.

```
android sslpinning disable
```



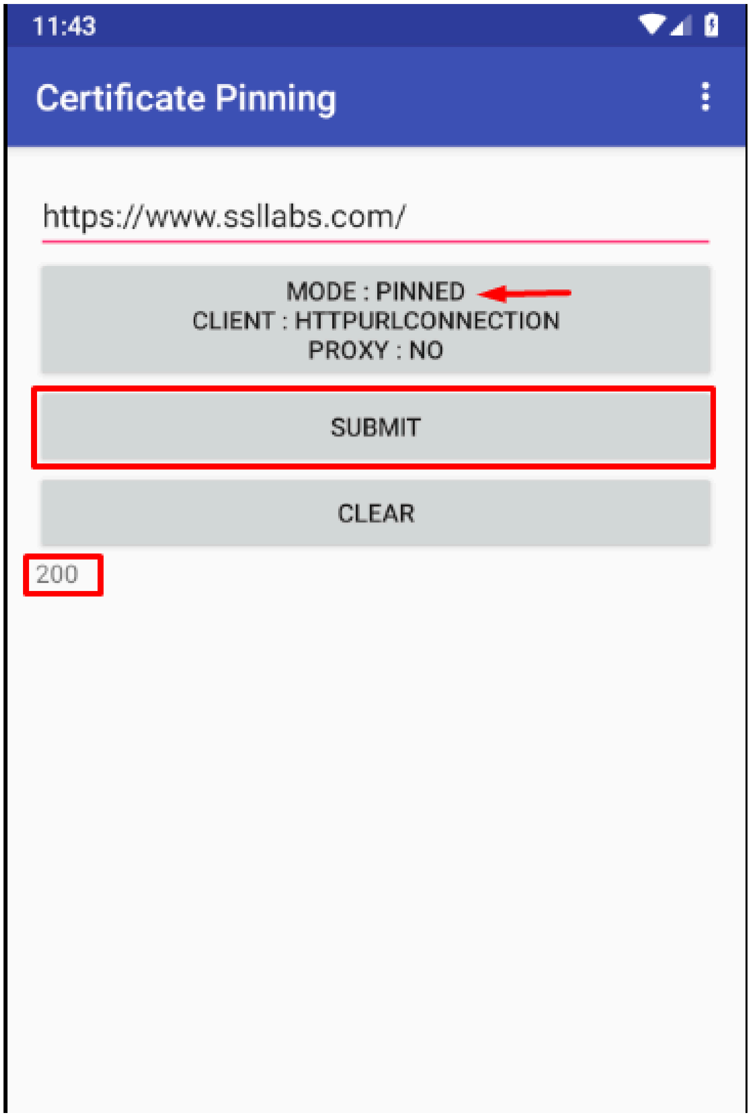
```
com.osfg.certificatepinning on (Android: 9) [usb] # android sslpinning disable
(agent) Custom TrustManager ready, overriding SSLContext.init()
(agent) Found com.android.org.conscrypt.TrustManagerImpl, overriding TrustManagerImpl.verifyCha
(agent) Found com.android.org.conscrypt.TrustManagerImpl, overriding TrustManagerImpl.checkTrus
(agent) Registering job 7637315190475. Type: android-sslpinning-disable
```

Once it is done, click on submit again in the app and check burp for any possible communication it has intercepted.



```
GET / HTTP/1.1
User-Agent: Dalvik/2.1.0 (Linux; U; Android 9; Google Pixel 2 Build/PI)
Host: www.ssllabs.com
Connection: close
Accept-Encoding: gzip, deflate
```

Now, we see a status code of 200, i.e., connection successfully established with no certificate error. Hence, we have bypassed SSLPinning!



Android Hooking

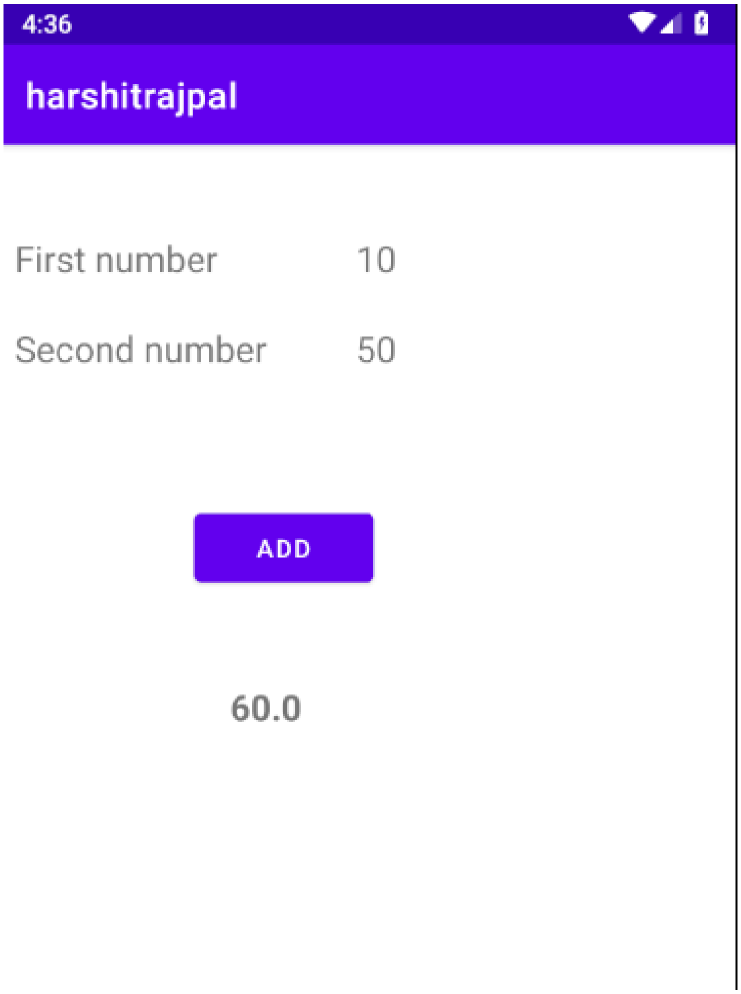
As explained briefly in the SSLPinning Bypass method above, hooking is the process of inserting our code (or hook) while the application is running to modify its behaviour. Hooking enables a client to see the software to hardware communications, modify output/ functions of the methods being used. Frida uses javascript to hook into methods of the application running, so we'll demonstrate hooking as a concept for a small android application we wrote and use javascript to hook into methods.

Here, I have written a small code in Java to calculate sum of 10 and 50.

```
package com.example.harshitrajpal;
import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;
public class MainActivity extends AppCompatActivity {
    Button add_button;
    TextView a,b,sum;
    double add=0;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        a=(TextView)findViewById(R.id.num1);
        b=(TextView)findViewById(R.id.num2);
        add_button=(Button) findViewById(R.id.add_button);
        sum=(TextView)findViewById(R.id.sum);
        add_button.setOnClickListener(new View.OnClickListener(){
            public void onClick(View v){
                sum.setText(Double.toString(returnValue()));
            }
        });
    }
    double returnValue(){
        double a1 = 10;
        double b1 = 50;
        add= a1 + b1;
        return add;
    }
}
```

As you can see it is taking two numbers as input and displaying its result.

Note: You can download the source code of the application from **here**.



