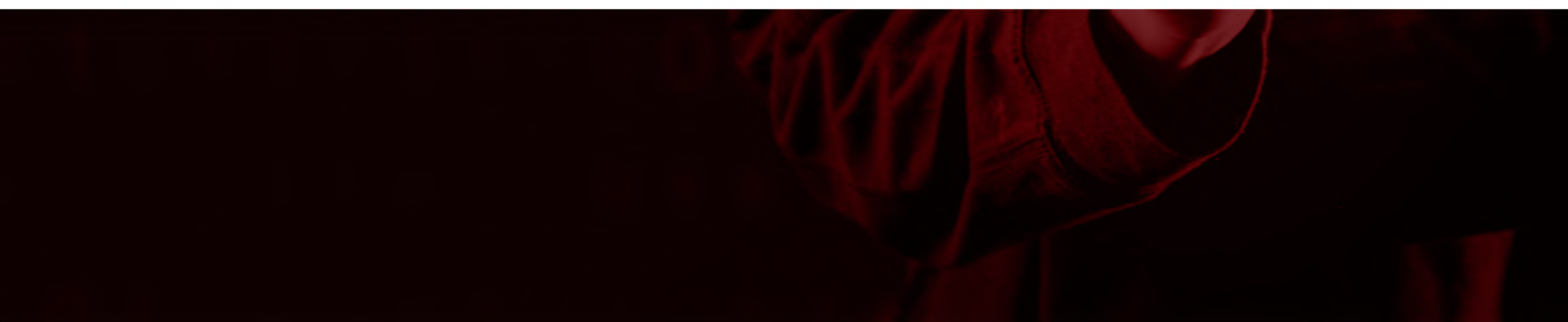
FRIDA:

A Guide to Android App Hooking

Payatu



About the Author

Anubhav Singh

Anubhav Singh, known online by the alias @dn0m1n8tor, works as a Security Engineer. He is a dedicated learner, always eager to explore and work across various technologies and domains. His expertise lies in identifying security issues in diverse applications, including web, API, mobile, cloud, and thick client. He actively contributes to the infosec community. He is always excited to learn, experiment, and collaborate with like-minded individuals to broaden his knowledge spectrum.

Table of Contents

Chapter 1

What is Frida?

Frida is a dynamic code instrumentation toolkit for developers, reverse engineers, and security researchers. It lets you inject snippets of JavaScript or your own library into a process from which you can interactively inspect and change running processes. Frida is available for many platforms, such as Windows, macOS, GNU/Linux, iOS, watchOS, tvOS, Android, FreeBSD, and QNX. It also provides you with some simple tools built on top of the Frida API.

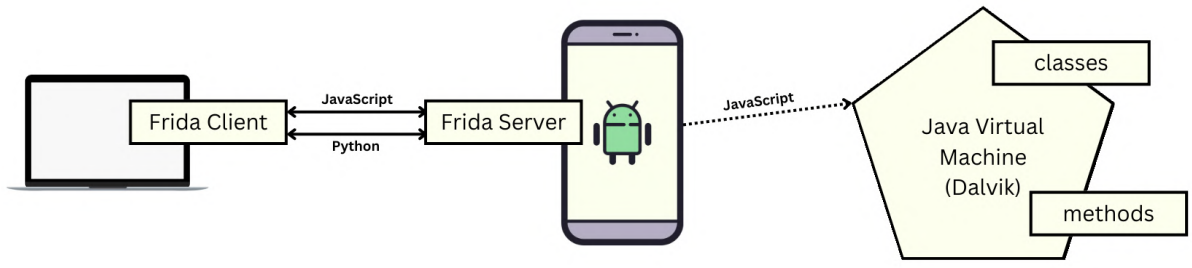
In this eBook, we will discuss Frida for Android, but the concept remains the same for other platforms.

Frida allows you to interact with the Android APK so that you can inject JavaScript snippets to bypass many of the techniques developers use to secure the apps. Some examples include bypassing the login screen to authenticate without a password or disabling SSL pinning to allow the hacker to see all the network traffic between your app and any backend servers. Many of the method calls you make in your Android app can be hijacked or overridden by Frida that were never intended. By injecting JavaScript code, Frida can disable or enable a switch or pass fake parameters and change the flow of a method to gain access to information that would not otherwise be available.

How Frida Works?

Java Virtual Machine (JVM) manages the whole Java Code, for example, if you write an application code in Java, then all the code which contains classes, methods, instances, variables, etc., will be held in JVM. This JVM is running on your Android phone, and it stores all the application codes in it. Suppose you are using a class in the application, such as an Activity class, then the Activity class will be loaded from Java Virtual Machine.

What Frida does here is it hooks into the JVM and performs any runtime manipulation in the code, such as listing out all the methods of a class, passing a different parameter to a method, seeing the parameters that are getting passed, changing the return value, checking the return value, etc.



Frida consists of two components Frida Client and Frida Server. Frida Server is a binary file that we need to run on an Android device, and Frida Client should be installed on your laptop.

To communicate between Client and Server, we can use JavaScript or Python (we will use JavaScript). Now the server hooks into the JVM, and this is where all the Java Code is present. Now the Frida server can modify or change the code in a runtime as it has access to classes, methods, variables, etc.

Frida is written in C and injects a JavaScript engine into the app. Now what does that mean?

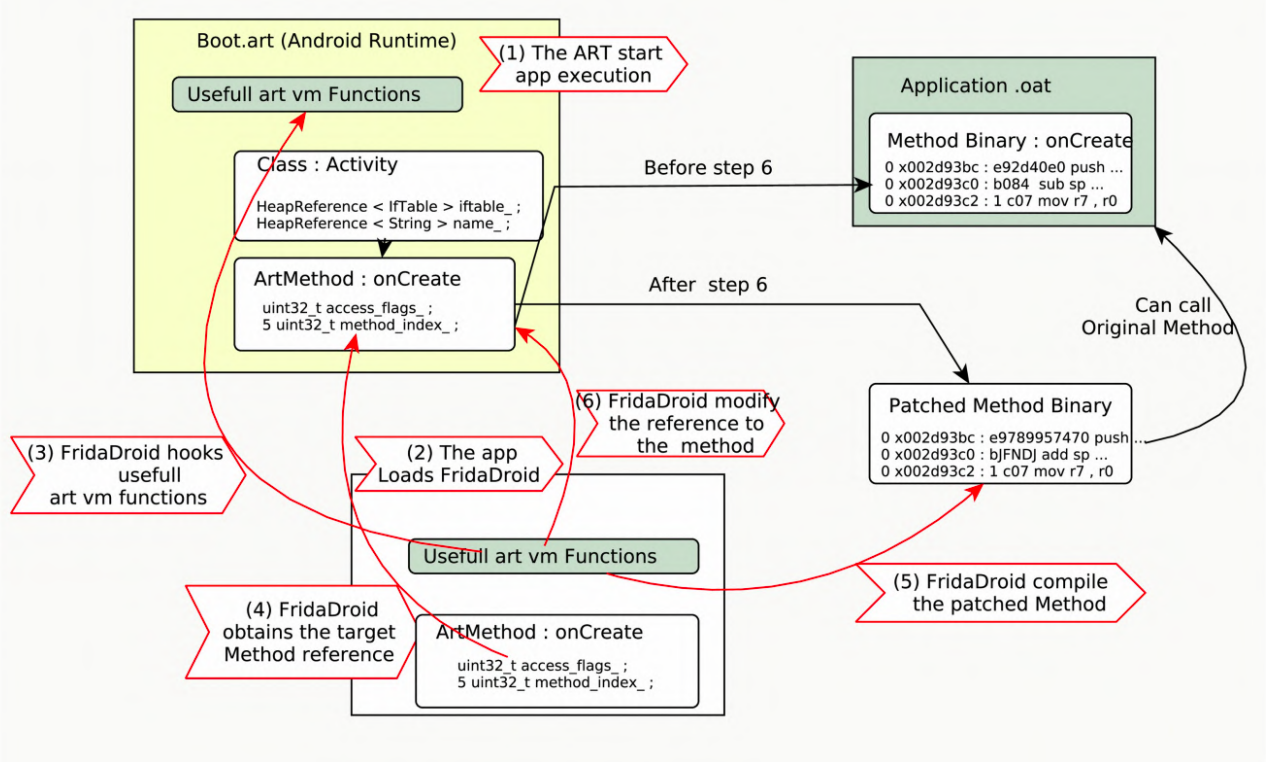
When the application is running, it has the following:
1. Application code
2. Shared libraries in memory that are used to access native functions of Android like JAVA.Android.util.log, JAVA.Android.util.sharedpreferences

This type of API call is embedded in those shared libraries. Now what Frida does here is, it injects its own shared library into the application.

But why does Frida do that?

Because as you know both iOS and Android implement sandboxing, so one application, while running, can't access the memory along with the file of another application. Frida injects that shared library into the memory in process and then the shared library gets access to the whole memory space of that process. Frida is then able to see other shared libraries it holds, what kind of dependency it has, what system call it makes, etc.

Let's Understand the Working of Frida in Detail



Suppose you have a device in which Frida and an application is installed and you want to hook a method from the application.

1. When you run the application (as you know, ART runs the application), all the useful ART VM functions get loaded.
2. The application loads the Frida library (fridadroid / frida agent).
3. After loading the Frida library, the fridaDroid/frida agent hooks all the useful ART VM functions.
4. After hooking all the virtual functions, it opens the target method reference (Let's say, if we want to hook any specific method, Frida obtains the method reference by some pointer, and after getting the reference, it hooks that specific method).
5. After hooking a specific method, it compiles and patches the method and creates a binary method (patching the method means writing a Frida script to modify it).
6. As a binary is created, it gets executed. After execution, it can call the original method, making the application run normally.

Installation

You need to install two things to set up Frida to work.

1. Frida Client

command: pip install frida-tools

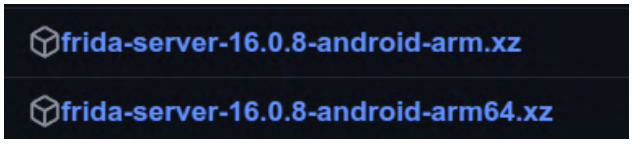
```
❯ pip install frida-tools
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: frida-tools in /home/anubhav/.local/lib/python3.8/site-packages (12.0.1)
Collecting frida<17.0.0,>=16.0.0
  Downloading frida-16.0.8-cp37-abi3-manylinux_2_5_x86_64.manylinux1_x86_64.whl (19.2 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 19.2/19.2 MB 5.6 MB/s eta 0:00:00
```

2. Frida Server

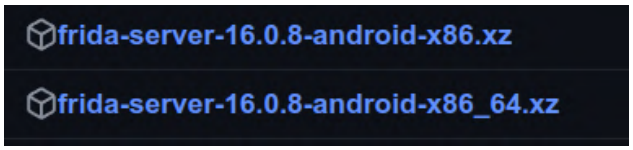
Download the Frida server for your Android platform from this Github Repo.

NOTE:

- If you are using a physical Android device, then you need to install an ARM-based Frida server. You can choose any one of these Frida servers that matches the architecture of your phone.



- If you are using any emulator devices like Genymotion or Android Emulator, then you need to install the Frida server based on your architecture.



- You can confirm your device architecture using the following command.

```
❯ adb shell getprop ro.product.cpu.abi
x86
```

After downloading the appropriate Frida server, move it to your device and run the server.

Now you are ready to explore the power of Frida.

Chapter 2

Modes of Operation

There are 3 modes of operation that Frida provides us with

1. Injected

2. Embedded

3. Preloaded

1. Injected

Suppose you want to spawn an existing program, attach it to a running program, or hijack one as it's being spawned, and then run your instrumentation logic inside of it then we can use injected mode. As this is such a common way to use Frida, it is what most of the documentation focuses on. We have already seen how to run the Frida server using the injected mode in the installation part. One disadvantage of the injected mode is that it requires a rooted device.

2. Embedded

If you are using an Android device which is not rooted, then it's not possible to run the injected mode. In this case, we need to use the embedded mode of operation, which doesn't require a rooted device, but here we need to patch the app with "frida-gadget" (a shared library) to be embedded with your app.

We can perform embedded operations in two ways

1. Using Objection
2. Manual

We will look into both ways

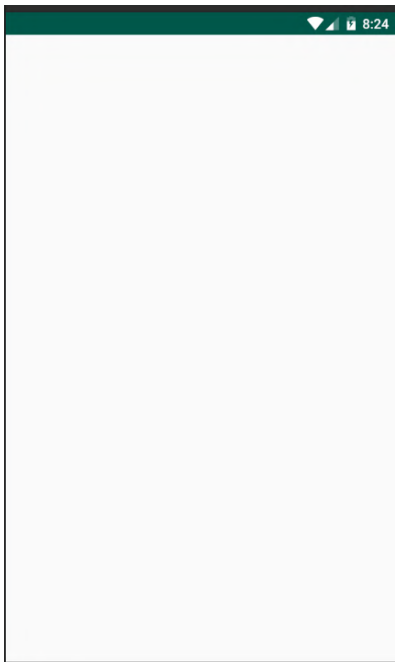
Using Objection

1. Patch the APK file with objection using the following command

Command: `objection patchapk -source <apk file name>`

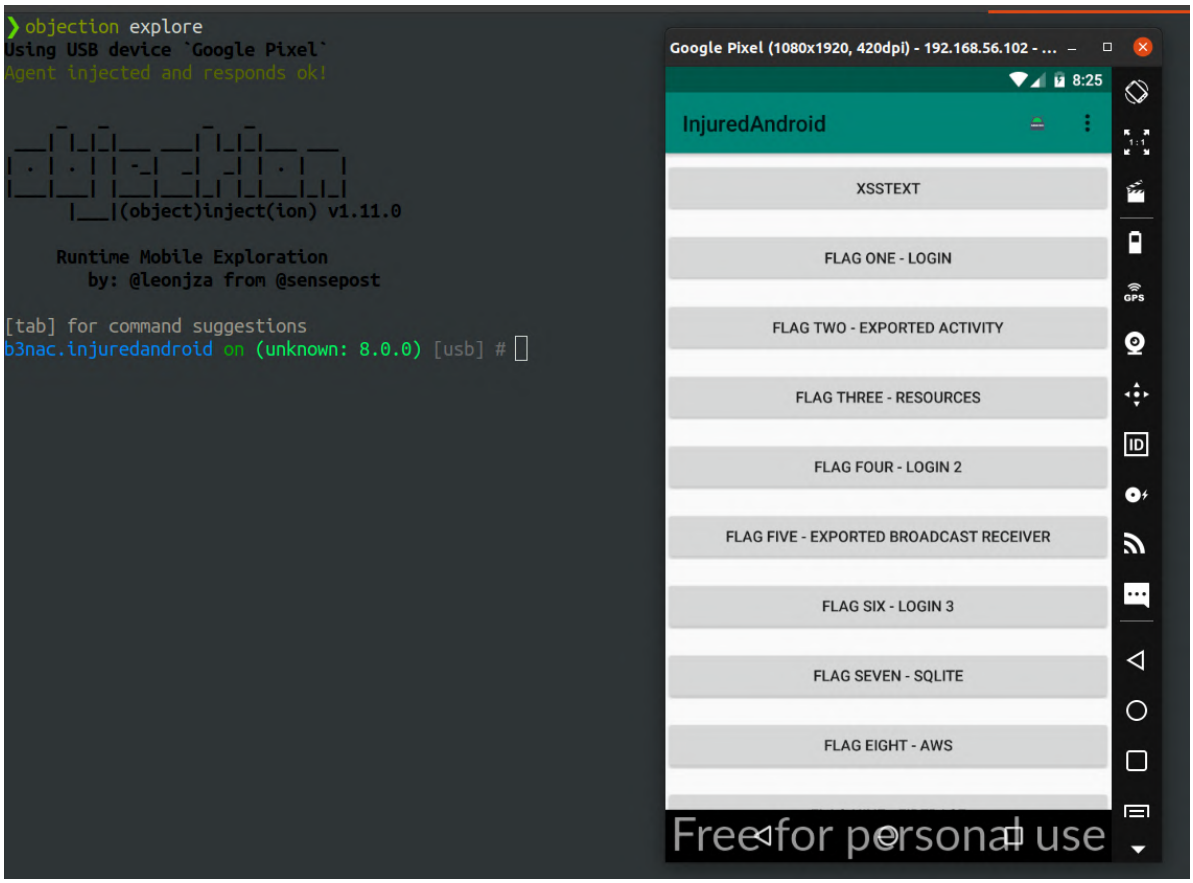
```
objection patchapk --source InjuredAndroid-1.0.12-release.apk
No architecture specified. Determining it using `adb`...
Detected target device architecture as: x86
Using latest Github gadget version: 16.0.8
Patcher will be using Gadget version: 16.0.8
Detected apktool version as: 2.4.1
Running apktool empty-framework-dir...
I: Removing 1.apk framework file...
Unpacking InjuredAndroid-1.0.12-release.apk
App already has android.permission.INTERNET
Target class not specified, searching for launchable activity instead...
Reading smali from: /tmp/tmp8jzpzsmm.apktemp/smali/b3nac/injuredandroid/MainActivity.smali
Injecting loadLibrary call at line: 10
Attempting to fix the constructors .locals count
Current locals value is 0, updating to 1:
Writing patched smali back to: /tmp/tmp8jzpzsmm.apktemp/smali/b3nac/injuredandroid/MainActivity.smali
Creating library path: /tmp/tmp8jzpzsmm.apktemp/lib/x86
Copying Frida gadget to libs path...
Rebuilding the APK with the frida-gadget loaded...
Built new APK with injected loadLibrary and frida-gadget
Performing zipalign
Zipalign completed
Signing new APK.
Signed the new APK
Copying final apk from /tmp/tmp8jzpzsmm.apktemp.aligned.objection.apk to InjuredAndroid-1.0.12-release.objection.apk in current directory...
Cleaning up temp files...
```

2. The APK file is patched now. Open the application, and you will notice that the application is stuck.



3. Use objection to resume the application.

command: `objection explore`

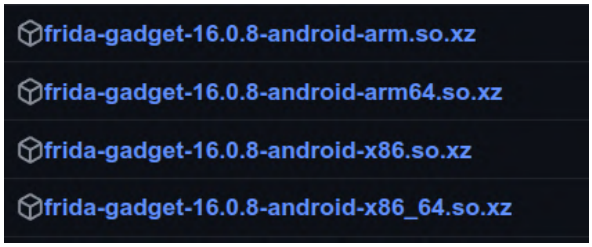


Manual

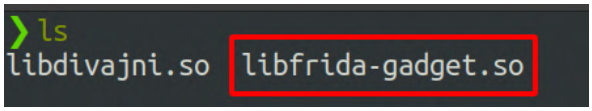
1. Decompile the application using the apktool.

```
❯ apktool d InjuredAndroid-1.0.12-release.apk
I: Using Apktool 2.4.1 on InjuredAndroid-1.0.12-release.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: /home/anubhav/.local/share/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
I: Copying META-INF/services directory
```

2. Download frida-gadget according to the architecture of your mobile. Now, download the x86 file as the device used to demonstrate is of x86 architecture.



3. Go to the lib/x86 folder of your decompiled APK and place the frida-gadget in that folder by naming it as "lib<name of frida gadget file>"



NOTE: After downloading the frida-gadget file from GitHub, rename that file to “frida-gadget.so”

1. Inject a System.loadLibrary("frida-gadget") call into the bytecode of the app, ideally before any other bytecode executes or any native code is loaded. A suitable place is typically the static initializer of the entry point classes of the app, e.g. the main application activity, found via the manifest.

We need to make some changes in the MainActivity of the application as the file gets loaded when the application starts.

For the Injured Android application, you need to go to “/smali/b3nac/injuredAndroid” and change the file content of MainActivity.smali

```
.class public final Lb3nac/injuredandroid/MainActivity;
.super Landroidx/appcompat/app/c;
.source ""


# instance fields
.field private w:I


# direct methods
.method public constructor <init>()V
    .locals 0

    invoke-direct {p0}, Landroidx/appcompat/app/c;-><init>()V

    const-string v0, "frida-gadget"
invoke-static {v0}, Ljava/lang/System;->loadLibrary(Ljava/lang/String;)V

    return-void
.end method

.method private final F()V
    .locals 3

    invoke-virtual {p0}, Landroid/app/Activity;->getIntent()Landroid/content/Intent;

    move-result-object v0

    const-string v1, "intent"
```

You need to paste the following 2 lines at the start of a method

```
const-string v0, "frida-gadget"
invoke-static {v0}, Ljava/lang/System;->loadLibrary(Ljava/lang/String;)V
```

These two lines are going to invoke shared object when the application starts.

1. Save the file
2. Add the Internet permission to the manifest if it's not there already so that the frida-gadget can open a socket.

```
<uses-permission android:name="android.permission.INTERNET" />
```

3. Repackage the application:

```
❯ apktool b InjuredAndroid-1.0.12-release -o injured_Patched.apk
I: Using Apktool 2.4.1
I: Checking whether sources has changed...
I: Checking whether resources has changed...
I: Copying libs... (/lib)
I: Building apk file...
I: Copying unknown files/dir...
I: Built apk...
```

4. Sign the updated APK using your own keys and zipalign.

A. **If you don't have a keystore already, here's how to create one**

```
$ keytool -genkey -v -keystore demo.keystore -alias demokeys  -keyalg RSA -keysize
2048 -validity 10000
```

When the password is asked, 123456 is given

B. **Sign the APK**

```
$ jarsigner -sigalg SHA1withRSA -digestalg SHA1 -keystore demo.keystore -storepass
123456 injured_Patched.apk demokeys
```

C. **Verify the signature you just created**

```
$ jarsigner -verify injured_Patched.apk
```

D. **Zipalign the APK**

```
$ zipalign 4 injured_Patched.apk injured_Patchedfinal.apk
```

5. Install the updated application to a device
6. The APK file is now patched. Open the application and you will notice that the application is stuck.
7. Use objection to resume the application.

```
$ objection explore
```

3. Preloaded

Perhaps you're familiar with LD_PRELOAD, or DYLD_INSERT_LIBRARIES? Wouldn't it be cool if there was JS_PRELOAD?

JS_PRELOAD can be used to preload a script that will be executed before the main application starts. This can be used to inject code into a running application or to modify the behavior of the application. This is where frida-gadget, the shared library discussed in the previous section, is useful when configured to run autonomously by loading a script from the filesystem.

Chapter 3

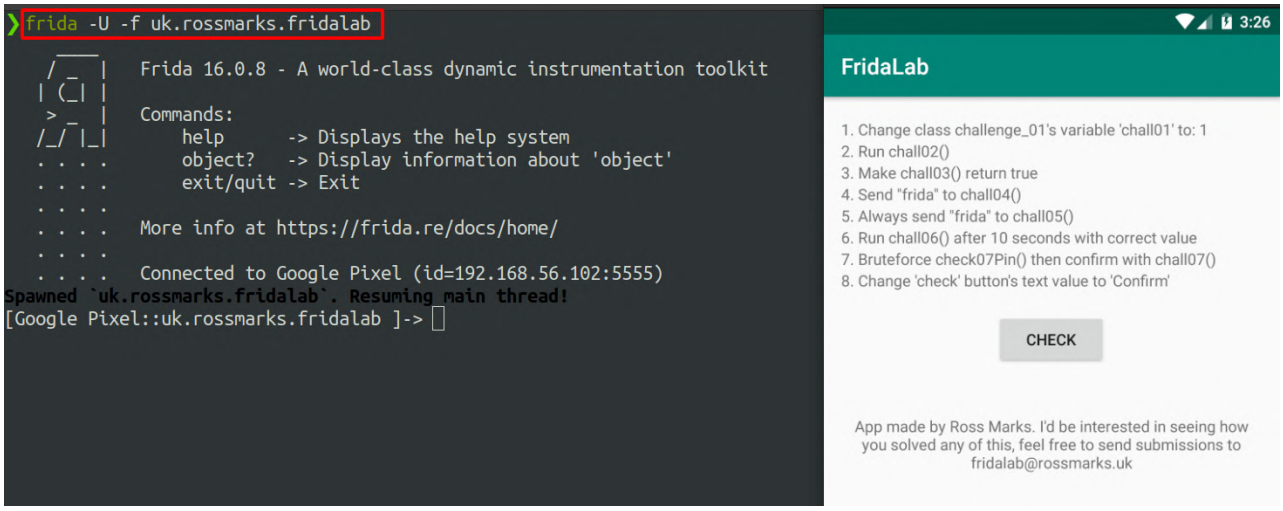
Frida Toolkit

Frida Tools are the Frida Components that are used to Discover/List/kill process, connecting, listing devices, Monitoring/tracing function calls of the application, etc.

1. Frida CLI
2. frida-ps
3. frida-trace
4. frida-discover
5. frida-ls-devices
6. frida-kill

1. Frida CLI

It is a REPL (read eval print loop) interface that aims to emulate a lot of the nice features of IPython (or Cycript), which tries to get you closer to your code for rapid prototyping and easy debugging. In simple words frida-cli is the one from which we will attach/spawn the process.



2. frida-ps

It is used to list the processes running on your installed device.

U: We are communicating with the Frida server over USB

a: list only applications

i: include all installed applications

```
frida-ps -Uai
 PID  Name                             Identifier
4  ---------------------------------  ------------------------------
5185  Email                            com.android.email
5128  Files                            com.android.documentsui
7580  FridaLab                         uk.rossmarks.fridalab
   -  Amaze                            com.amaze.filemanager
   -  Calculator                       com.android.calculator2
   -  Calendar                         com.android.calendar
   -  Camera                           com.android.camera2
   -  Clock                            com.android.deskclock
   -  Contacts                         com.android.contacts
   -  Custom Locale                    com.android.customlocale2
   -  Dev Tools                        com.android.development
   -  Development Settings             com.android.development_settings
```

3. frida-trace

It is a tool for dynamically tracing function calls.

Suppose we have an application in which we have a class, and in that class there are some methods that we want to monitor at run time. Monitor here means when you click on any button then what methods gets called or when we click on any event then what methods are getting called. So to monitor and debug all the methods on a specific class or all classes we can use Frida trace.

Suppose we have an application with a class containing methods that we want to monitor at runtime. Here, "monitor" means tracking which methods are called when any button is clicked or any event is triggered. To monitor and debug all the methods in a specific class or across all classes, we can use Frida trace.

This won't work for high level function calls such as loadLibrary() from java.lang.system() class in Java.
1) You want to hook a specific method of a class in an application

```
$ frida-trace -U -f <package name> -j 'classname!methodname'
```

2) You want to hook all methods of a class in an application

```
$ frida-trace -U -f <package name> -j 'classname!*'
```

3) You want to hook all methods of all class in an application

```
$ frida-trace -U -f <package name> -j '*!*'
```

TIP: If you want to hook a particular method of class and frida-trace is not able to find you the name of the package then you can try to provide running PID of that application.

Example: You want to trace `getChall01Int` method from `challenge_01` class then you can write the command as shown below.

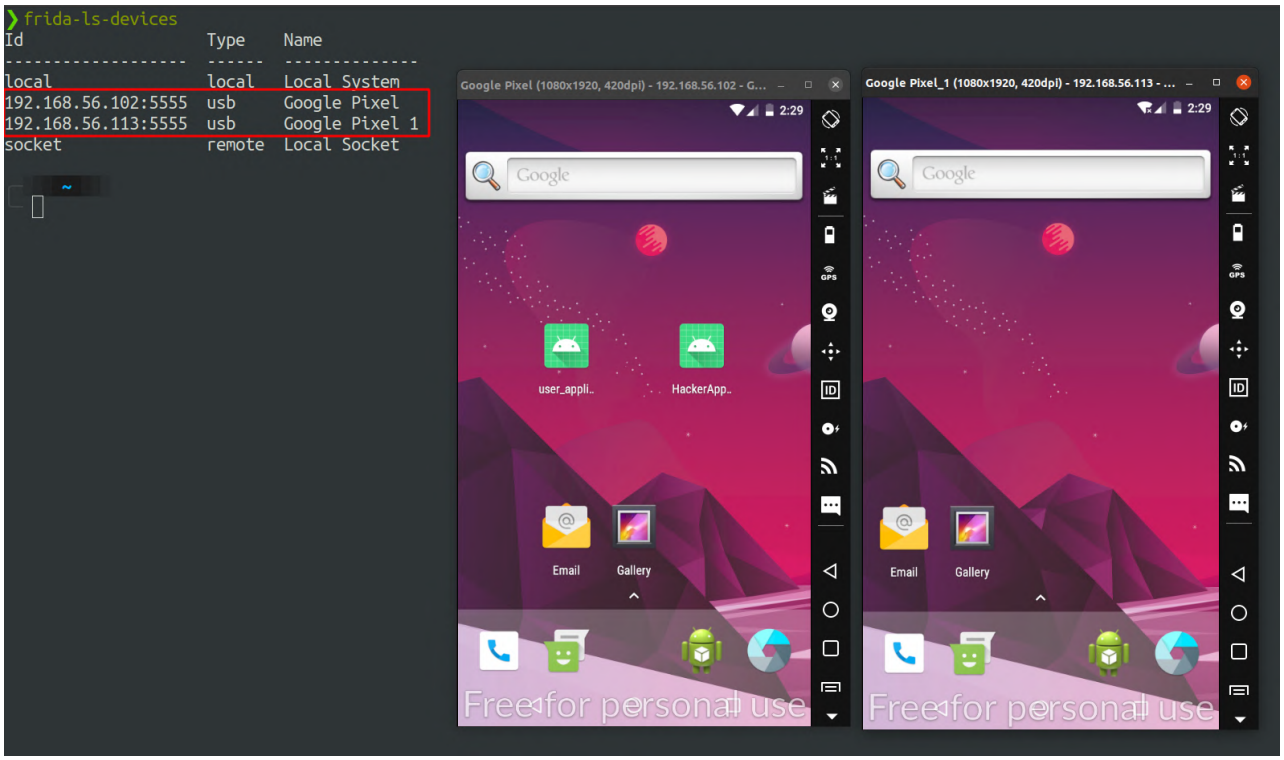
```
❯ frida-trace -U 9161  -j 'uk.rossmarks.fridalab.challenge_01!getChall01Int'
Instrumenting...
challenge_01.getChall01Int: Loaded handler at "/home/anubhav/company/research/frida/ebook/__handlers__/uk.rossmarks.fridalab.challenge_01/getChall01Int.js"
Started tracing 1 function. Press Ctrl+C to stop.
           /* TID 0x23c9 */
  3006 ms  challenge_01.getChall01Int()
  3007 ms  <= 0
```

4. frida-discover

frida-discover is a tool for discovering internal functions in a program, which can then be traced by using frida-trace.

5. frida-ls-devices

This is a command-line tool for listing attached devices, which is very useful when interacting with multiple devices.

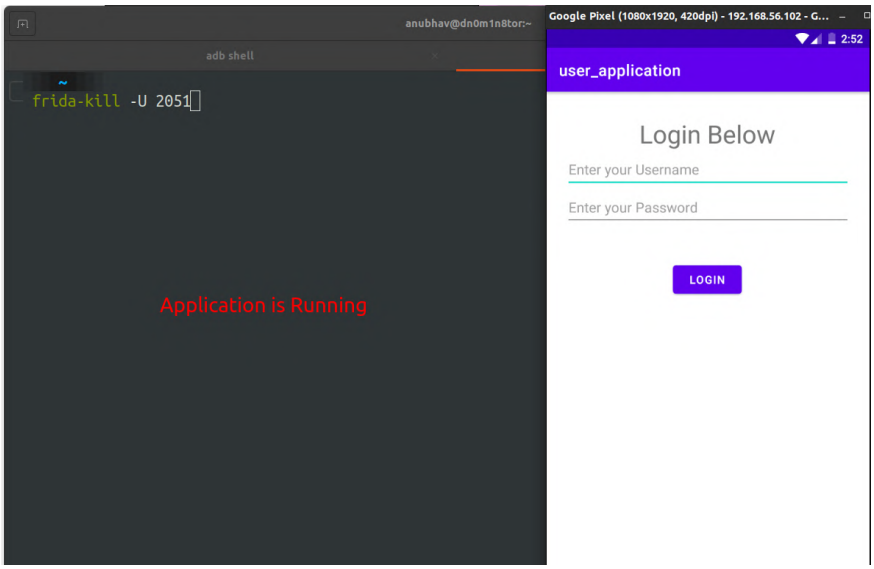


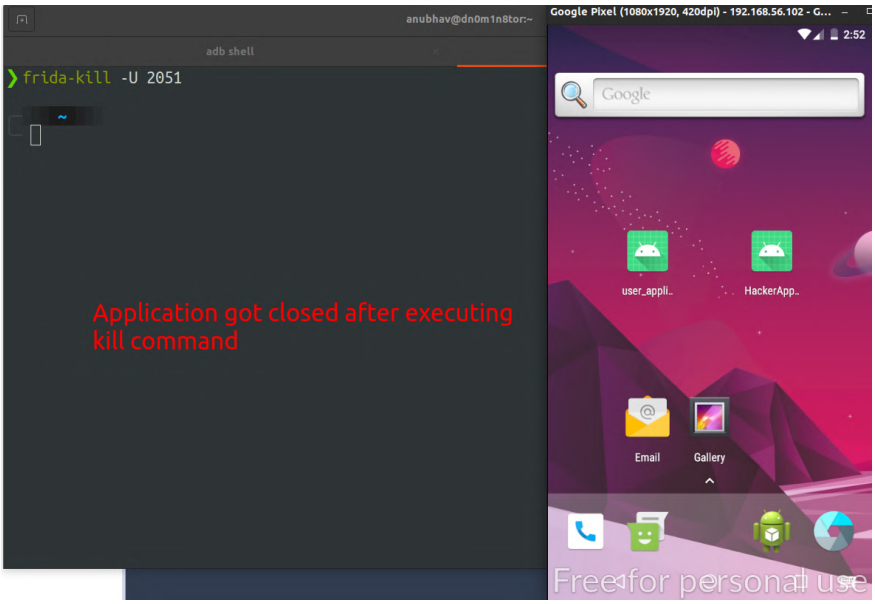
6. frida-kill

This is a command-line tool for killing processes. Suppose you want to kill a particular process using Frida then you can use frida-kill, it takes PID as an argument. We can get PID by running frida-ps which we saw previously.

```
❯ frida-ps -Ua
  PID  Name              Identifier
4  ----------------  --------------------------
1442  Calendar          com.android.calendar
1244  Clock             com.android.deskclock
1488  Email             com.android.email
1513  Messaging         com.android.messaging
1293  Phone             com.android.dialer
1580  Superuser         com.genymotion.superuser
2051  user_application  com.example.user_application
```

To kill this process using PID we can run a simple command as : frida-kill -U 2051





Chapter 4

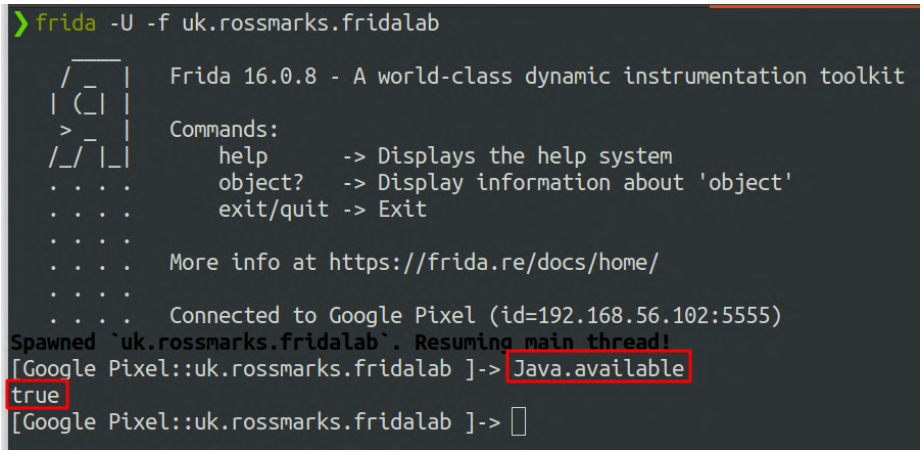
Frida APIs

Frida API Overview

1. Java.available()
2. Java.perform(fn)
3. Java.performNow(fn)
4. Java.use(className)
5. Java.choose(classname, callbacks)
6. Java.androidVersion()
7. Java.enumerateLoadedClasses(callback)
8. Java.enumerateLoadedClassesSync(callback)
9. Java.scheduleOnMainThread(func)
10. Java.enumerateMethods(query)
11. Java.isMethodThread()
12. Java.array(datatype, [])

1. Java.available()

It returns a boolean specifying whether the current process has a Java VM loaded, i.e. Dalvik or ART.



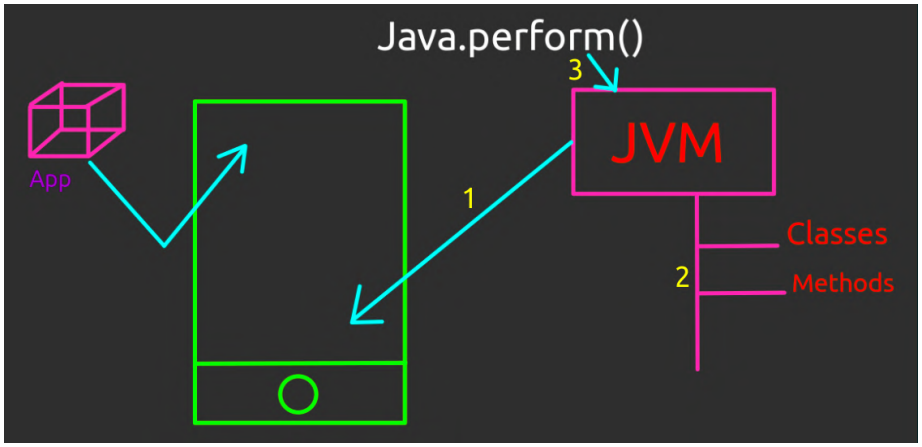
As you can see in the above image Java.available returns true which means JavaVM is loaded in the current process. By verifying this we can make sure that we won't invoke any other Java properties or methods unless Java.available returns true. We will use this while writing custom JavaScript snippets to check if JavaRuntime is available or not.

2. Java.perform()

Java.perform attaches JVM to a current thread and once that is done we are going to call a function and write JavaScript snippets in it so that we can manipulate the application in runtime. If we don't

attach the JVM using Java.perform then we will not be able to access classes and methods. And if we are not are able to access these methods then we will not be able to manipulate them.

Lets understand its working in depth:



1. There is an Android phone that has a JVM (Java Virtual Machine) in it.
2. This JVM hosts or loads all those Java classes and their methods, etc. of installed application.
3. When we write Java.perform, Frida gets attached to the JVM and because of this we now have access to all those classes and methods.
4. Now we can manipulate the behavior of the application by overwriting any class methods via writing JavaScript snippets.

Understand the Structure of Java.perform:

Here we are calling Java.perform() and not defining it, as you know when we define the function the syntax is like this

```
    Java.perform() {
    //Function working
};
```

But when we call any function the syntax is like shown below.

```
    Java.perform();
```

Now in this Java.perform we pass Anonymous Function as a parameter.

```
Java.perform(function() {
    //Here we are passing Anonymous Function as a parameter
});
```

You can write your JavaScript code inside Anonymous function.

```
Java.perform(function() {

    // What ever javascript we write here will get executed
    console.log("I can do it");
});
```

3. Java.performNow(fn)

Its same as Java.perform() but it is useful when you want to perform an early instrumentation i.e. when you want to hook into the implementation of system classes and function. Suppose you want to hook "java.lang.system" class and you don't want to wait till your application classes are loaded or you don't want to hook application class, then you can use Java.performNow(). It also ensures that the current thread is attached to the VM and call fn.

Example :

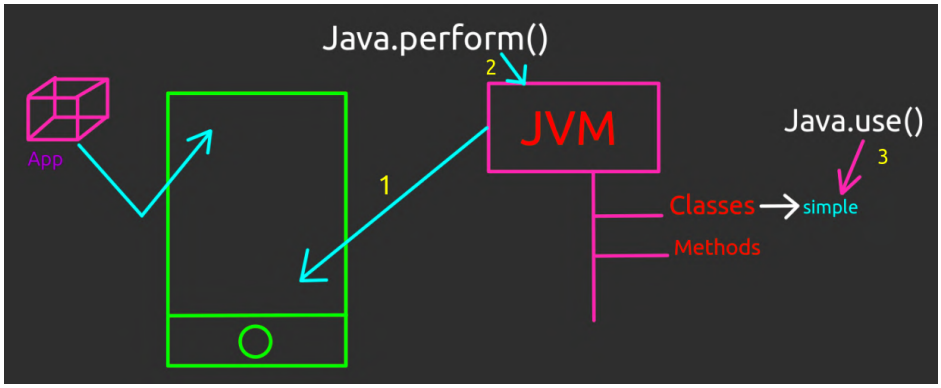
```
Java.performNow(function() {
    var system = Java.use("java.lang.System");
    system.exit.implementation = function() {
        console.log("Exit function is called");
    }
});
```

4. Java.use(class name)

It is another function call that takes the parameter as a class name and creates a wrapper of that class. In simple words java.use returns the reference of the given class name from the JVM.

```
    Java.perform(function() {

    const hello = Java.use("com.example.sample.simple");
    console.log(hello);
})
```

Lets understand its working in depth:



1. There is an Android phone that has a JVM(Java Virtual Machine) in it. This JVM hosts or loads all those Java classes and their methods, etc. on the installed application.
2. When we write Java.perform, Frida gets attached to the JVM and because of this we now have access to all those classes and methods.
3. When we write Java.use and pass class name to it, it returns a reference of the particular class which we passed from the JVM.