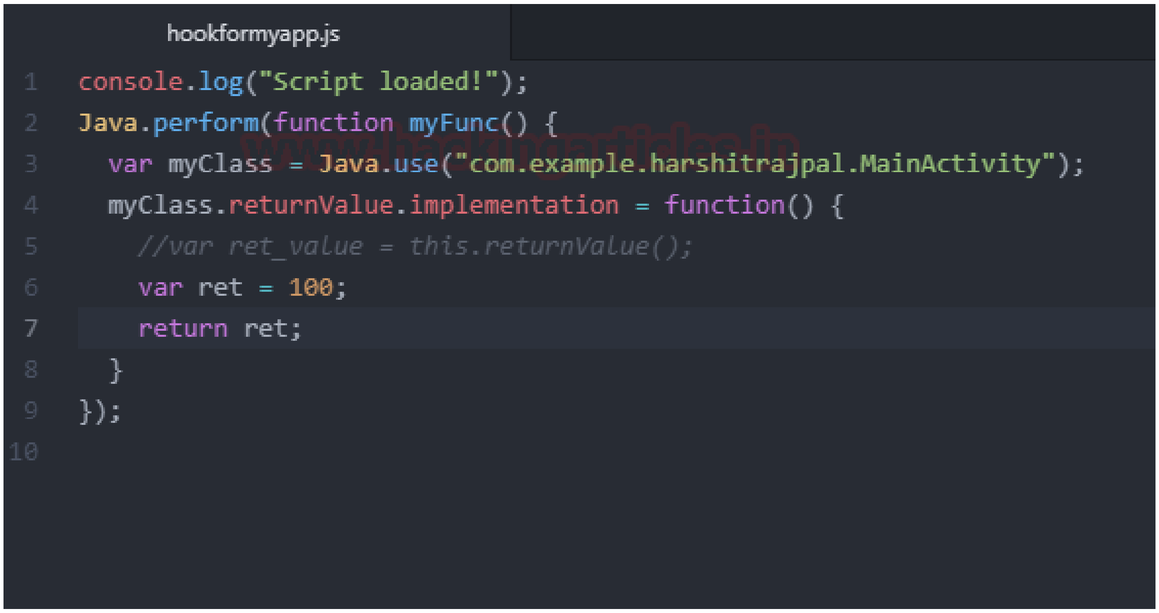
First number 10

Second number 50

60.0

Now we'll create a hook in javascript and try to tamper with the output of the two numbers! As you can see the app is using a function called returnValue to calculate. Maybe we can tamper it.

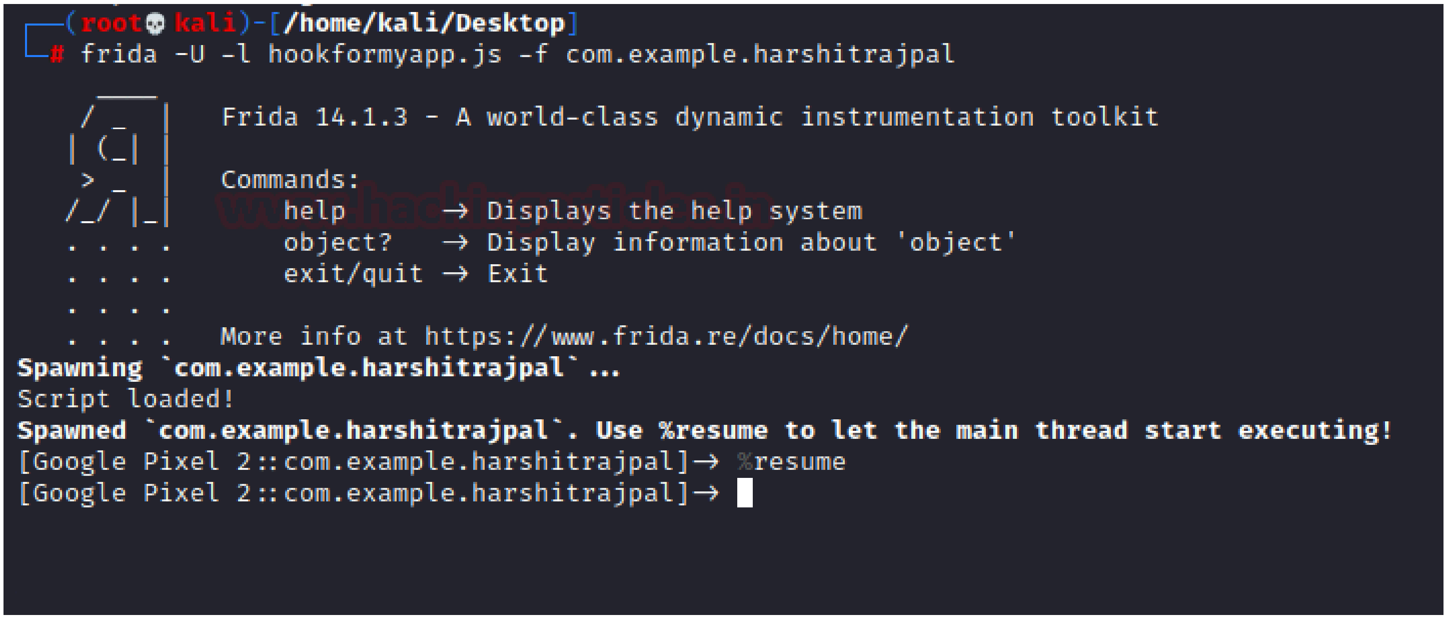
Following is the hook that I have created in javascript for the above program in which I have changed the return value to 100:



```
console.log("Script loaded!");
Java.perform(function myFunc() {
  var myClass = Java.use("com.example.harshitrajpal.MainActivity");
  myClass.returnValue.implementation = function() {
    //var ret_value = this.returnValue();
    var ret = 100;
    return ret;
  }
});
```

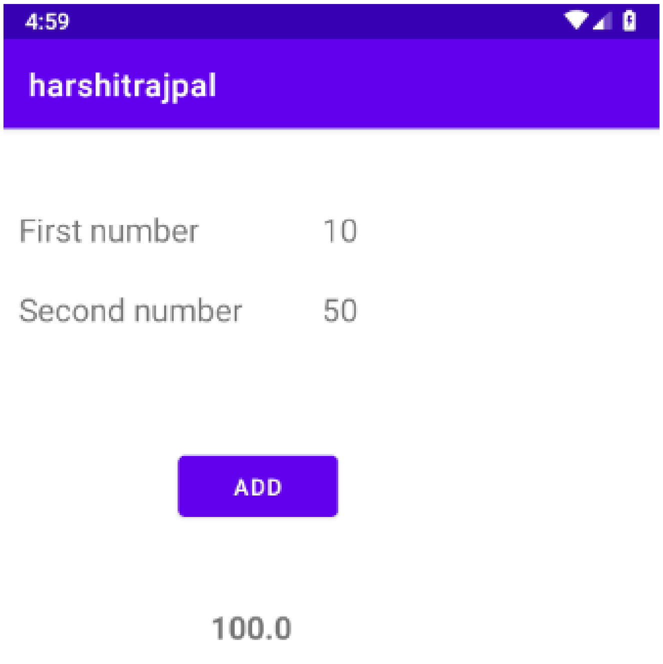
You can download the code for this javascript file from **here**.