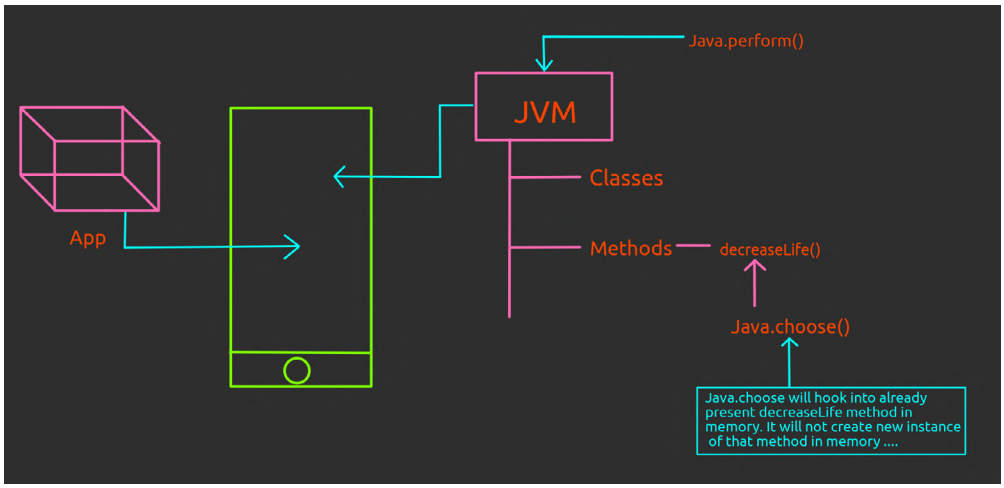
5. Java.choose()

It gets a wrapper to an already created class instance.

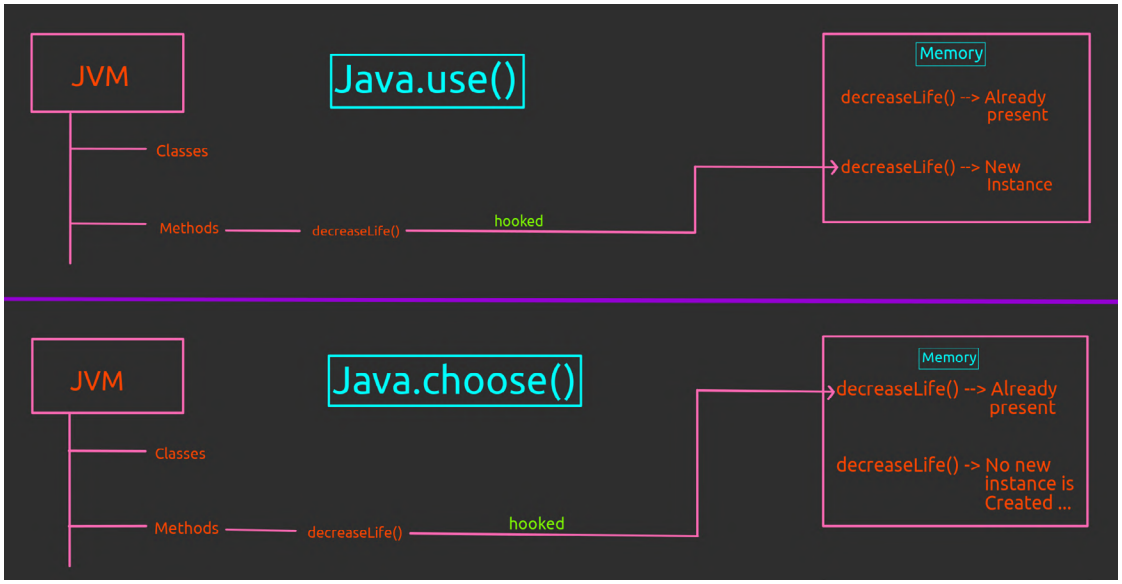
This API enumerates live instances of the `class name` class by scanning the Java heap, where `callback` is an object specified in `java.choose`.

```
    Java.perform(function() {
    Java.choose("com.example.fridalab.MAinActivity", {
        onMatch: function(instance) {
            console.log(instance);
        },
        onComplete: function() {
            console.log("end")
        }
    });

})
```

Lets understand its working in depth:



Difference between Java.use and Java.choose



If you want to hook an already created instance of a method in memory then use Java.choose(). For example there is a method named decreaseLife whose class object is created already. So if you use Java.use() then again an object of a class of decreaseLife method is created. So you will not be able to hook the original decreaseLife method which was of original object method.

If you want to hook a method which is not related to any existing object then we can use Java.use API to hook it.

6. Java.androidVersion()

It will return the Android operating system version of the emulator or a device which is connected.

```
[Google Pixel::uk.rossmarks.fridalab ]->
[Google Pixel::uk.rossmarks.fridalab ]-> Java.androidVersion
"8.0.0"
[Google Pixel::uk.rossmarks.fridalab ]->
```

I am using Genymotion here, you can see it has Android Version 8.0.

7. Java.enumerateLoadedClasses(callback)

It enumerates all the classes loaded and where callback is an object that executes functions in each of them.

Understand the structure of Java.enumerateLoadedClasses():

In JavaScript you can create an object using `{}` and inside the object you can put key-value pairs.

- This is the basic structure of Java.enumerateLoadedClasses(callback)

```
Java.perform(()=>{
    Java.enumerateLoadedClasses({

        key_1 : value_1,
        key_2 : value_2,
    })

});
```

- This is the structure of Java.enumerateLoadedClasses(callback) replacing the key value pair with onMatch and onComplete.

```
Java.perform(()=>{
    Java.enumerateLoadedClasses({

        onMatch : function(name, handle){},
        onComplete : function() {},
    })

});
```

- This is the structure of Java.enumerateLoadedClasses(callback) by onMatch and onComplete defination.

```
Java.perform(()=>{
    Java.enumerateLoadedClasses({
        onMatch : function(name, handle){
                console.log(name);

        },
        onComplete : function(){
            console.log("--- done ---");
        }
    });
});
```

Example:

You want to enumerate all the classes when the application named “fridalab” gets loaded.

```
Java.perform(()=>{
    Java.enumerateLoadedClasses({
        onMatch : function(name, handle){
                console.log(name);

        },
        onComplete : function(){
            console.log("--- done ---");
        }
    });
});
```

If you run above script using Frida, you will find all the classes that are invoked by the application will be displayed on CLI.

```
❯ frida -U -n "FridaLab" -l script1.js
     ____
    / _  |   Frida 16.0.8 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Google Pixel (id=192.168.56.102:5555)
Attaching...
org.apache.http.ProtocolVersion
org.apache.http.HttpResponse
org.apache.http.message.AbstractHttpMessage
org.apache.http.HttpHost
org.apache.http.conn.params.ConnPerRoute
org.apache.http.impl.conn.tsccm.RefQueueWorker
org.apache.http.conn.params.ConnManagerParams
org.apache.http.params.AbstractHttpParams
org.apache.http.impl.conn.IdleConnectionHandle
org.apache.http.conn.ConnectionReleaseTrigger
org.apache.http.HttpRequestInterceptor
org.apache.commons.logging.impl.WeakHashtable
org.apache.commons.logging.Log
org.apache.http.conn.params.ConnManagerPNames
org.apache.http.message.BasicHeader
org.apache.http.StatusLine
org.apache.http.client.params.HttpClientParams
org.apache.http.client.methods.HttpUriRequest
org.apache.http.conn.ClientConnectionManager
org.apache.http.HttpEntity
```

But let's say you only want to see the classes that have the package name “uk.rossmarks.fridalab”. In that case, you can `includes()` method which returns `true` if a string contains a specified string.

```
Java.perform(()=>{
    Java.enumerateLoadedClasses({
        onMatch : function(name, handle){
            if(name.includes("uk.rossmarks.fridalab")){
                console.log(name);
            }

        },
        onComplete : function(){
            console.log("--- done ---");
        }
    });
});
```

If you run the above script using Frida, you will only find the classes that have the package name “uk.rossmarks.fridalab”.

```
❯ frida -U -n "FridaLab" -l script.js
     ____
    / _  |   Frida 16.0.8 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Google Pixel (id=192.168.56.102:5555)
Attaching...
uk.rossmarks.fridalab.challenge_01
uk.rossmarks.fridalab.challenge_06
uk.rossmarks.fridalab.challenge_07
uk.rossmarks.fridalab.MainActivity
uk.rossmarks.fridalab.MainActivity$1
uk.rossmarks.fridalab.MainActivity$2
--- done ---
[Google Pixel::FridaLab ]->
```

8. Java.enumerateLoadedClassesSync(callback)

It is similar to enumerateLoadedClasses but returns an array of all the loaded classes.

`Java.enumerateLoadedClasses()` is an asynchronous function, which means it will return immedi-

-ately and call the provided callback functions (`onMatch` and `onComplete`) when the results are available, allowing the script to continue executing the other code while the enumeration is running in the background.

On the other hand, `Java.enumerateLoadedClassesSync()` is a synchronous function, which means it will block the execution of the script until the results are available. The script will wait for the enumeration to complete before moving on to the next line of code.

Example:

```
Java.perform(function () {
  var allClasses = Java.enumerateLoadedClassesSync();
  send("Loaded " + allClasses.length + " classes!");
  for(var i=0;i<allClasses.length;i++){
            console.log(allClasses[i])
        };
});
```

```
❯ frida -U -f "uk.rossmarks.fridalab" -l script5.js

     ____
    / _  |   Frida 16.0.8 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Google Pixel (id=192.168.56.102:5555)
Spawned `uk.rossmarks.fridalab`. Resuming main thread!
[Google Pixel::uk.rossmarks.fridalab ]-> message: {'type': 'send', 'payload': 'Loaded 4678 classes!'} data: None
org.apache.http.ProtocolVersion
org.apache.http.HttpResponse
org.apache.http.message.AbstractHttpMessage
org.apache.http.HttpHost
org.apache.http.conn.params.ConnPerRoute
org.apache.http.impl.conn.tsccm.RefQueueWorker
org.apache.http.conn.params.ConnManagerParams
org.apache.http.params.AbstractHttpParams
org.apache.http.impl.conn.IdleConnectionHandler
org.apache.http.conn.ConnectionReleaseTrigger
org.apache.http.HttpRequestInterceptor
org.apache.commons.logging.impl.WeakHashtable
org.apache.commons.logging.Log
org.apache.http.conn.params.ConnManagerPNames
org.apache.http.message.BasicHeader
org.apache.http.StatusLine
org.apache.http.client.params.HttpClientParams
org.apache.http.client.methods.HttpUriRequest
org.apache.http.conn.ClientConnectionManager
org.apache.http.HttpEntity
org.apache.http.conn.ClientConnectionOperator
```

9. Java.scheduleOnMainThread(func)

In an Android UI, a function must run on the Main UI Thread and if you want to tamper Android UI then you can use Java.scheduleOnMainThread() API to create an instance.
Run the function on the main thread as it is useful for tinkering around the objects that must execute in

in the main thread.

Syntax:

```
Java.scheduleOnMainThread(function() {
        //What ever function you want to execute in your UI Thread.
});
```

10. Java.enumerateMethods(query)

This API enumerate method matching a given query. The query is specified as “class!method”.

Example:

- If you want to enumerate classes and methods with class names starting with `uk.roosmarks.frida-lab , you will notice the `*`, which instructs Frida to match any class beginning with `uk.roosmarks.fridalab . If you want all methods of those classes, you can use this API: `Java.enumerateMethods('uk.roosmarks.fridalab*!*')`.

```
Java.perform(() => {
  const groups = Java.enumerateMethods('uk.rossmarks.fridalab*!*')
  console.log(JSON.stringify(groups, null, ' '));
});
```

```
❯ frida -U -f "uk.rossmarks.fridalab" -l enumerateMethods.js
     ____
    / _  |   Frida 16.0.8 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Google Pixel (id=192.168.56.102:5555)
Spawned `uk.rossmarks.fridalab`. Resuming main thread!
[Google Pixel::uk.rossmarks.fridalab ]-> []
[
 {
  "loader": "<instance: java.lang.ClassLoader, $className: dalvik.system.PathClassLoader>",
  "classes": [
   {
    "name": "uk.rossmarks.fridalab.challenge_06",
    "methods": [
     "$init",
     "addChall06",
     "confirmChall06",
     "startTime"
    ]
   },
   {
    "name": "uk.rossmarks.fridalab.challenge_07",
    "methods": [
     "$init",
     "check07Pin",
     "setChall07"
    ]
   },
   {
    "name": "uk.rossmarks.fridalab.MainActivity",
```

All methods of that class

All methods of that class

- Suppose you want the methods that start with `chall` in the same class

```
Java.perform(() => {
  const groups = Java.enumerateMethods('uk.rossmarks.fridalab*!chall*')
  console.log(JSON.stringify(groups, null, ' '));
});
```

```
[
 {
  "loader": "<instance: java.lang.ClassLoader, $className: dalvik.system.PathClassLoader>",
  "classes": [
   {
    "name": "uk.rossmarks.fridalab.MainActivity",
    "methods": [
     "chall02",
     "chall03",
     "chall04",
     "chall05",
     "chall06",
     "chall07",
     "chall08"
    ]
   }
  ]
 }
]
```

- For classes that chall in their class name.

```
Java.perform(() => {
  const groups = Java.enumerateMethods('*chall*!*')
  console.log(JSON.stringify(groups, null, ' '));
});
```


```
[
 {
  "loader": "<instance: java.lang.ClassLoader, $className: dalvik.system.PathClassLoader>",
  "classes": [
   {
    "name": "uk.rossmarks.fridalab.challenge_06",
    "methods": [
     "$init",
     "addChall06",
     "confirmChall06",
     "startTime"
    ]
   },
   {
    "name": "uk.rossmarks.fridalab.challenge_07",
    "methods": [
     "$init",
     "check07Pin",
     "setChall07"
    ]
   }
  ]
 }
]
```


11. Java.isMainThread()

To determine whether the caller is running on the main thread or not

```
[Google Pixel::uk.rossmarks.fridalab ]-> Java.isMainThread()
false
```

12. Java.array(datatype, elements)

It's useful when dealing with java arrays. It creates a Java array with elements of the specified `type`, from a JavaScript array `elements`. The resulting Java array behaves like a JS array, but can be passed by reference to Java APIs in order to allow them to modify its contents.

```
1) const values = Java.array('int', [ 1003, 1005, 1007 ]);
2) var javaStringArray = Java.array('java.lang.String', [ "Test" ]);

3) const JString = Java.use('java.lang.String');
   const str = JString.$new(Java.array('byte', [ 0x48, 0x65, 0x69 ]));
```

13. Java.registerClass(spec)

It creates a new Java class and returns a wrapper for it, where `spec` is an object containing:

- **`name:`** String specifying the name of the class.
- **`superClass:`** (optional) Super-class. Omit to inherit from **`java.lang.Object`**.
- **`implements:`** (optional) Array of interfaces implemented by this class.
- **`fields:`** (optional) Object specifying the name and type of each field to expose.
- **`methods:`** (optional) Object specifying methods to implement.

The method takes two arguments: the class name (as a string) and an options object. The options object can be used to specify the methods and fields of the class that should be exposed to Frida.

Here is an example of how to use **`Java.registerClass`** to register a class called "com.example.FridaClass":

```
var MainActivity4 = Java.registerClass({
     name:'com.example.fridalearn.MainActivity4',
     methods:{
         sayHello:function(){
            console.log("MainActivity4 is called ");
         }
     }
 });
```

In this example, the **`Java.registerClass`** method is used to register the class "com.example.fridalearn.MainActivity4" and expose its method "sayHello" to Frida.

Then you can use the function in your script and call sayHello as an instance of MainActivity4.

You can use the function using **`Java.use(class_name)`**

```
var MainActivity4Class = Java.use('com.example.fridalearn.MainActivity4')
MainActivity4Class.sayHello();
```

Chapter 5

Manipulating Android App Elements

Let's try to apply the concepts learned thus far in real-world scenarios.

1. Manipulating Static and Non Static Variable Values

A. Static Variable

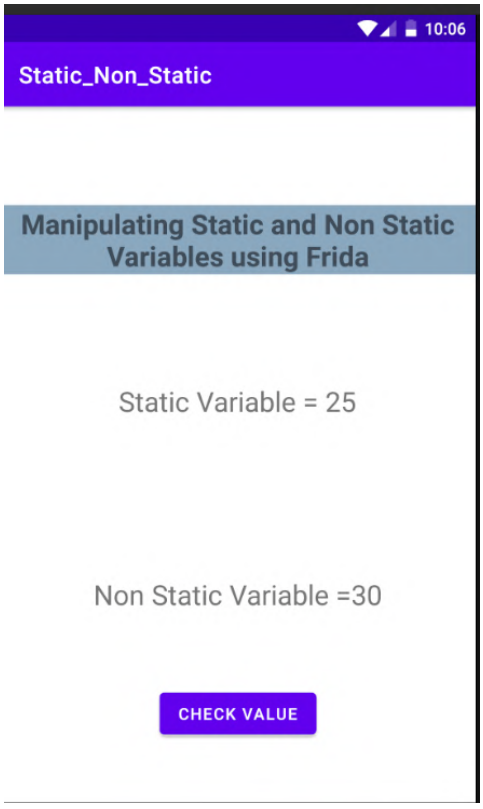
a. Read Variable Value

We can directly read static variable values by getting the reference of the class and then accessing the variable

(i) This is a class where we have defined the values

```
class Testing_Variables {

    static int hello_stat=25;
    int hello_non_stat = 30;

}
```

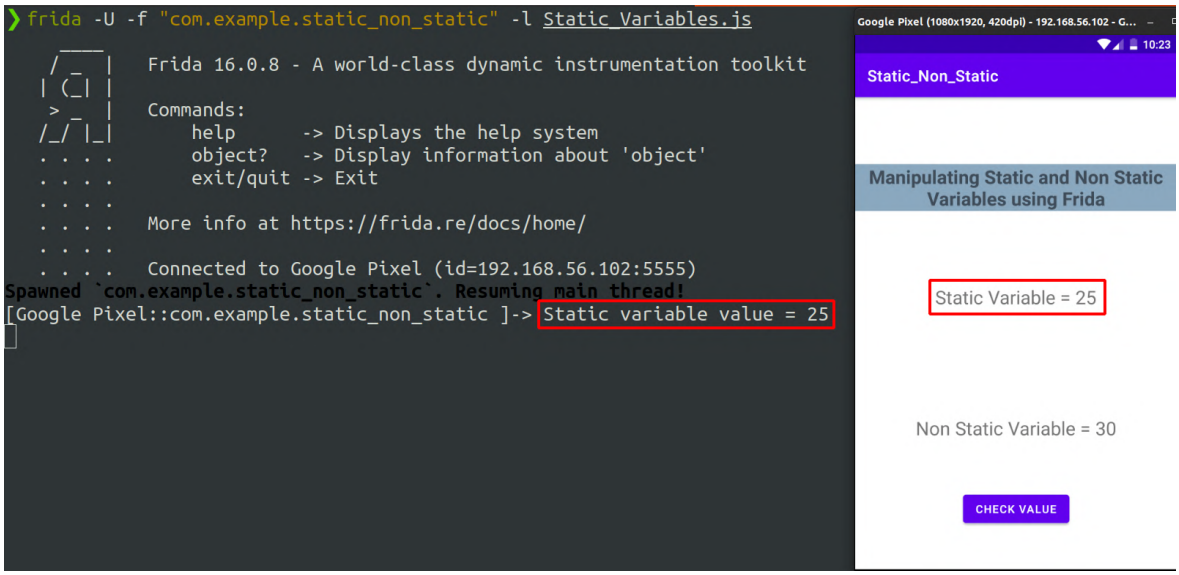
(ii) In this activity we are displaying these values



(iii) We will write a custom Frida script to read the `hello_stat` variable value which is a static variable

```
Java.perform(function(){

// Accessing static class variable

//Creating a reference for class "Testing_Variables"
var c1 = Java.use("com.example.static_non_static.Testing_Variables");

        //Accessing static variable using syntax as "Class_name.Variable_name.value"

        console.log("Static variable value = " + c1.hello_stat.value);

})
```

(iv) When we execute the script, you will get the value of that static variable



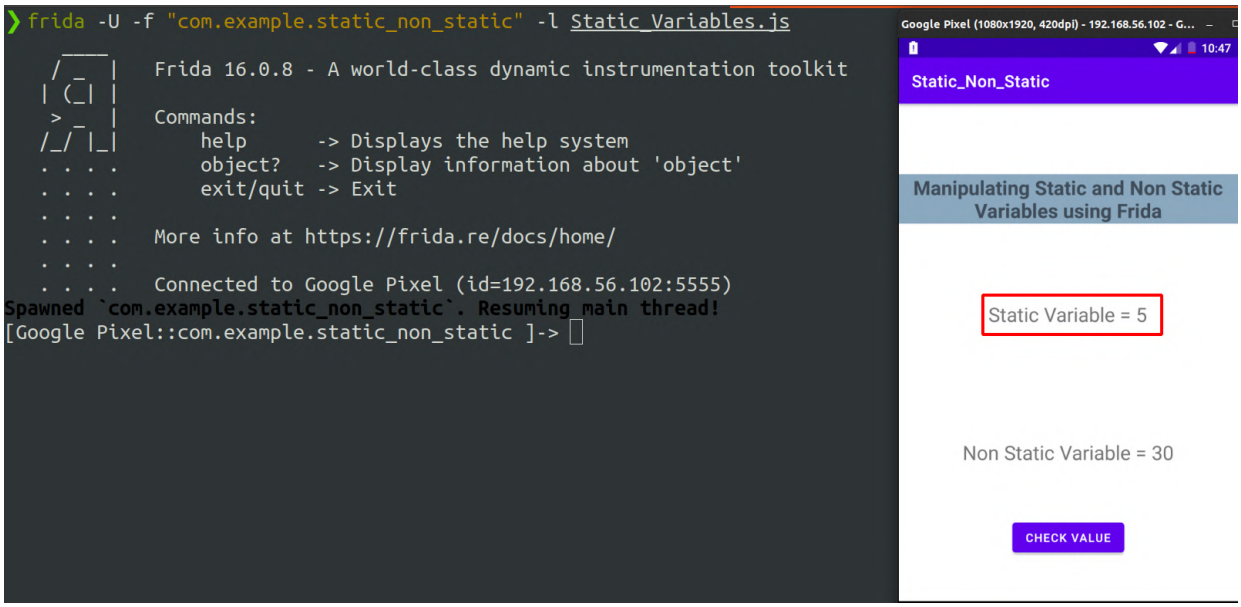
(v) So you can use this method to dynamically know in runtime what value is getting assigned to what variable as sometimes secret keys or sensitive values are also stored in such variables

b. Change Variable Value

(i) You can directly change the value:

```
Java.perform(function(){

// Accessing static class variable

//Creating a reference for class "Testing_Variables"
var c1 = Java.use("com.example.static_non_static.Testing_Variables");

        //Changing static variable value using syntax as "Class_name.Variable_name.value = value"

        c1.hello_stat.value = 5;

})
```

(ii) You can see the value has changed from 25 to 5



B. Non Static Variable

a. Read Variable Value

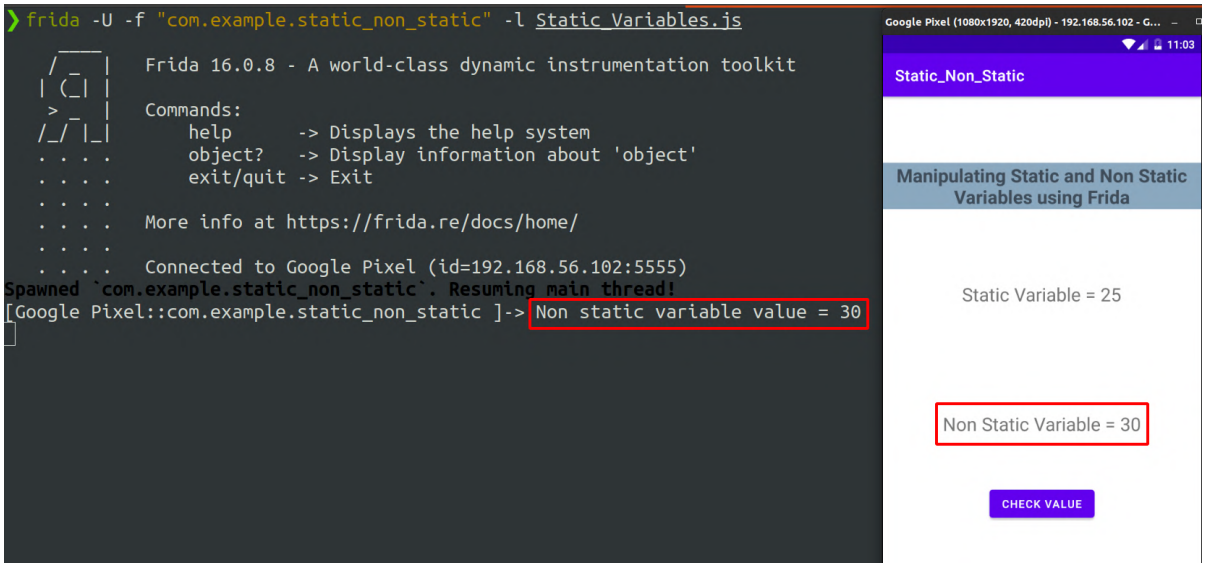
(i) We cannot directly read the non static variable value, we need to create an object of that class and then its access variables

```
class Testing_Variables {

    static int hello_stat=25;
    int hello_non_stat = 30;

}
```

(ii) The script that is going to create references of the class and access non static variables value

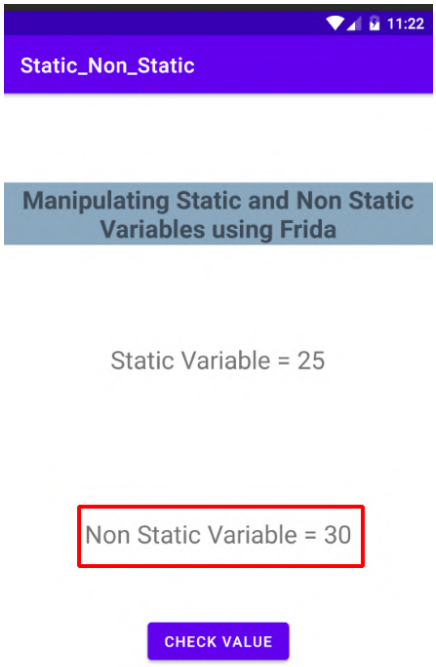
```
Java.perform(function(){

// Accessing static class variable

//Creating a reference for class "Testing_Variables"
var c1 = Java.use("com.example.static_non_static.Testing_Variables");

        var c2 = c1.$new();  //We are creating object of Testing_Variables class

        // We are accessing value of hello_non_stat variable
        console.log("Non static variable value = " + c2.hello_non_stat.value);

})
```

(iii) Running the script



b. Read Variable Value

(i) When we launch the application you will see that the value of `hello_non_stat` is 30. Now we need to change this value according to us.



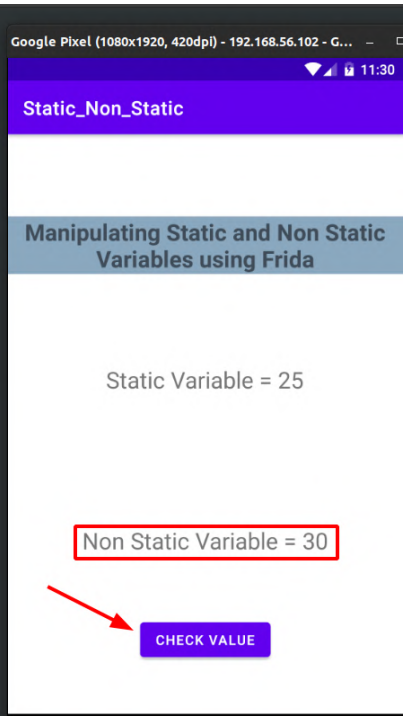
(ii) As this is a non static variable we need to use the `java.choose()` API so that we can get the instance of the created object of `Testing_Variables` and then we can manipulate anything in it.

(iii) We have written the script here, which will change the value of `hello_non_stat` variable to 6000.

```
Java.perform(function(){

// Accessing static class variable

//Creating a reference for class "Testing_Variables"

        Java.choose('com.example.static_non_static.Testing_Variables',{

        onMatch: function(instance) {

                //Assigning 6000 as value for hello_non_stat variable
                instance.hello_non_stat.value = 6000;

                send("The hello_non_stat value = " + instance.hello_non_stat.value );
        },

        onComplete: function() {
                console.log("Done!!");
        }
})
})
```

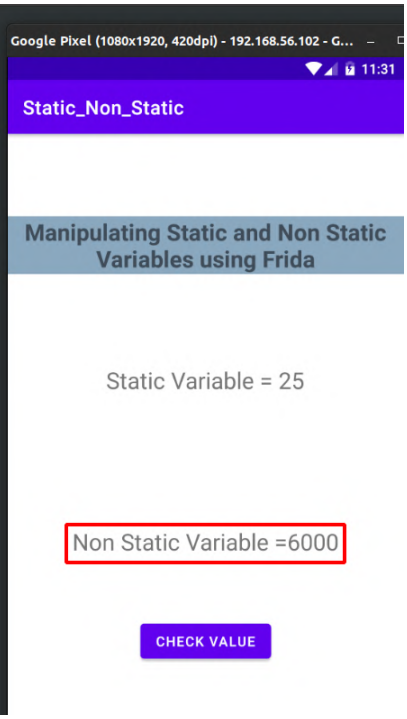
(iv) When we execute the script and click on the check value button, you will see that the value has changed to 6000

```
❯ frida -U -f "com.example.static_non_static" -l Static_Variables.js
     ____
    / _  |   Frida 16.0.8 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Google Pixel (id=192.168.56.102:5555)
Spawned `com.example.static_non_static`. Resuming main thread!
[Google Pixel::com.example.static_non_static ]-> Done!!
Done!!
message: {'type': 'send', 'payload': 'The hello_non_stat value = 6000'} data: None
```



After clicking on the 'check value' button you will notice that the value has changed to 6000

```
❯ frida -U -f "com.example.static_non_static" -l Static_Variables.js
     ____
    / _  |   Frida 16.0.8 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Google Pixel (id=192.168.56.102:5555)
Spawned `com.example.static_non_static`. Resuming main thread!
[Google Pixel::com.example.static_non_static ]-> Done!!
Done!!
message: {'type': 'send', 'payload': 'The hello_non_stat value = 6000'} data: None
```



NOTE: Suppose we use the **`Java.use`** API, create a new object of that class, and assign a new value to the **`hello_non_stat`** variable, you will notice it gives a cannot access an instance error. **`Java.use`** will create a new object but we need to change the value of the variable in an existing object so that's why it's not possible to use **`Java.use`** to manipulate non static variables or methods.

Changing non static variable value using **`Java.use`** using the below script

```
Java.perform(function() {

    var c1 = Java.use("com.example.static_non_static.Testing_Variables");

    c1.hello_non_stat.value = 6000;

    console.log(c1.hello_non_stat.value);

})
```

You will get an error like this -

```
❯ frida -U -f "com.example.static_non_static" -l Static_Variables.js
     ____
    / _  |   Frida 16.0.8 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Google Pixel (id=192.168.56.102:5555)
Spawned `com.example.static_non_static`. Resuming main thread!
[Google Pixel::com.example.static_non_static ]-> Error: Cannot access an instance field without an instance
    at set (frida/node_modules/frida-java-bridge/lib/class-factory.js:1143)
    at <anonymous> (/home/anubhav/company/research/frida/ebooooook/Static_Variables.js:3)
    at <anonymous> (frida/node_modules/frida-java-bridge/lib/vm.js:12)
    at _performPendingVmOps (frida/node_modules/frida-java-bridge/index.js:250)
    at <anonymous> (frida/node_modules/frida-java-bridge/index.js:242)
    at apply (native)
    at ne (frida/node_modules/frida-java-bridge/lib/class-factory.js:620)
    at <anonymous> (frida/node_modules/frida-java-bridge/lib/class-factory.js:598)
[Google Pixel::com.example.static_non_static ]->
```

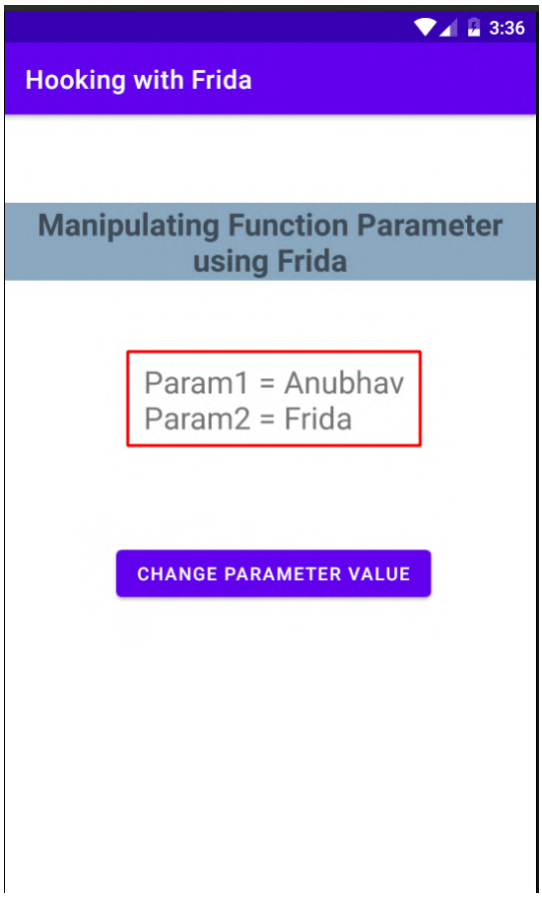
2. Reading Method Parameters

Suppose you are analyzing the code in Jadx-GUI for an XYZ application and you got a function that seems interesting. Now you want to know what parameters are getting passed to that function. Let's see how you can see those parameters.

A. This is the code you find while analyzing the code. The parameter is getting passed to the `chall01` method

```
public String chall01(String str, String str1) {
    String a = str;
    String b = str1;
    String c = "Param1 = " + a + "\nParam2 = " + b ;

    return c;
}
```

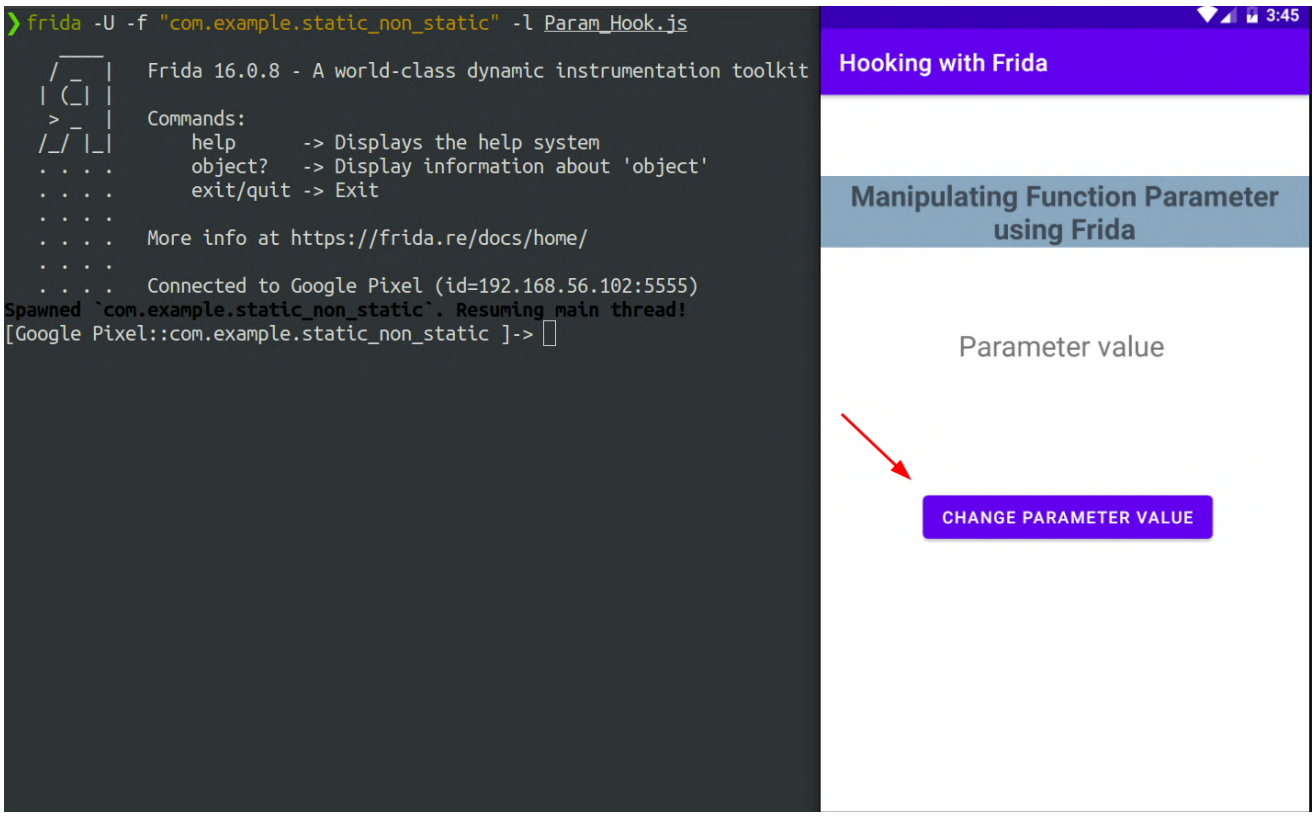
B. As you can see in 'Activity' that two params are printed.



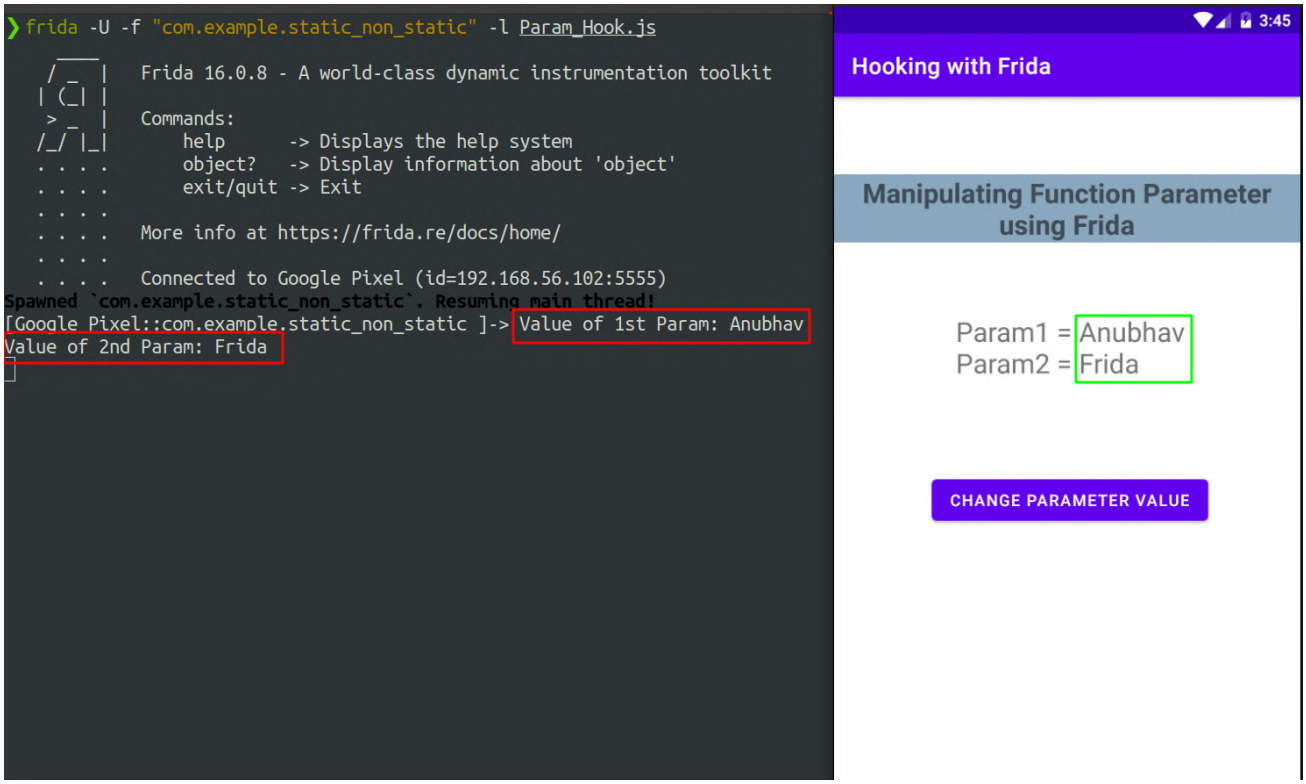
C. We will hook this method using Frida and log this value in the Frida console.

```
            Java.perform(function(){

            //We have taken the reference of class Parameter_Pass_hook in which
chall01 method is defiend
            var class_Hook = Java.use("com.example.static_non_static.Parameter_
Pass_hook");

            // To manipulate any method you need to use this .implementation
            class_Hook.chall01.implementation = function(a,b) {

              // We have only passed two parameter in this function as a,b
because only two paramater where passed in actual method. If 3 parameter where
passed to this method then also need to provide 3 params here as a,b,c

              console.log("Value of 1st Param: " + a);
              console.log("Value of 2nd Param: " + b);

              // We are returing the actual implementation of the function and
due this whatever we have manipulated to this function will not affect the actual
implementation of the function in runtime
              return this.chall01(a,b);
            }
            })
```

D. Execute the script and click on change the value



E. Params value are logged in the console



Suppose you got a method that has 3 params, then you will have to just pass 1 more params to the function.

```
Java.perform(function(){

var class_Hook = Java.use("<classname>");

// To manipulate any method you need to use this .implementation
<classname>.<methodname>.implementation = function(a,b,**c**) {

  console.log("Value of 1st Param: " + a);
  console.log("Value of 2nd Param: " + b);
  console.log("Value of 2nd Param: " + c);

  // We are returing the actual implementation of the function and
due this whatever we have manipulated to this function will not affect the actual
implementation of the function in runtime
  return this.chall01(a,b,c);
}
})
```

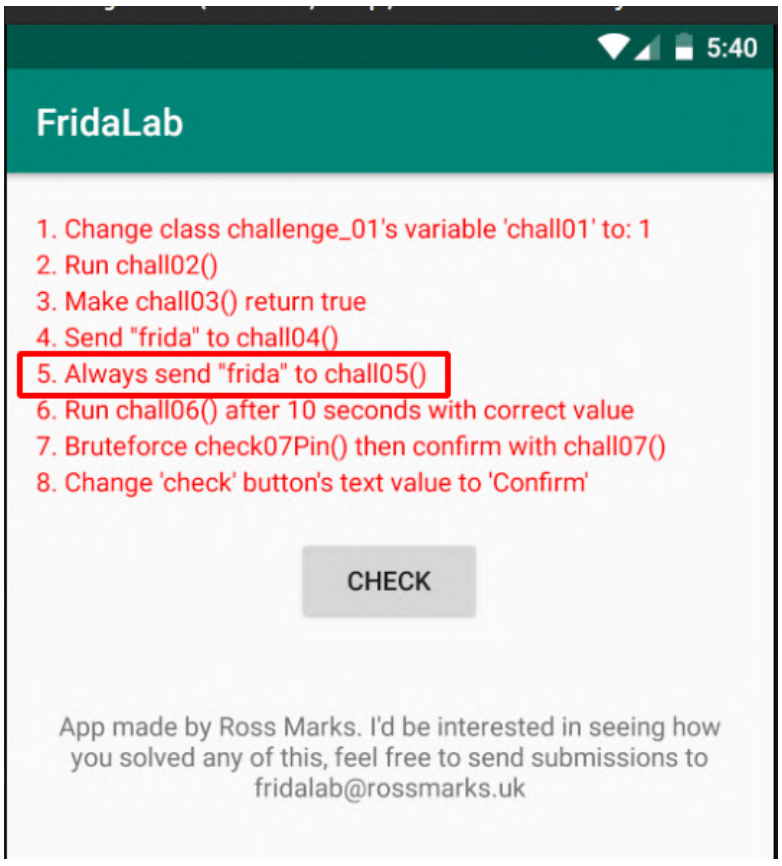
3. Replacing Arguments of a Method by Our Choice

Suppose you are analyzing the code in Jadx-GUI for an XYZ application, you got a function that checks for a specific value which comes from the parameter. Based on that it checks for certain functionalities. So in this case we want to pass the parameter of our choice which satisfies the condition and execute the operation.