Now, when we manually run this code using frida we see the following:



```
┌──(root💀 kali)-[/home/kali/Desktop]
└─# frida -U -l hookformyapp.js -f com.example.harshitrajpal
     ____
    / _  |   Frida 14.1.3 - 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/
Spawning `com.example.harshitrajpal`...
Script loaded!
Spawned `com.example.harshitrajpal`. Use %resume to let the main thread start executing!
[Google Pixel 2::com.example.harshitrajpal]-> %resume
[Google Pixel 2::com.example.harshitrajpal]->
```

And sure enough, when we click on add we see that hook has worked!



4:59

harshitrajpal

First number 10

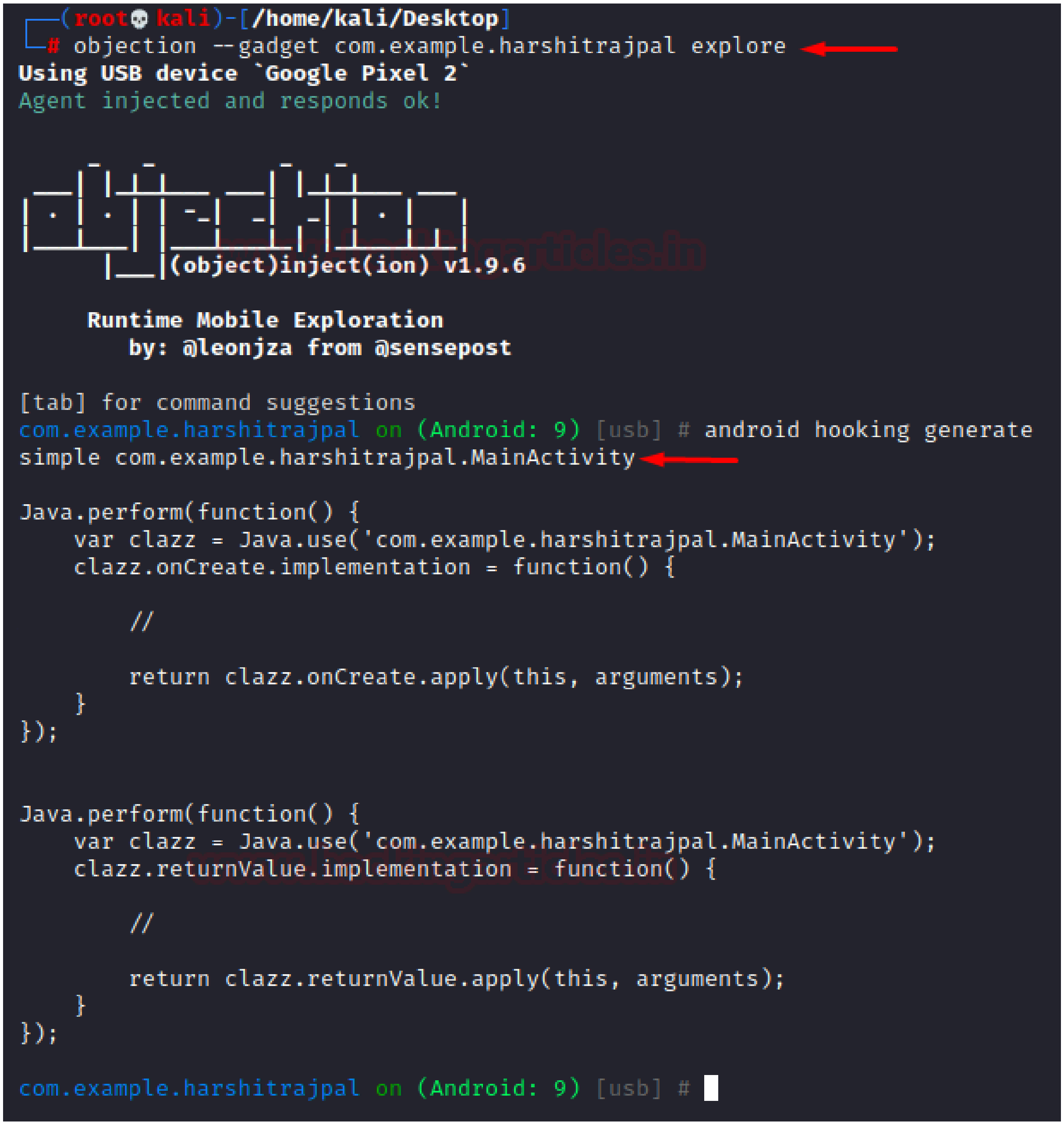
Second number 50

100.0

To make javascript hook code creation easy objection can automate this process. Let's explore what all options do objection have for hooking. Let's attach the gadget in objection and explore all related hooking options:

```
objection --gadget com.example.harshitrajpal explore
android hooking generate simple com.example.harshitrajpal.MainActivity
```

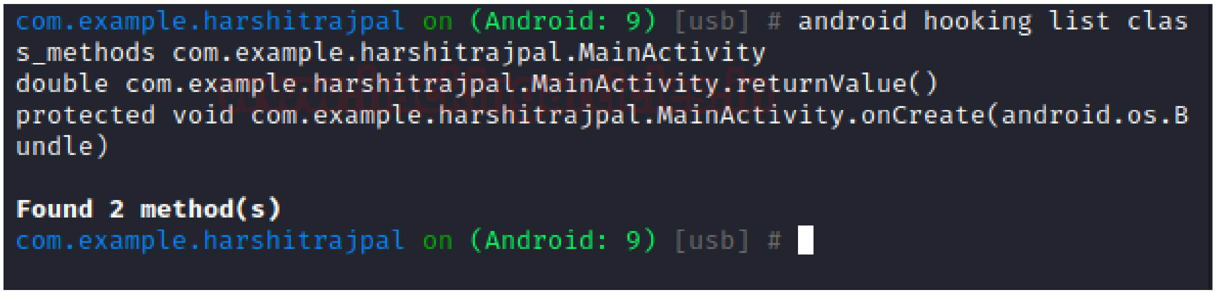
Now, this code needs to be modified only at the places of comments and we can perform whichever function we'd like the code to do so, while at run time! Interesting isn't it?



```
┌──(root💀kali)-[/home/kali/Desktop]
└─# objection --gadget com.example.harshitrajpal explore
Using USB device `Google Pixel 2`
Agent injected and responds ok!

     _   _         _   _
 ___| |_|_|___ ___| |_|_|___ ___
| . | . | | -_|  _|  _| | . |   |
|___|___| |___|___|_| |_|___|_|_|
      |___|(object)inject(ion) v1.9.6

     Runtime Mobile Exploration
        by: @leonjza from @sensepost

[tab] for command suggestions
com.example.harshitrajpal on (Android: 9) [usb] # android hooking generate
simple com.example.harshitrajpal.MainActivity

Java.perform(function() {
    var clazz = Java.use('com.example.harshitrajpal.MainActivity');
    clazz.onCreate.implementation = function() {

        //

        return clazz.onCreate.apply(this, arguments);
    }
});


Java.perform(function() {
    var clazz = Java.use('com.example.harshitrajpal.MainActivity');
    clazz.returnValue.implementation = function() {

        //

        return clazz.returnValue.apply(this, arguments);
    }
});

com.example.harshitrajpal on (Android: 9) [usb] #
```

It is worth exploring some other related functions in objection which can make our life easy. For example, exploring all the methods in the application which won't be normally visible unless we decompile the app. This goes like:

```
android hooking list class_methods com.example.harshitrajpal.MainActivity
```