A. You can see there is a method that checks for the param value `frida`. If this methods gets the param value as `frida`, the colour of the challenge changes to green in the application's UI.

```
public void chall05(String str) {
    if (str.equals("frida")) {
        this.completeArr[4] = 1;
    } else {
        this.completeArr[4] = 0;
    }
}
```

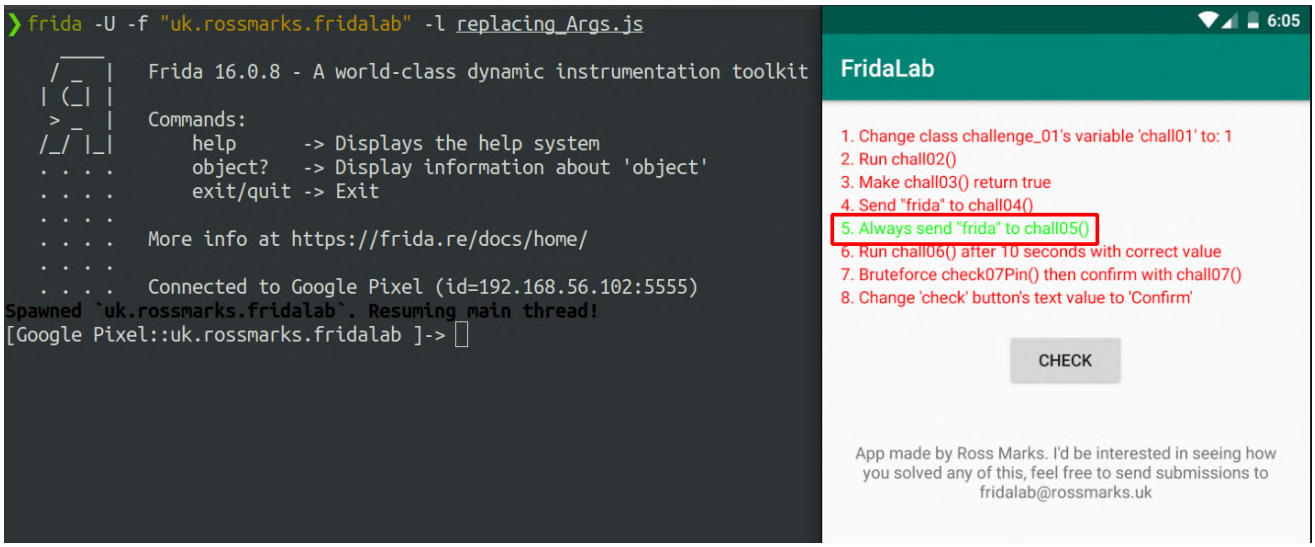
B. If above methods get param value as `frida` the colour of the challenge changes to green in the application. As of now its red.



C. Execute the below Frida script to perform the action

```
            Java.perform(function(){

            //We have taken the reference of class MainActivity in which chall05
  method is defiend
            var class_Hook = Java.use("uk.rossmarks.fridalab.MainActivity");

            // To manipulate any method you need to use this .implementation
            class_Hook.chall05.implementation = function (arg0) {

            // We are calling again chall05 method using this keyword and passing
  "frida" as an param
            this.chall05("frida");
            }

            })
```

D. After executing the script, you will see we have bypassed the check.



The above given example a was simple one, but let's say there is a class that has defined two or more methods that have the same name but different parameters know as overloading methods. So in this class, the above given script will not work. You will need to overload the method with proper argument type.

Let's understand this with an example

- There are total 3 display methods, all three methods take different arguments.

```
        display( a: 1);
        display( a: "Hello");
        display( a: "Hello", b: 1);

    }

    private void display(int a){
        TextView textView5 = (TextView) findViewById(R.id.textView5);
        textView5.setText("1st : Got Integer data as : " + a);
    }

    private void display(String a){
        TextView textView6 = (TextView) findViewById(R.id.textView6);
        textView6.setText("2nd : Got String object as : " + a);
    }

    private void display(String a, int b){
        TextView textView7 = (TextView) findViewById(R.id.textView7);
        textView7.setText("3rd : Got String object as :" + a + "\nand Integer data as : " + b );
    }

}
```

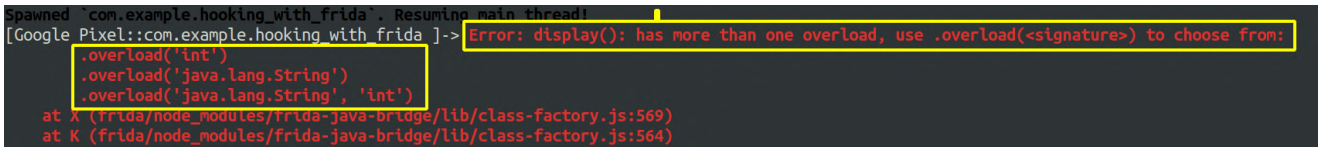
- We have hooked the display method using the previous script.

```
Java.perform(function(){

var class_Hook = Java.use("com.example.hooking_with_frida.replacing_Args");
class_Hook.display.implementation = function (a) {

this.display(a);

}
})
```

- You will notice that Frida throws an error such as

```
❯ frida -U -f "com.example.hooking_with_frida" -l replacing_Args.js
     ____
    / _  |   Frida 16.0.8 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Google Pixel (id=192.168.56.102:5555)
Spawned `com.example.hooking_with_frida`. Resuming main thread!
[Google Pixel::com.example.hooking_with_frida ]-> Error: display(): has more than one overload, use .overload(<signature>) to choose from:
        .overload('int')
        .overload('java.lang.String')
        .overload('java.lang.String', 'int')
    at X (frida/node_modules/frida-java-bridge/lib/class-factory.js:569)
    at K (frida/node_modules/frida-java-bridge/lib/class-factory.js:564)
    at set (frida/node_modules/frida-java-bridge/lib/class-factory.js:932)
    at <anonymous> (/home/anubhav/company/research/frida/ebooook/replacing_Args.js:3)
    at <anonymous> (frida/node_modules/frida-java-bridge/lib/vm.js:12)
    at _performPendingVmOps (frida/node_modules/frida-java-bridge/index.js:250)
    at <anonymous> (frida/node_modules/frida-java-bridge/index.js:242)
    at apply (native)
    at ne (frida/node_modules/frida-java-bridge/lib/class-factory.js:620)
    at <anonymous> (frida/node_modules/frida-java-bridge/lib/class-factory.js:598)
```

- If you observe the error properly, you will see that display(): has more than one overload.

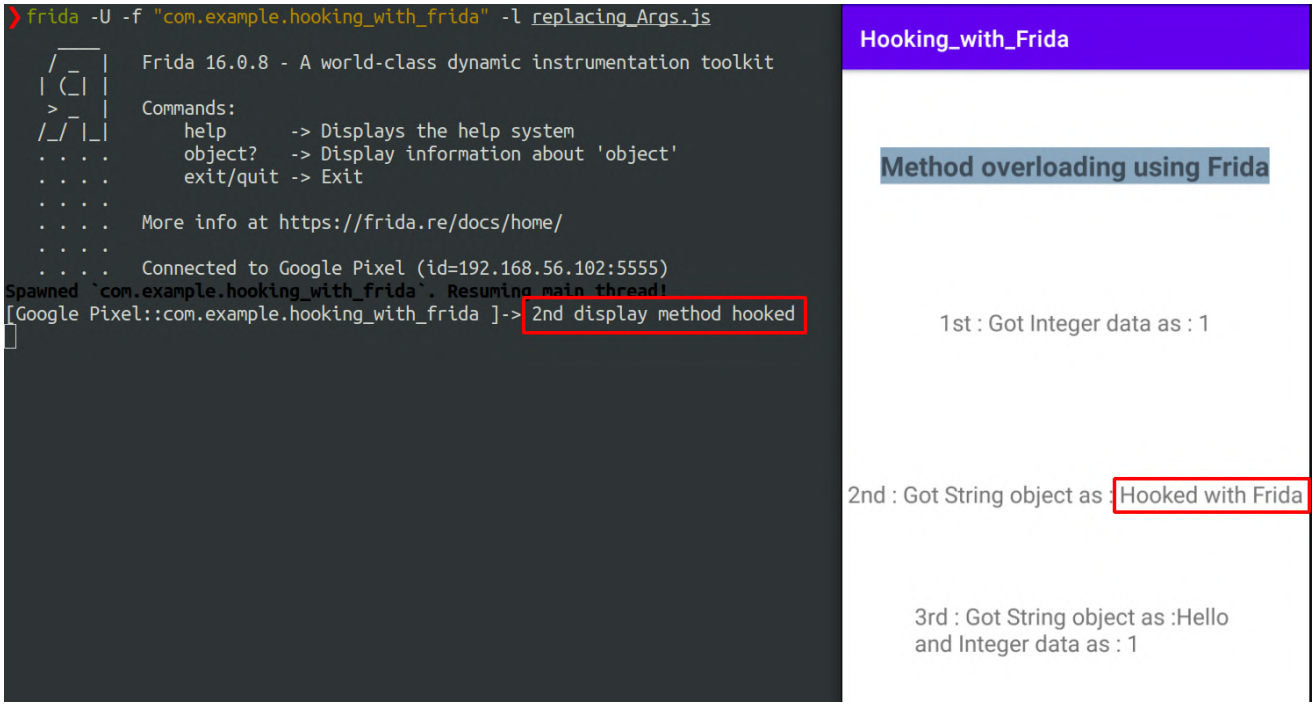


- We now need to specify to Frida that we want to hook the `display` method specifically; otherwise, Frida will give us an error.

```
            //1st method

            Java.perform(function() {

                  var class_Hook = Java.use("com.example.hooking_with_frida.
replacing_Args");

                  class_Hook.display.overload('int').implementation = function
(a) {

                  this.display(25);
                  console.log("1st display method hooked")

                  }

            })

            //2nd method

            Java.perform(function() {

                  var class_Hook = Java.use("com.example.hooking_with_frida.
replacing_Args");

                  //We have choosen 2nd method of display to hook so we provided
overload('java.lang.String'), suppose we want to hook first method then we would
have provided as overload('int')
                  class_Hook.display.overload('java.lang.String').implementation
= function (a) {

                  this.display("Hooked with Frida");
                  console.log("2nd display method hooked")

                  }

            })
```

```
            //3rd method

            Java.perform(function() {

                  var class_Hook = Java.use("com.example.hooking_with_frida.
replacing_Args");

                  class_Hook.display.overload('java.lang.String', 'int').
implementation = function (a) {

                  this.display("Hooked with Frida",65);
                  console.log("3rd display method hooked")

                  }

            })
```

- Executing the script for the 2nd method.



NOTE: We need to pass what type of argument the method is taking - `java.lang.String`, `android.os.Bundle`, etc. So to find what type of argument it is, we can take help of different methods. Here I will show you using objection as it is the easiest method.

- You just need to attach the objection with the application.

```
❯ objection --gadget com.example.hooking_with_frida explore
Using USB device `Google Pixel`
Agent injected and responds ok!

     _   _         _   _
 ___| |_|_|___ ___| |_|_|___ ___
| . | . | | -_|  _|  _| | . |   |
|___|___| |___|___|_| |_|___|_|_|
      |___|(object)inject(ion) v1.11.0

     Runtime Mobile Exploration
        by: @leonjza from @sensepost

[tab] for command suggestions
com.example.hooking_with_frida on (Android: 8.0.0) [usb] # a
```

- Instruct objection to watch out the class you were looking for `android hooking watch class <classname>`

```
com.example.hooking_with_frida on (Android: 8.0.0) [usb] # android hooking watch class com.example.hooking_with_frida.replacing_Args
(agent) Hooking com.example.hooking_with_frida.replacing_Args.display(int)
(agent) Hooking com.example.hooking_with_frida.replacing_Args.display(java.lang.String)
(agent) Hooking com.example.hooking_with_frida.replacing_Args.display(java.lang.String, int)
(agent) Hooking com.example.hooking_with_frida.replacing_Args.onCreate(android.os.Bundle)
(agent) Registering job 360789. Type: watch-class for: com.example.hooking_with_frida.replacing_Args
com.example.hooking_with_frida on (Android: 8.0.0) [usb] #
```

- Above are all the parameter types listed for display method.

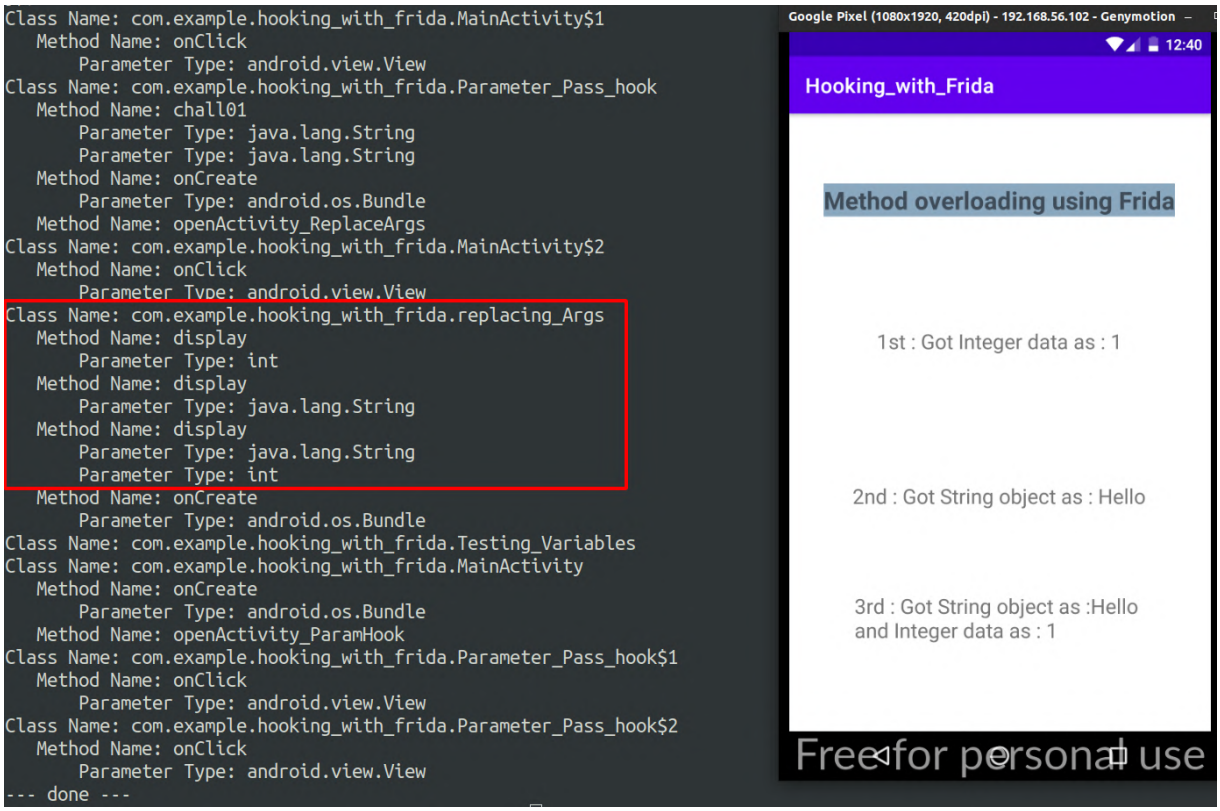
```
package com.example.hooking_with_frida;

import ...

public class replacing_Args extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_replacing_args);

        display( a: 1);
        display( a: "Hello");
        display( a: "Hello", b: 1);

    }

    private void display(int a){
        TextView textView5 = (TextView) findViewById(R.id.textView5);
        textView5.setText("1st : Got Integer data as : " + a);
    }

    private void display(String a){
        TextView textView6 = (TextView) findViewById(R.id.textView6);
        textView6.setText("2nd : Got String object as : " + a);
    }

    private void display(String a, int b){
        TextView textView7 = (TextView) findViewById(R.id.textView7);
        textView7.setText("3rd : Got String object as :" + a + "\nand Integer data as : " + b );
    }

}
```

There are other methods using the Frida script from which you can find the parameter types using APIs such as Java.enumerateLoadedClasses, Java.enumerateLoadedClassesSync()

For example,

```
        Java.perform(()=>{
            Java.enumerateLoadedClasses({
                onMatch : function(className, handle){
                    if(className.includes("com.example.hooking_with_frida")){

                        console.log("Class Name: " + className);
                        var clazz = Java.use(className);

                        var methods = clazz.class.getDeclaredMethods();
                        methods.forEach(function (method) {
                            console.log("    Method Name: " + method.getName());

                            var parameters = method.getParameterTypes();
                             parameters.forEach(function (parameter){
                                console.log("        Parameter Type: " + parameter.
getName());
                             });
                        });
                    }

                },
                onComplete : function(){
                    console.log("--- done ---");
                }
            });
        });
```

The output -

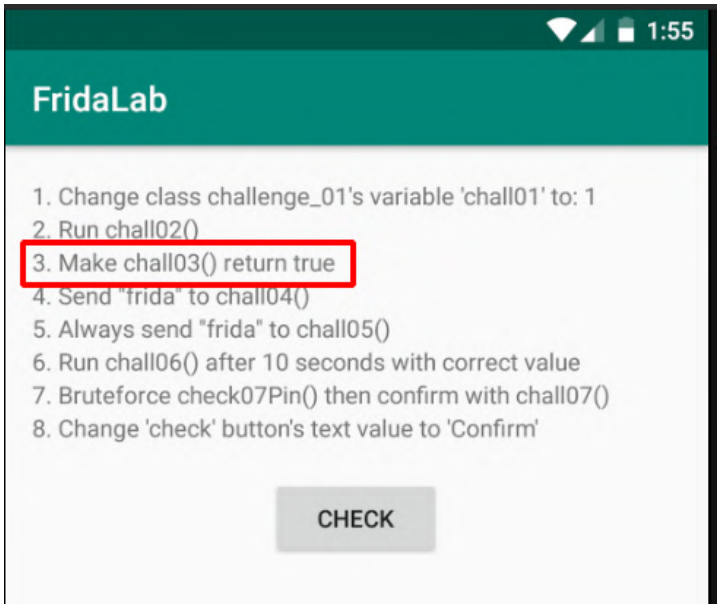


4. Manipulating Return Values

To change the return value of a function, Frida can be of help.

Let's look at an example:

A. The 3rd challenge of this APK you need to return `true`. It means in the actual code it's returning `false` value.



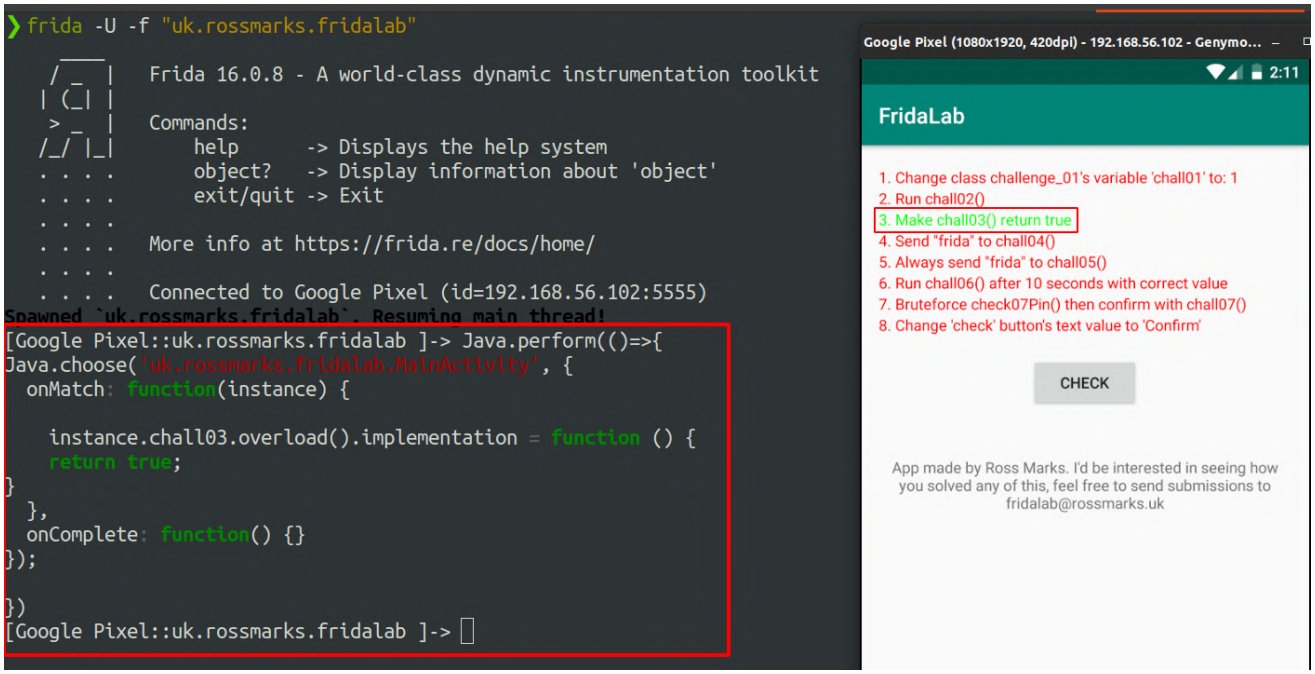
B. It is returning false and now we need to change this value to true

```
public boolean chall03() {
    return false;
}
```

C. Write a simple Frida script

```
Java.perform(() => {
    Java.choose('uk.rossmarks.fridalab.MainActivity', {
        onMatch: function(instance) {

            instance.chall03.overload().implementation = function() {

                //changing the return value to true
                return true;
            }
        },
        onComplete: function() {}
    });

})
```

D. Execute the script

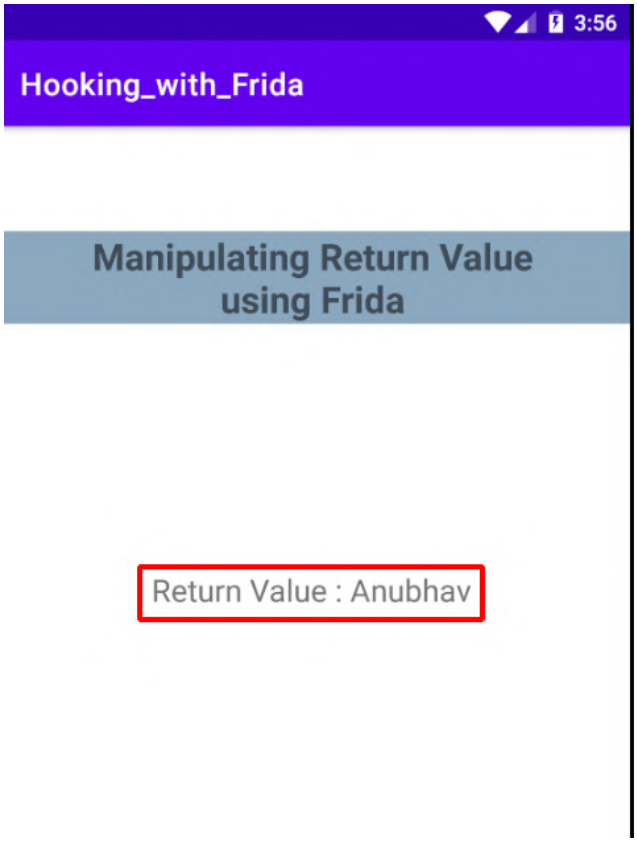


Let's see another example:

a. We need to change the return value from `Anubhav` to `frida`

```
package com.example.hooking_with_frida;

import ...

public class Return_Change extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_return_change);
        TextView textView5 = (TextView) findViewById(R.id.textView8);
        String a = changeReturn();
        textView5.setText("Return Value : " + a);

    }

    private String changeReturn(){

        String value = "Anubhav";
        return value;

    }
}
```

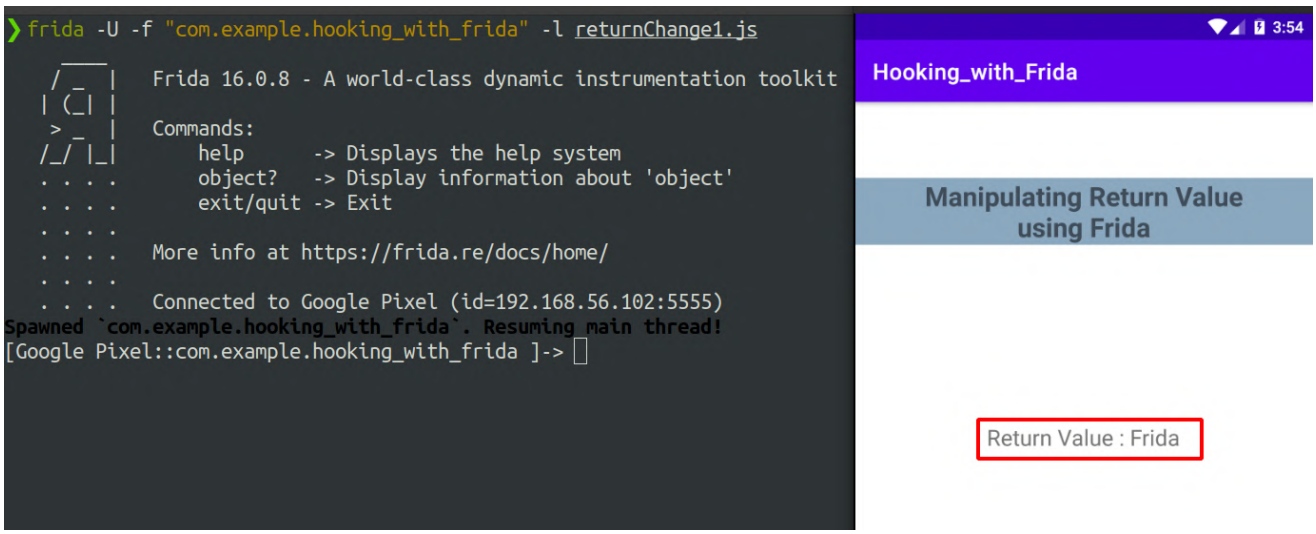
b. This is the activity



c. Write a script to change the return value

```
Java.perform(function() {

  var class_Hook = Java.use("com.example.hooking_with_frida.Return_
Change");

  class_Hook.changeReturn.implementation = function () {

      this.changeReturn();
      return "Frida";

      };
})
```

d. The return value changed to **`frida`**



This is how you can manipulate the return value of any function.

E. Instrumenting constructors

You can use Frida to hook the constructor of a class in a running process.

A constructor in a class is a special method that is automatically called when an object of that class is created. It's used to initialize the state of the object, allocate memory, and perform any other setup

required for the object to be used. The constructor in a class has the same name as the class itself, and is used to create an instance of the class. For example, if you have a class named `Person` , the constructor might look something like this:

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}
```

To create an instance of the `Person` class, you would call the constructor:

```
let person = new Person("John Doe", 30);
```

This creates a new instance of the `Person` class with the name `"John Doe"` **and the age** `30`. The constructor sets the values of the `name` **and** `age` properties of the new `Person` object.

So to understand the hooking of the constructor we have created a class `ChangeConstructor` and have defined two constructors of that class.

```
package com.example.hooking_with_frida;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.TextView;

public class ChangeConstructor extends AppCompatActivity {

    int length;
    int breadth;

    public ChangeConstructor() {
        System.out.println("Default Constructor is called\n");
    }

    public ChangeConstructor(int l, int b) {
        this.length = l;
        this.breadth = b;

        System.out.println("Constructor with Args is called\n");
    }
```

Constructor with No Args

Constructor with Args

To hook the constructor we need to use a keyword named like `$init` since the method name is same as the className.

a. Constructor with No Args

Script:

```
        Java.perform(function() {

            var ClassReference = Java.use('com.example.hooking_with_frida.
ChangeConstructor');
            ClassReference.$init.overload().implementation = function() {

                send(" Default constructor hooked ");
                this.$init();
            }
        });
```

Explanation:

(i) Get the reference of the class ChangeConstructor.

```
            Java.perform(function() {

                var ClassReference = Java.use('com.example.hooking_with_frida.
ChangeConstructor');

                }
            });
```

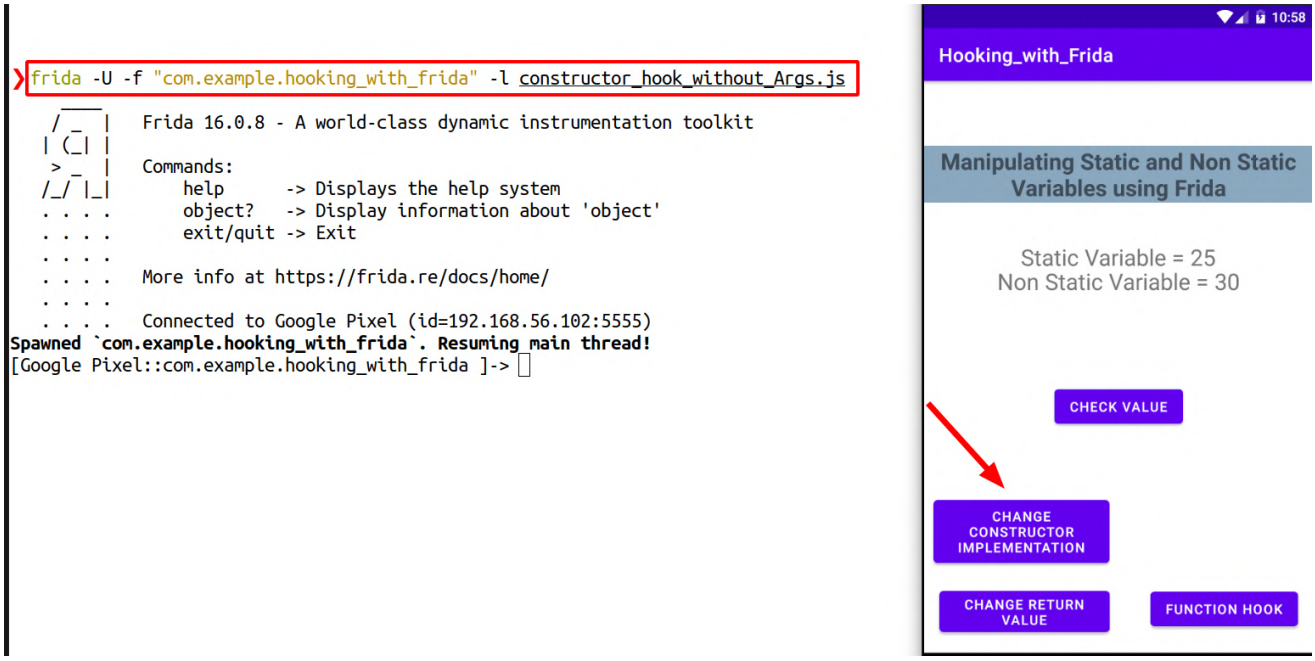
(ii) Now use the keyword `$init` to hook the constructor.

```
            Java.perform(function() {

                var ClassReference = Java.use('com.example.hooking_with_frida.
ChangeConstructor');

                //As there are two constructor defined we need to use .overload to
provide which constructor method we need to hook
                ClassReference.$init.overload().implementation = function() {

                }
            });
```

(iii) Implement the changes and call the original function.

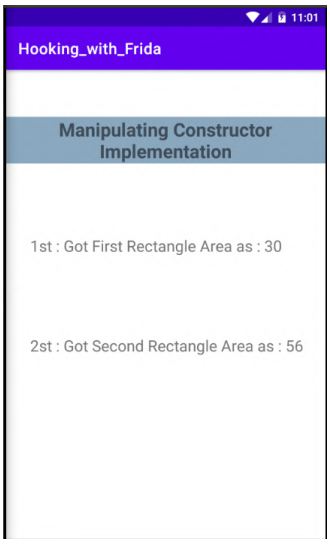
```
            Java.perform(function() {

                var ClassReference = Java.use('com.example.hooking_with_frida.
ChangeConstructor');
                ClassReference.$init.overload().implementation = function() {

                    send(" Default constructor hooked ");
                    this.$init();
                }
            });
```

(iv) Run the script



(v) Hook the default constructor

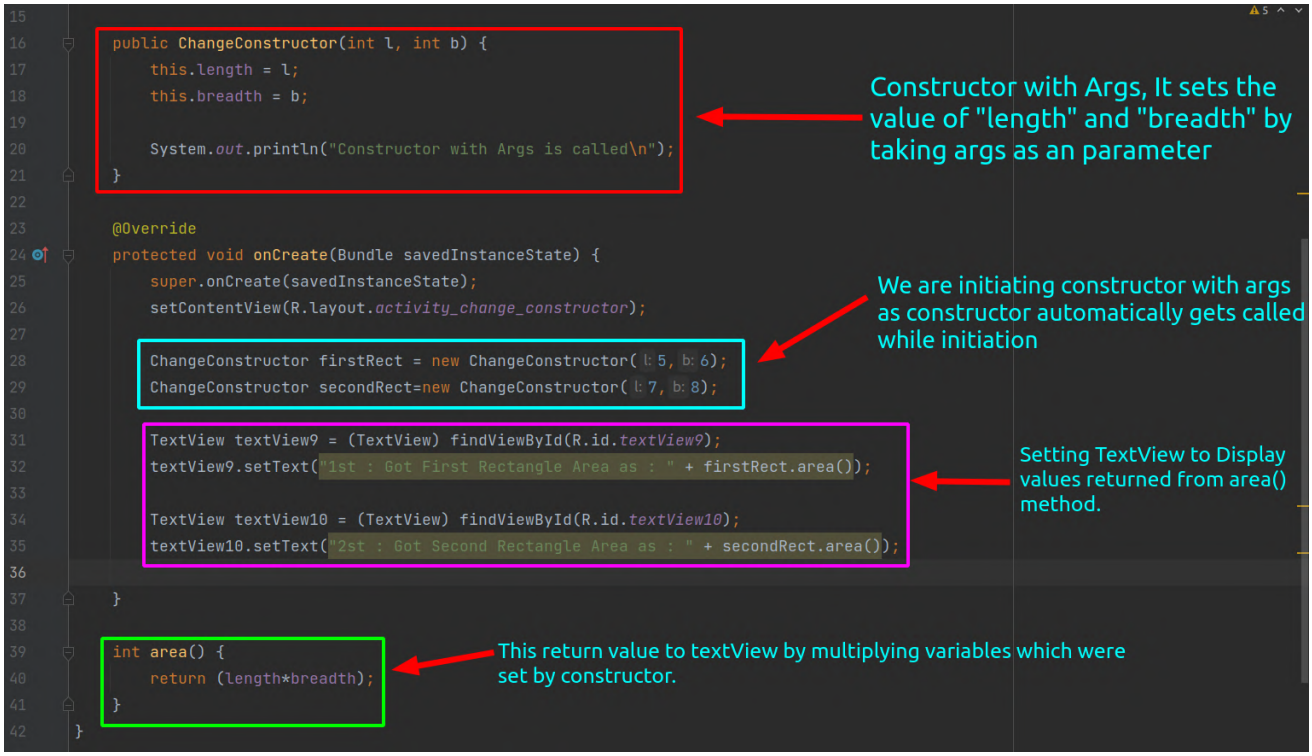
```
❯ frida -U -f "com.example.hooking_with_frida" -l constructor_hook_without_Args.js
     ____
    / _  |   Frida 16.0.8 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Google Pixel (id=192.168.56.102:5555)
Spawned `com.example.hooking_with_frida`. Resuming main thread!
[Google Pixel::com.example.hooking_with_frida ]-> message: {'type': 'send', 'payload': ' Default constructor
 hooked '} data: None
[Google Pixel::com.example.hooking_with_frida ]->
```



b. Constructor with Args

It's the same as the previous one, except we need to overload the constructor as it takes two arguments.

The explanation of code for the activity

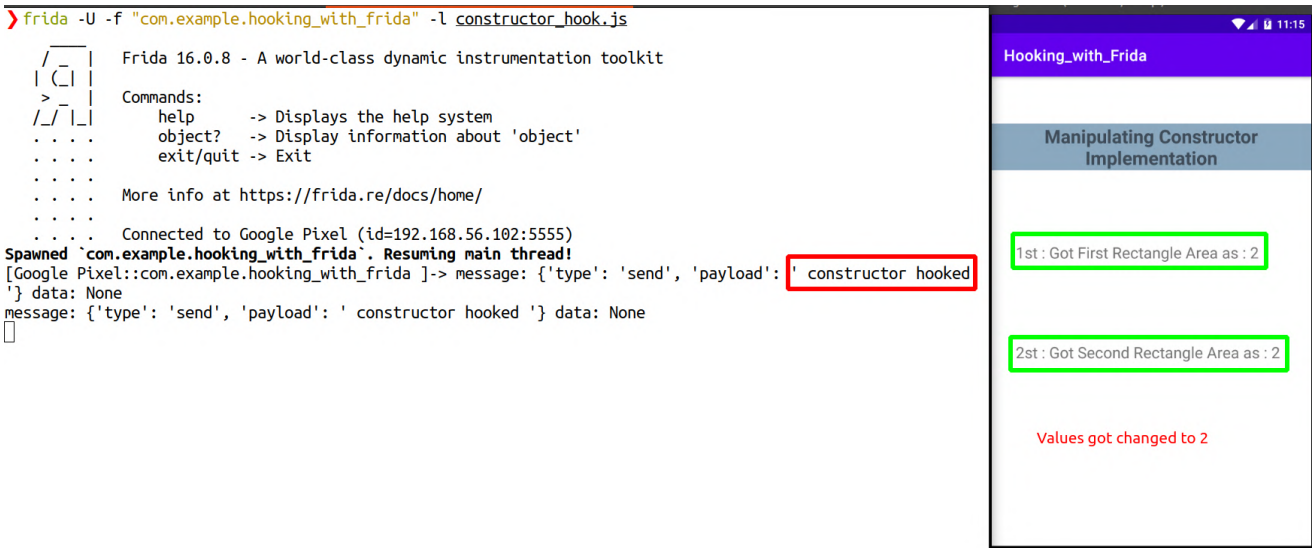


Activity output:

Script to hook the method and change the value of the length and the breadth:

```
            Java.perform(function() {

                var ClassReference = Java.use('com.example.hooking_with_frida.
ChangeConstructor');

                // Overloaded the Constructor as it was taking two 'int' type
args
                ClassReference.$init.overload('int','int').implementation =
function(a,b) {

                    send(" constructor hooked ");

                    //Again calling constructor with different args which will
change the value of length and breadth variable.
                    this.$init(1,2);
                }
            });
```

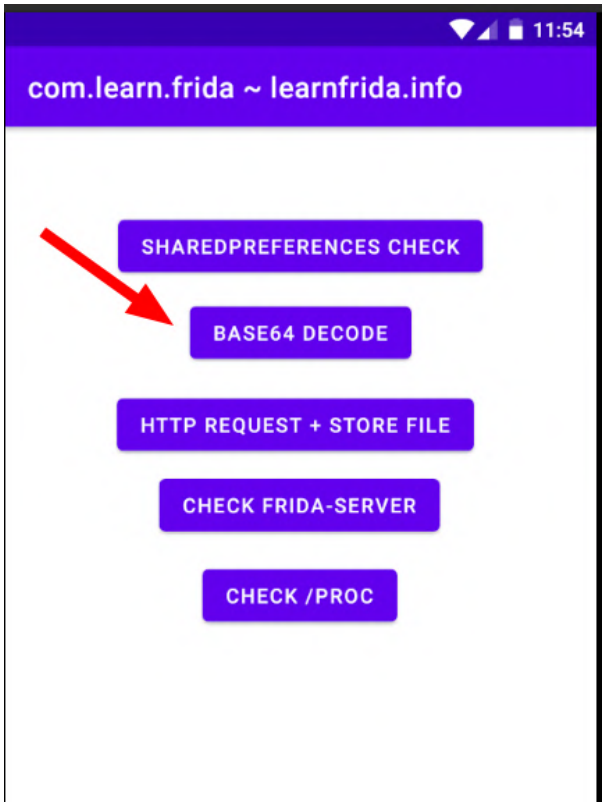
The output after executing the script:

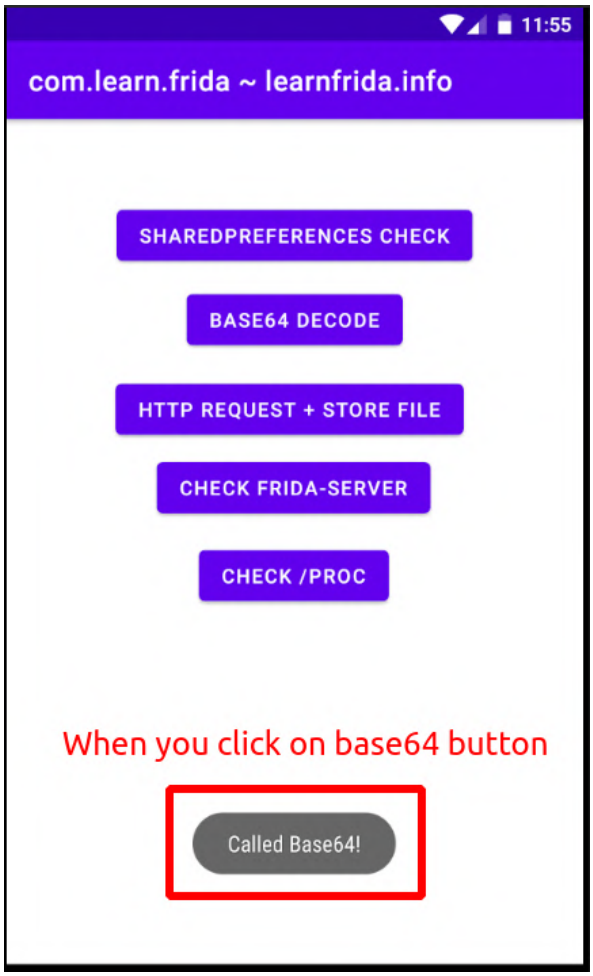


F. Handle ByteArrays value

A common pattern due to how JVM methods are usually designed is that they returned arrays of bytes or `byte[]`. To learn how to handle this data-type, the button base64 decode is available in the below app.

As you can see there is a button names as `BASE64 DECODE`





Lets understand what's' happening at the code level in this app

```
    /* JADX INFO: Access modifiers changed from: private */
    /* renamed from: onCreate$lambda-3  reason: not valid java name */
79  public static final void m315onCreate$lambda3(MainActivity this$0, View it) {
        Intrinsics.checkNotNullParameter(this$0, "this$0");
80      byte[] encodedString = "SSBob3BlIHlvdSBhcmUgaGF2aW5nIGZ1biB3aXRoIEZyaWRhIQ==".getBytes(Charsets.UTF_8);
        Intrinsics.checkNotNullExpressionValue(encodedString, "this as java.lang.String).getBytes(charset)");
81      Base64.Decoder b64decoder = Base64.getDecoder();
82      byte[] decode = b64decoder.decode(encodedString);
        Intrinsics.checkNotNullExpressionValue(decode, "b64decoder.decode(encodedString)");
        new String(decode, Charsets.UTF_8);
83      Toast.makeText(this$0, "Called Base64!", 0).show();
    }
```

Explanation of the code:

This is a Java code for a method named `m315onCreate$lambda3` in a class `MainActivity`. This method takes two arguments, `this$0` of type `MainActivity` and `it` of type `View`.

The method starts by checking if the `this$0` parameter is not null using the `Intrinsics.checkNotNullParameter` method.

Next, it converts a Base64 encoded string "SSBob3BlIHlvdSBhcmUgaGF2aW5nIGZ1biB3aX-RoIEZyaWRhIQ" to a byte array using the `getBytes` method and the `UTF-8` charset. Then, it uses the `Base64.getDecoder()` method to obtain a Base64 decoder, and uses this decoder to decode the byte array obtained in the previous step.

Finally, it converts the decoded bytes back to a string using the `UTF-8` charset, and displays a Toast message "Called Base64!" on the screen.

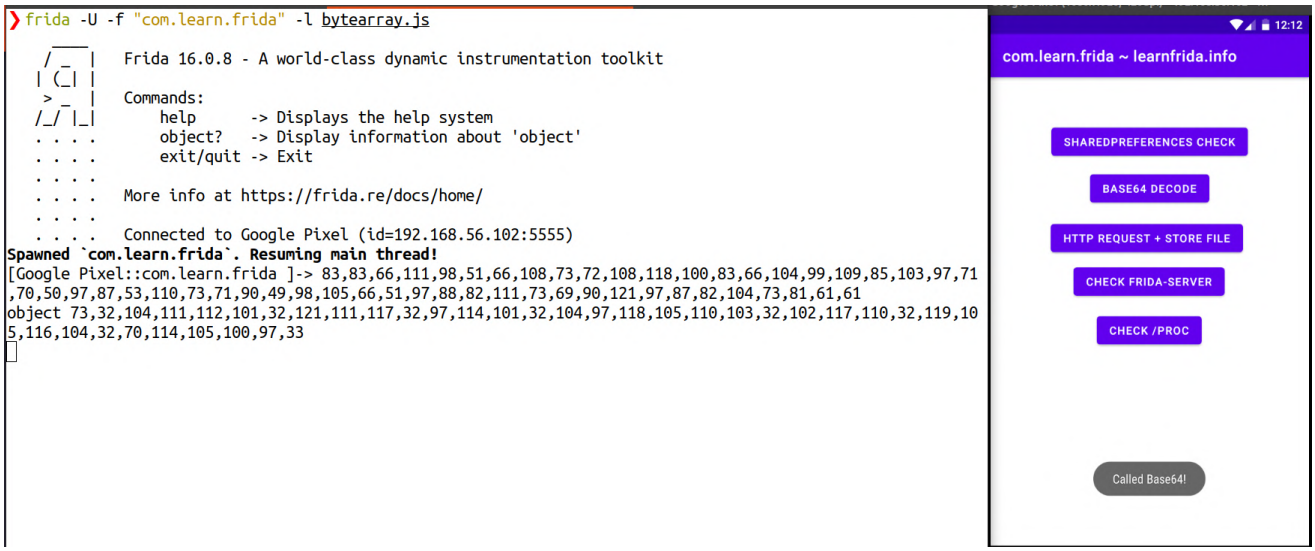
So we want to hook this `.decode` method and get the value passed.

```
/* JADX INFO: Access modifiers changed from: private */
/* renamed from: onCreate$lambda-3  reason: not valid java name */
public static final void m315onCreate$lambda3(MainActivity this$0, View it) {
    Intrinsics.checkNotNullParameter(this$0, "this$0");
    byte[] encodedString = "SSBob3BlIHlvdSBhcmUgaGF2aW5nIGZ1biB3aXRoIEZyaWRhIQ==".getBytes(Charsets.UTF_8);
    Intrinsics.checkNotNullExpressionValue(encodedString, "this as java.lang.String).getBytes(charset)");
    Base64.Decoder b64decoder = Base64.getDecoder();
    byte[] decode = b64decoder.decode(encodedString);
    Intrinsics.checkNotNull
    new String(decode, Char
    Toast.makeText(this$0,
}
```

"The `.decode()` method can be found within the `java.util.Base64` class as you can see in the above image, and it can be accessed through a wrapper obtained with the `getDecoder()` method. To instrument this method using Frida, the target wrapper becomes `java.util.Base64$Decoder`. The argument for the `.decode()` **method is a byte array, represented in Frida as** `[B` With this information, we can now proceed to instrumentation. Here's the basic structure for instrumenting this method:

```
// frida -l ins.js -U learnfrida --no-pause
Java.perform(() =>{
const b64Claz = Java.use('java.util.Base64$Decoder');
    b64Claz.decode.overload('[B').implementation = function (inputString) {
        const retval = this.decode(inputString);
        console.log(inputString);
        console.log(typeof(retval), retval);
        return retval;
    }
});
```

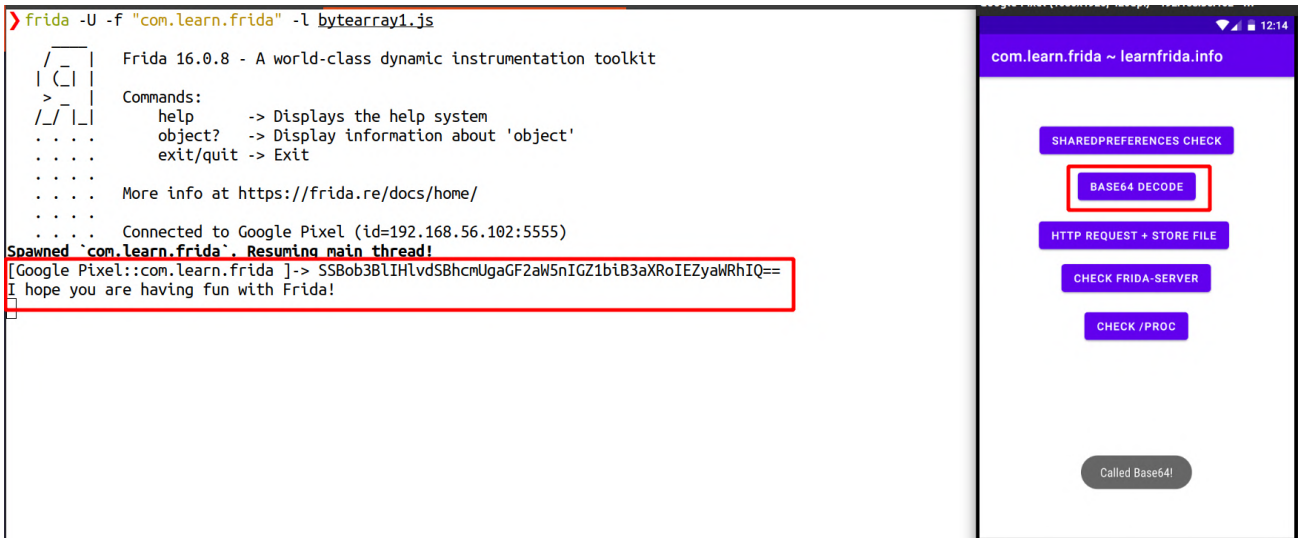
Notice how the overload receives a single `[B` parameter. This is what was previously indicated as the representation of a bytearray data-type. When instrumenting and pressing the Base64 button Frida's REPL returns the following output:



The first set of bytes corresponds the input that the method `.decode()` receives and the second line is the result after calling the function, which is the string to decode. Of course, both of these strings are not really useful for us the way they are now. To get meaningful data the best options are either using an external tool or just do the conversion right within our script. Let's try the latter option!

```
Java.perform(() =>{
const b64Claz = Java.use('java.util.Base64$Decoder');
    b64Claz.decode.overload('[B').implementation = function (inputString) {
        const retval = this.decode(inputString);

        let asciiInputString = []
        for(let i = 0; i < inputString.length; i++) {
            asciiInputString.push(String.fromCharCode(inputString[i]));
        }
        console.log(asciiInputString.join(""));
        let asciiOutputString = []
        for(let i = 0; i < retval.length; i++) {
            asciiOutputString.push(String.fromCharCode(retval[i]));
        }
        console.log(asciiOutputString.join(""));
        return retval;
    }
});
```

After iterating each value of the `Object` representing the bytearray we get the following output:



StackTraces

One way to streamline instrumentation is to capture a trace of prior function calls. This approach can reveal additional functions and code paths of interest without the need for static analysis. Let's take the previous base64 button as an example to demonstrate this concept.

Script for getting the StackTrace of the implementation:

```
Java.perform(() =>{
    const thread = Java.use('java.lang.Thread').$new();

    const b64Claz = Java.use('java.util.Base64$Decoder');
        b64Claz.decode.overload('[B').implementation = function (inputString) {
            const retval = this.decode(inputString);
            const stacktrace = thread.currentThread().getStackTrace();
            return retval;
        }
});
```

The first requirement is to get an instanced Thread object by using java.lang.Thread's wrapper and then invoking $new() to create an instance of it. With the thread object, it is now possible to call its methods including the current thread's StackTrace via .getStackTrace(). This StackTrace is returned as a list thus the output needs postprocessing, the simple way to achieve this is by iterating and printing it via .forEach:

```
Java.perform(() =>{
            const thread = Java.use('java.lang.Thread').$new();

        const b64Claz = Java.use('java.util.Base64$Decoder');
            b64Claz.decode.overload('[B').implementation = function (inputString) {
                const retval = this.decode(inputString);
                const stacktrace = thread.currentThread().getStackTrace();
                stacktrace.forEach( (element) => {
                    console.log(element);
                });
                return retval;
            }
        });
```

After instrumenting the application and pressing the button, Frida's REPL prints the following output:

```
[Android Emulator 5554::learnfrida ]-> dalvik.system.VMStack.
getThreadStackTrace(Native Method)
        java.lang.Thread.getStackTrace(Thread.java:1736)
        java.util.Base64$Decoder.decode(Native Method)
        com.learn.frida.MainActivity.onCreate$lambda-3(MainActivity.kt:72)
        com.learn.frida.
MainActivity.$r8$lambda$b7XbkR9rj5AI57kvCCFslXJKa7Q(Unknown Source:0)
        com.learn.frida.MainActivity$$ExternalSyntheticLambda0.onClick(Unknown
Source:2)
        android.view.View.performClick(View.java:7448)

com.google.android.material.button.MaterialButton.performClick(MaterialButton.
java:1194)
        android.view.View.performClickInternal(View.java:7425)
        android.view.View.access$3600(View.java:810)
        android.view.View$PerformClick.run(View.java:28305)
        android.os.Handler.handleCallback(Handler.java:938)
        android.os.Handler.dispatchMessage(Handler.java:99)
        android.os.Looper.loop(Looper.java:223)
        android.app.ActivityThread.main(ActivityThread.java:7656)
        java.lang.reflect.Method.invoke(Native Method)
        com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.
java:592)
        com.android.internal.os.ZygoteInit.main(ZygoteInit.java:947)
```

As it can be seen, the provided information is indeed very valuable (despite our controlled simple example). What can be observed from the StackTrace is that the event that triggered the call is `.onClick`within `onCreate` and then our instrumented function was called.

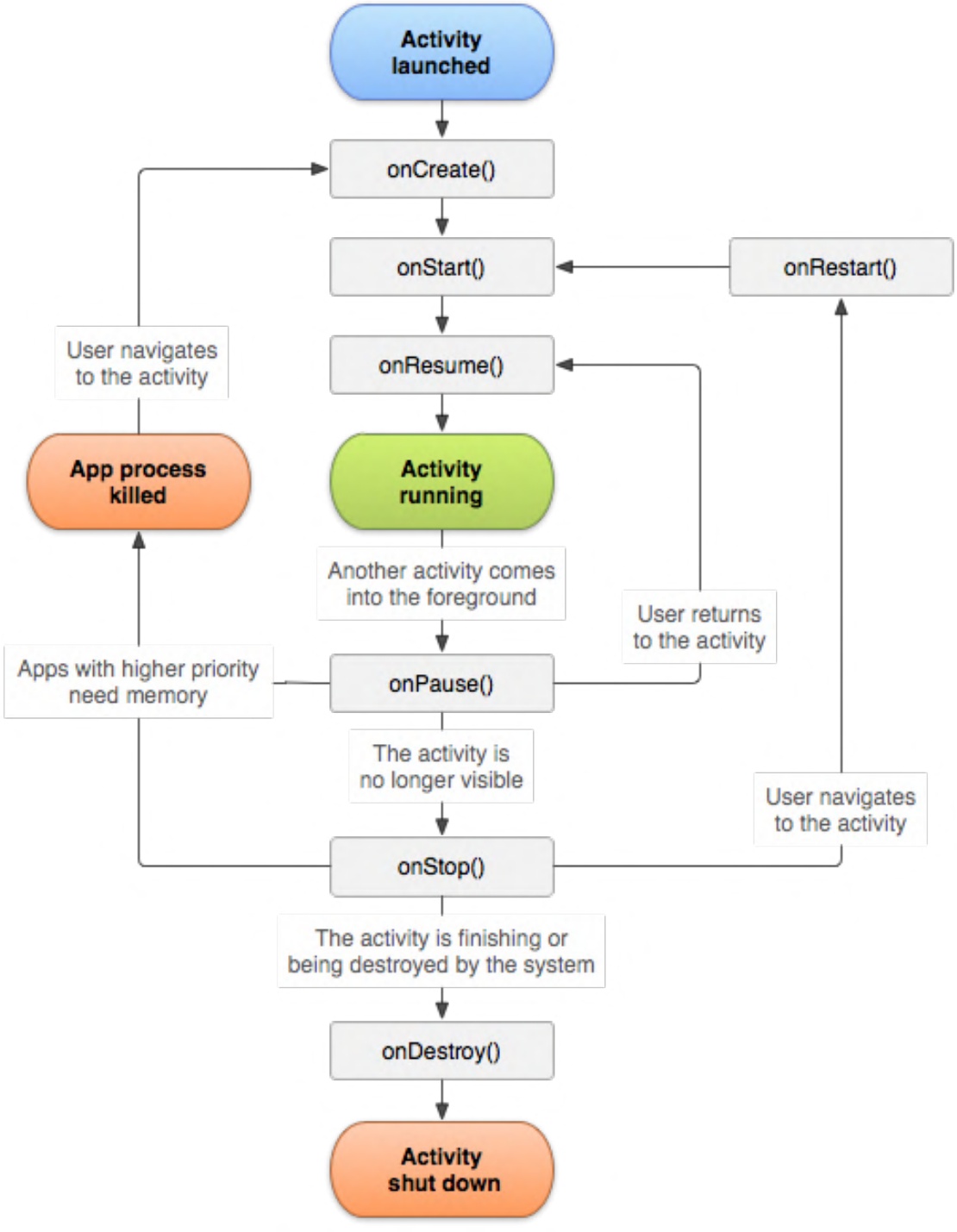
Chapter 6

Monitoring Android App Components

1. Hooking into the Android Lifecycle

The Android lifecycle describes the series of states that an Android app goes through from the moment it is launched until it is closed by the user or the system. An Android activity goes through six major lifecycle stages or callbacks. These are: **onCreate() , onStart() , onResume() , onPause() , onStop() , and onDestroy()**.

You can learn about all these lifecycles from this article.



Let's take an application in which we will hook the Android Activity Lifecycle methods.

So let's see how the application works.

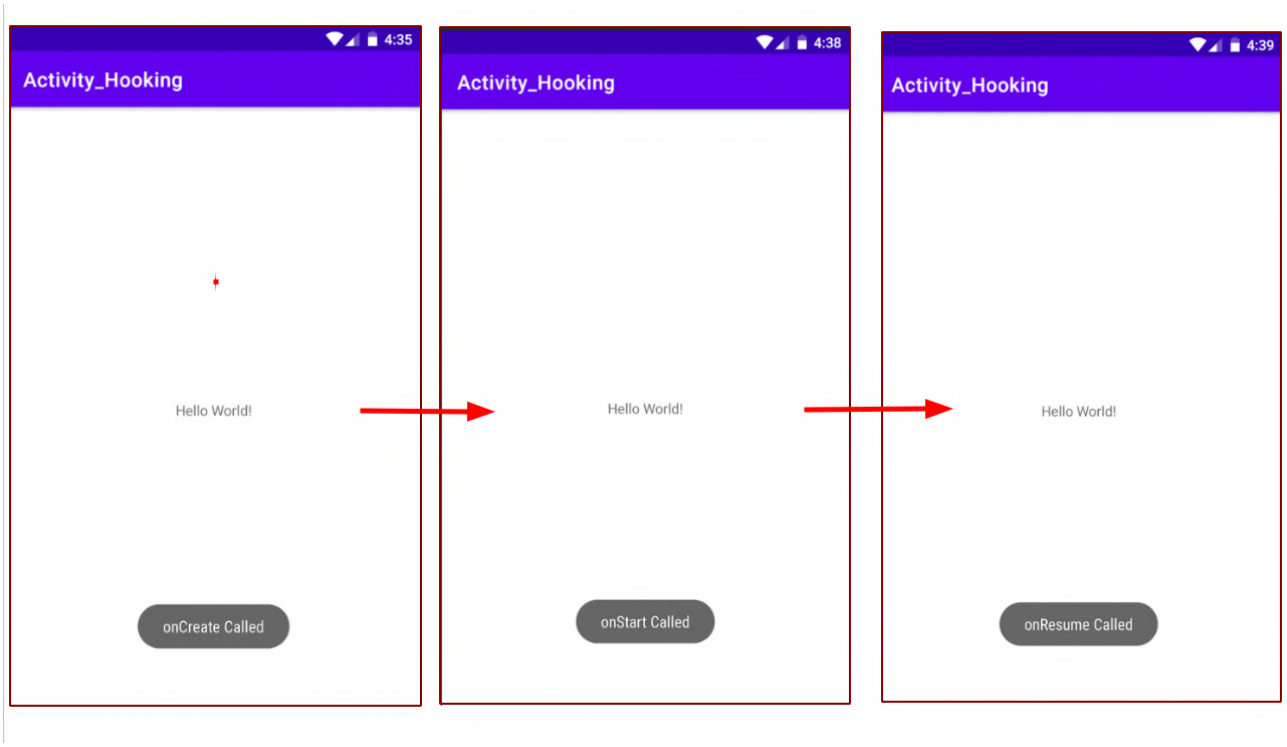
1. When you open the application you will see the `onCreate()` method getting executed, is called

2. As you are familiar with the Activity Lifecycle, below is the implementation of methods:

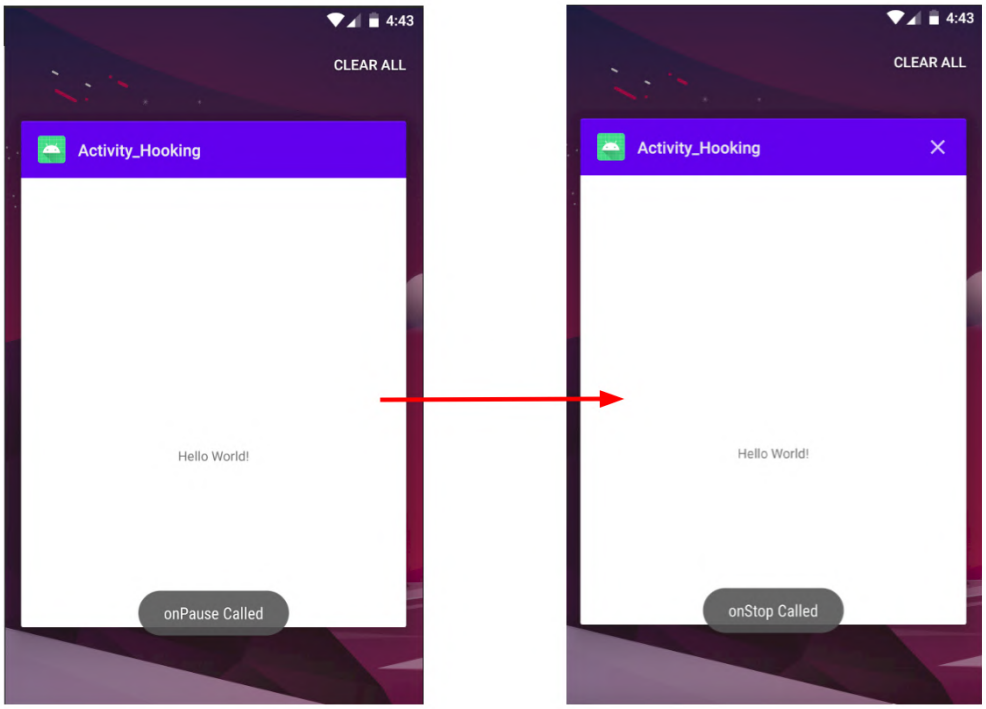
A. For all the methods we have implemented a toast message

```
    @Override
// androidx.fragment.app.FragmentActivity, androidx.activity.ComponentActivity, andr
Activity
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toast.makeText(getApplicationContext(), "onCreate Called", 1).show();
    }

    /* JADX INFO: Access modifiers changed from: protected */
    @Override // androidx.appcompat.app.AppCompatActivity, androidx.fragment.app.Fra
    public void onStart() {
        super.onStart();
        Toast.makeText(getApplicationContext(), "onStart Called", 1).show();
    }

    @Override // android.app.Activity
    protected void onRestart() {
        super.onRestart();
        Toast.makeText(getApplicationContext(), "onRestart Called", 1).show();
    }
```

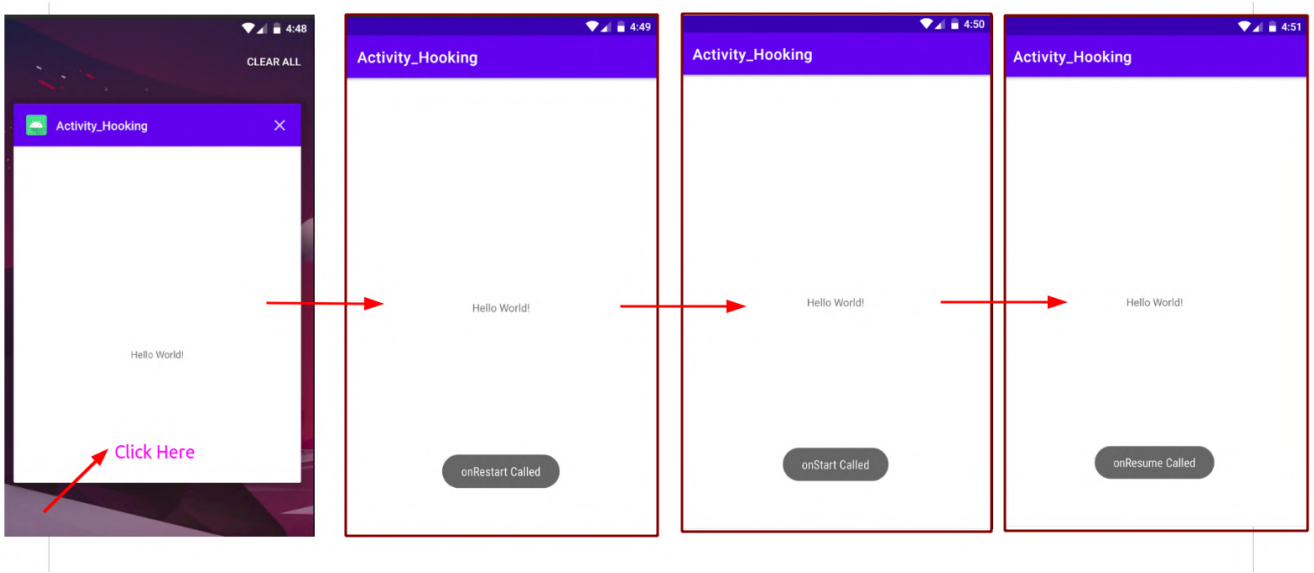
B. When you open the application : `onCreate()` → `onStart()` → `onResume()` method gets called.



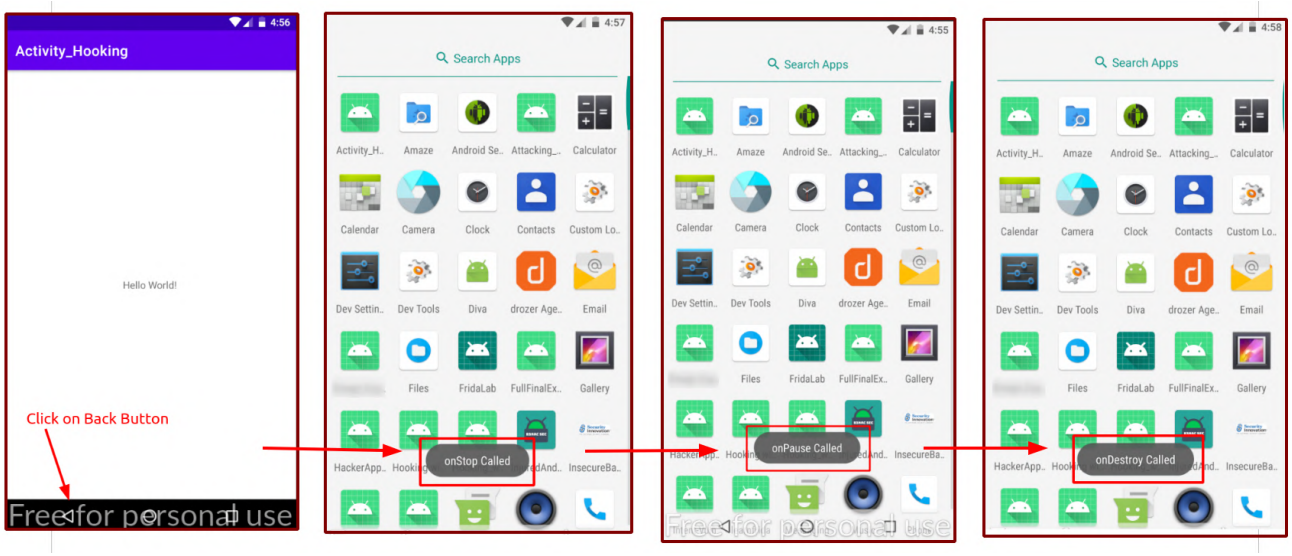
C. When you move the application to minimize the methods, the methods called are : `onPause()` → `onStop()`



D. When you take your application from minimizing the menu to the application, the methods called are : `onRestart()` → `onStart()` → `onResume()`



E. When you press the back button the activity gets destroyed, the methods called are : `onStop` → `onPause()` → `onDestroy()`



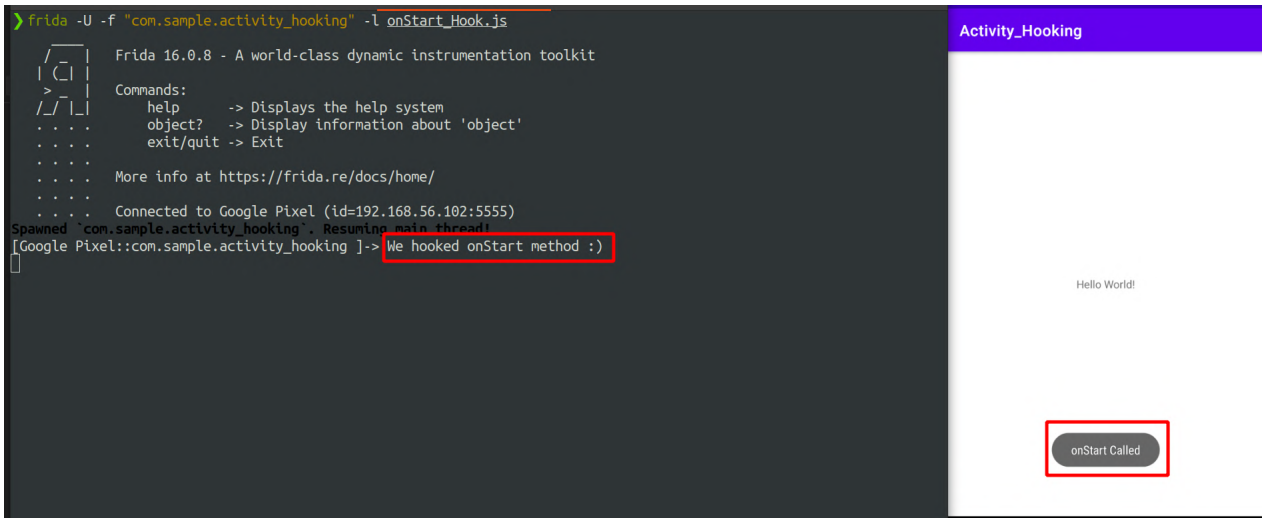
3. Now we are going to hook the `onStart` method of the Activity Lifecycle. The reset of the methods are same to hook nothing much rather than changing method name in Javascript file.

Now, we are going to hook the `onStart` method of the Activity Lifecycle. The rest of the methods are the same to hook, with no significant changes other than modifying the method names in the JavaScript file.
So here we are going to hook the **`onStart`** method and log some text to the console from which we will come to know the onStart method is hooked.

4. We are going to hook the method using the below script

```
        Java.perform(function() {

            //Getting Reference of Acitivity Class
            var Activity = Java.use("android.app.Activity");

            //Changing the implementation of onStart Method
            Activity.onStart.implementation = function() {

                // This line is calling the onStart method of the current object
(i.e. this), and storing the result in a variable called ret.
                var ret = this.onStart();
                // Logging something in console so that we can know this method
is hooked
                console.log("We hooked onStart method :)");
                //return the result of onStart method which is stored in ret
                return ret;

            };
        });
```

5. Executing the above script



For other methods, you can achieve the same by hooking, but you need to change the method name and write the logic you want to implement.

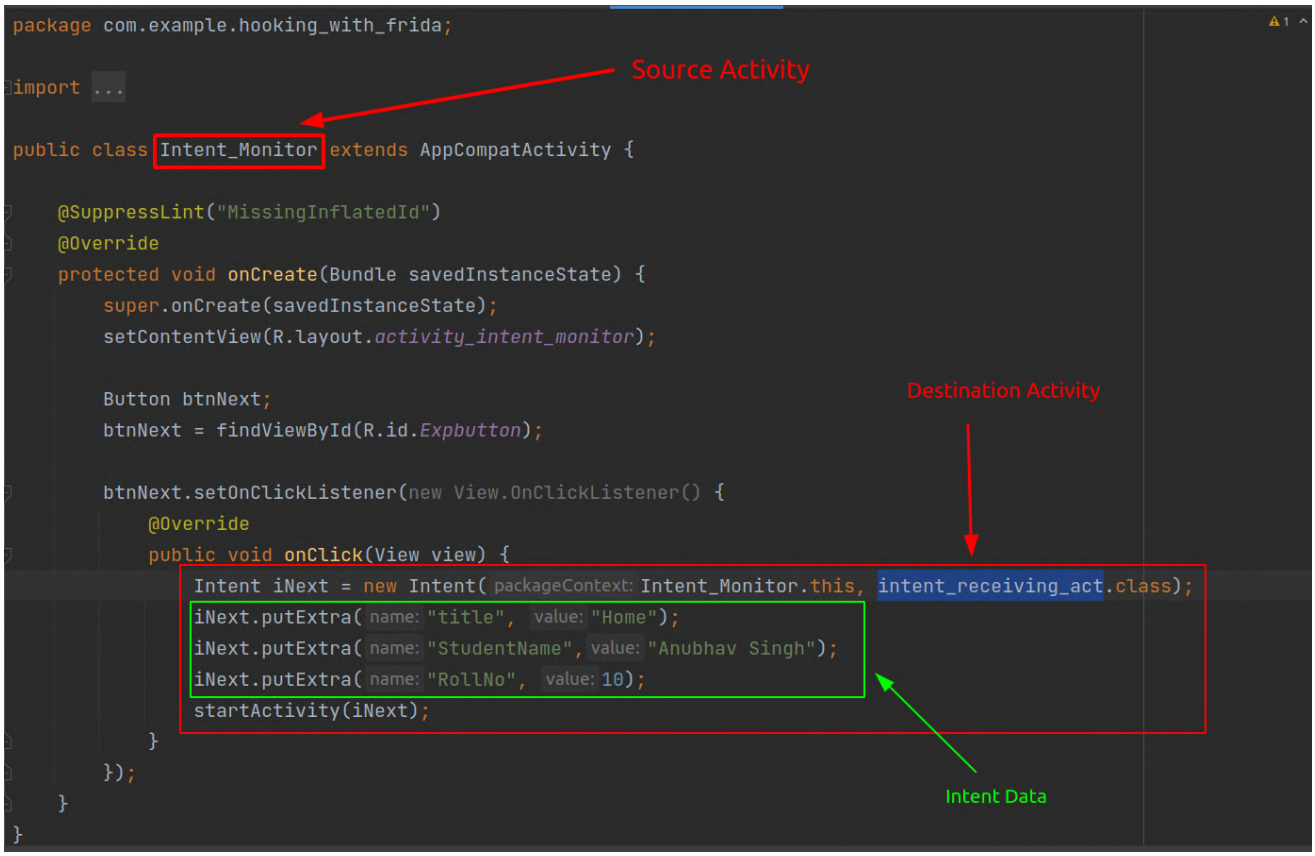
2. Monitor Intents

A. Outgoing Intent

In Android, Incoming Intent refers to an object that is sent from one component to another within an application or between different applications. It is a fundamental concept in the Android operating system, and is used to facilitate communication and data exchange between different components of an app, such as activities, services, and broadcast receivers. Outgoing Intent refers to an Intent object that is used to start an activity or service inside or outside of the current application. For example, an Android application might use an Outgoing Intent to launch a web browser with a specific URL, or to open the camera app to take a photo.

In this example, we are going to hook the Intent data which is going to another activity. So let's see how we can perform this.

a. We have created an application in which we have one activity named `Intent_Monitor`. In this activity we have defined the Intent with extra data which will be passed to the `intent_receiving_act` activity. We need to hook into this Intent and determine both the source activity and the destination activity to which this Intent data is being passed, as well as the data being transmitted between the activities using Frida.



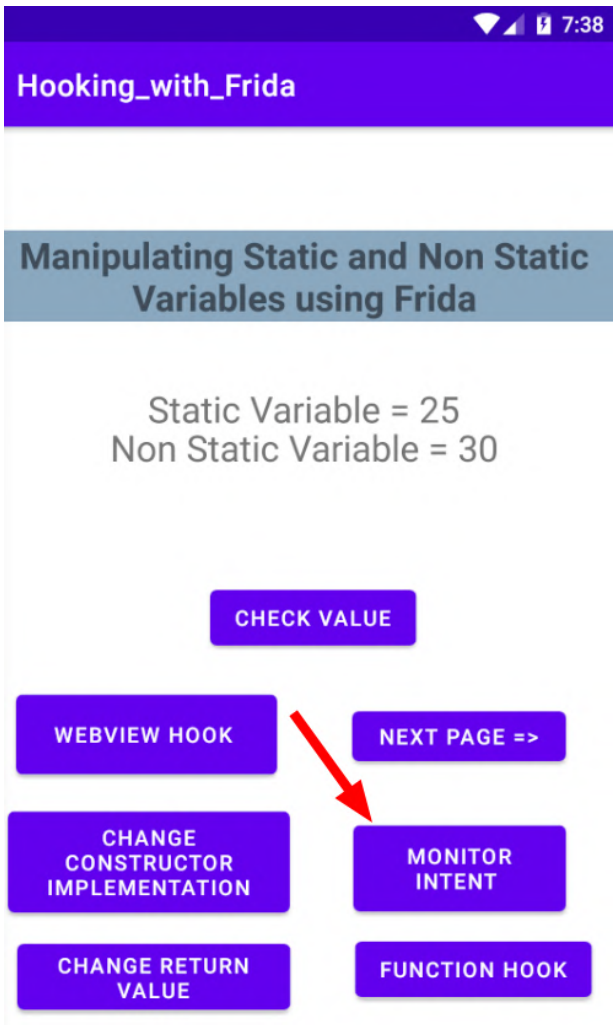

```
package com.example.hooking_with_frida;

import ...

public class Intent_Monitor extends AppCompatActivity {

    @SuppressLint("MissingInflatedId")
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_intent_monitor);

        Button btnNext;
        btnNext = findViewById(R.id.Expbutton);

        btnNext.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Intent iNext = new Intent( packageContext: Intent_Monitor.this, intent_receiving_act.class);
                iNext.putExtra( name: "title",  value: "Home");
                iNext.putExtra( name: "StudentName", value: "Anubhav Singh");
                iNext.putExtra( name: "RollNo",  value: 10);
                startActivity(iNext);
            }
        });
    }
}
```

b. Writing a Frida script to hook this Intent and find the data.

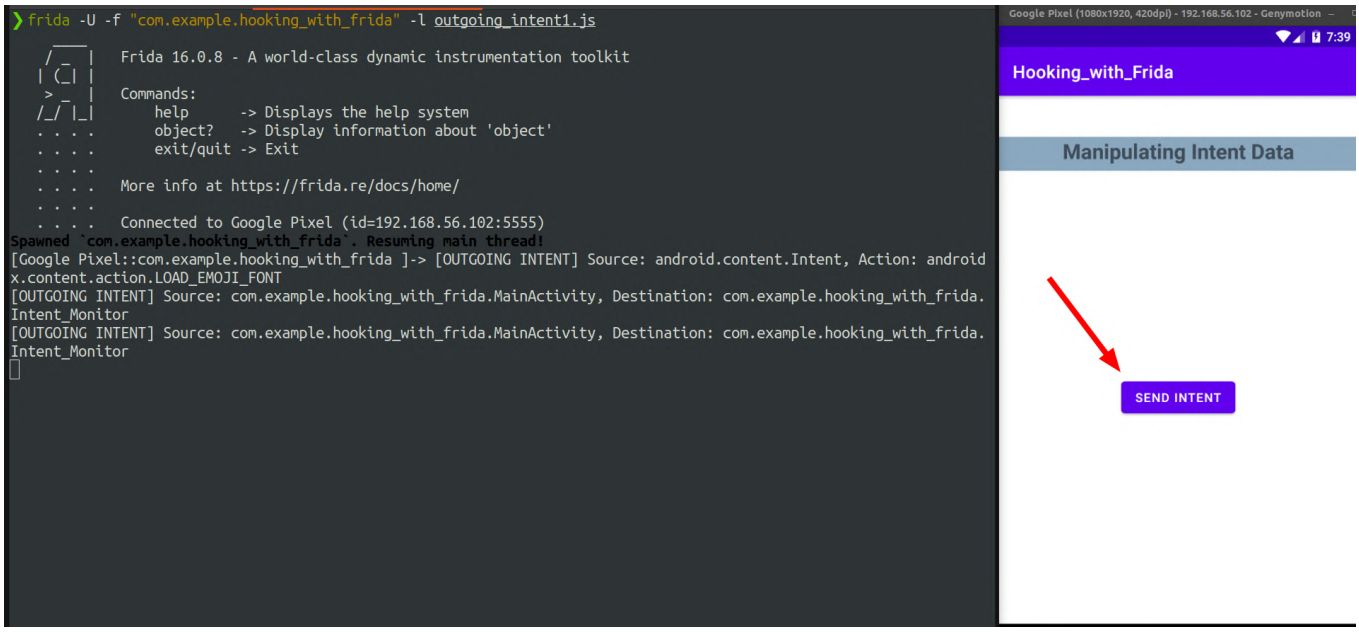
```
            Java.perform(function() {

              // Getting reference of intent class
              var Intent = Java.use("android.content.Intent");

              // Overloaded constructor with paramater as java.lang.String type
to get action of intent
              Intent.$init.overload('java.lang.String').implementation =
function(action) {
                console.log("[OUTGOING INTENT] Source: " + this.$className + ",
Action: " + action);
                return this.$init(action);
              };
              // Overloaded constructor with paramater as android.content.Context
and java.lang.Class type
              Intent.$init.overload('android.content.Context', 'java.lang.
Class').implementation = function(context, cls) {
                console.log("[OUTGOING INTENT] Source: " + context.getClass().
getName() + ", Destination: " + cls.getName());
                return this.$init(context, cls);
              };
              // Overloaded putExtra with two paramater with parameter type both
as string as we have passed string values in 1st and 2nd putextra method (title
and StudentName).
```

```
                Intent.putExtra.overload('java.lang.String', 'java.lang.String').
implementation = function(name, value) {
                  console.log("[EXTRA] " + name + " = " + value);
                  return this.putExtra(name, value);
                };

                // Overloaded putExtra with two paramater with parameter type one
as as string and one as int as we have passed int value in 3rd putextra method
(RollNo).
                Intent.putExtra.overload('java.lang.String', 'int').implementation
= function(name, value) {
                  console.log("[EXTRA] " + name + " = " + value);
                  return this.putExtra(name, value);
                };


            });
```

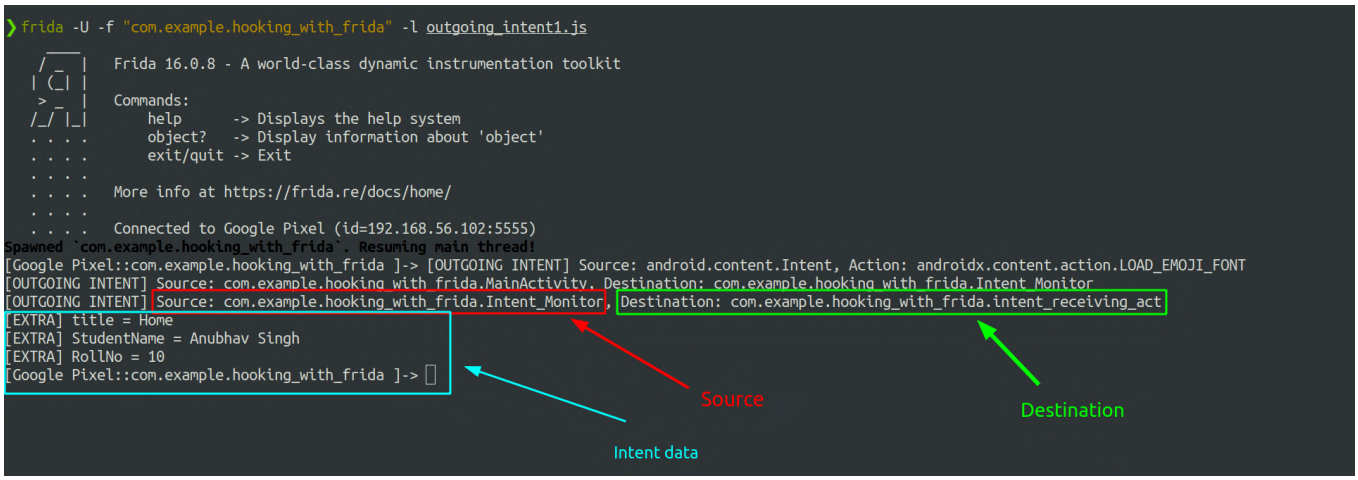
c. Click on Monitor Intent



d. Click on Send Intent



e. Hook intent using Frida



This is how you can hook Intent in Android and can get values associated with it. This is very useful while performing a pentest as sometimes sensitive data gets passed between intent like keys, secrets, etc. so by hooking Intent we can directly get it.

B. Incoming Intent

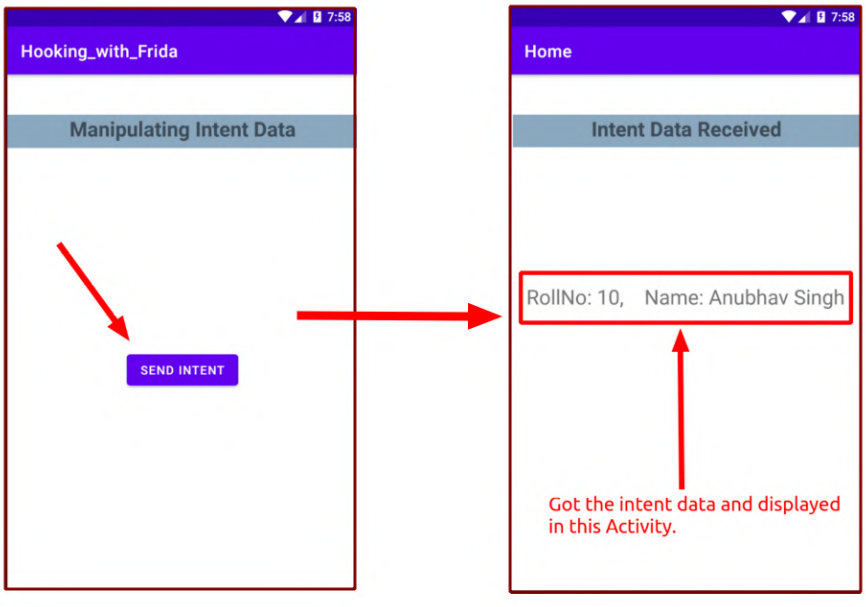
In Android development, an "Incoming Intent" refers to an Intent object that is received by an activity, service, or broadcast receiver. An Intent is a messaging object that can be used to communicate between components of an application or between different applications.

When an Incoming Intent is received, the component can extract information such as the action to be performed, any data included in the Intent, and any flags or parameters that modify the behavior of the Intent.

For example, an Android activity might receive an incoming Intent that requests it to display a specific piece of data or perform a specific action, such as to view a web page or display a photo. Similarly, a service might receive an Incoming Intent that instructs it to perform a background task, such as to download data or upload a file.

So we are going to hook incoming Intent here using Frida.
Example:
a. We have created an application with one activity that receives an Intent. By extracting the data from the Intent, it displays it on the activity using a TextView.