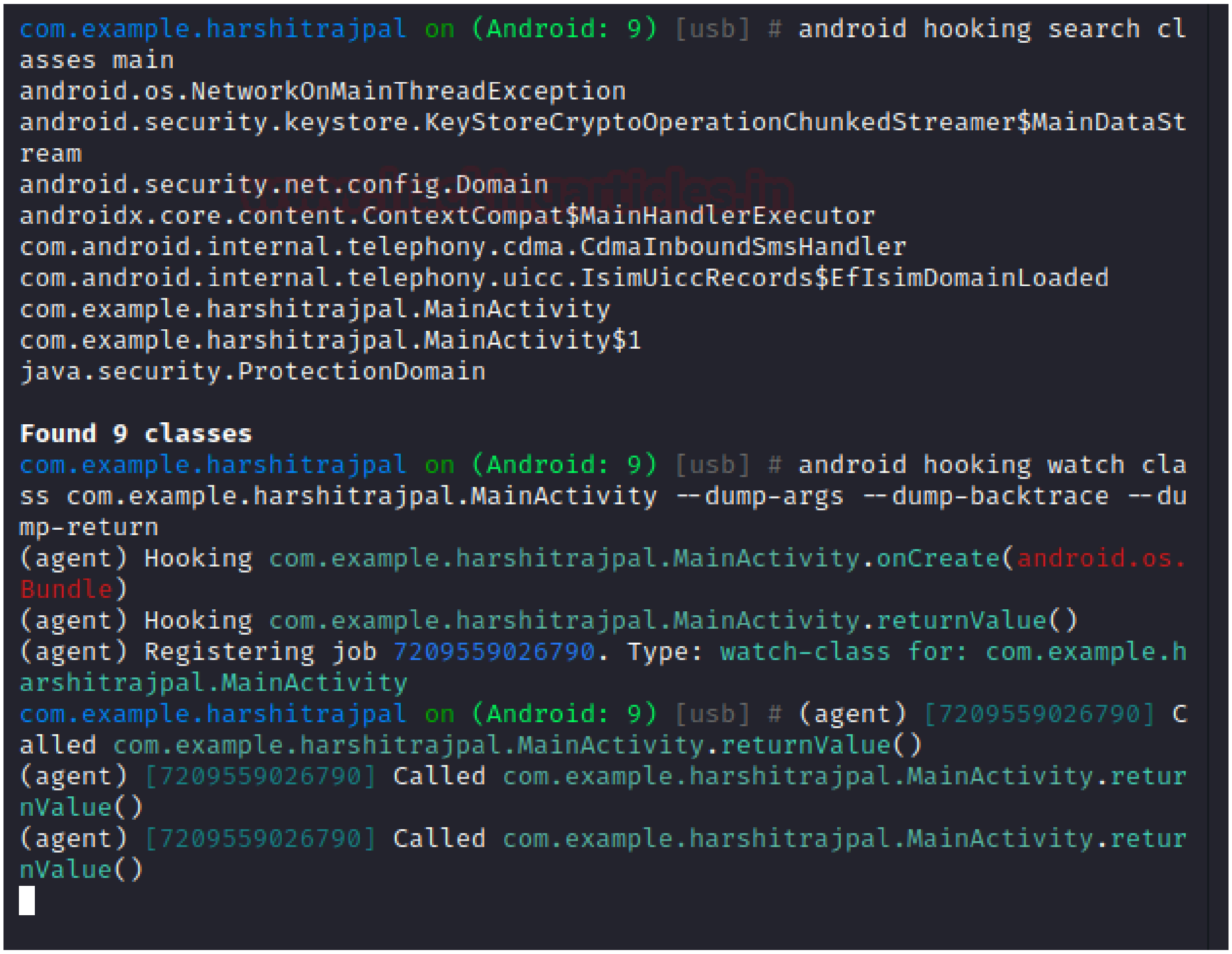
```
com.example.harshitrajpal on (Android: 9) [usb] # android hooking list clas
s_methods com.example.harshitrajpal.MainActivity
double com.example.harshitrajpal.MainActivity.returnValue()
protected void com.example.harshitrajpal.MainActivity.onCreate(android.os.B
undle)

Found 2 method(s)
com.example.harshitrajpal on (Android: 9) [usb] #
```

Similarly, we can also search for all the classes that start have the word “main” in them via the following command:

```
android hooking search classes main
```

Now, if we want to monitor a particular activity to see what all functions the activity calls and in what logical sequence to better understand how to create hooks, we’d type the following command:

```
android hooking watch class com.example.harshitrajpal.MainActivity --dump-args --d
```

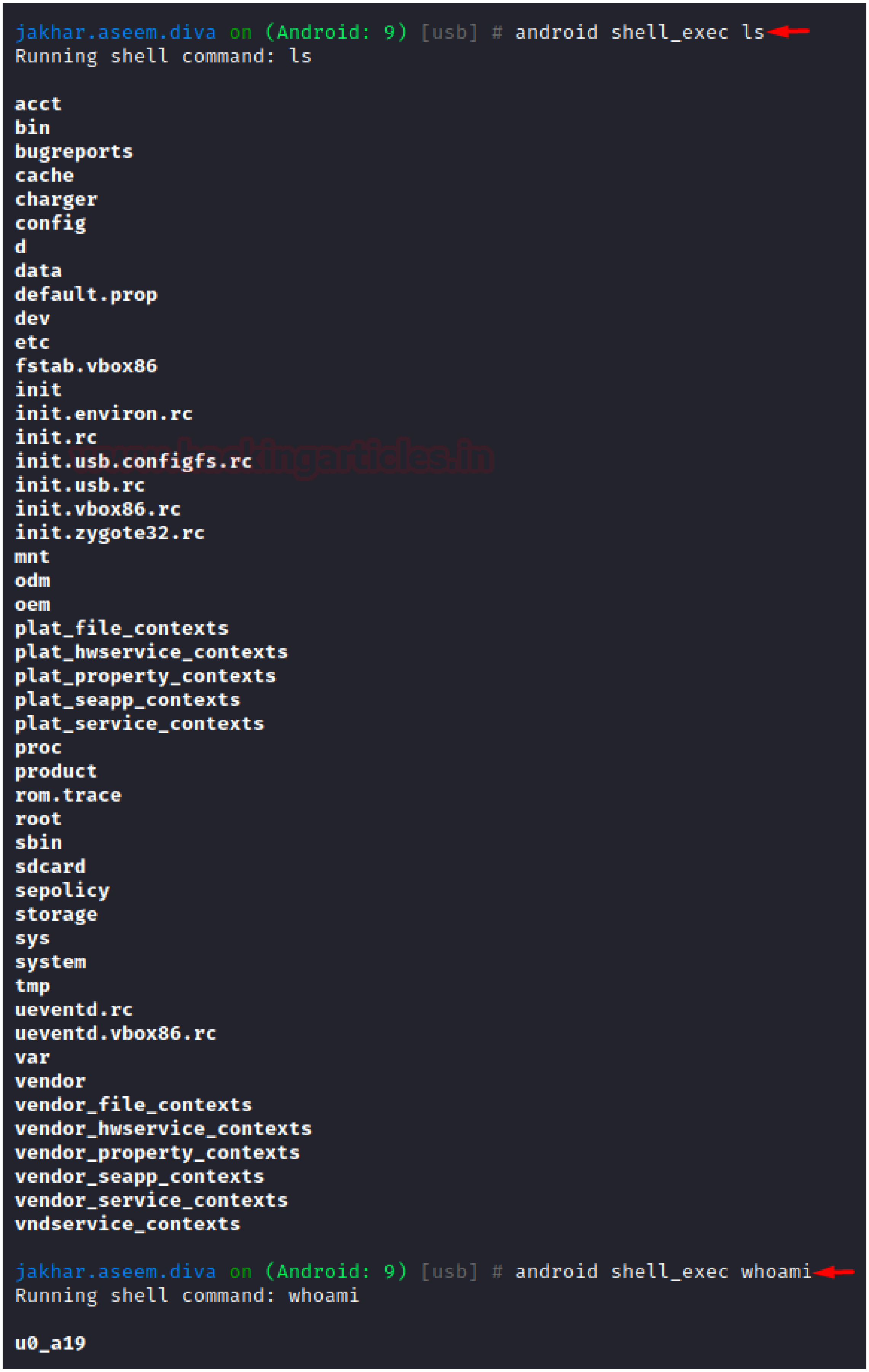


```
com.example.harshitrajpal on (Android: 9) [usb] # android hooking search cl
asses main
android.os.NetworkOnMainThreadException
android.security.keystore.KeyStoreCryptoOperationChunkedStreamer$MainDataSt
ream
android.security.net.config.Domain
androidx.core.content.ContextCompat$MainHandlerExecutor
com.android.internal.telephony.cdma.CdmaInboundSmsHandler
com.android.internal.telephony.uicc.IsimUiccRecords$EfIsimDomainLoaded
com.example.harshitrajpal.MainActivity
com.example.harshitrajpal.MainActivity$1
java.security.ProtectionDomain

Found 9 classes
com.example.harshitrajpal on (Android: 9) [usb] # android hooking watch cla
ss com.example.harshitrajpal.MainActivity --dump-args --dump-backtrace --du
mp-return
(agent) Hooking com.example.harshitrajpal.MainActivity.onCreate(android.os.
Bundle)
(agent) Hooking com.example.harshitrajpal.MainActivity.returnValue()
(agent) Registering job 7209559026790. Type: watch-class for: com.example.h
arshitrajpal.MainActivity
com.example.harshitrajpal on (Android: 9) [usb] # (agent) [7209559026790] C
alled com.example.harshitrajpal.MainActivity.returnValue()
(agent) [7209559026790] Called com.example.harshitrajpal.MainActivity.retur
nValue()
(agent) [7209559026790] Called com.example.harshitrajpal.MainActivity.retur
nValue()
```