```
package com.example.hooking_with_frida;

import ...

public class intent_receiving_act extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_intent_receiving);

        Intent fromAct = getIntent();
        String title = fromAct.getStringExtra( name: "title");
        String studentName = fromAct.getStringExtra( name: "StudentName");
        System.out.println(title);
        System.out.println(studentName);

        int rollNo = fromAct.getIntExtra( name: "RollNo",  defaultValue: 0);

        TextView txtStudentInfo = findViewById(R.id.txtStudentInfo);
        txtStudentInfo.setText("   RollNo: "+rollNo+",    Name: "+ studentName);
        getSupportActionBar().setTitle(title);
```

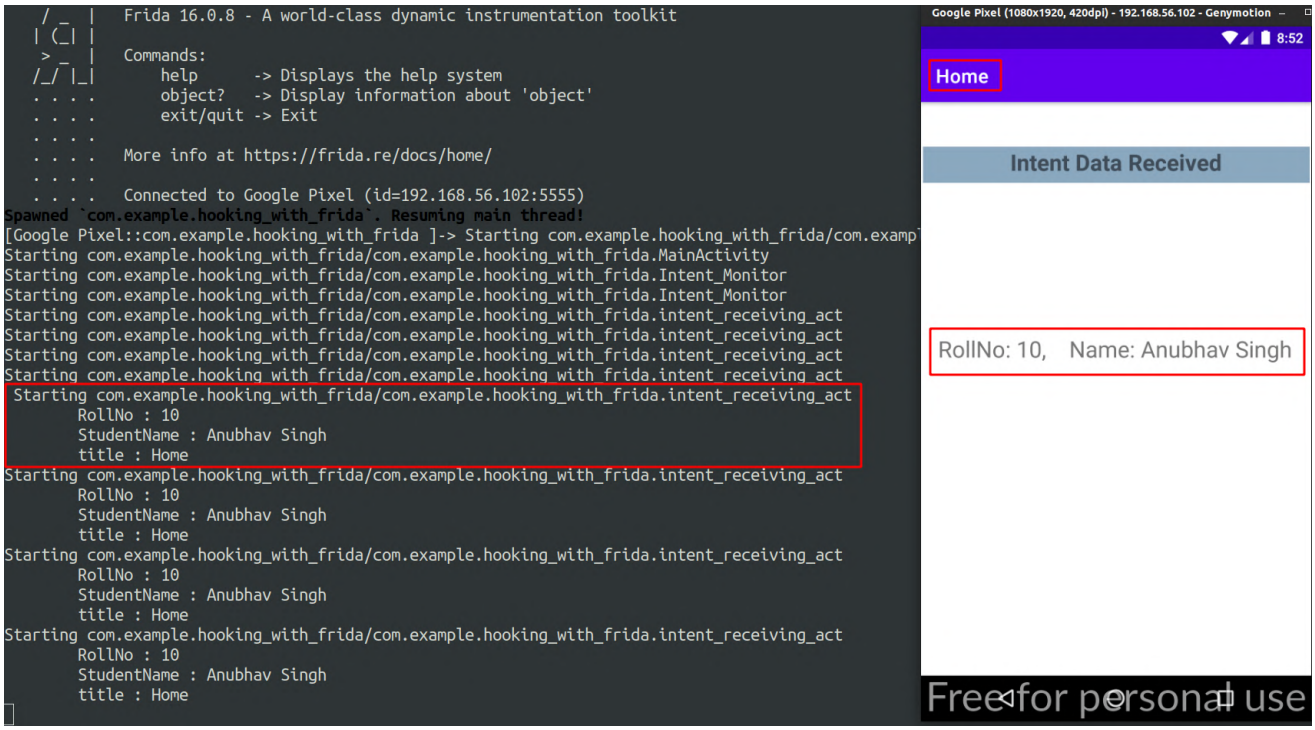


b. Let's write a Frida script to hook this Incoming Intent so that we can see the being received.

```
            Java.perform(function() {
                var act = Java.use("android.app.Activity");

                act.getIntent.implementation = function() {
                    var intent = this.getIntent() // We are getting the retun
value of this method so that we can pass this wvalue when script is executed so
that working of the method will not be disturbed

                    // We can retrive components of intent using this
                    var cp = intent.getComponent()

                    // console.log(cp)
                    // --> ComponentInfo{com.example.hooking_with_frida/com.
example.hooking_with_frida.intent_receiving_act}

                    console.log("Starting " + cp.getClassName())
                   // -> Starting com.example.hooking_with_frida.intent_
receiving_act

                    var ext = intent.getExtras();

                    // console.log(ext)
                    // Bundle[{RollNo=10, StudentName=Anubhav Singh, title=Home}]

                    // Now we need to iterate this Bundle object
                    if (ext) {
                        var keys = ext.keySet()

                        var iterator = keys.iterator()

                        while (iterator.hasNext()) {
                            var k = iterator.next().toString()

                            // console.log(k)
                            // --> Roll No
                            //     StudentName
                            //     title

                            var v = ext.get(k)

                            // console.log(v)
                            // --> 10
                            //     Anubhav
                            //     Hooking
                            if (v) {

                                console.log("\t" + k + ' : ' + v.toString())
```

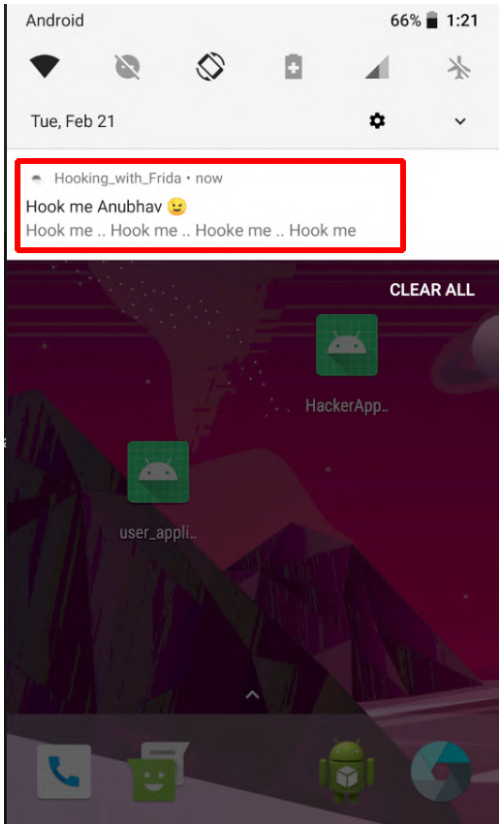
```
                                // Roll No : 10 -> StudentName : Anubhav -> title
    : Intent Hooking

                            } else {
                                console.log("\t" + +k + ' : null')
                            }
                        }
                    }
                    return intent;
                };
            })
```

c. Executing the above script

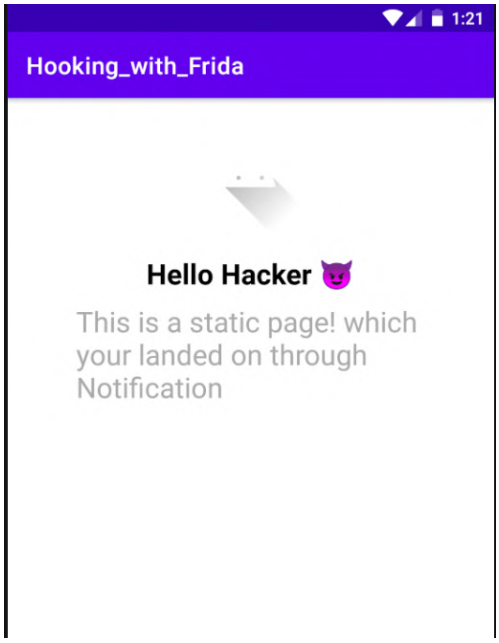


d. As you can see above we have logged the Incoming Intent data.

a. You can see a notification.



b. When a user will click on this notification, an activity will open.



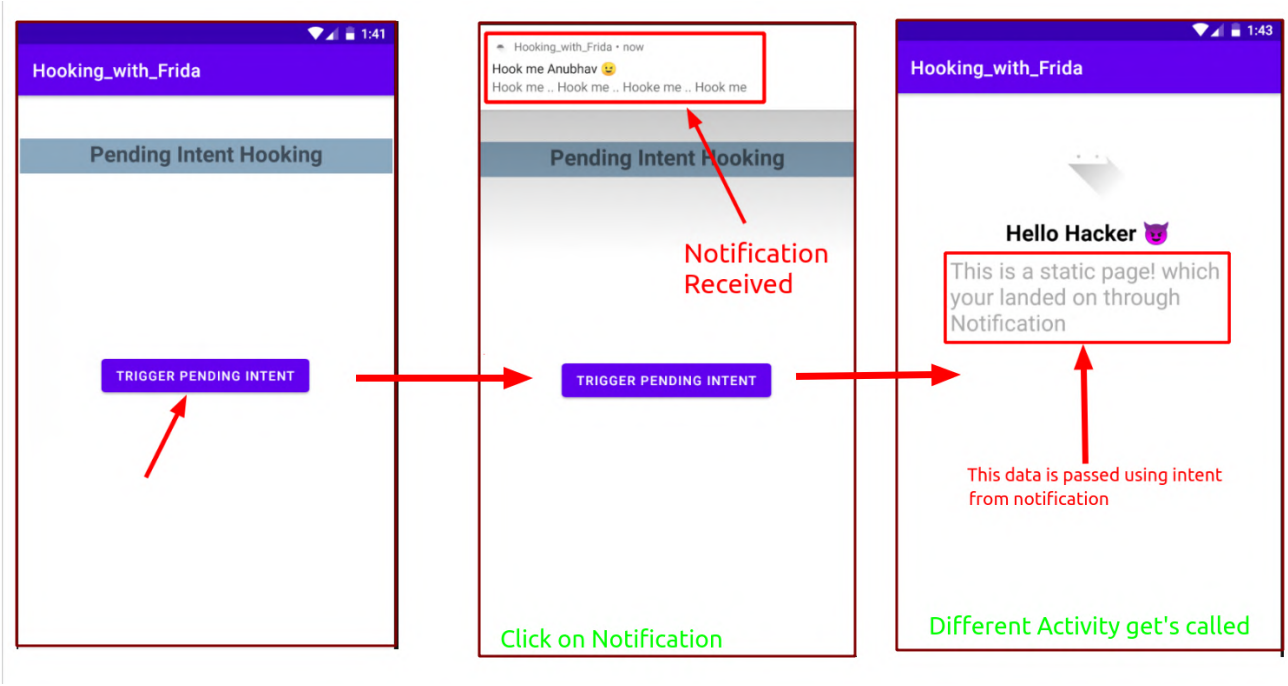
So what we are going to do here is hook this pending Intent and log in the console which activity this pending Intent is calling and what data is being passed through it.

Let's start

(i) We have created an **`pending_Intent`** activity in which we are creating the pending Intent and setting up an OnClickListener so that we can trigger the Intent.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_pending_intent);

    trigger_pendInt = (Button) findViewById(R.id.trigger_pendInt);
    trigger_pendInt.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {

            createNotificationChannel();

            // Create a notification
            NotificationCompat.Builder builder = new NotificationCompat.Builder( context: pending_Intent.this,  channelId: "channelId")
                    .setSmallIcon(R.drawable.ic_launcher_foreground)
                    .setContentTitle("Hook me Anubhav \uD83D\uDE09")
                    .setContentText("Hook me .. Hook me .. Hooke me .. Hook me")
                    .setPriority(NotificationCompat.PRIORITY_HIGH);

            // Create a pending intent
            Intent intent = new Intent( packageContext: pending_Intent.this, Pending_Intent_Second.class);
            intent.putExtra( name: "message",  value: "This is a static page! which your landed on through Notification");
            PendingIntent pendingIntent = PendingIntent.getActivity( context: pending_Intent.this, REQUEST_CODE, intent, PendingIntent.FLAG_UPDATE_CURRENT);

            // Set the pending intent to the notification
            builder.setContentIntent(pendingIntent);
            builder.setAutoCancel(true);
```

(ii) This is the application.



(iii) The data is passed from notification to activity using **`intent.putExtra()`**

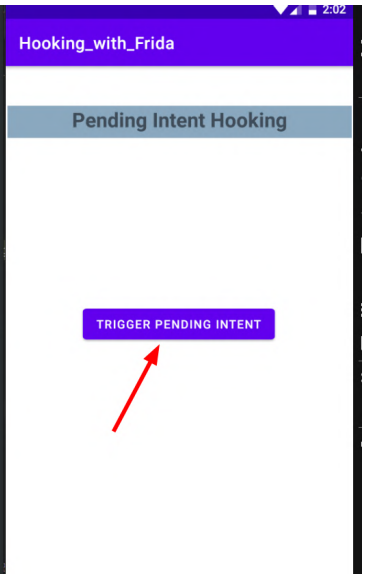
```
// Create a pending intent
Intent intent = new Intent( packageContext: pending_Intent.this, Pending_Intent_Second.class);
intent.putExtra( name: "message",  value: "This is a static page! which your landed on through Notification");
PendingIntent pendingIntent = PendingIntent.getActivity( context: pending_Intent.this, REQUEST_CODE, intent, PendingIntent.FLAG_UPDATE_CURRENT);
```

(iv) Now we are going to hook this pending Intent and log data using the below script.

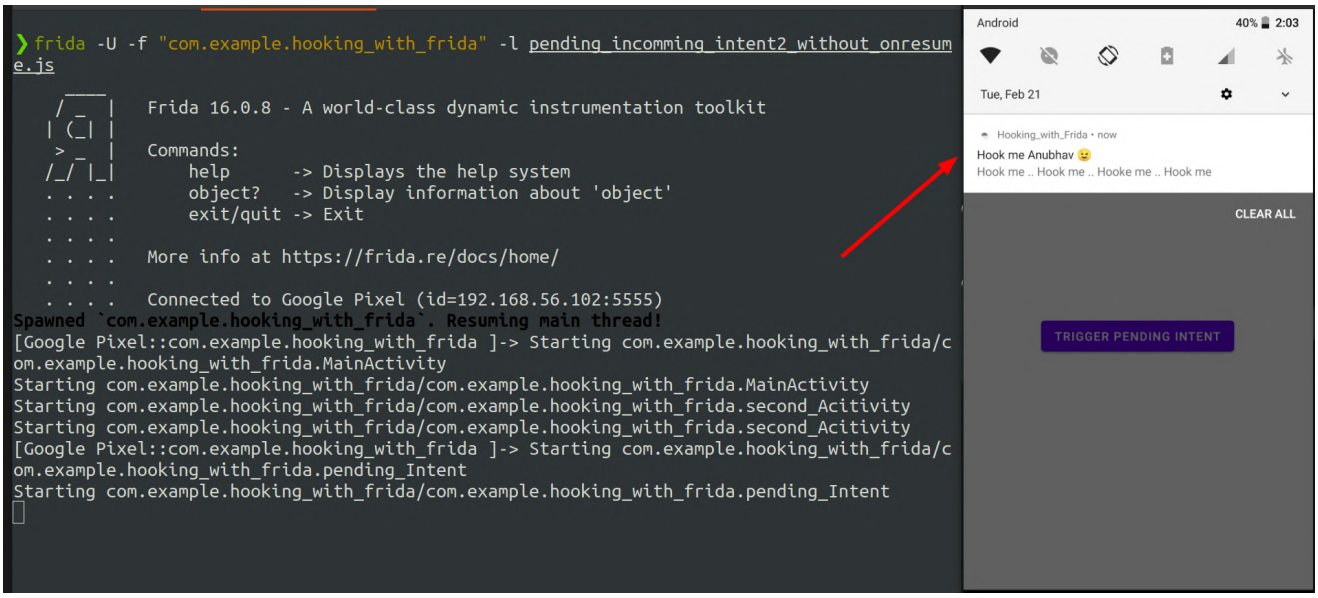
```
Java.perform(function() {
    var act = Java.use("android.app.Activity");
    act.getIntent.overload().implementation = function() {
        var intent = this.getIntent()
        var cp = intent.getComponent()

        console.log("Starting " + cp.getPackageName() + "/" + cp.getClassName())

        var ext = intent.getExtras();

        if (ext) {
            var keys = ext.keySet()

            var iterator = keys.iterator()

            while (iterator.hasNext()) {
                var k = iterator.next().toString()

                var v = ext.get(k)
                if (v) {
                    console.log("\t" + k + ' : ' + v.toString())
                } else {
                    console.log("\t" + +k + ' : null')
                }
            }
        }
        return intent;
    };
});
```

(v) Run the script and click on this button

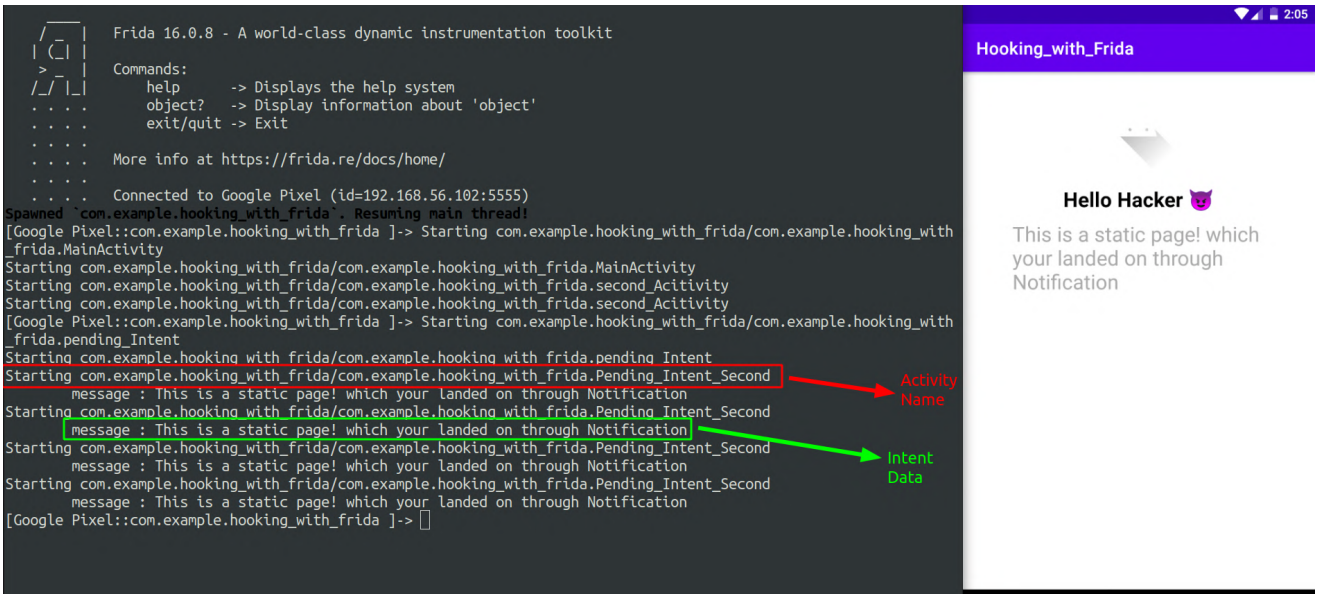
```
❯ frida -U -f "com.example.hooking_with_frida" -l pending_incomming_intent2_without_onresume.js
     ____
    / _  |   Frida 16.0.8 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Google Pixel (id=192.168.56.102:5555)
Spawned `com.example.hooking_with_frida`. Resuming main thread!
[Google Pixel::com.example.hooking_with_frida ]-> Starting com.example.hooking_with_frida/com.example.hooking_with_frida.MainActivity
Starting com.example.hooking_with_frida/com.example.hooking_with_frida.MainActivity
Starting com.example.hooking_with_frida/com.example.hooking_with_frida.second_Acitivity
Starting com.example.hooking_with_frida/com.example.hooking_with_frida.second_Acitivity
[Google Pixel::com.example.hooking_with_frida ]-> Starting com.example.hooking_with_frida/com.example.hooking_with_frida.pending_Intent
Starting com.example.hooking_with_frida/com.example.hooking_with_frida.pending_Intent
```



(vi) Click on the notification



(vii) We have hooked the activity name and Intent data using Frida.

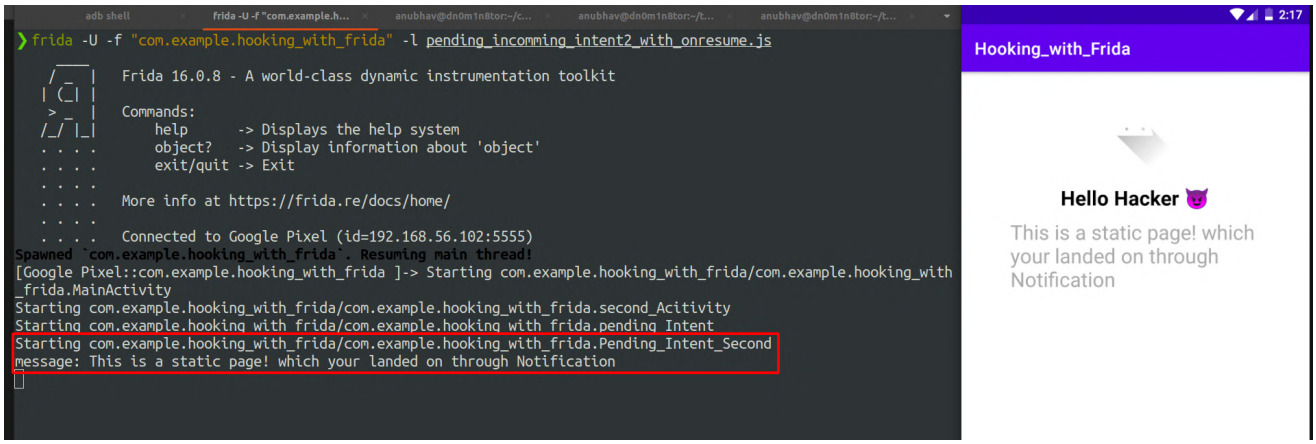


But you can see this data is getting logged multiple times because we are hooking into the `getIntent()` method which can be called multiple times during the lifecycle of an activity. To avoid this, you can check if the activity you are interested in is currently in the foreground before logging the Intent data. So we can solve this issue by making some changes in our script.

```
Java.perform(function() {
    var Activity = Java.use("android.app.Activity");

    //We are just hooking now in onResume method of Activity
    Activity.onResume.implementation = function() {
```

```
            this.onResume();
            var intent = this.getIntent();
            if (intent) {
                var component = intent.getComponent();

                    console.log("Starting " + component.getPackageName() + "/" +
component.getClassName());
                    var extras = intent.getExtras();
                    if (extras) {
                        var keys = extras.keySet().toArray();
                        for (var i = 0; i < keys.length; i++) {
                            var key = keys[i].toString();
                            var value = extras.get(key);
                            console.log(key + ": " + value);
                        }
                    }

            }
        };
    });
```

Executing the script

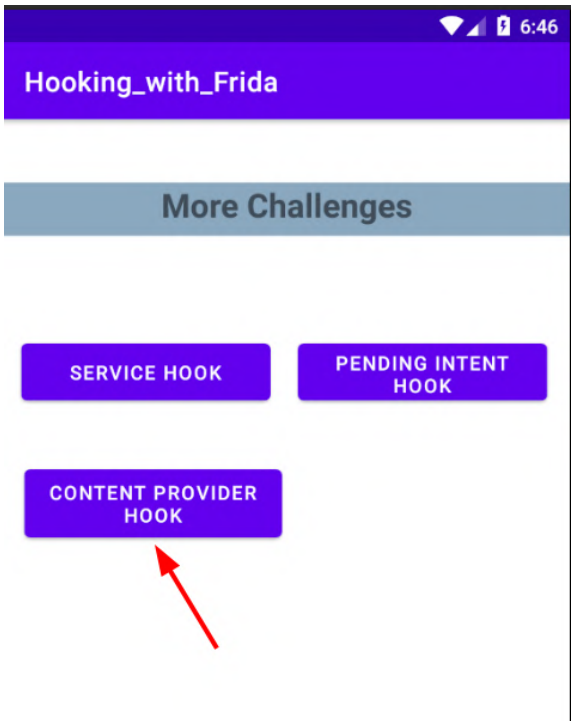


3. Monitor Content Provider

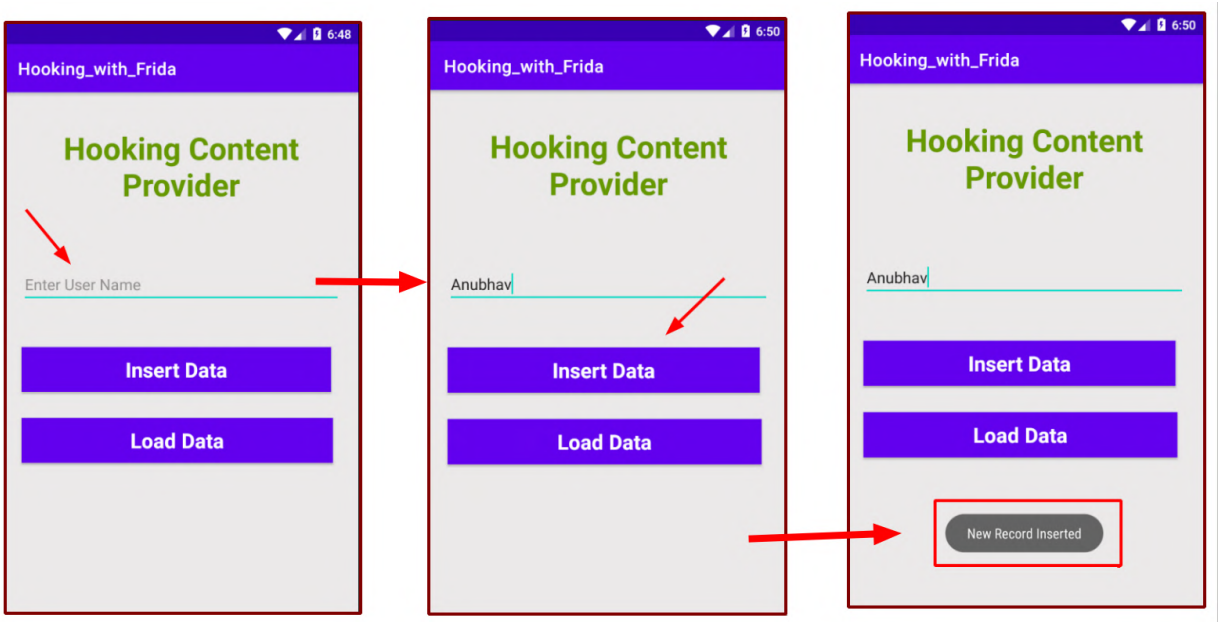
Content Providers play a crucial role in Android by providing a centralised repository for application data, acting as a relational database. They allow secure access to data from other applications based on user requirements. Android provides several options to store application data, including images, audio, videos, and personal contact information in SQLite Database, files, or network storage. Content providers have permissions that grant or restrict access to the data, facilitating data sharing between applications. You can read more about this here.

So we are going to monitor the content provider using Frida. Let's understand the application and hook the content provider.

A. We have created a content provider activity which we will open with this button.



B. You can insert the data using this activity.



To insert the data use the below code

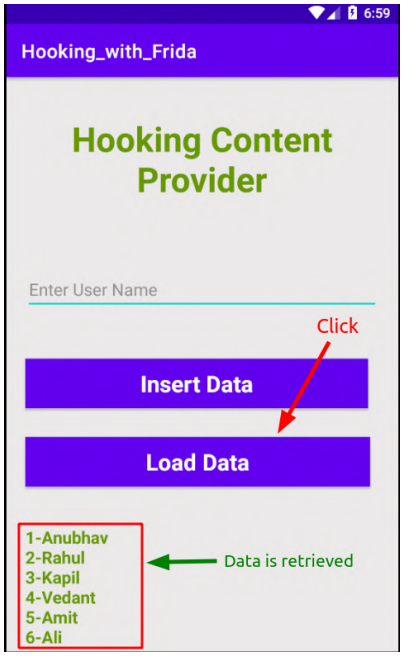
a. This `onClickAddDetails` method gets called when we click on `Insert Data`

```
public void onClickAddDetails(View view) {

    // class to add values in the database
    ContentValues values = new ContentValues();

    // fetching text from user
    values.put(MyContentProvider.name, ((EditText) findViewById(R.id.textName)).getText().toString());

    // inserting into database through content URI
    getContentResolver().insert(MyContentProvider.CONTENT_URI, values);

    // displaying a toast message
    Toast.makeText(getBaseContext(), text: "New Record Inserted", Toast.LENGTH_LONG).show();
}
```

b. This is the code of insert query.

```
// adding data to the database
@Override
public Uri insert(Uri uri, ContentValues values) {
    long rowID = db.insert(TABLE_NAME, nullColumnHack: "", values);
    if (rowID > 0) {
        Uri _uri = ContentUris.withAppendedId(CONTENT_URI, rowID);
        getContext().getContentResolver().notifyChange(_uri, observer: null);
        return _uri;
    }
    throw new SQLiteException("Failed to add a record into " + uri);
}
```

c. You can also retrieve the inserted data using the `Load Data` button.



The code behind this is

d. The `onClickShowDetails` method gets called when we click on `Load Data`

```
@SuppressLint("Range")
public void onClickShowDetails(View view) {
    // inserting complete table details in this text field
    TextView resultView= (TextView) findViewById(R.id.res);

    // creating a cursor object of the
    // content URI
    Cursor cursor = getContentResolver().query(Uri.parse(String.valueOf(MyContentProvider.CONTENT_URI)), projection: null, selection: null, selectionArgs: null, sortOrder: null);

    // iteration of the cursor
    // to print whole table
    if(cursor.moveToFirst()) {
        StringBuilder strBuild=new StringBuilder();
        while (!cursor.isAfterLast()) {
            strBuild.append("\n"+cursor.getString(cursor.getColumnIndex( columnName: "id"))+ "-"+ cursor.getString(cursor.getColumnIndex( columnName: "name")));
            cursor.moveToNext();
        }
        resultView.setText(strBuild);
    }
    else {
        resultView.setText("No Records Found");
    }
}
}
```

e. This is a method for implementing the `query()` method in a Content Provider in Android. It is responsible for handling queries from clients of the Content Provider and returning a cursor containing the results.

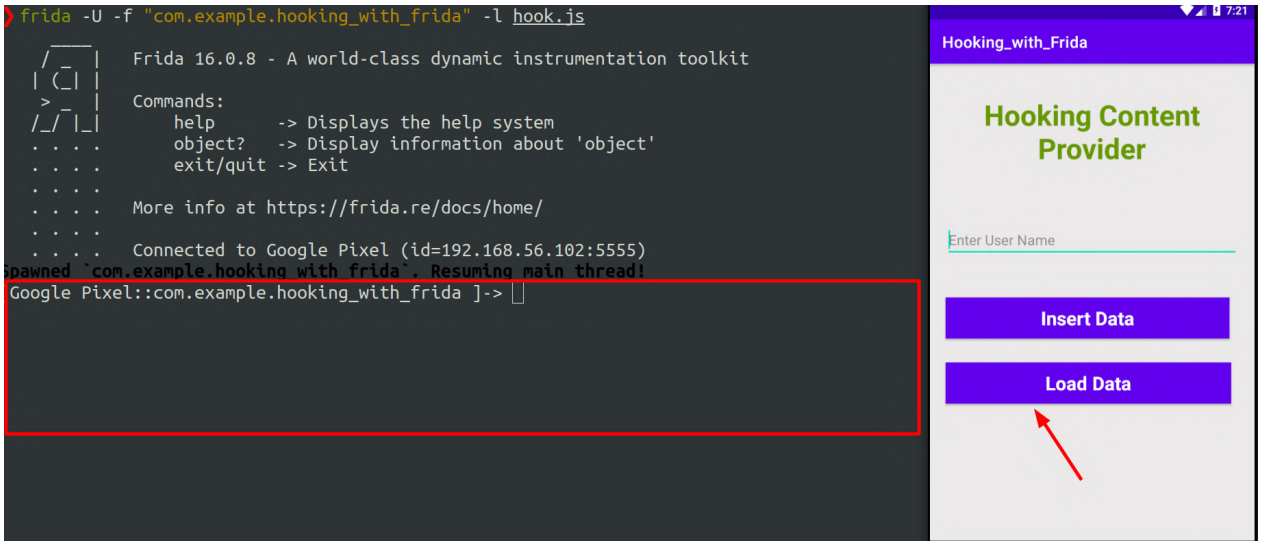
```
@Override
public Cursor query(Uri uri, String[] projection, String selection,
                    String[] selectionArgs, String sortOrder) {
    SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
    qb.setTables(TABLE_NAME);
    switch (uriMatcher.match(uri)) {
        case uriCode:
            qb.setProjectionMap(values);
            break;
        default:
            throw new IllegalArgumentException("Unknown URI " + uri);
    }
    if (sortOrder == null || sortOrder == "") {
        sortOrder = id;
    }
    Cursor c = qb.query(db, projection, selection, selectionArgs, groupBy: null,
            having: null, sortOrder);
    c.setNotificationUri(getContext().getContentResolver(), uri);
    return c;
}
```

f. So let's hook the query method of the content provider to log the data when this method gets called, such as the data we saw for: Anubhav, Kapil, Rahul, etc. Below is the script from which we can hook the query method of the content provider.

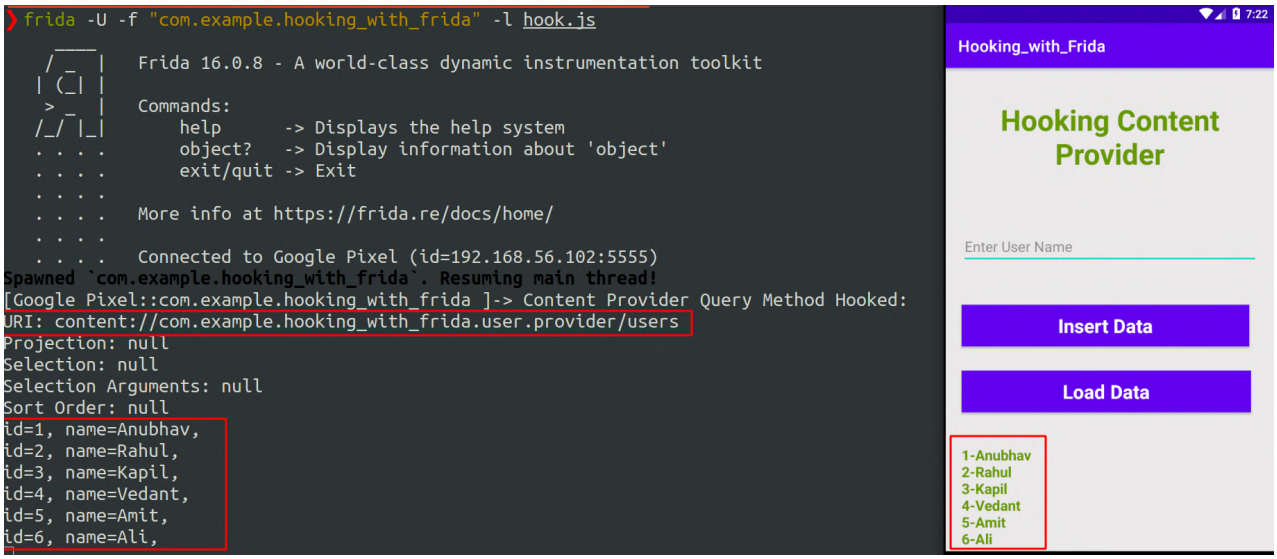
```
        Java.perform(function() {

          // Change this to the class name of the Content Provider you want to
hook
          var contentProviderClass = Java.use("com.example.hooking_with_frida.
MyContentProvider");

          // Hook the query method of the Content Provider
          contentProviderClass.query.overload('android.net.Uri', '[Ljava.lang.
String;', 'java.lang.String', '[Ljava.lang.String;', 'java.lang.String').
implementation = function(uri, projection, selection, selectionArgs, sortOrder) {

            // Print the parameters passed to the query method
            console.log("Content Provider Query Method Hooked:");
            console.log("URI: " + uri);
            console.log("Projection: " + projection);
            console.log("Selection: " + selection);
            console.log("Selection Arguments: " + selectionArgs);
            console.log("Sort Order: " + sortOrder);

          // Call the original implementation of the method
            var cursor = this.query(uri, projection, selection, selectionArgs,
sortOrder);

            // Log the retrieved data
            if (cursor) {
              var columnNames = cursor.getColumnNames();
              while (cursor.moveToNext()) {
                var rowData = "";
                for (var i = 0; i < columnNames.length; i++) {
                  var columnName = columnNames[i];
                  var columnValue = cursor.getString(cursor.
getColumnIndex(columnName));
                  rowData += columnName + "=" + columnValue + ", ";
                }
                console.log(rowData);
              }
            }

            return cursor;

          };
        });
```

Executing the script

(i) Click on the button



(ii) Data logged



We can hook other methods too of content provider such as update, delete, etc to monitor the content or tamper with the data.