Shell Execution

Objection is also able to execute a local shell within its own interface. It’s a handy little trick that saves the hassle of opening new tabs.

```
android shell_exec ls
android shell_exec whoami
```

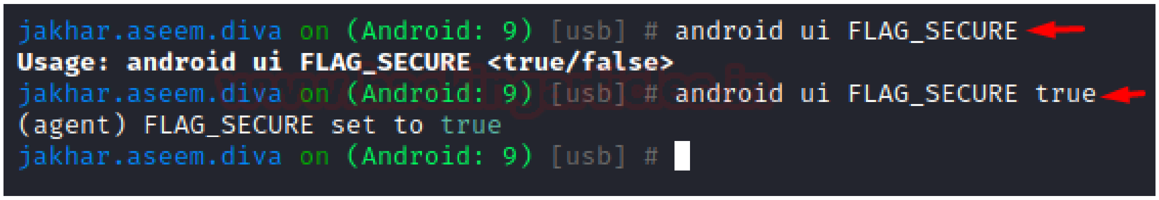


```
jakhar.aseem.diva on (Android: 9) [usb] # android shell_exec ls
Running shell command: ls

acct
bin
bugreports
cache
charger
config
d
data
default.prop
dev
etc
fstab.vbox86
init
init.environ.rc
init.rc
init.usb.configfs.rc
init.usb.rc
init.vbox86.rc
init.zygote32.rc
mnt
odm
oem
plat_file_contexts
plat_hwservice_contexts
plat_property_contexts
plat_seapp_contexts
plat_service_contexts
proc
product
rom.trace
root
sbin
sdcard
sepolicy
storage
sys
system
tmp
ueventd.rc
ueventd.vbox86.rc
var
vendor
vendor_file_contexts
vendor_hwservice_contexts
vendor_property_contexts
vendor_seapp_contexts
vendor_service_contexts
vndservice_contexts

jakhar.aseem.diva on (Android: 9) [usb] # android shell_exec whoami
Running shell command: whoami

u0_a19
```

FLAG_SECURE Bypass

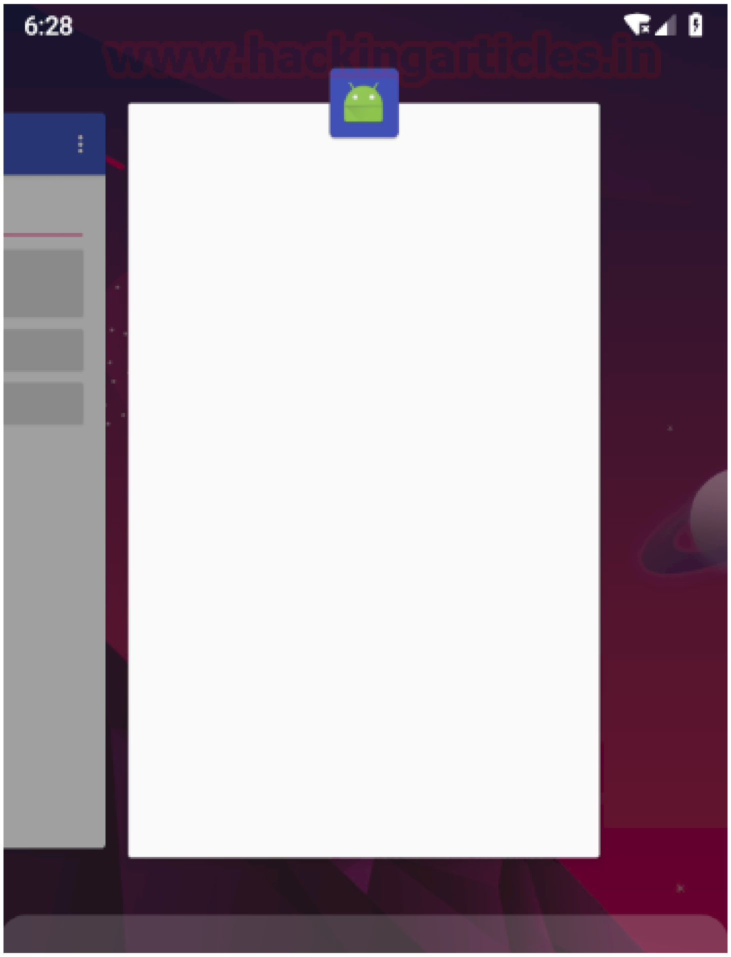
Often times android developers set a check-in their code known as “FLAG_SECURE” that prevents users to watch contents of an application while it is minimized. This is a critical function as a lot of apps don’t allow users to view materials while the app is minimized. Switching this flag would toggle that security. Objection has the ability to toggle this flag in run time. First, let’s set the flag to true and see what happens:

```
android ui FLAG_SECURE
android ui FLAG_SECURE true
```



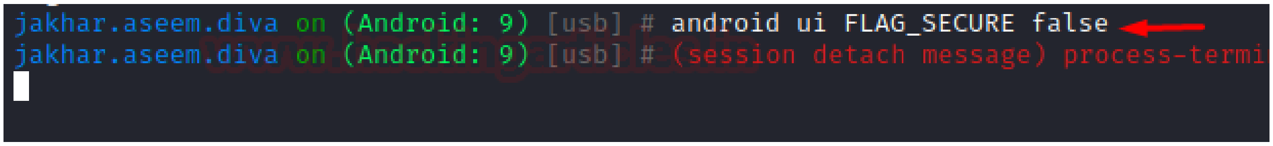
```
jakhar.aseem.diva on (Android: 9) [usb] # android ui FLAG_SECURE
Usage: android ui FLAG_SECURE <true/false>
jakhar.aseem.diva on (Android: 9) [usb] # android ui FLAG_SECURE true
(agent) FLAG_SECURE set to true
jakhar.aseem.diva on (Android: 9) [usb] #
```

Now let’s minimize the app to the drawer and see what this has done. Pretty obvious that we are not able to watch any content while it is minimized



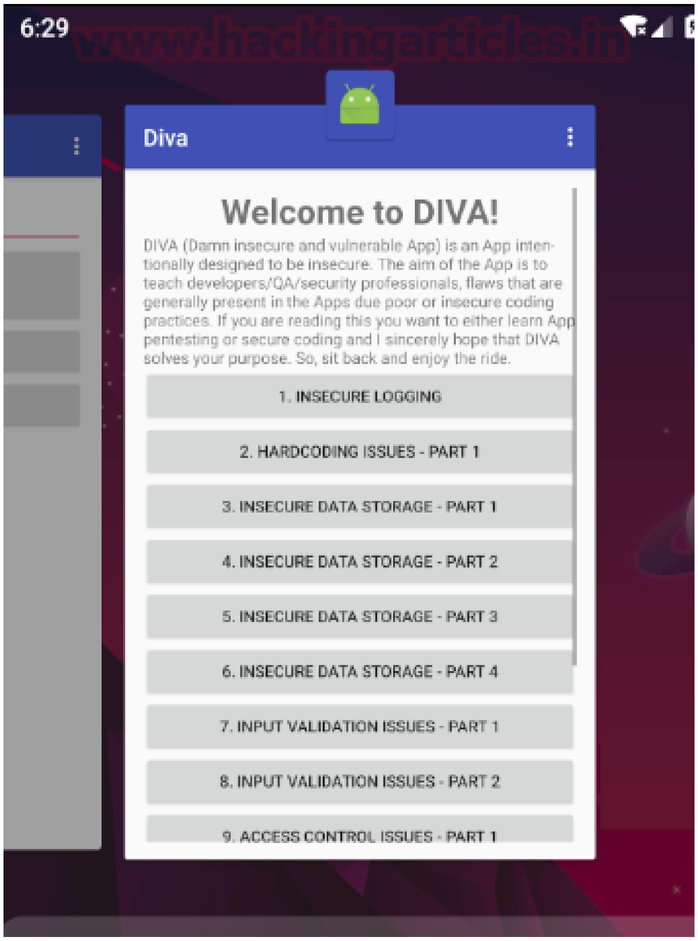
Now let’s turn the flag off

```
    android ui FLAG_SECURE false
```



```
jakhar.aseem.diva on (Android: 9) [usb] # android ui FLAG_SECURE false
jakhar.aseem.diva on (Android: 9) [usb] # (session detach message) process-termi
```

Let’s see what happens to the app now when it is minimized. Sure enough, now we are able to see the contents!



6:29

Diva

Welcome to DIVA!

DIVA (Damn insecure and vulnerable App) is an App intentionally designed to be insecure. The aim of the App is to teach developers/QA/security professionals, flaws that are generally present in the Apps due poor or insecure coding practices. If you are reading this you want to either learn App pentesting or secure coding and I sincerely hope that DIVA solves your purpose. So, sit back and enjoy the ride.

1. INSECURE LOGGING

2. HARDCODING ISSUES - PART 1

3. INSECURE DATA STORAGE - PART 1

4. INSECURE DATA STORAGE - PART 2

5. INSECURE DATA STORAGE - PART 3

6. INSECURE DATA STORAGE - PART 4

7. INPUT VALIDATION ISSUES - PART 1

8. INPUT VALIDATION ISSUES - PART 2

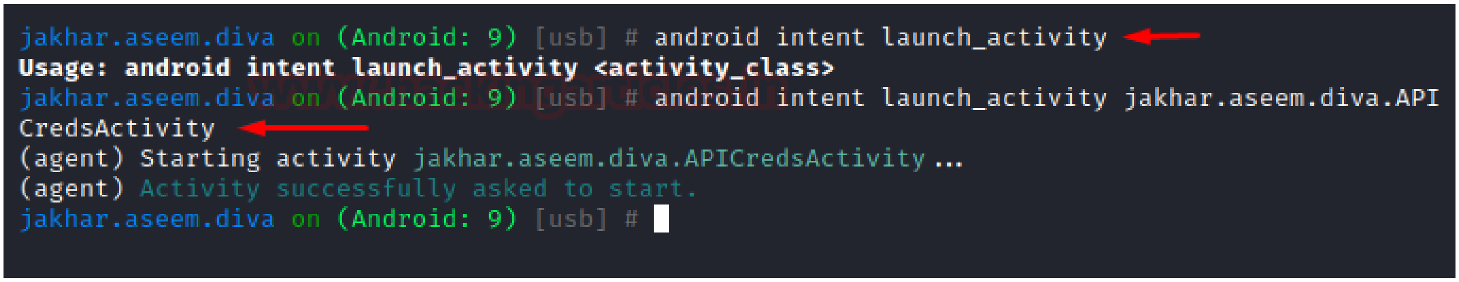
9. ACCESS CONTROL ISSUES - PART 1

Launching Activity Using Objection

Objection can also be used to launch an activity from within its interface. This is the same as gdb’s am tool with -n option as we saw in the **previous** article. To start an activity, we type:

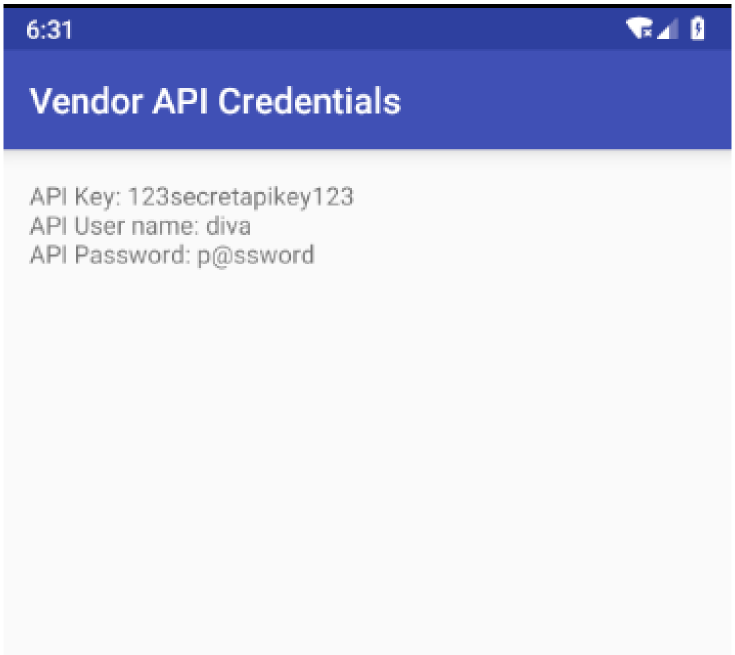
```
    android intent launch_activity
    android intent launch_activity jakhar.aseem.diva.APICredsActivity
```