4. Monitor Services

Android service is a component that is used to perform operations on the background such as playing music, handling network transactions, interacting with content providers, etc. It doesn't have any UI (user interface).

The service runs in the background indefinitely even if the application is destroyed. Moreover, the service can be bounded by a component to perform interactivity and inter process communication (IPC).

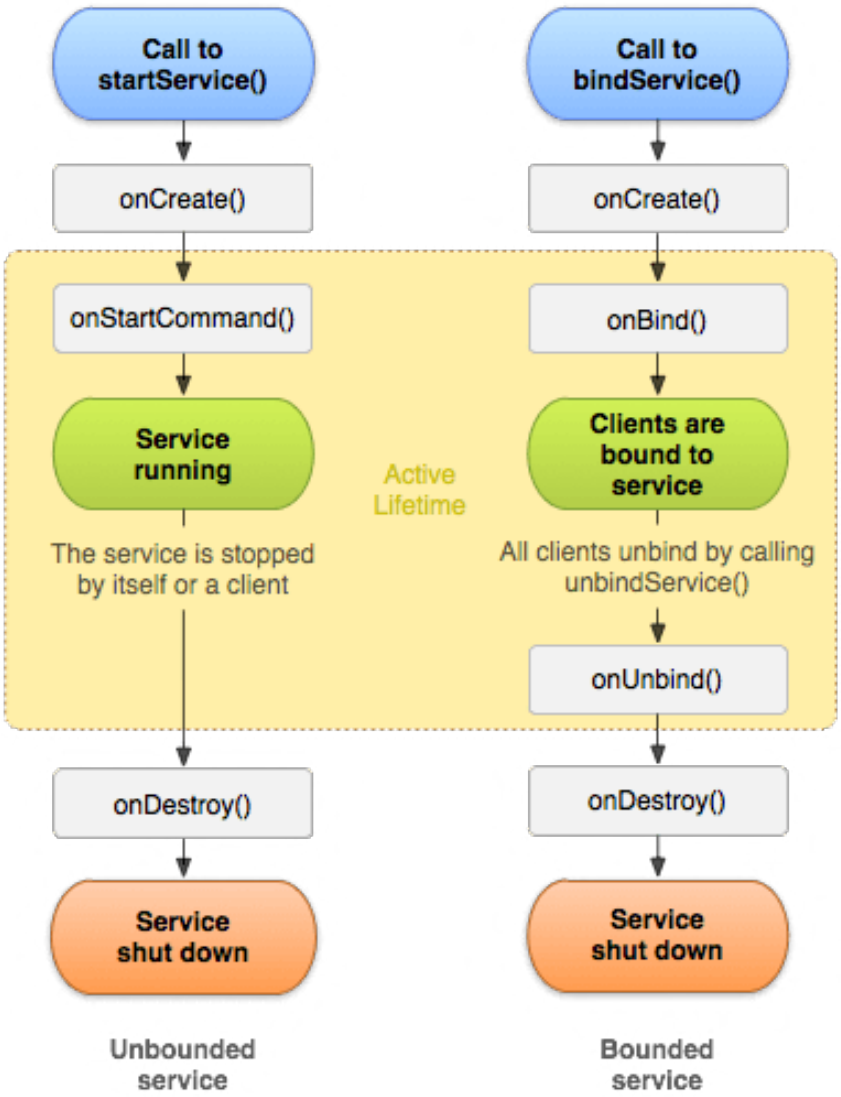
There are two forms of service - started and bound service.

A. Started Service

A service is started when a component (like activity) calls the **startService()** method, and runs in the background indefinitely. It is stopped by the **stopService()** method. The service can stop itself by calling the stopService() method.

B. Bound Service

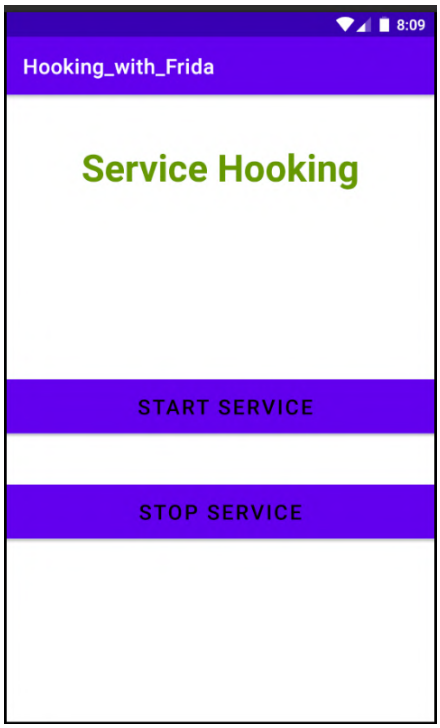
A service is bound when another component (e.g. client) calls the **bindService()** method. The client can unbind the service by calling the **unbindService()**
method. The service cannot be stopped until all clients unbind the service.



You can read more about this here. We are going to look at Started service and hook `startService` and `stopService` methods.

Here's an example:

a. We have two buttons to start and stop the service in this activity.



b. When we click on start service a music starts playing and when we click on stop, the music stops playing.

```
public void onClick(View view) {

    // process to be performed
    // if start button is clicked
    if(view == start){

        // starting the service
        startService(new Intent( packageContext: this, NewService.class ) );
    }

    // process to be performed
    // if stop button is clicked
    else if (view == stop){

        // stopping the service
        stopService(new Intent( packageContext: this, NewService.class ) );

    }
}
```

These methods are present in the `Service_Hooking` class.

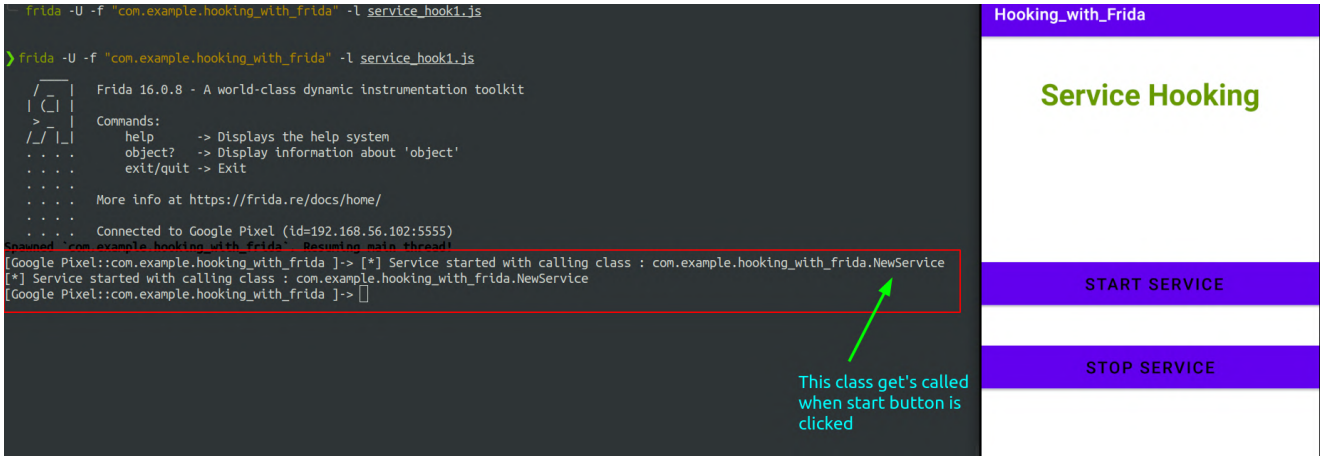
```java
package com.example.hooking_with_frida;

import ...

public class Service_Hooking extends AppCompatActivity {
    // declaring objects of Button class
    private Button start, stop;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_service_hooking);

        // assigning ID of startButton
        // to the object start
        start = (Button) findViewById( R.id.startButton );

        // assigning ID of stopButton
        // to the object stop
        stop = (Button) findViewById( R.id.stopButton );
```

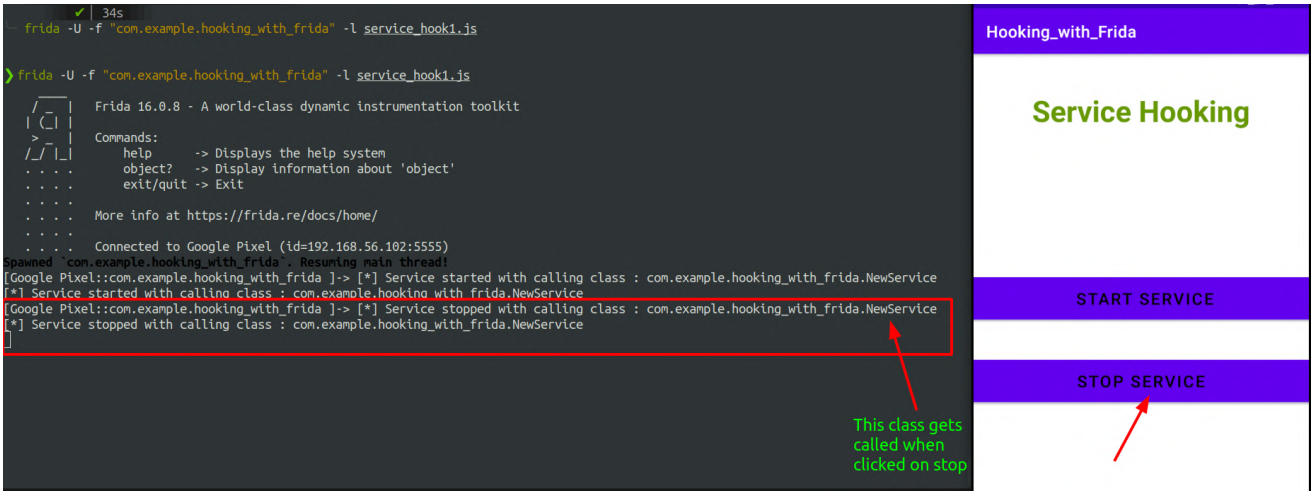
c. Let's write a script for hooking.

```
        Java.perform(function() {

          // Get a reference to the Context class
          var contextClass = Java.use("com.example.hooking_with_frida.Service_
Hooking");

              // Hook the startService method
              contextClass.startService.overload('android.content.Intent').
implementation = function(intent) {

              // Log the service being started
              console.log("[*] Service started with calling class : " + intent.
getComponent().getClassName());

              // Call the original implementation of the method
              return this.startService(intent);
            }
```

```
                // Hook the stopService method
                contextClass.stopService.overload('android.content.Intent').
    implementation = function(intent) {

                // Log the service being stopped
                console.log("[*] Service stopped with calling class : " + intent.
    getComponent().getClassName());

                // Call the original implementation of the method
                return this.stopService(intent);
            }

        });
```

d. Executing the script





This is how you can hook methods of services.

5. Monitor the Broadcast Receiver

In Android, a broadcast receiver is a component that listens for system-wide or application-specific broadcast messages. When a broadcast message is sent, the system automatically sends the message to all registered broadcast receivers for that message, allowing them to respond to the message appropriately.

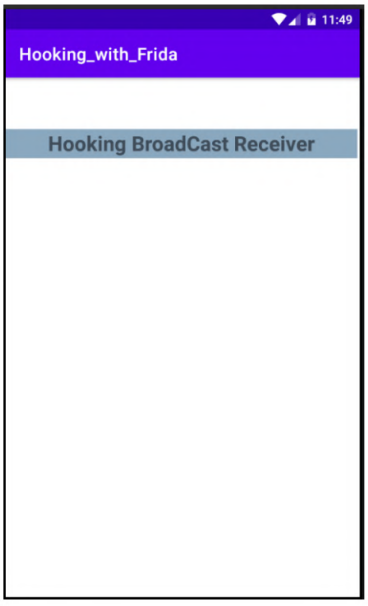
Broadcast receivers are often used to receive system events, such as a change in network connectivity, device power status, or the installation or removal of an application. They can also be used to receive custom messages that are broadcast within an application or between different applications.

To use a broadcast receiver in an Android app, you need to define a class that extends the `BroadcastReceiver` class and override the `onReceive()` method to handle the broadcast message. You then need to register the broadcast receiver with the system by specifying an Intent filter that defines the types of broadcast messages the receiver should listen for.

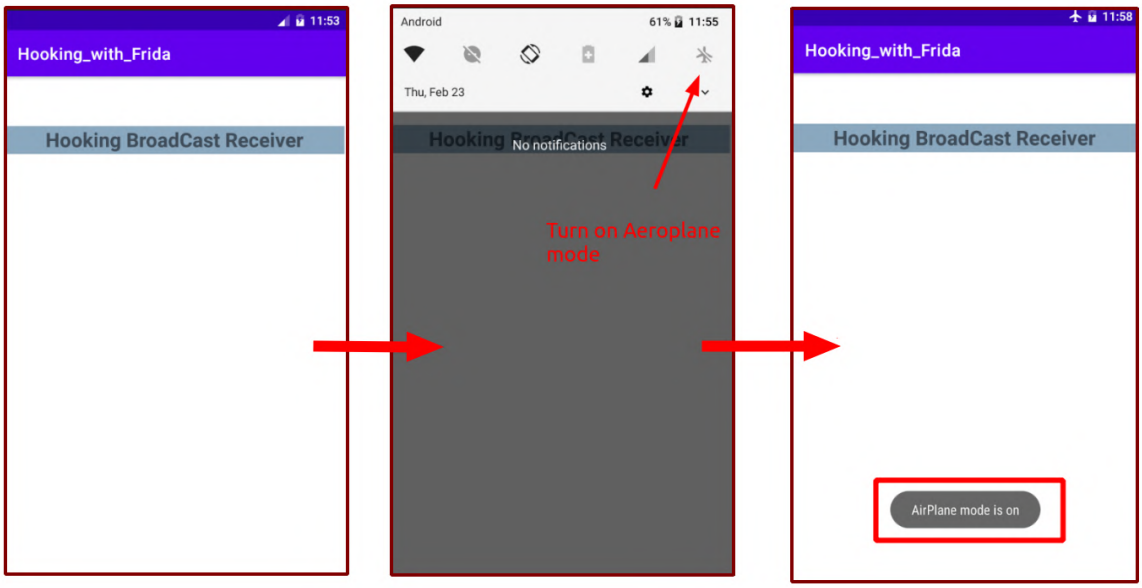
Let's see how the broadcast Receiver works

A. This is our activity, which registers itself for airplane mode changes.

```
public class BroadCast_Receiver_Hook extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_broad_cast_receiver_hook);
    }

    AirplaneModeChangeReceiver airplaneModeChangeReceiver = new AirplaneModeChangeReceiver();

    @Override
    protected void onStart() {
        super.onStart();
        IntentFilter filter = new IntentFilter(Intent.ACTION_AIRPLANE_MODE_CHANGED);
        registerReceiver(airplaneModeChangeReceiver, filter);
    }

    @Override
    protected void onStop() {
        super.onStop();
        unregisterReceiver(airplaneModeChangeReceiver);
    }
}
```



B. Now when the airplane mode is turned on you will see a toast message which we have defined in the AirplaneModeChangeReceiver class.



C. Monitor this BroadCast using Frida then use a code snippet provided on codeshare.

```
        function registerReceiver(intentFilter) {
            Java.perform(function() {
                var cxt = getContext();
                if (cxt) {
                    const parent = BroadcastReceiver().$new();
                    cxt.registerReceiver(ChildBroadcastReceiver().$new(parent),
Java.use('android.content.IntentFilter').$new(intentFilter));
                }
            });
        }

        function registerReceiverForAll() {
            Java.perform(function() {
                var Modifier = Java.use('java.lang.reflect.Modifier');
                var String = Java.use('java.lang.String');
                const parent = BroadcastReceiver().$new();
                var intent = Java.use('android.content.Intent').$new();
                var fields = intent.getClass().getDeclaredFields();
                for (var i = 0; i < fields.length; i++) {
                    var field = fields[i];
                    var modifiers = field.getModifiers();
                    if (Modifier.isPublic(modifiers) && Modifier.isStatic(modifiers)
&& Modifier.isFinal(modifiers) && field.getType().equals(String.class)) {
                        var filter = field.get(intent);
                        if (filter) {
                            console.log('Global BroadcastReceiver -> Registered:
' + filter);
                            getContext().
registerReceiver(ChildBroadcastReceiver().$new(parent), Java.use('android.
content.IntentFilter').$new(filter));
                        }
                    }
                }
            });
        }
```

```
        function hookReceiver(receiverClassID) {
            if (typeof receiverClassID == 'undefined') {
                receiverClassID = 'android.content.BroadcastReceiver';
            }
            Java.perform(function() {
                var ReceiverClass = Java.use(receiverClassID);
                if (ReceiverClass) {
                    ReceiverClass.onReceive.implementation = function(context,
intent) {
                        printIntent(intent);
                        return this.onReceive(context, intent);
                    };
                    console.log("[+] BroadcastReceiver (" + receiverClassID + ") hook
loaded.");
                }
            });
        }

        function hookSender() {
            Java.perform(function() {
                var Context = Java.use('android.content.Context');
                Context.sendBroadcast.overload('android.content.Intent').
implementation = function(intent) {
                    printIntent(intent);
                    return this.sendBroadcast(intent);
                };
                console.log("[+] Context.sendBroadcast() hook loaded.");
            });
        }

        function getContext() {
            return Java.use('android.app.ActivityThread').currentApplication().
getApplicationContext();
        }

        function BroadcastReceiver() {
            const ParentBroadcastReceiver = Java.registerClass({
                name: 'ParentBroadcastReceiver',
                superClass: Java.use('android.content.BroadcastReceiver'),
                fields: {
                    parent: 'android.content.BroadcastReceiver'
                },
                methods: {
                    onReceive: [{
                        returnType: 'void',
                        argumentTypes: ['android.content.Context', 'android.content.
Intent'],
                        implementation: function(context, intent) {
                            printIntent(intent);
                        }
                    }]
                },
            });
            return ParentBroadcastReceiver;
        }
```

```
        function ChildBroadcastReceiver() {
            const ChildBroadcastReceiver = Java.registerClass({
                name: 'ChildBroadcastReceiver',
                superClass: Java.use('android.content.BroadcastReceiver'),
                fields: {
                    parent: 'android.content.BroadcastReceiver'
                },
                methods: {
                    '<init>': [{
                        returnType: 'void',
                        argumentTypes: [],
                        implementation: function() {}
                    }, {
                        returnType: 'void',
                        argumentTypes: ['android.content.BroadcastReceiver'],
                        implementation: function(parent) {
                            this.parent.value = parent;
                        }
                    }],
                    onReceive: [{
                        returnType: 'void',
                        argumentTypes: ['android.content.Context', 'android.
content.Intent'],
                        implementation: function(context, intent) {
                            this.parent.value.onReceive(context, intent);
                        }
                    }]
                },
            });
            return ChildBroadcastReceiver;
        }

        function printIntent(intent) {
            console.log("\n[*] Intent received. Action: " + intent.getAction());
            var keys = intent.getExtras().keySet();
            var extras = "";
            var iterator = keys.iterator();
            for (var i = 0; i < keys.size(); i++) {
                if (iterator.hasNext()) {
                    var key = iterator.next();
                    extras += "\n" + key + " = " + intent.getExtras().get(key);
                }
            }
            console.log("[*] Extras:" + extras);
        }
```

D. You can use the above script in 4 different ways.

Android Broadcast Receiver

a. Registering a broadcast receiver with a specified Intent filter:

```
registerReceiver(IntentFilter);
registerReceiver('com.example.customAction');
```

b. Registering a "catch all" broadcast receiver:

Android SDK does not allow the registration of a receiver without specifying an Intent filter and does not support wildcard filters. This restriction can be circumvented using Java's reflection features to enumerate all applicable events. Then a custom receiver can be registered for each event type. The limitation of this method is the inability to catch non-standard actions/categories/extras. In such cases, use "registerReceiver(IntentFilter);".

Otherwise you can use "registerReceiverForAll();" so that we register receiver for all.

c. Hook an existing broadcast receiver:

```
hookReceiver();
hookReceiver(BroadcastReceiverClass);
```

d. Hook an existing broadcast sender (via the Context.sendBroadcast() method):

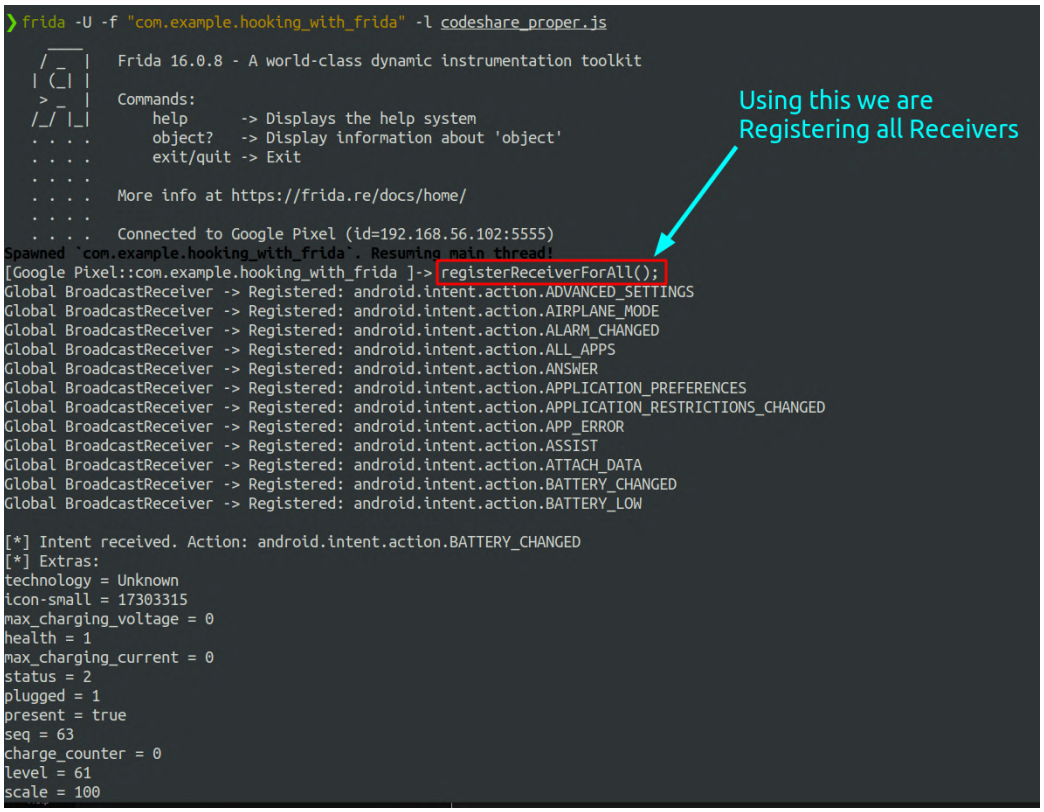
Note that Intents sent with LocalBroadcastManager will not be hooked.

```
hookSender();
```

E. I will show you two methods

a. Registering All Receiver

(i) Executing the script with `registerReceiverForAll()`




```
❯ frida -U -f "com.example.hooking_with_frida" -l codeshare_proper.js
     ____
    / _  |   Frida 16.0.8 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Google Pixel (id=192.168.56.102:5555)
Spawned `com.example.hooking_with_frida`. Resuming main thread!
[Google Pixel::com.example.hooking_with_frida ]-> registerReceiverForAll();
Global BroadcastReceiver -> Registered: android.intent.action.ADVANCED_SETTINGS
Global BroadcastReceiver -> Registered: android.intent.action.AIRPLANE_MODE
Global BroadcastReceiver -> Registered: android.intent.action.ALARM_CHANGED
Global BroadcastReceiver -> Registered: android.intent.action.ALL_APPS
Global BroadcastReceiver -> Registered: android.intent.action.ANSWER
Global BroadcastReceiver -> Registered: android.intent.action.APPLICATION_PREFERENCES
Global BroadcastReceiver -> Registered: android.intent.action.APPLICATION_RESTRICTIONS_CHANGED
Global BroadcastReceiver -> Registered: android.intent.action.APP_ERROR
Global BroadcastReceiver -> Registered: android.intent.action.ASSIST
Global BroadcastReceiver -> Registered: android.intent.action.ATTACH_DATA
Global BroadcastReceiver -> Registered: android.intent.action.BATTERY_CHANGED
Global BroadcastReceiver -> Registered: android.intent.action.BATTERY_LOW

[*] Intent received. Action: android.intent.action.BATTERY_CHANGED
[*] Extras:
technology = Unknown
icon-small = 17303315
max_charging_voltage = 0
health = 1
max_charging_current = 0
status = 2
plugged = 1
present = true
seq = 63
charge_counter = 0
level = 61
scale = 100
```

(ii) All receivers are registered and now Frida is ready to monitor them

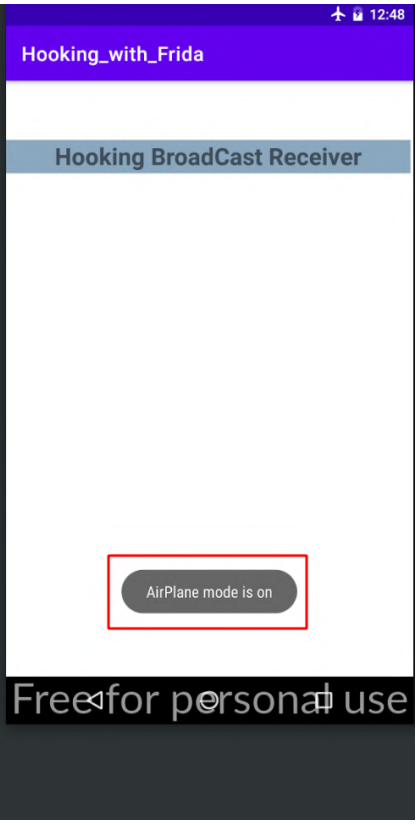
```
Global BroadcastReceiver -> Registered: android.intent.extra.TITLE
Global BroadcastReceiver -> Registered: android.intent.extra.UID
Global BroadcastReceiver -> Registered: android.intent.extra.UNINSTALL_ALL_USERS
Global BroadcastReceiver -> Registered: android.intent.extra.USER
Global BroadcastReceiver -> Registered: android.intent.extra.user_handle
Global BroadcastReceiver -> Registered: android.intent.extra.USER_ID
Global BroadcastReceiver -> Registered: android.intent.extra.USER_REQUESTED_SHUTDOWN
Global BroadcastReceiver -> Registered: android.intent.extra.VERIFICATION_BUNDLE
Global BroadcastReceiver -> Registered: android.intent.extra.VERSION_CODE
Global BroadcastReceiver -> Registered: radioTechnology
Global BroadcastReceiver -> Registered: voiceRegState
Global BroadcastReceiver -> Registered: voiceRoamingType
Global BroadcastReceiver -> Registered: android.intent.extra.WIPE_EXTERNAL_STORAGE
Global BroadcastReceiver -> Registered: android.dock_home
Global BroadcastReceiver -> Registered: android.SETUP_VERSION
[Google Pixel::com.example.hooking_with_frida ]->
[*] Intent received. Action: android.intent.action.TIME_TICK
[*] Extras:
android.intent.extra.ALARM_COUNT = 1

[*] Intent received. Action: android.intent.action.TIME_TICK
[*] Extras:
android.intent.extra.ALARM_COUNT = 1

[*] Intent received. Action: android.intent.action.TIME_TICK
[*] Extras:
android.intent.extra.ALARM_COUNT = 1

[*] Intent received. Action: android.intent.action.TIME_TICK
[*] Extras:
android.intent.extra.ALARM_COUNT = 1

[*] Intent received. Action: android.intent.action.TIME_TICK
[*] Extras:
android.intent.extra.ALARM_COUNT = 1
[Google Pixel::com.example.hooking_with_frida ]->
```

(iii) We will create a broadCast such as turning on the airplane mode to see what information gets logged.

```
Global BroadcastReceiver -> Registered: android.intent.extra.shortcut.INTENT
Global BroadcastReceiver -> Registered: android.intent.extra.shortcut.NAME
Global BroadcastReceiver -> Registered: android.intent.extra.SHUTDOWN_USERSPACE_ONLY
Global BroadcastReceiver -> Registered: android.intent.extra.SIM_ACTIVATION_RESPONSE
Global BroadcastReceiver -> Registered: android.intent.extra.SPLIT_NAME
Global BroadcastReceiver -> Registered: android.intent.extra.STREAM
Global BroadcastReceiver -> Registered: android.intent.extra.SUBJECT
Global BroadcastReceiver -> Registered: systemId
Global BroadcastReceiver -> Registered: android.intent.extra.TASK_ID
Global BroadcastReceiver -> Registered: android.intent.extra.TEMPLATE
Global BroadcastReceiver -> Registered: android.intent.extra.TEXT
Global BroadcastReceiver -> Registered: android.intent.extra.THERMAL_STATE
Global BroadcastReceiver -> Registered: android.intent.extra.TIME_PREF_24_HOUR_FORMAT
Global BroadcastReceiver -> Registered: android.intent.extra.TITLE
Global BroadcastReceiver -> Registered: android.intent.extra.UID
Global BroadcastReceiver -> Registered: android.intent.extra.UNINSTALL_ALL_USERS
Global BroadcastReceiver -> Registered: android.intent.extra.USER
Global BroadcastReceiver -> Registered: android.intent.extra.user_handle
Global BroadcastReceiver -> Registered: android.intent.extra.USER_ID
Global BroadcastReceiver -> Registered: android.intent.extra.USER_REQUESTED_SHUTDOWN
Global BroadcastReceiver -> Registered: android.intent.extra.VERIFICATION_BUNDLE
Global BroadcastReceiver -> Registered: android.intent.extra.VERSION_CODE
Global BroadcastReceiver -> Registered: radioTechnology
Global BroadcastReceiver -> Registered: voiceRegState
Global BroadcastReceiver -> Registered: voiceRoamingType
Global BroadcastReceiver -> Registered: android.intent.extra.WIPE_EXTERNAL_STORAGE
Global BroadcastReceiver -> Registered: android.dock_home
Global BroadcastReceiver -> Registered: android.SETUP_VERSION
[Google Pixel::com.example.hooking_with_frida ]->
[*] Intent received. Action: android.intent.action.AIRPLANE_MODE
[*] Extras:
state = true

[*] Intent received. Action: android.intent.action.TIME_TICK
[*] Extras:
android.intent.extra.ALARM_COUNT = 1
```

This broadCast Intent is received

State is true which means airplane mode is ON

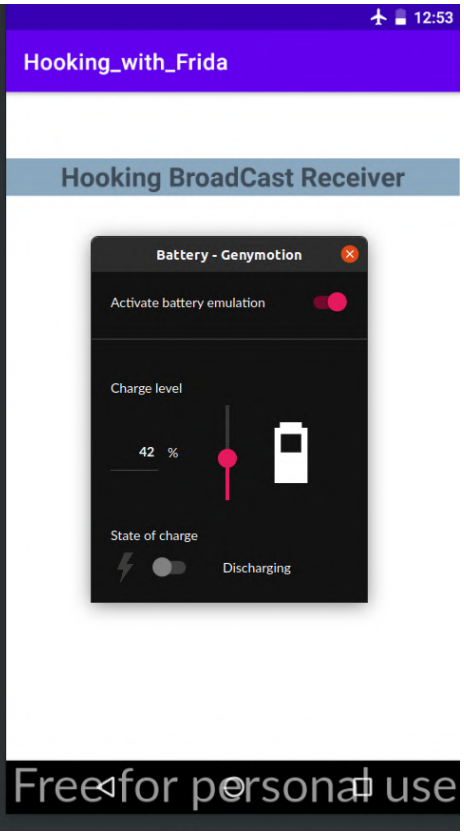


(iv) You can also monitor other Broadcasts such as Battery Low or battery status.

```
status = 3
plugged = 0
present = true
seq = 67
charge_counter = 0
level = 11
scale = 100
temperature = 0
voltage = 10000
invalid_charger = 0

[*] Intent received. Action: android.intent.action.BATTERY_OKAY
[*] Extras:
seq = 68

[*] Intent received. Action: android.intent.action.BATTERY_CHANGED
[*] Extras:
technology = Unknown
icon-small = 17303301
max_charging_voltage = 0
health = 1
max_charging_current = 0
status = 3
plugged = 0
present = true
seq = 68
charge_counter = 0
level = 42
scale = 100
temperature = 0
voltage = 10000
invalid_charger = 0
```

This shows like charger is not pluged

The level of Battery percentage



This is how you monitor all the receivers.

b. Registering a receiver by providing the Intent Filter

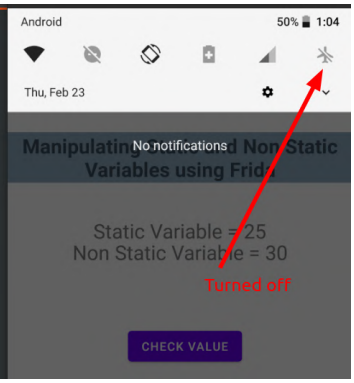
As seen previously, we used the `registerReceiverForAll()` which hooked all the receivers, but suppose you just want to monitor the broadcast receiver of the Airplane Mode. For that, you will pass the Intent for AIRPLANE_MODE and monitor it.

(i) Let's hook the receiver using the previous script but this time we will use the `registerReceiver()` by passing IntentFilter to it.

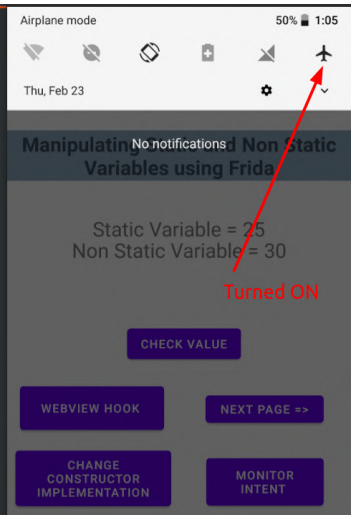
```
❯ frida -U -f "com.example.hooking_with_frida" -l codeshare_proper.js
     ____
    / _  |   Frida 16.0.8 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Google Pixel (id=192.168.56.102:5555)
Spawned `com.example.hooking_with_frida`. Resuming main thread!
[Google Pixel::com.example.hooking_with_frida ]-> registerReceiver('android.intent.action.AIRPLANE_MODE');
[Google Pixel::com.example.hooking_with_frida ]->
```

(ii) We will create the broadcast of the airplane mode and see what information get's logged.

```
❯ frida -U -f "com.example.hooking_with_frida" -l codeshare_proper.js
     ____
    / _  |   Frida 16.0.8 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Google Pixel (id=192.168.56.102:5555)
Spawned `com.example.hooking_with_frida`. Resuming main thread!
[Google Pixel::com.example.hooking_with_frida ]-> registerReceiver('android.intent.action.AIRPLANE_MODE');
[Google Pixel::com.example.hooking_with_frida ]->
[*] Intent received. Action: android.intent.action.AIRPLANE_MODE
[*] Extras:
state = false
```




```
❯ frida -U -f "com.example.hooking_with_frida" -l codeshare_proper.js
     ____
    / _  |   Frida 16.0.8 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Google Pixel (id=192.168.56.102:5555)
Spawned `com.example.hooking_with_frida`. Resuming main thread!
[Google Pixel::com.example.hooking_with_frida ]-> registerReceiver('android.intent.action.AIRPLANE_MODE');
[Google Pixel::com.example.hooking_with_frida ]->
[*] Intent received. Action: android.intent.action.AIRPLANE_MODE
[*] Extras:
state = false

[*] Intent received. Action: android.intent.action.AIRPLANE_MODE
[*] Extras:
state = true
```



This is how you can register a receiver for a particular broadcast. You can use this technique to hook other broadcast receivers by just changing the IntentFilter.

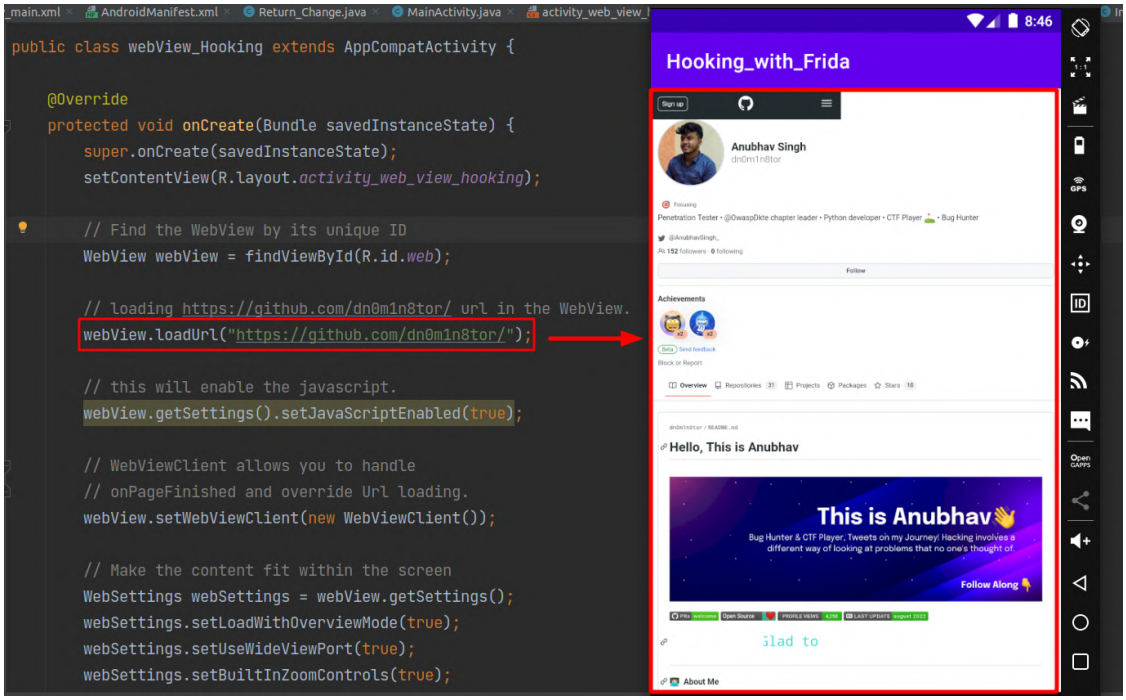
6. Monitor Webview

You want to monitor your application to check whether there are any WebView calls, so as to monitor it with Frida by writing scripts.

For example:

We are going to monitor an application for a webView call by writing a Frida Script which will give us a URL which are being loaded whenever a webview gets loaded. As you know to loadUrl in WebView we use **`webView.loadUrl(URL);`** method so from here we just want the URL value to check the URLs getting loaded.

A. We have defined the URL as https://github.com/dn0m1n8tor and loaded it in WebView.

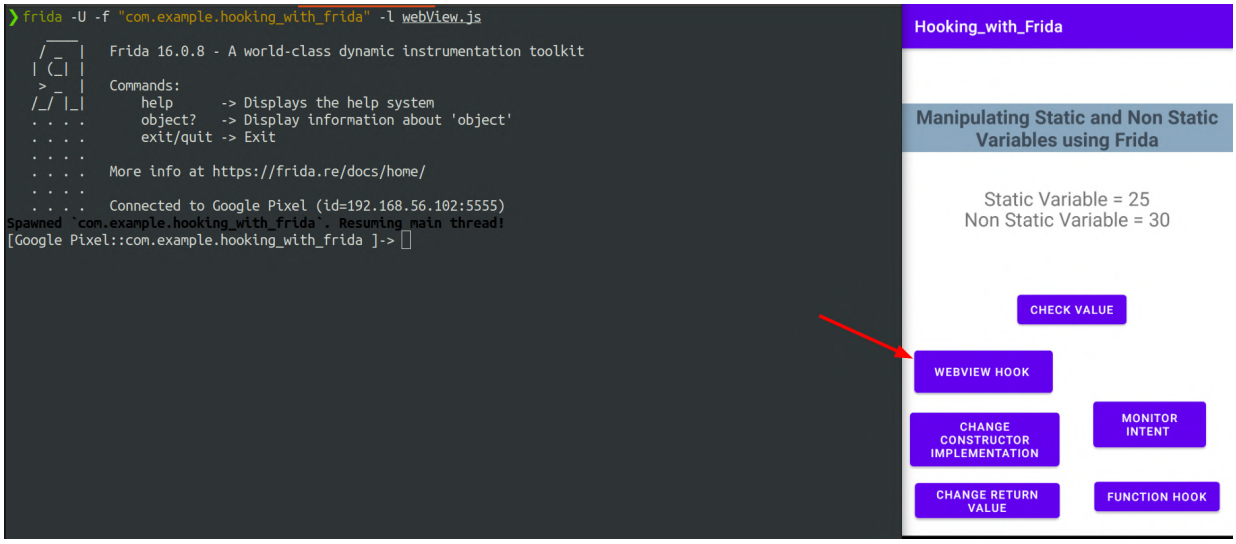


B. We will hook the loadUrl() method using our Frida Script.

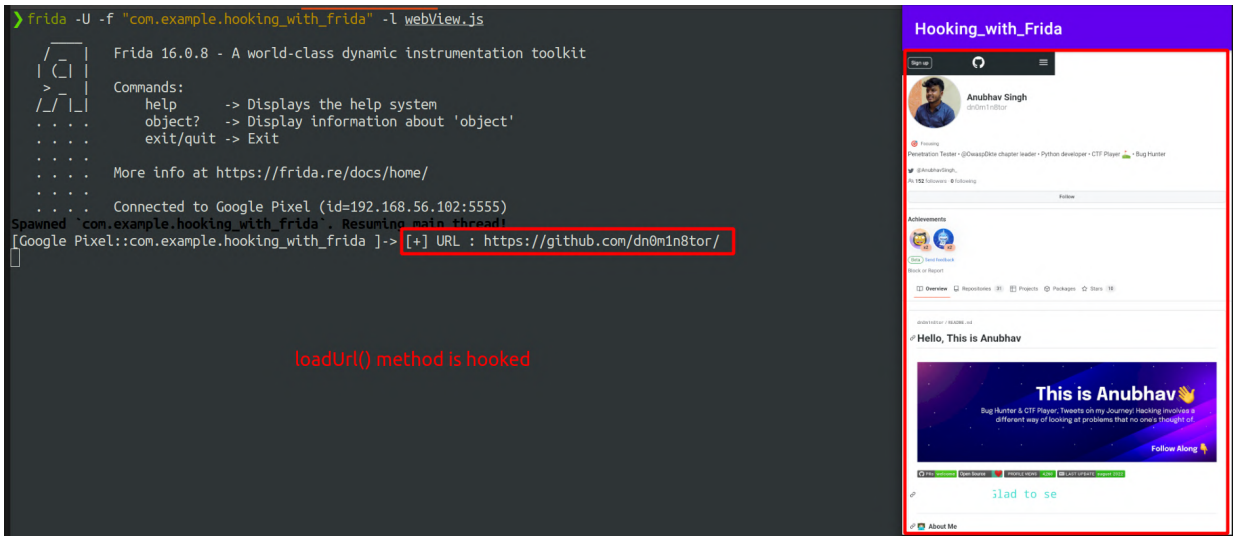
```
Java.perform(function() {

    //Reference of WebView class
    var target_class = Java.use("android.webkit.WebView");
```

```
            //Hooking into loadURL method as this method takes only 1 argumenet
    which is String so we have used overload here
            target_class.loadUrl.overload("java.lang.String").implementation =
    function(a) {

                // Printing URL in console
                console.log("[+] URL : " + a.toString());

                // Calling original method and returning the value
                return this.loadUrl(a);
            };

        })
```

C. Run the Frida script against the application and click on the WebView Hook button



D. The LoadUrl method is hooked.



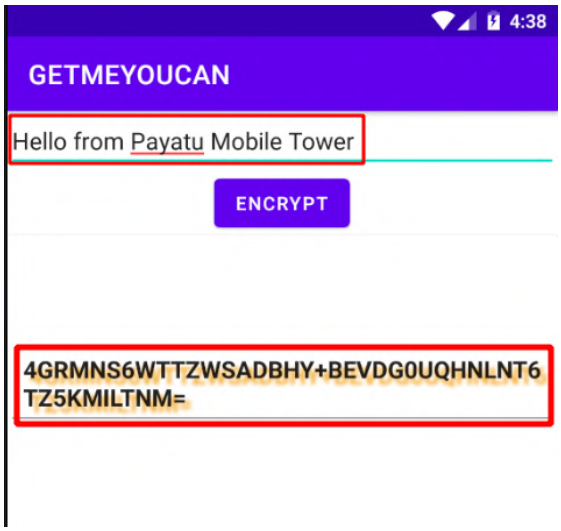
This is how we can hook the webView using Frida.

7. Monitor the Crypto Classes

You want to monitor the Crypto classes of the application for things like what's the encryption algorithm, secret key, etc. You can monitor it all with Frida.

For example, we need to retrieve a secret key -

A. This is an application that takes input from the user and encrypts as well as displays the output.



B. Let's see the code of the application

```
28
29          @Override // android.view.View.OnClickListener
30          public void onClick(View view) {
31              byte[] bArr;
32              String valueOf = String.valueOf(this.f2015b.getText());
33              try {
34                  SecretKeySpec secretKeySpec = new SecretKeySpec(x0.a.a(new byte[]{74, 101, -23, 39, 22, -29, -71, 109, -94, -
    119, -19, -100, 66, -35, -70, 60, 10, -25, 119, -70, -91, -6, 115, -97, 126, 26, 45, 47, -29, -104, -47, -92}).getBytes(),
    "AES");
35                  Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
36                  cipher.init(1, secretKeySpec);
37                  bArr = cipher.doFinal(valueOf.getBytes());
38              } catch (Exception e2) {
39                  System.out.println(e2.toString());
40                  bArr = null;
41              }
42              System.out.println(new String(b2.a.c(bArr)));
43              String str = new String(b2.a.c(bArr));
44              Log.d(x0.a.a(new byte[]{19, 125, -100, -5, -72, 19, -42, -2, -55, -114, -39, 98, -39, 125, -123, -110}), str);
45              this.f2016c.setText(str);
46          }
47      }
48
49      @Override // androidx.fragment.app.q, androidx.activity.ComponentActivity, x.e, android.app.Activity
50      public void onCreate(Bundle bundle) {
51          super.onCreate(bundle);
52          setContentView(R.layout.activity_main);
53          ((Button) findViewById(R.id.click)).setOnClickListener(new a(this, (EditText) findViewById(R.id.text), (EditText)
    findViewById(R.id.textView)));
54      }
55  }
```

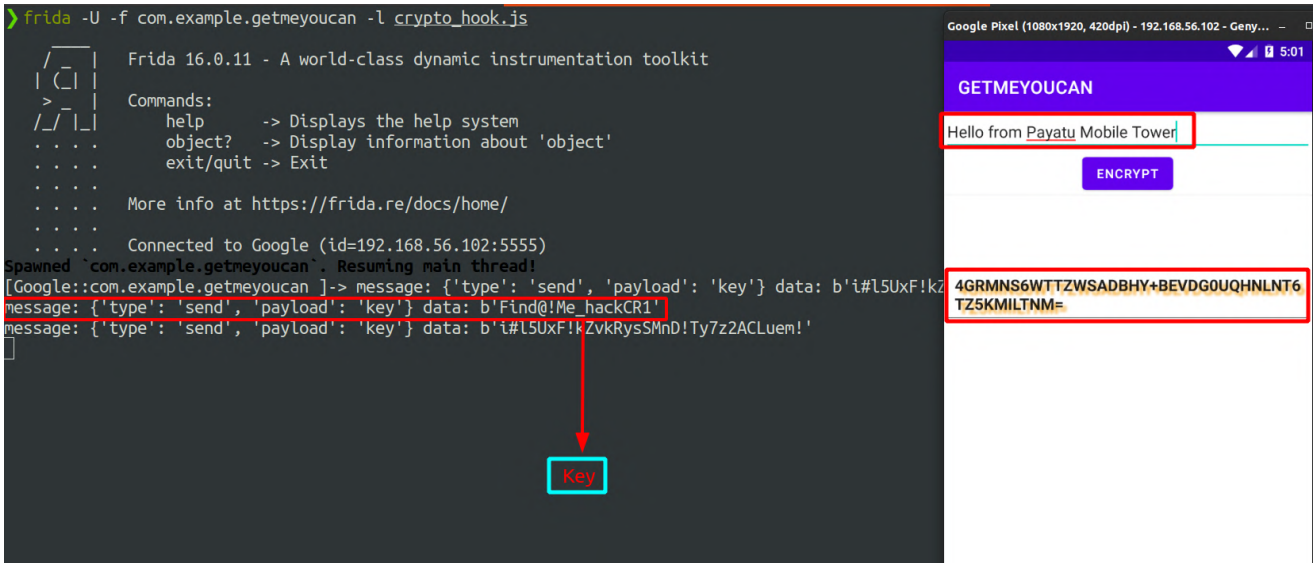
After analyzing the code we can see that an object is made of secretkeyspec and two parameters are passed where one is a key and the second is an algorithm.

Let's hook the constructor of secretkeyspec to get the value of the key.

C. Write a script to hook.

```
        Java.perform(function () {

        //Got the reference of secretKeySpec class
        var secretKeySpec = Java.use('javax.crypto.spec.SecretKeySpec');

        // Hooked into constructor
        secretKeySpec.$init.overload('[B', 'java.lang.String').implementation =
function (a, b) {
            send('key', new Uint8Array(a))
        return this.$init(a,b);
        }

        })
```

D. Execute the script



We are able to log the key from the application.

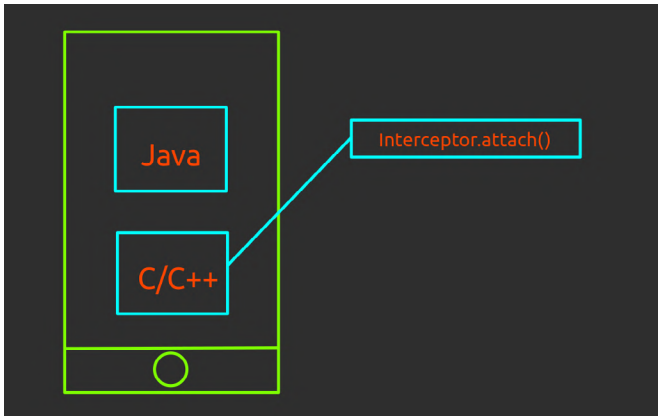
We can also perform various other things using Frida in the crypto world.

Chapter 7

Hooking the NDK Function

We have explored how we can hook into Java code using Frida with methods like `Java.perform()` and `Java.choose()`. However, developers sometimes utilize the Android NDK (Native Development Kit) to implement certain logic or code in C or C++ languages and access those functionalities within the APK. These implementations can often be found in .so files. Since it's not feasible to decompile and retrieve the source code of a compiled C/C++ native library, we can utilize Frida APIs to explore and interact with these libraries.

Using the interceptor.attach() API we can hook the native code. This API is similar to the Java.choose() API.



Let's examine how the C code is incorporated into Java and how communication occurs between Java and C.

```
package com.example.fridanative;

import ...

public class MainActivity extends AppCompatActivity implements View.OnClickListener {
    EditText firstNumberEditText, secondNumberEditText;
    TextView resultTextView;
    Button addButton, subtractButton;

    static {
        System.loadLibrary( libname: "fridanative");
    }

    public native int add(int x, int y);
    public native int sub(int x, int y);

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Initialize the views
        firstNumberEditText = findViewById(R.id.firstNumberEditText);
        secondNumberEditText = findViewById(R.id.secondNumberEditText);
        resultTextView = findViewById(R.id.resultTextView);
        addButton = findViewById(R.id.addButton);
        subtractButton = findViewById(R.id.subtractButton);
```

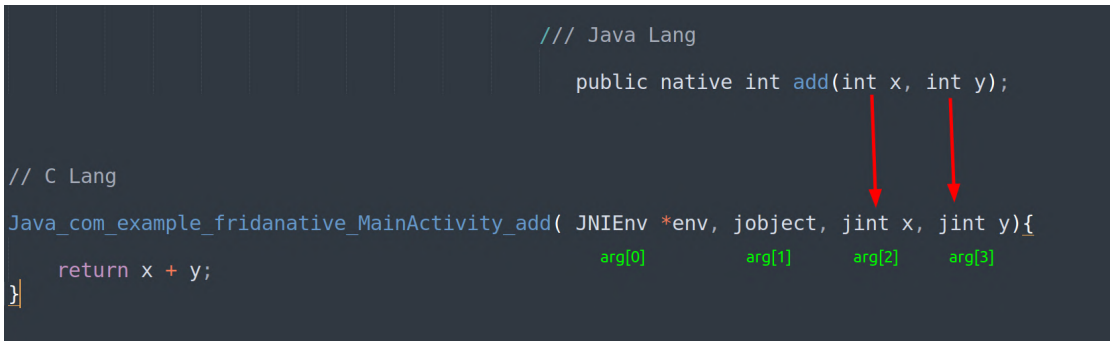
We have loaded here shared library which will be present when our application we get compiled.

The above loaded library contains these add and sub methods but to them in Java we need to specify "native" using the method

So when you want to see which methods are declared in the Native part, you can simply search for "native" in Jadx. If you find any method, then the implementation of that method will be defined in a .so file.

Let's see how the code is implemented in C for add and sub method.

Let's understand what are the 4 parameter passing in the function.



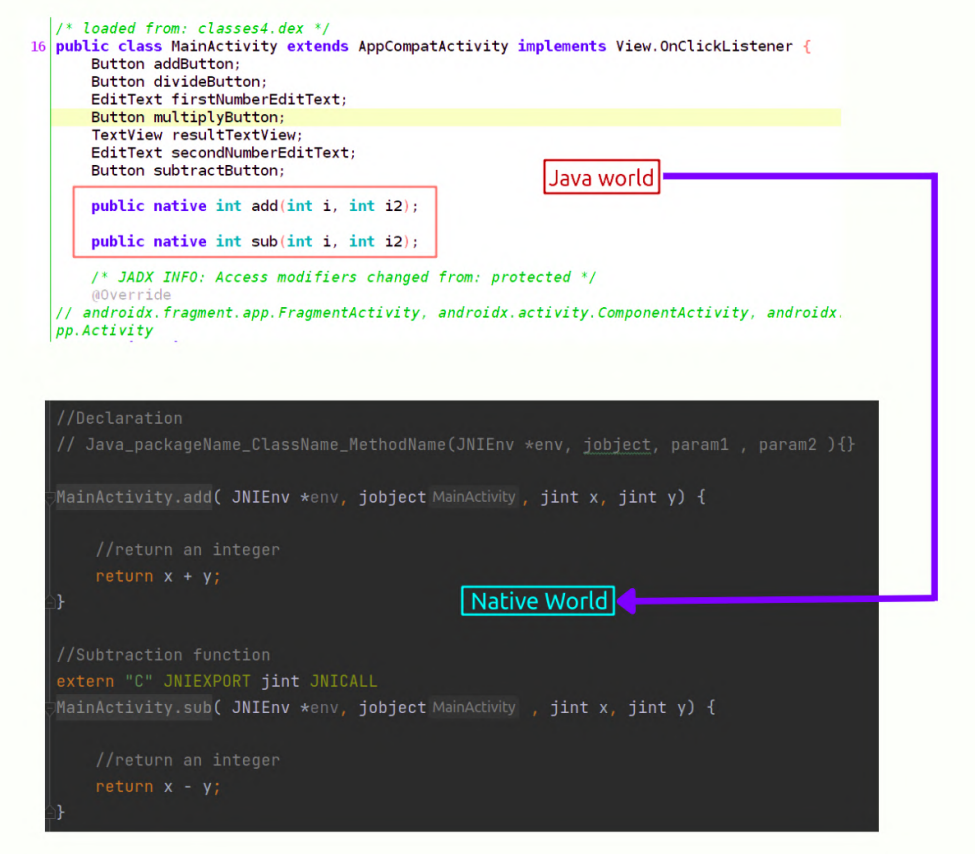
1. `JNIEnv *env:` A pointer to the JNI (Java Native Interface) environment. This parameter provides access to the JNI functions that enable communication between Java and native code.
2. `jobject:` An object that represents the class that contains the native method being called. This parameter is typically used to call other methods or access fields of the class.
3. `jint x:` An integer value that represents the first argument of the native method. In this case, it represents the value of the x parameter passed to the add method.
4. `jint y:` An integer value that represents the second argument of the native method. In this case, it represents the value of the y parameter passed to the add method.

There are total 3 ways in which we are going to look for hooking native functions

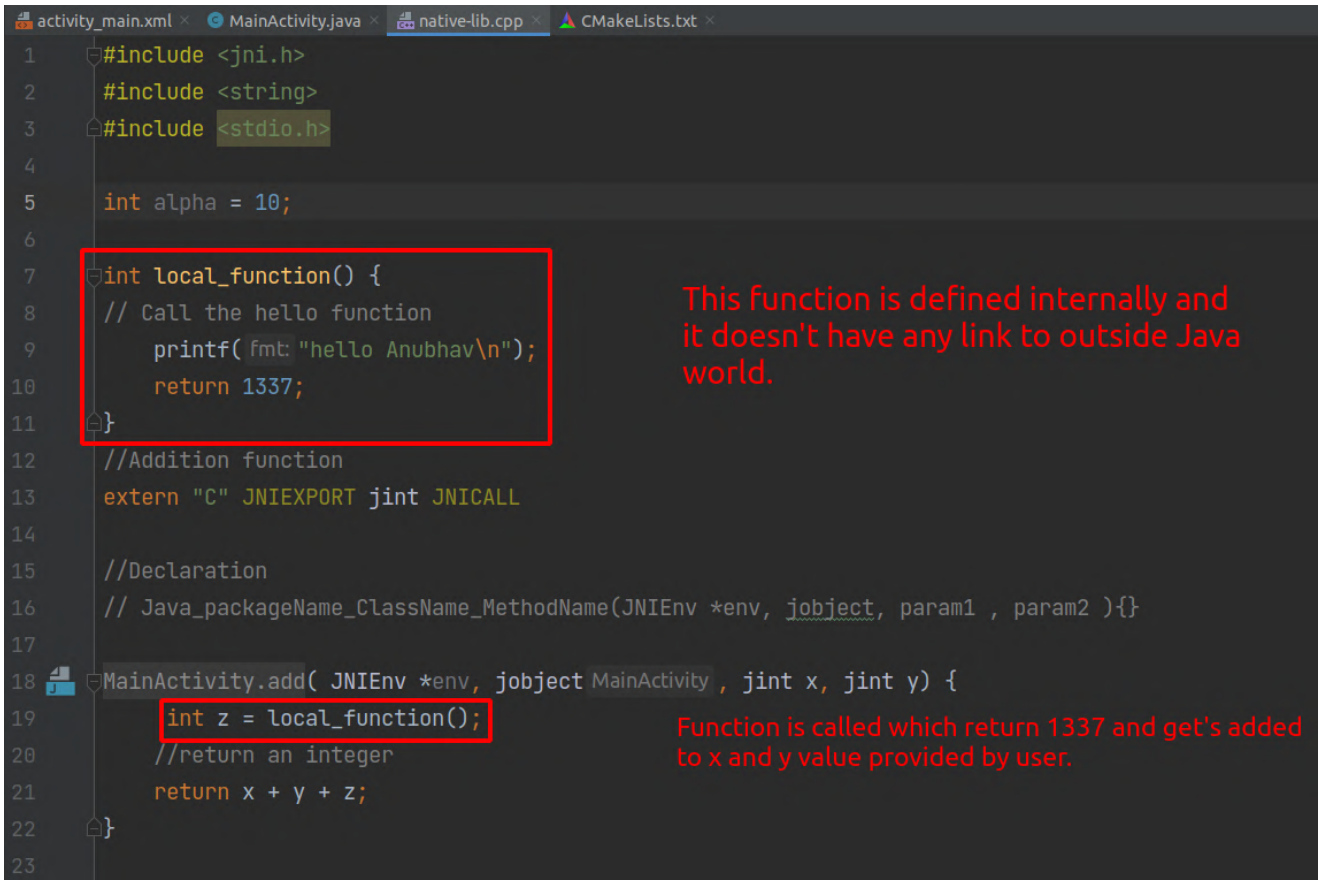
1. Using Java APIs (Easy)
2. Using Native APIs (Difficult)
3. Using Ghidra

1. Using Java APIs

This is the easiest method to hook native functions using Java APIs. As you can see in the image below, there's a transition between the Java world and the C/C++ world. We declare the function in the Java world, and its definition is written in the native library.



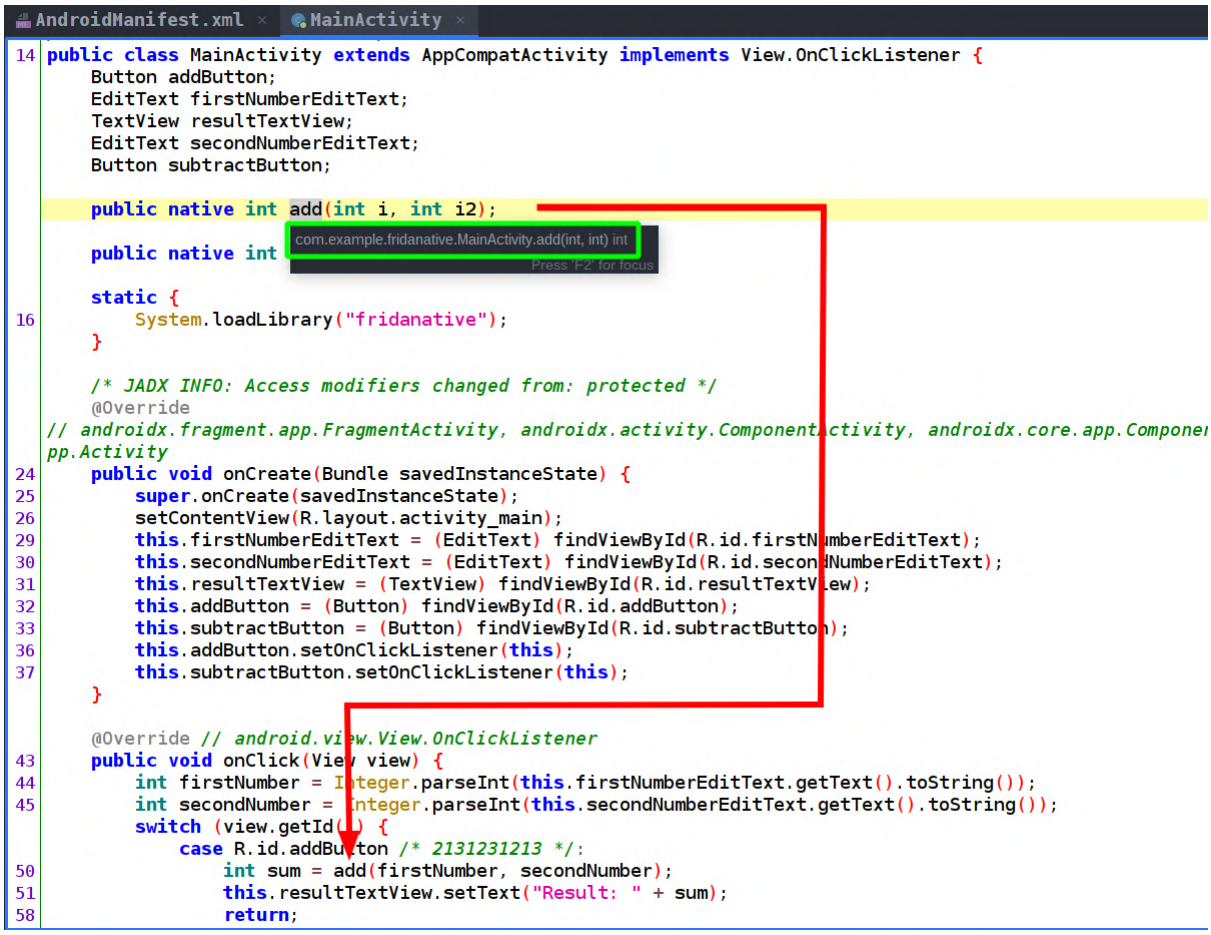
The main motive to hook a function is to retrieve passed parameter values and any parameters you want to pass, as well as to observe the return value. We can accomplish all of this using Java APIs. We are able to do this because there is a declaration of that native function in a Java file. However, suppose there is a function in a native library that is only evaluated within that library and doesn't have any declaration in Java. In that case, we can't use Java APIs to hook that function.



As you can see above, the `local_function()` is declared in a C file, which doesn't have a link to the Java file like we have for add and subtract. That's why we can't hook `local_function()` using Java APIs, but we can hook it using Native APIs, which we will see in the second method.

Let's hook the add native method using the Java APIs and see their parameter value and return value.

A. This is MainActivity.java where the native function named add is declared. We intend to hook this method to observe the function parameters and return value it yields.

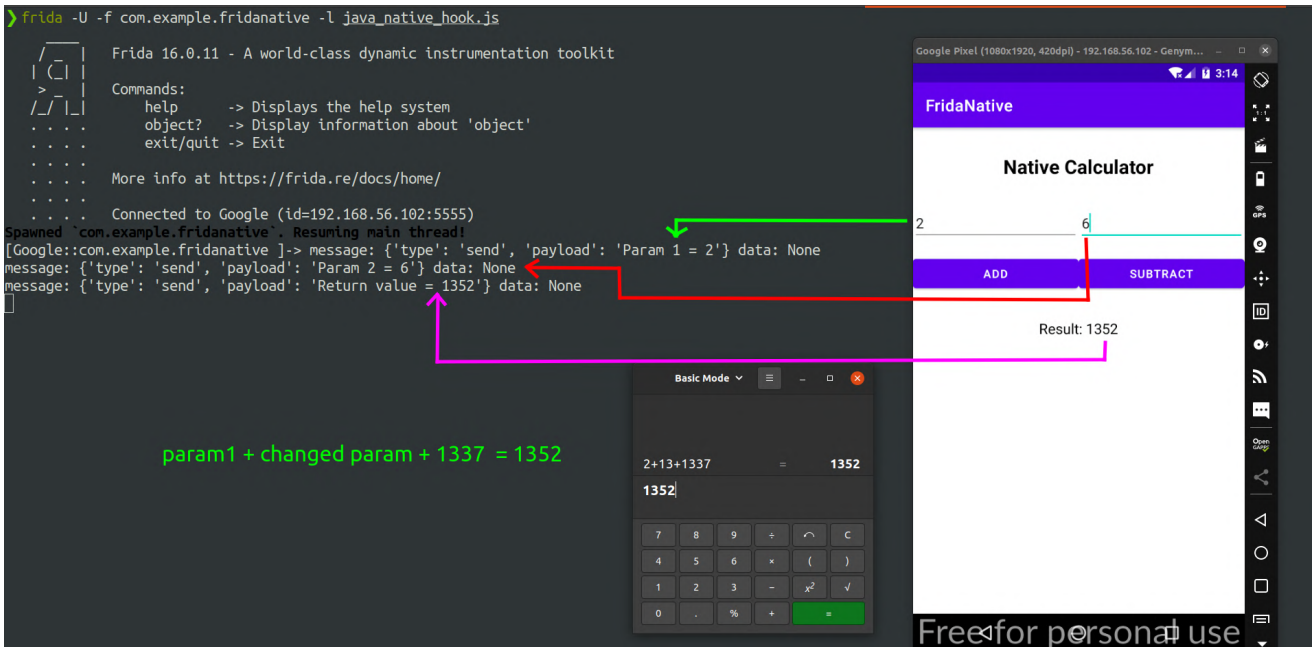



```
14 public class MainActivity extends AppCompatActivity implements View.OnClickListener {
       Button addButton;
       EditText firstNumberEditText;
       TextView resultTextView;
       EditText secondNumberEditText;
       Button subtractButton;

       public native int add(int i, int i2);

       public native int

       static {
16         System.loadLibrary("fridanative");
       }

       /* JADX INFO: Access modifiers changed from: protected */
       @Override
// androidx.fragment.app.FragmentActivity, androidx.activity.ComponentActivity, androidx.core.app.Componer
pp.Activity
24     public void onCreate(Bundle savedInstanceState) {
25         super.onCreate(savedInstanceState);
26         setContentView(R.layout.activity_main);
29         this.firstNumberEditText = (EditText) findViewById(R.id.firstNumberEditText);
30         this.secondNumberEditText = (EditText) findViewById(R.id.secondNumberEditText);
31         this.resultTextView = (TextView) findViewById(R.id.resultTextView);
32         this.addButton = (Button) findViewById(R.id.addButton);
33         this.subtractButton = (Button) findViewById(R.id.subtractButton);
36         this.addButton.setOnClickListener(this);
37         this.subtractButton.setOnClickListener(this);
       }

       @Override // android.view.View.OnClickListener
43     public void onClick(View view) {
44         int firstNumber = Integer.parseInt(this.firstNumberEditText.getText().toString());
45         int secondNumber = Integer.parseInt(this.secondNumberEditText.getText().toString());
           switch (view.getId()) {
               case R.id.addButton /* 2131231213 */:
50                 int sum = add(firstNumber, secondNumber);
51                 this.resultTextView.setText("Result: " + sum);
58                 return;
```

B. Writing a JavaScript snippet to hook the above method.

```
// Easy way

Java.perform(function() {

    // Class reference of the Mainactivity
    var mainActivity = Java.use('com.example.fridanative.MainActivity');
    mainActivity.add.implementation = function(param1, param2) {

        // Logging parameter values
        send("Param 1 = " + param1);
        send("Param 2 = " + param2);

        // Changing Second parameter value
        var returnValue = this.add(param1, 13);

        // Logging return value
        send("Return value = " + returnValue);

        // returning return value
        return returnValue;
    }

})
```

C. You can see the output after hooking the method.



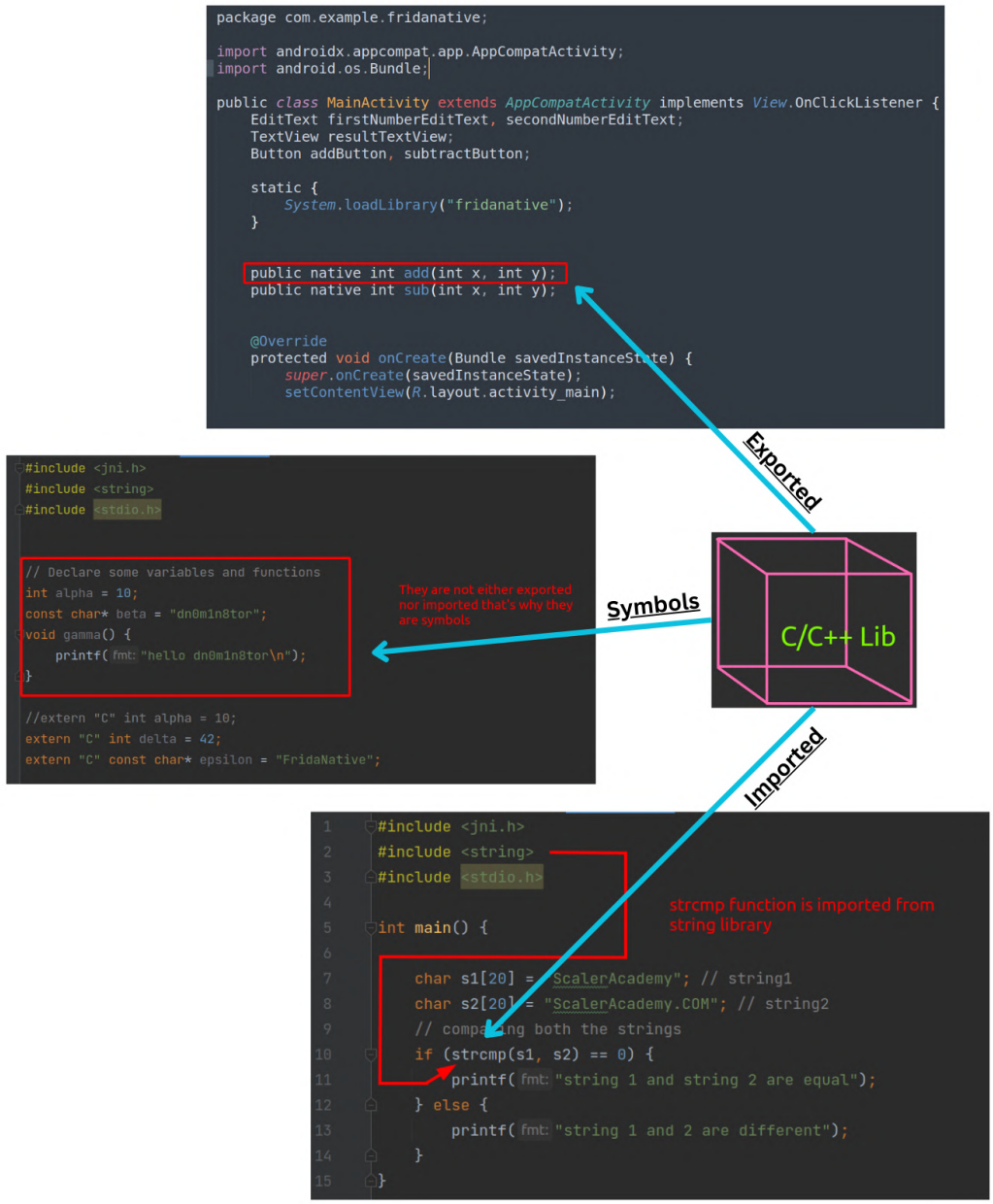
D. This is how you can hook native functions using Java APIs.

2. Using Native APIs

In the native part of Android, which involves using programming languages like C and C++, there are certain concepts related to importing and exporting symbols that are important to understand before moving ahead with hooking part. When a C or C++ library is compiled, it contains symbols, which are essentially names of functions and variables defined within the library. These symbols can be either exported or kept internal.

Exported symbols are made available to other libraries or executable, which means that they can be accessed and used by other parts of the system. On the other hand, internal symbols are only accessible within the library they are defined in and cannot be accessed from outside.

Let's understand this with an example,



Looking at the above image you would have understood what the exported, imported and symbols are,

1. "Exported" refers to functions written in a native library but made accessible for use in the Java world.
2. "Imported" refers to functions brought into a library file through imports, such as strcmp from `lib string`, hence they are referred to as imported.
3. "Symbols" are functions neither exported nor imported; they are solely declared in the library and called within the library.

To hook native functions, Frida has provided us with these APIs. Therefore, we first need to understand how to utilize them.

1. Module.enumerateExports(libname)

Using this API, we are able to enumerate exported functions and variables. This API returns an array with three entries: address, name, and type, indicating whether it is a function or variable.

```
[Google::com.example.fridanative ]-> Module.enumerateExports("libfridanative.so")
[
    {
        "address": "0xd0378a00",
        "name": "Java_com_example_fridanative_MainActivity_sub",
        "type": "function"
    },
    {
        "address": "0xd0378990",
        "name": "_Z17internal_functionv",
        "type": "function"
    },
    {
        "address": "0xd0378840",
        "name": "_Z5gammav",
        "type": "function"
    },
    {
        "address": "0xd0378870",
        "name": "main",
        "type": "function"
    },
    {
        "address": "0xd03789c0",
        "name": "Java_com_example_fridanative_MainActivity_add",
        "type": "function"
    },
    {
        "address": "0xd0378960",
        "name": "_Z14local_functionv",
        "type": "function"
    },
    {
        "address": "0xd037abd0",
        "name": "alpha",
        "type": "variable"
    },
```

2. Module.enumerateImports(libname)

Using this API, we can enumerate imported functions and variables. This API returns an array with four entries: address, module, name, and the type, indicating whether it is a function or variable.

```
[Google::com.example.fridanative ]-> Module.enumerateImports("libfridanative.so")
[
    {
        "address": "0xef505dd0",
        "module": "/system/lib/libc.so",
        "name": "__cxa_atexit",
        "type": "function"
    },
    {
        "address": "0xef505f20",
        "module": "/system/lib/libc.so",
        "name": "__cxa_finalize",
        "type": "function"
    },
    {
        "address": "0xef4f04c0",
        "module": "/system/lib/libc.so",
        "name": "__register_atfork",
        "type": "function"
    },
    {
        "address": "0xef4f41d0",
        "module": "/system/lib/libc.so",
        "name": "__stack_chk_fail",
        "type": "function"
    },
    {
        "address": "0xef49b200",
        "module": "/system/lib/libc.so",
        "name": "memcpy",
        "type": "function"
    },
    {
        "address": "0xef503120",
        "module": "/system/lib/libc.so",
        "name": "printf",
        "type": "function"
    },
    {
        "address": "0xef497dc0",
        "module": "/system/lib/libc.so",
        "name": "strcmp",
        "type": "function"
    }
]
[Google::com.example.fridanative ]->
```

I hope you have understood the concept and are ready to hook native functions using native APIs.

We will look into

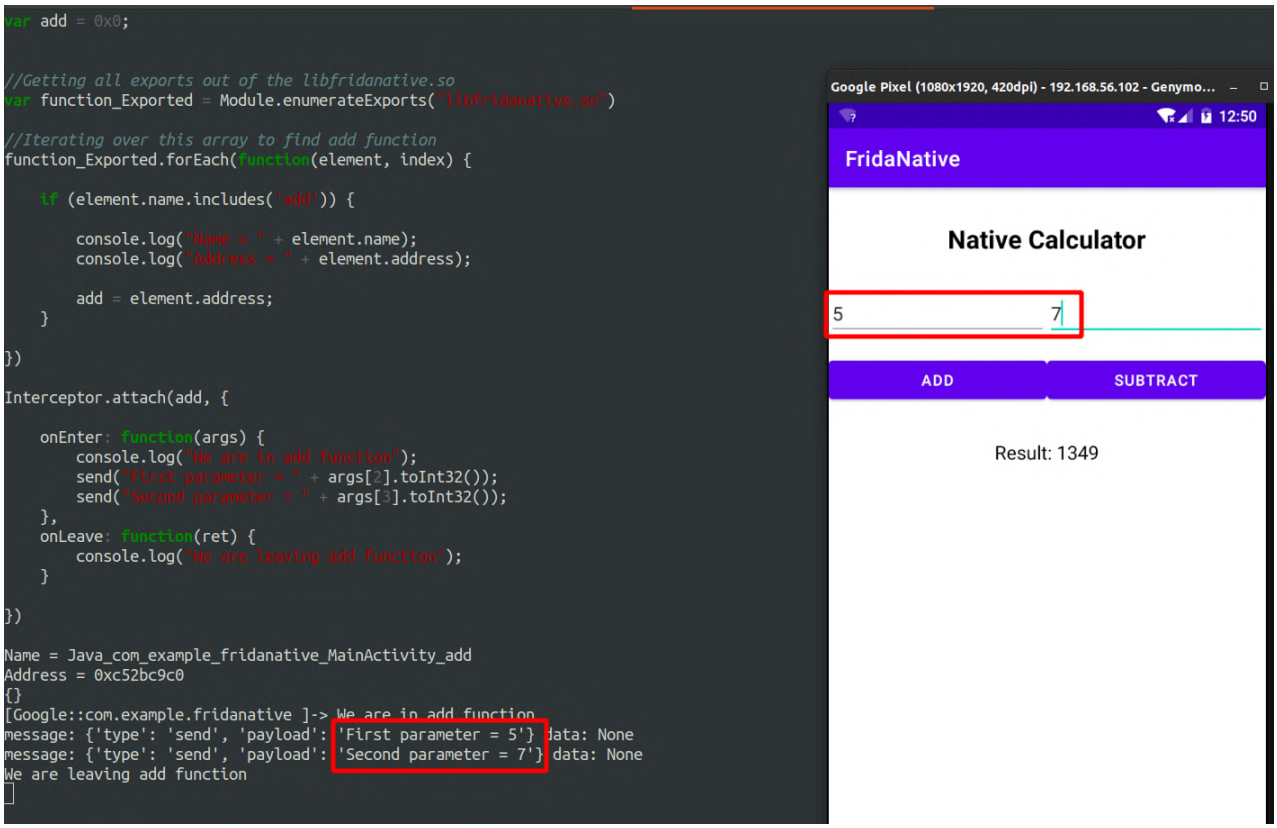
1. Reading Function Parameter
2. Reading Return Value
3. Pass Function Parameter
4. Changing Return Value

1. Reading Function Parameter

Using the below script we can hook the native function add

```
//Address of the function which we want to hook
var add_func_address = 0x0;

//Getting all exports out of the libfridanative.so
var function_Exported = Module.enumerateExports("libfridanative.so")

//Iterating over this array to find add function
function_Exported.forEach(function(element, index) {

    if (element.name.includes('add')) {

        console.log("Name = " + element.name);
        console.log("Address = " + element.address);

        add_func_address = element.address;
    }

})

Interceptor.attach(add_func_address, {

    onEnter: function(args) {
        console.log("We are in add function");
        //casts this NativePointer (args[2]) to a signed 32-bit integer
        send("First parameter = " + args[2].toInt32());
        send("Second parameter = " + args[3].toInt32());
    },
    onLeave: function(ret) {
        console.log("We are leaving add function");
    }

})
```

Launch the application using Frida, then paste the above Frida script to hook the native function.



You can see above, the parameters are listed.

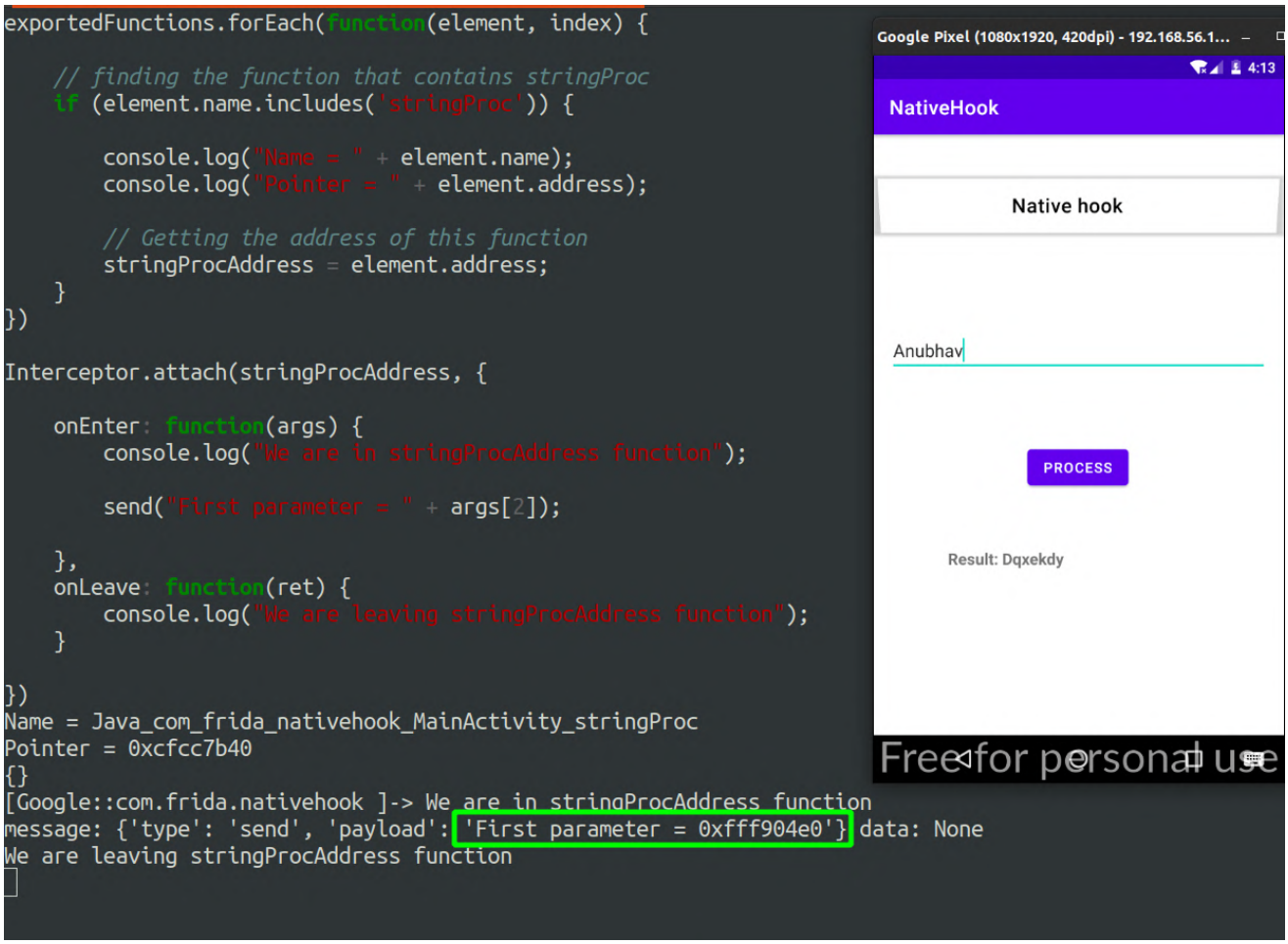
But what if the parameters are of string type? Will we be able to read them using the above method? Let's find out.