Note, to launch DIVA’s activity, please attach its gadget first



```
jakhar.aseem.diva on (Android: 9) [usb] # android intent launch_activity
Usage: android intent launch_activity <activity_class>
jakhar.aseem.diva on (Android: 9) [usb] # android intent launch_activity jakhar.aseem.diva.API
CredsActivity
(agent) Starting activity jakhar.aseem.diva.APICredsActivity...
(agent) Activity successfully asked to start.
jakhar.aseem.diva on (Android: 9) [usb] #
```

And sure enough, the activity has now started



Vendor API Credentials

API Key: 123secretapikey123
API User name: diva
API Password: p@ssword

Root Detection Bypass

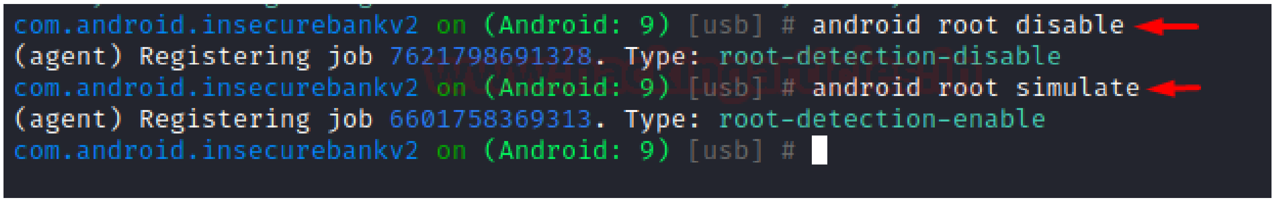
Root detection is a feature that developers code in their application to restrict their apps from running on rooted devices for security purposes. There are again two methods to bypass root detection, one is through reverse-engineering the application and the other is to use a tool like objection.

To do this in objection, I'll be using the following commands:

```
android root disable
```

To simulate root detection again:

```
android root simulate
```



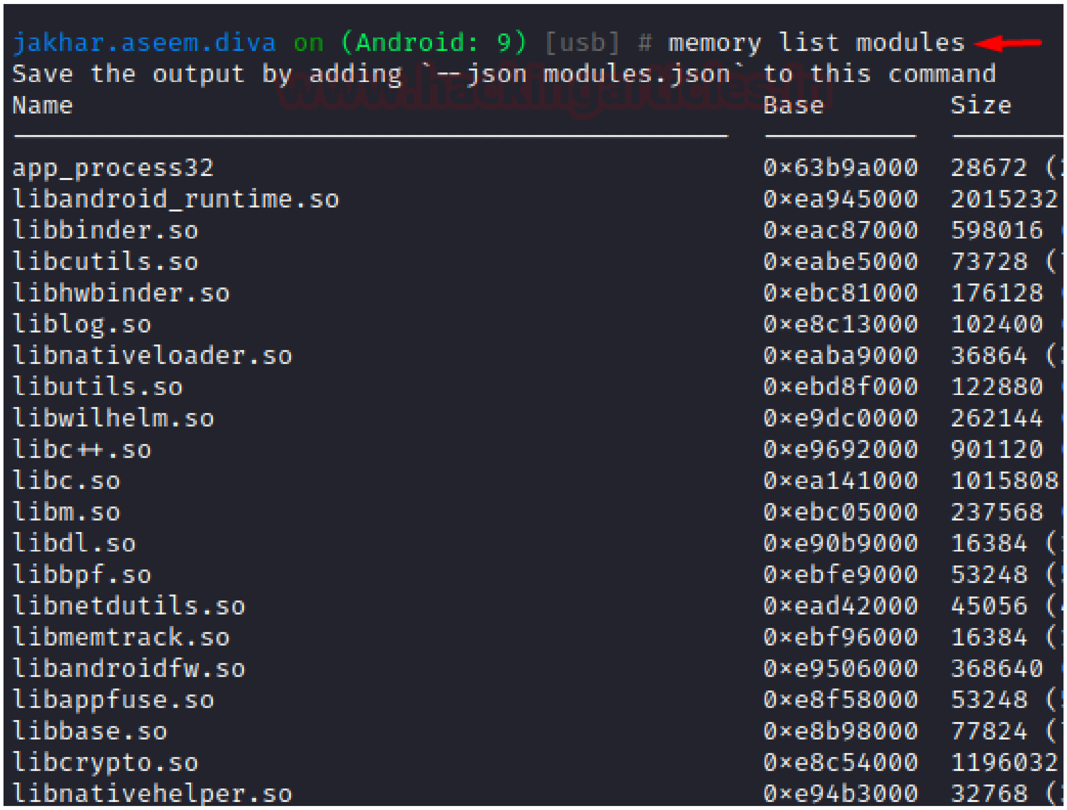
```
com.android.insecurebankv2 on (Android: 9) [usb] # android root disable
(agent) Registering job 7621798691328. Type: root-detection-disable
com.android.insecurebankv2 on (Android: 9) [usb] # android root simulate
(agent) Registering job 6601758369313. Type: root-detection-enable
com.android.insecurebankv2 on (Android: 9) [usb] #
```

Memory Related Tasks

To play with low-level memory, objection has a handy tool called memory module that allows a person to see what a process is exporting to memory, where it is writing and more!

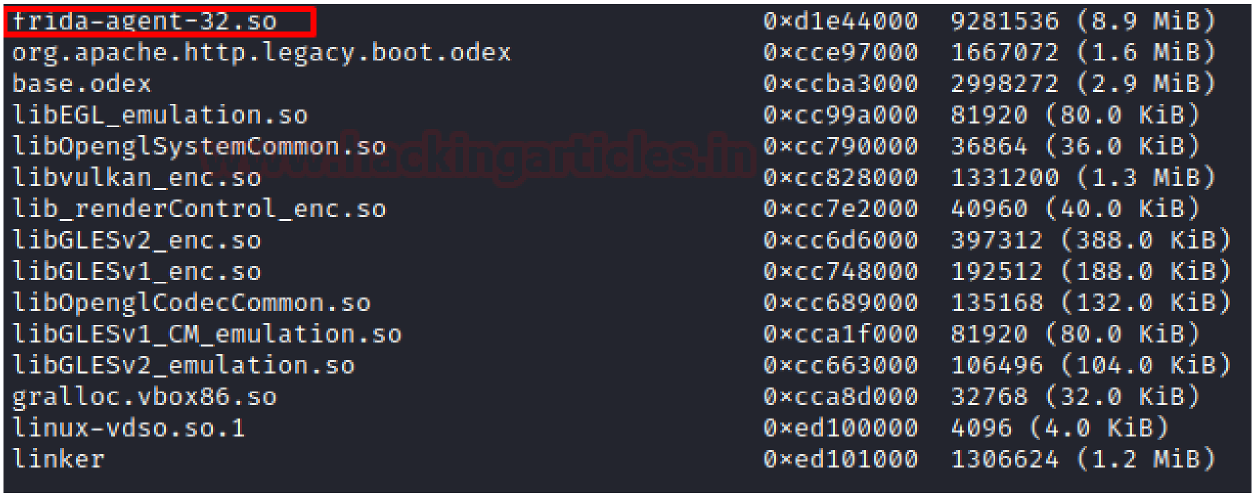
Here, let’s see what all modules are loaded into the memory while the gadget is attached

```
memory list modules
```



```
jakhar.aseem.diva on (Android: 9) [usb] # memory list modules
Save the output by adding `--json modules.json` to this command
Name                                              Base         Size
────────────────────────────────────────────────  ───────────  ──────
app_process32                                     0×63b9a000   28672 (
libandroid_runtime.so                             0×ea945000   2015232
libbinder.so                                      0×eac87000   598016
libcutils.so                                      0×eabe5000   73728 (
libhwbinder.so                                    0×ebc81000   176128
liblog.so                                         0×e8c13000   102400
libnativeloader.so                                0×eaba9000   36864 (
libutils.so                                       0×ebd8f000   122880
libwilhelm.so                                     0×e9dc0000   262144
libc++.so                                         0×e9692000   901120
libc.so                                           0×ea141000   1015808
libm.so                                           0×ebc05000   237568
libdl.so                                          0×e90b9000   16384 (
libbpf.so                                         0×ebfe9000   53248 (
libnetdutils.so                                   0×ead42000   45056 (
libmemtrack.so                                    0×ebf96000   16384 (
libandroidfw.so                                   0×e9506000   368640
libappfuse.so                                     0×e8f58000   53248 (
libbase.so                                        0×e8b98000   77824 (
libcrypto.so                                      0×e8c54000   1196032
libnativehelper.so                                0×e94b3000   32768 (
```

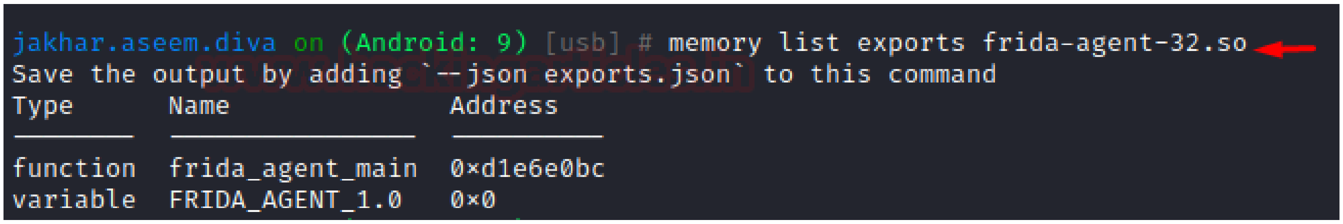
At the bottom, we would see a Frida module as well.



```
frida-agent-32.so                                 0×d1e44000   9281536 (8.9 MiB)
org.apache.http.legacy.boot.odex                  0×cce97000   1667072 (1.6 MiB)
base.odex                                         0×ccba3000   2998272 (2.9 MiB)
libEGL_emulation.so                               0×cc99a000   81920 (80.0 KiB)
libOpenglSystemCommon.so                          0×cc790000   36864 (36.0 KiB)
libvulkan_enc.so                                  0×cc828000   1331200 (1.3 MiB)
lib_renderControl_enc.so                          0×cc7e2000   40960 (40.0 KiB)
libGLESv2_enc.so                                  0×cc6d6000   397312 (388.0 KiB)
libGLESv1_enc.so                                  0×cc748000   192512 (188.0 KiB)
libOpenglCodecCommon.so                           0×cc689000   135168 (132.0 KiB)
libGLESv1_CM_emulation.so                         0×cca1f000   81920 (80.0 KiB)
libGLESv2_emulation.so                            0×cc663000   106496 (104.0 KiB)
gralloc.vbox86.so                                 0×cca8d000   32768 (32.0 KiB)
linux-vdso.so.1                                   0×ed100000   4096 (4.0 KiB)
linker                                            0×ed101000   1306624 (1.2 MiB)
```

Now, to see what Frida is exporting and its respective address:

```
memory list exports Frida-agent-32.so
```

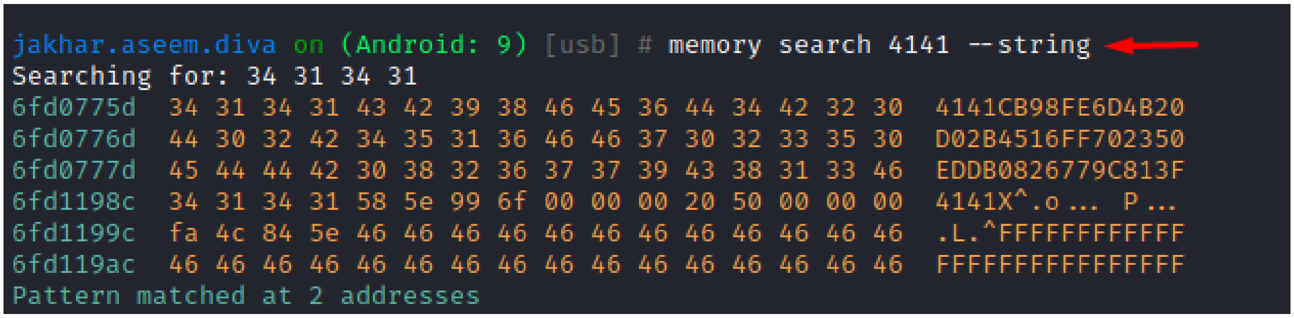


```
jakhar.aseem.diva on (Android: 9) [usb] # memory list exports frida-agent-32.so
Save the output by adding `--json exports.json` to this command
Type      Name              Address
--------  ----------------  ----------

function  frida_agent_main  0×d1e6e0bc
variable  FRIDA_AGENT_1.0   0×0
```

Similarly, we can also search a particular string in the memory and even write over it. To search a string we type:

```
memory search 4141 –string
```

And we'd see objection has returned the memory address of the memory block containing the defined string.



```
jakhar.aseem.diva on (Android: 9) [usb] # memory search 4141 --string
Searching for: 34 31 34 31
6fd0775d  34 31 34 31 43 42 39 38 46 45 36 44 34 42 32 30  4141CB98FE6D4B20
6fd0776d  44 30 32 42 34 35 31 36 46 46 37 30 32 33 35 30  D02B4516FF702350
6fd0777d  45 44 44 42 30 38 32 36 37 37 39 43 38 31 33 46  EDDB0826779C813F
6fd1198c  34 31 34 31 58 5e 99 6f 00 00 00 20 50 00 00 00  4141X^.o... P...
6fd1199c  fa 4c 84 5e 46 46 46 46 46 46 46 46 46 46 46 46  .L.^FFFFFFFFFFFF
6fd119ac  46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46  FFFFFFFFFFFFFFFF
Pattern matched at 2 addresses
```

To write over the memory we have the following command:

```
memory write <address> <string> --string
```

I'll not be writing into the memory right now and in-depth coverage of memory-related tasks would be covered in detail in an upcoming article.

Conclusion

In this article, we saw a handy tool objection that automates various tasks while doing static and dynamic APK analysis. We also got acquainted with SSLPinning bypass and android hooking. Hope you liked the article.