To demonstrate this, I have created another application. In this application, you will notice an input field. When you input the character "a," it will display the result as "d." Essentially, it shifts each alphabet three positions forward, such as a → d, b → e, c → f, and so forth.

So, using the script below, which is the same as the one we used previously.

```
//Address of the function which we want to hook
var add_func_address = 0x0;

//Getting all exports out of the libfridanative.so
var function_Exported = Module.enumerateExports("libfridanative.so")

//Iterating over this array to find add function
function_Exported.forEach(function(element, index) {

    if (element.name.includes('add')) {

        console.log("Name = " + element.name);
        console.log("Address = " + element.address);

        add_func_address = element.address;
    }

})

Interceptor.attach(add_func_address, {

    onEnter: function(args) {
        console.log("We are in add function");

        send("First parameter = " + args[2]);
    },
    onLeave: function(ret) {
        console.log("We are leaving add function");
    }

})
```

You can see that we haven't obtained the value we desire here.

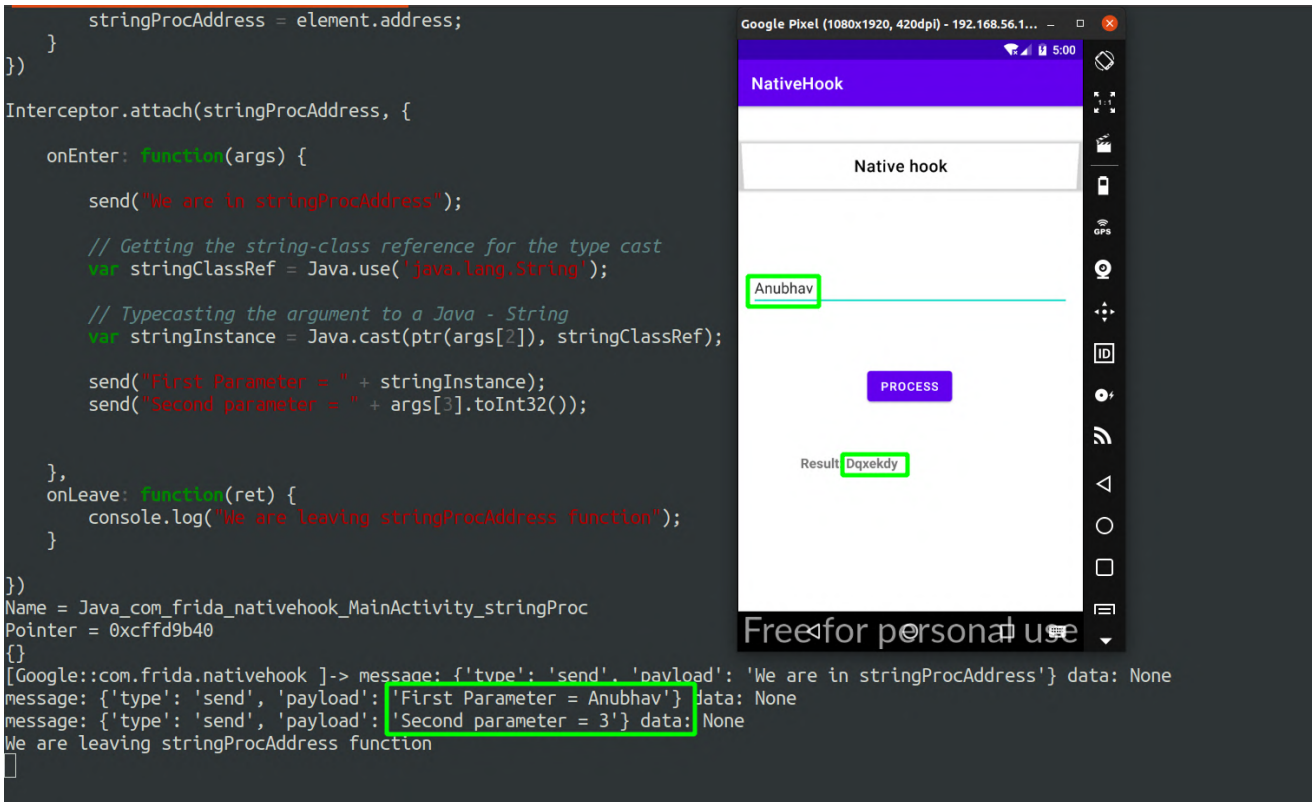


So to display the value we just need to typecast it to the java string.

```
// Address of the function we want to hook
var stringProcAddress = 0x0;

// Getting all exports out of the libnative-lib.so
var exportedFunctions = Module.enumerateExports("libnativehook.so")

// Iterating over this array trying to find stringProc
exportedFunctions.forEach(function(element, index) {

    // finding the function that contains stringProc
    if (element.name.includes('stringProc')) {

        console.log("Name = " + element.name);
        console.log("Pointer = " + element.address);

        // Getting the address of this function
        stringProcAddress = element.address;
    }
})
```

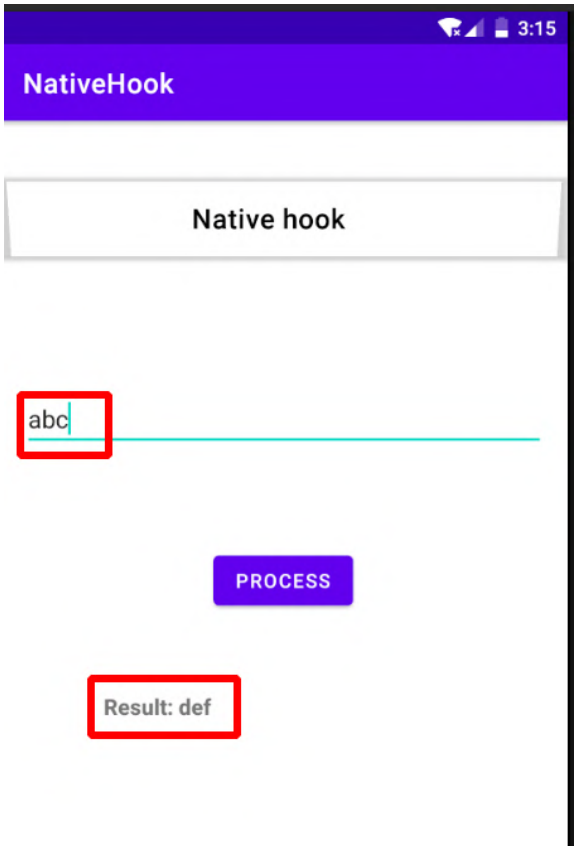
```
        Interceptor.attach(stringProcAddress, {

            onEnter: function(args) {

                send("We are in stringProcAddress");

                // Getting the string-class reference for the type cast
                var stringClassRef = Java.use('java.lang.String');

                // Typecasting the argument to a Java - String
                var stringInstance = Java.cast(ptr(args[2]), stringClassRef);

                send("First Parameter = " + stringInstance);
                send("Second parameter = " + args[3].toInt32());

            },
            onLeave: function(ret) {
                console.log("We are leaving stringProcAddress function");
            }

        })
```

Executing the above script we are able to get the parameters value.



2. Reading Return Value

Here, we only need to read the return value of the function; "def" is returned.



Now, using Frida, we aim to read the return value. Let's proceed.

A. We aim to identify the function responsible for processing all inputs. Therefore, we will utilize the Module.enumerateExports(libname) API to locate it.

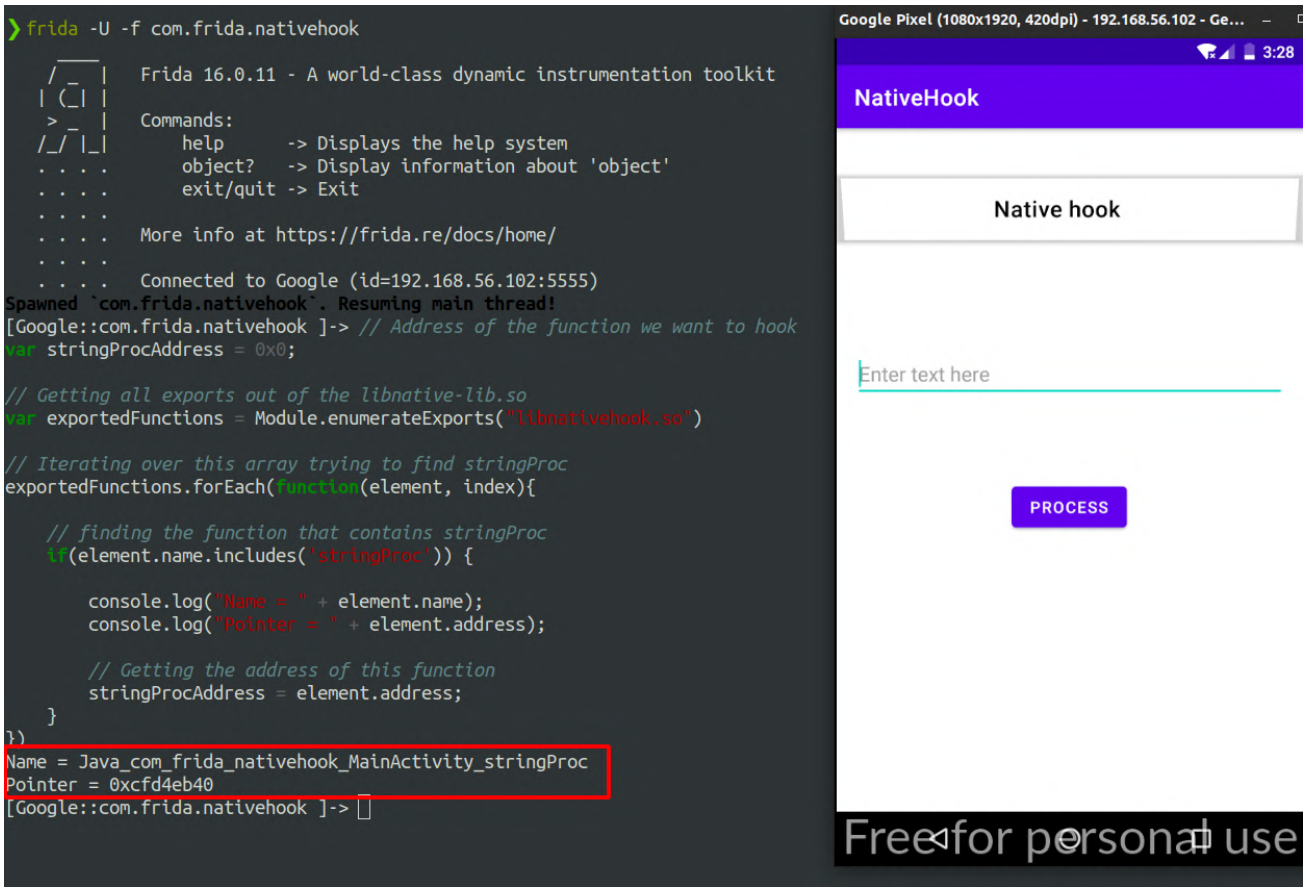
```
❯ frida -U -f com.frida.nativehook
     ____
    / _  |   Frida 16.0.11 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Google (id=192.168.56.102:5555)
Spawned `com.frida.nativehook`. Resuming main thread!
[Google::com.frida.nativehook ]-> Module.enumerateExports("libnativehook.so")
[
    {
        "address": "0xcffe3220",
        "name": "_ZnajRKSt9nothrow_t",
        "type": "function"
    },
    {
        "address": "0xcffda550",
        "name": "_ZTSPw",
        "type": "variable"
    },
```

```
},
{
    "address": "0xcffe2b40",
    "name": "Java_com_frida_nativehook_MainActivity_stringProc",
    "type": "function"
},
{
    "address": "0xcffe2d50",
    "name": "_ZN7_JNIEnv21ReleaseStringUTFCharsEP8_jstringPKc",
    "type": "function"
},
{
    "address": "0xcffe3270",
    "name": "_ZdlPv",
    "type": "function"
},
```

B. Now that we've discovered that **`stringProc`** is the function processing our input string, we'll use a Frida script to determine its address. Remember, to hook native APIs, we need to utilize the **`interceptor.attach`** API, and for this, we require the address of the function we intend to hook.

```
// Address of the function we want to hook
var stringProcAddress = 0x0;

// Getting all exports out of the libnative-lib.so
var exportedFunctions = Module.enumerateExports("libnativehook.so")

// Iterating over this array trying to find stringProc
exportedFunctions.forEach(function(element, index){
```

```
        // finding the function that contains stringProc
        if(element.name.includes('stringProc')) {

            console.log("Name = " + element.name);
            console.log("Pointer = " + element.address);

            // Getting the address of this function
            stringProcAddress = element.address;
        }
    })
```

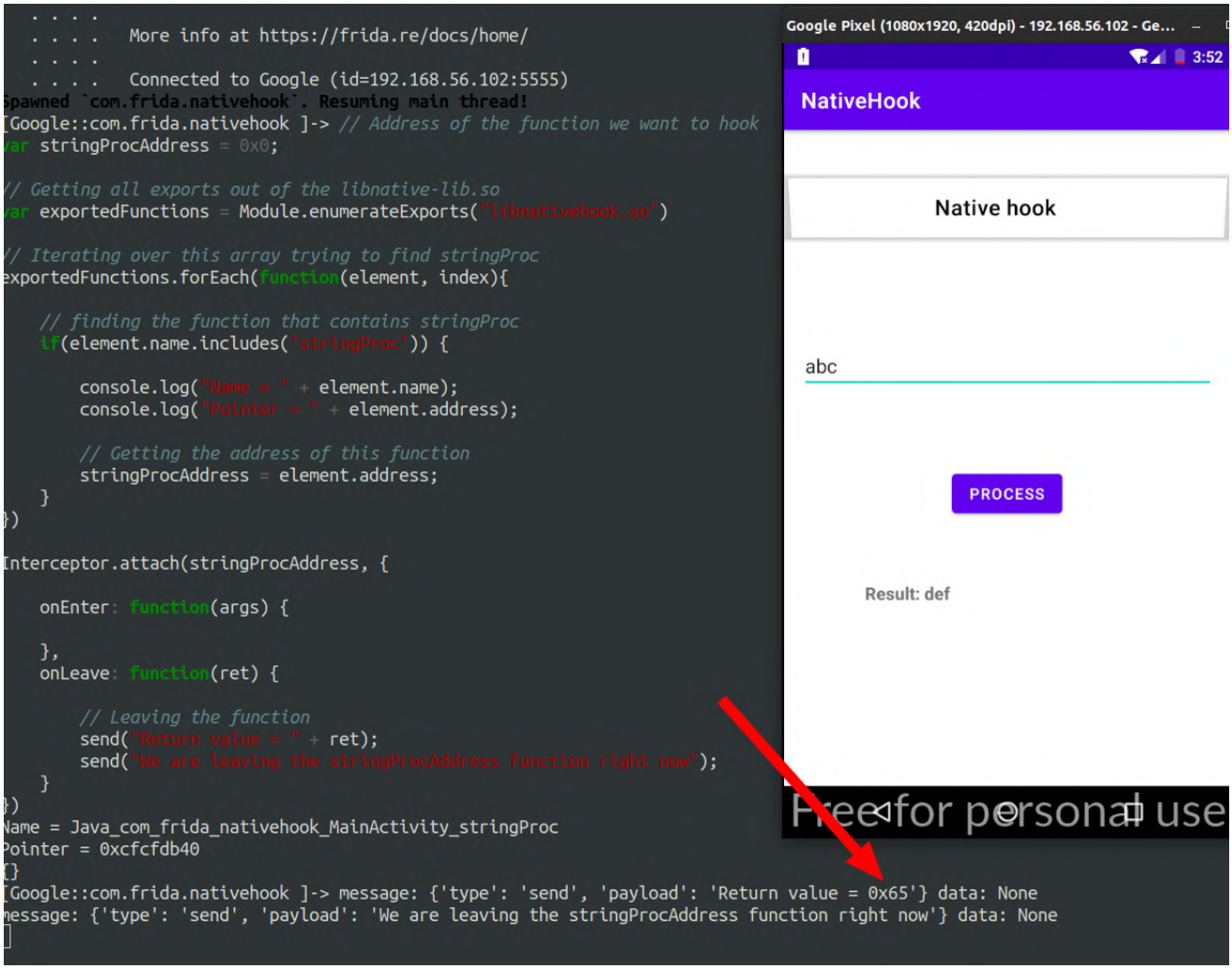


C. Now we will hook the **`stringProc`** function to view its return value using the script below.

```
    // Address of the function we want to hook
    var stringProcAddress = 0x0;

    // Getting all exports out of the libnative-lib.so
    var exportedFunctions = Module.enumerateExports("libnativehook.so")

    // Iterating over this array trying to find decryptString
    exportedFunctions.forEach(function(element, index){
```

```
        // finding the function that contains decryptString
        if(element.name.includes('stringProc')) {

            console.log("Name = " + element.name);
            console.log("Pointer = " + element.address);

            // Getting the address of this function
            stringProcAddress = element.address;
        }
    })

    Interceptor.attach(stringProcAddress, {

        onEnter: function(args) {


        },
        onLeave: function(ret) {

            send("Return value = " + ret);
            send("We are leaving the stringProcAddress function right now");
        }
    })
```

D. After executing the above script, as seen in the image below, we are receiving a hexadecimal value which is not desired. Instead, we aim to have "def" as a string value in our output.



E. To obtain the actual value, we must create a pointer to the specific memory address that is returned, and then typecast it to a string.

Here, we need to do two things:

a. Create a pointer, which can be done with `ptr()` in Frida.

`ptr()` is a function in the Frida toolkit that creates a pointer to a specific memory address in the target process. The `ptr()` function accepts one or more arguments that specify the memory address to which the pointer should point.

b. We need to typecast it to a string, which can be achieved with Java.cast() in Frida.

Java.cast(handle, klass): This function creates a JavaScript wrapper based on the existing instance at handle of the specified class `klass`, as returned from Java.use. Additionally, such a wrapper includes a class property for obtaining a wrapper for its class, and a $className property for retrieving a string representation of its class name.

```
    const Activity = Java.use('android.app.Activity');
    const activity = Java.cast(ptr('0x1234'), Activity);

    this is how we will use both of them
```

F. Let's implement it in our script.

```
// Address of the function we want to hook
var stringProcAddress = 0x0;

// Getting all exports out of the libnative-lib.so
var exportedFunctions = Module.enumerateExports("libnativehook.so")

// Iterating over this array trying to find stringProc
exportedFunctions.forEach(function(element, index){

    // finding the function that contains stringProc
    if(element.name.includes('stringProc')) {

        console.log("Name = " + element.name);
        console.log("Pointer = " + element.address);

        // Getting the address of this function
        stringProcAddress = element.address;
    }
})

Interceptor.attach(stringProcAddress, {

    onEnter: function(args) {

    },
    onLeave: function(ret) {

        // Getting the string-class reference for the type cast
        var stringClassRef = Java.use('java.lang.String');

        // Typecasting the return value to a Java - String
        var stringInstance = Java.cast(ptr(ret), stringClassRef);

        // Leaving the function
        send("Return value = " + stringInstance);
        send("We are leaving the stringProcAddress function right now");
    }
})
```

Output:

G. As you can see above, we are able to read the return value using Frida.

3. Pass Function Parameter

We are going to change the parameter of the function here.

In this application there are 2 parameters:

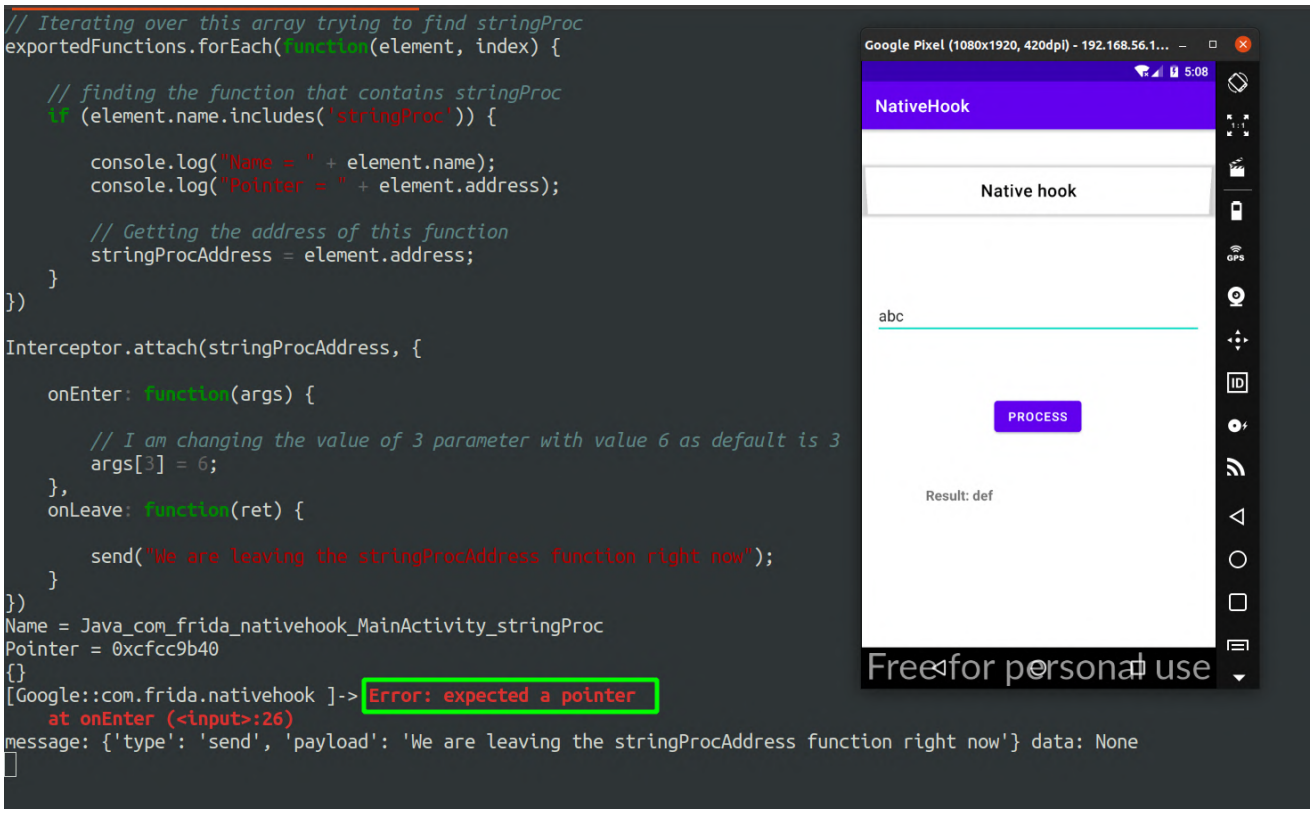
1. Our input
2. The number to which the alphabet is going to be shifted forward. For example, if the number is 3, then a → d, b → e, and c → f.

I have written the below script to hook the function and change the second parameter of the function.

```
// Address of the function we want to hook
var stringProcAddress = 0x0;

// Getting all exports out of the libnative-lib.so
var exportedFunctions = Module.enumerateExports("libnativehook.so")

// Iterating over this array trying to find stringProc
exportedFunctions.forEach(function(element, index){

    // finding the function that contains stringProc
    if(element.name.includes('stringProc')) {
```

```
            console.log("Name = " + element.name);
            console.log("Pointer = " + element.address);

            // Getting the address of this function
            stringProcAddress = element.address;
        }
    })

    Interceptor.attach(stringProcAddress, {

        onEnter: function(args) {

            // I am changing the value of 3 parameter with value 6 as default is 3
            args[3] = 6;
        },
        onLeave: function(ret) {

            send("We are leaving the stringProcAddress function right now");
        }
    })
```

Below, you can see that we encountered some errors related to pointers.



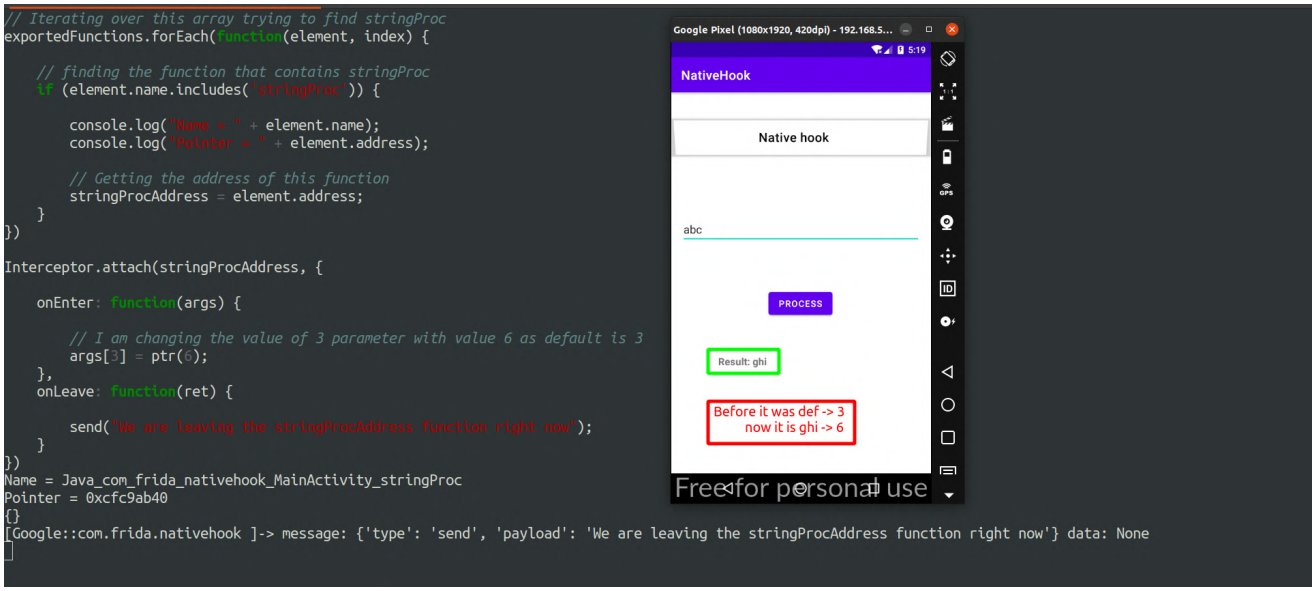
We encountered this error because we need to provide a pointer to args[3] rather than directly assigning a static value to it. Therefore, we can assign the pointer of 6 to args[3] using `prt()`.

Simple like `args[3] = prt(6);`

Modifying the script

```
// Address of the function we want to hook
var stringProcAddress = 0x0;

// Getting all exports out of the libnative-lib.so
var exportedFunctions = Module.enumerateExports("libnativehook.so")

// Iterating over this array trying to find stringProc
exportedFunctions.forEach(function(element, index){

    // finding the function that contains stringProc
    if(element.name.includes('stringProc')) {

        console.log("Name = " + element.name);
        console.log("Pointer = " + element.address);

        // Getting the address of this function
        stringProcAddress = element.address;
    }
})

Interceptor.attach(stringProcAddress, {

    onEnter: function(args) {

        // I am changing the value of 3 parameter with value 6 as default is 3
        args[3] = ptr(6);
    },
    onLeave: function(ret) {

        send("We are leaving the stringProcAddress function right now");
    }
})
```

Executing the script.

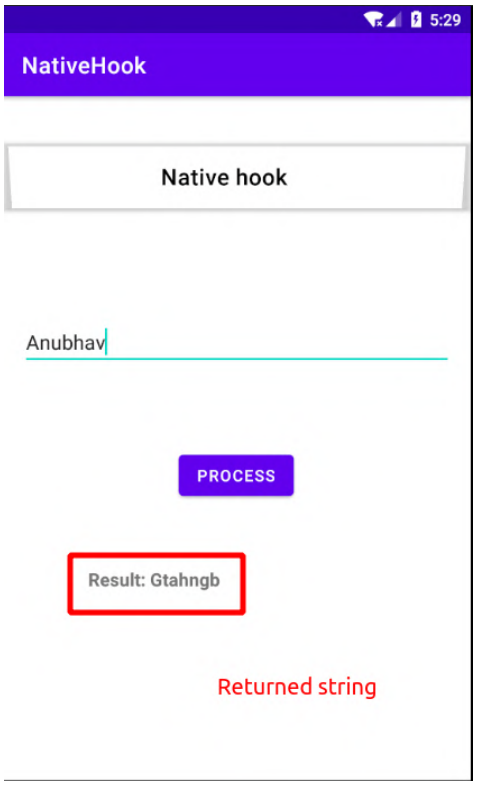


This is how we can change the function parameter using native APIs.

4. Changing Return Value

Here, we'll explore how we can alter the return value of a function using Frida native APIs. As you've observed, the application returns a processed string based on the input. However, what if I desire to return my own string? Let's delve into how we can achieve this using Frida.

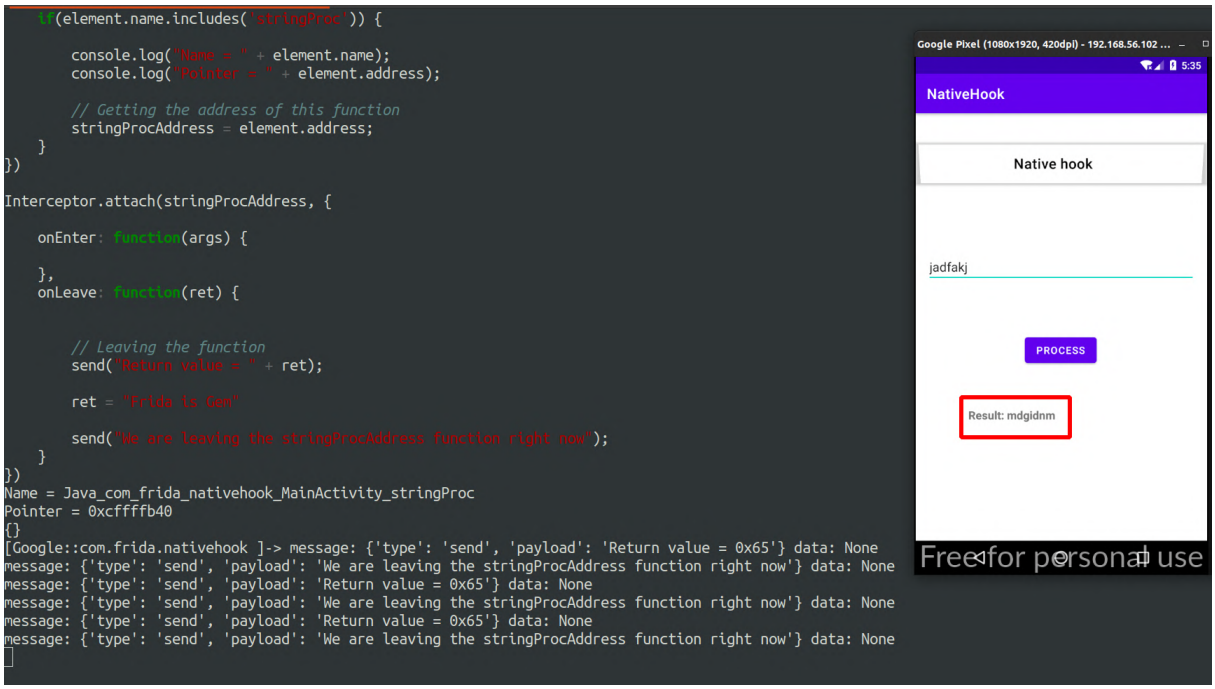
A. So our application takes input from us, advances each alphabet by three positions, and then displays the result on the screen.



B. Writing a Frida script to change the return value to “Frida is Gem”.

```
            // Address of the function we want to hook
            var stringProcAddress = 0x0;

            // Getting all exports out of the libnative-lib.so
            var exportedFunctions = Module.enumerateExports("libnativehook.so")

            // Iterating over this array trying to find stringProc
            exportedFunctions.forEach(function(element, index){

                // finding the function that contains stringProc
                if(element.name.includes('stringProc')) {

                    console.log("Name = " + element.name);
                    console.log("Pointer = " + element.address);

                    // Getting the address of this function
                    stringProcAddress = element.address;
                }
            })

            Interceptor.attach(stringProcAddress, {

                onEnter: function(args) {

                },
                onLeave: function(ret) {

                    // Leaving the function
                    send("Return value = " + ret);

                    // Assigning value to ret as it's a return value which is
going to be passed
                    ret = "Frida is Gem"

                    send("We are leaving the stringProcAddress function right
now");
                }
            })
```

C. You can see after executing the script the return value hasn't changed.

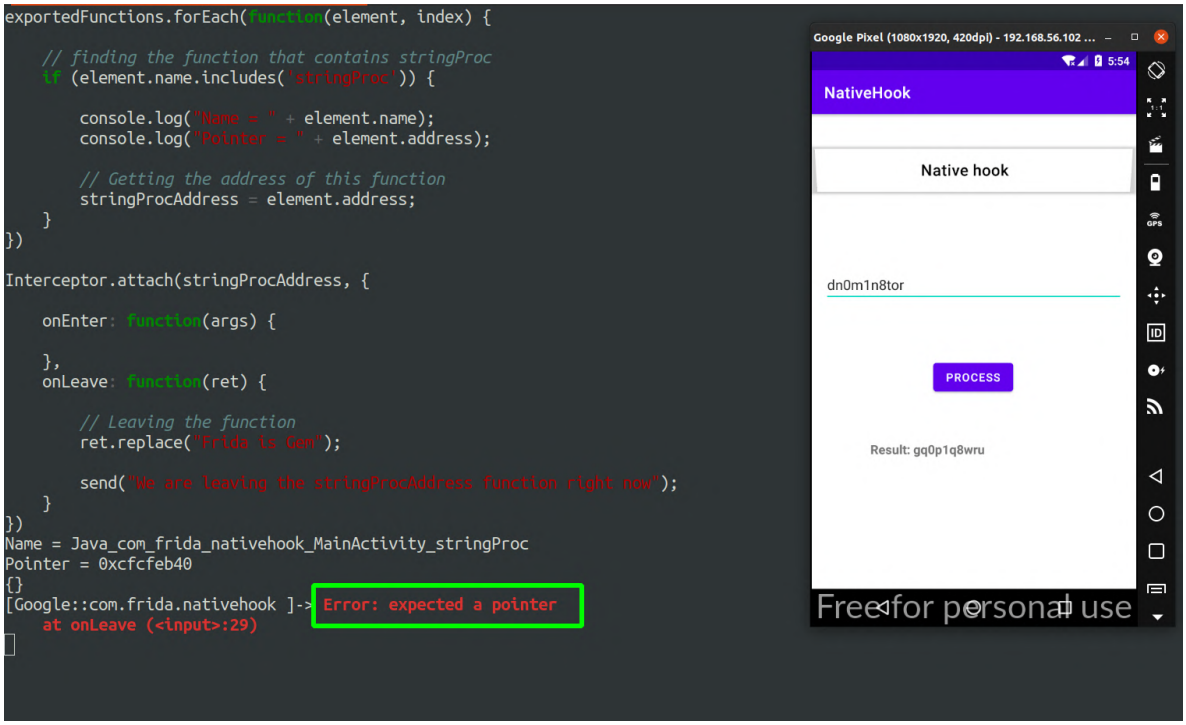


D. To replace the return value in the native world using Frida, we must utilize the `.replace()` function.

```
// Address of the function we want to hook
var stringProcAddress = 0x0;

// Getting all exports out of the libnative-lib.so
var exportedFunctions = Module.enumerateExports("libnativehook.so")

// Iterating over this array trying to find stringProc
exportedFunctions.forEach(function(element, index){

    // finding the function that contains stringProc
    if(element.name.includes('stringProc')) {

        console.log("Name = " + element.name);
        console.log("Pointer = " + element.address);

        // Getting the address of this function
        stringProcAddress = element.address;
    }
})

Interceptor.attach(stringProcAddress, {

    onEnter: function(args) {

    },
    onLeave: function(ret) {

        // Leaving the function
        ret.replace("Frida is Gem");
```

```
            send("We are leaving the stringProcAddress function right now");
        }
    })
```

Executing the above script



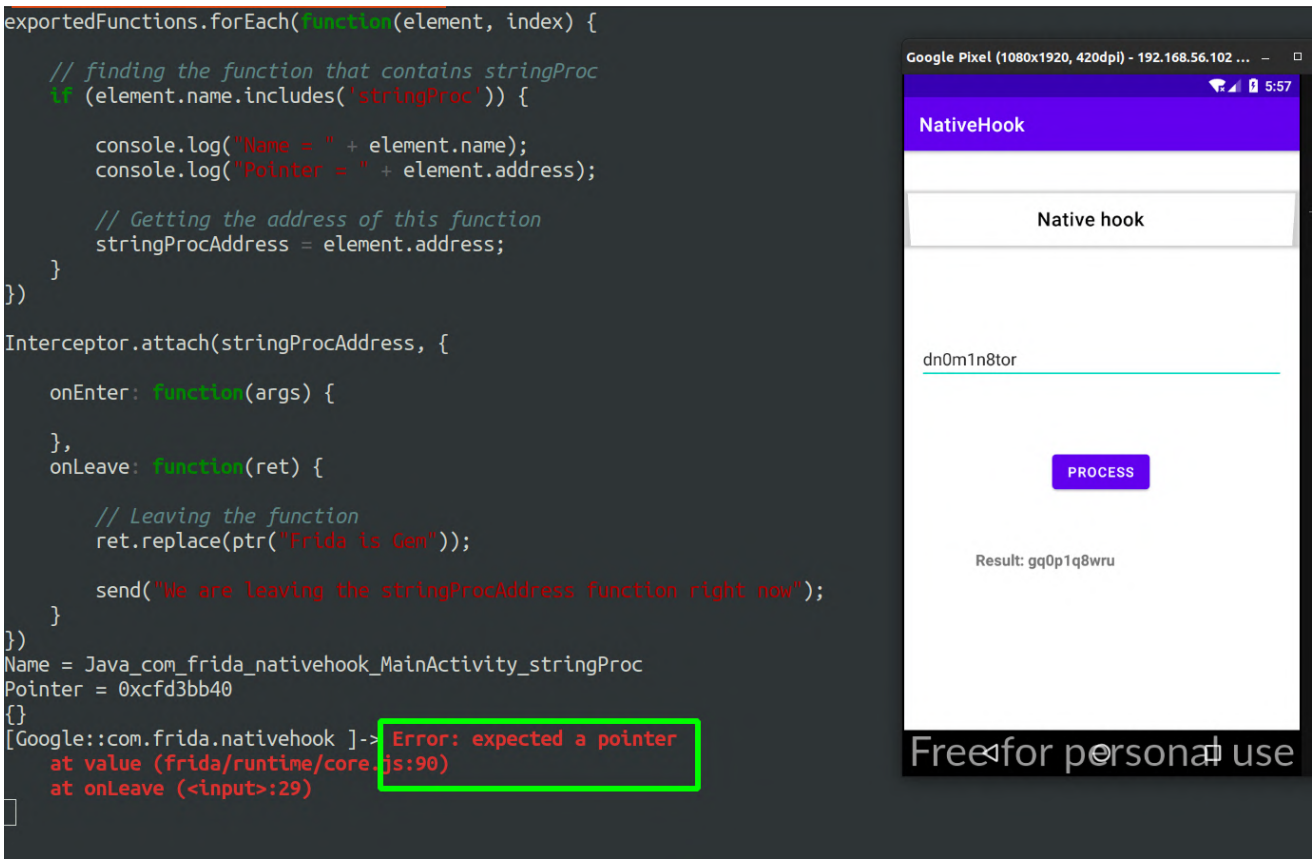
We got an error as expected a pointer.

E. Now we will use the ptr() function to determine if we can resolve this error.

```
        // Address of the function we want to hook
        var stringProcAddress = 0x0;

        // Getting all exports out of the libnative-lib.so
        var exportedFunctions = Module.enumerateExports("libnativehook.so")

        // Iterating over this array trying to find stringProc
        exportedFunctions.forEach(function(element, index) {

            // finding the function that contains stringProc
            if (element.name.includes('stringProc')) {

                console.log("Name = " + element.name);
                console.log("Pointer = " + element.address);

                // Getting the address of this function
                stringProcAddress = element.address;
            }
        })
```

```
        Interceptor.attach(stringProcAddress, {

            onEnter: function(args) {

            },
            onLeave: function(ret) {

                // Leaving the function
                ret.replace(ptr("Frida is Gem"));

                send("We are leaving the stringProcAddress function right now");
            }
        })
```

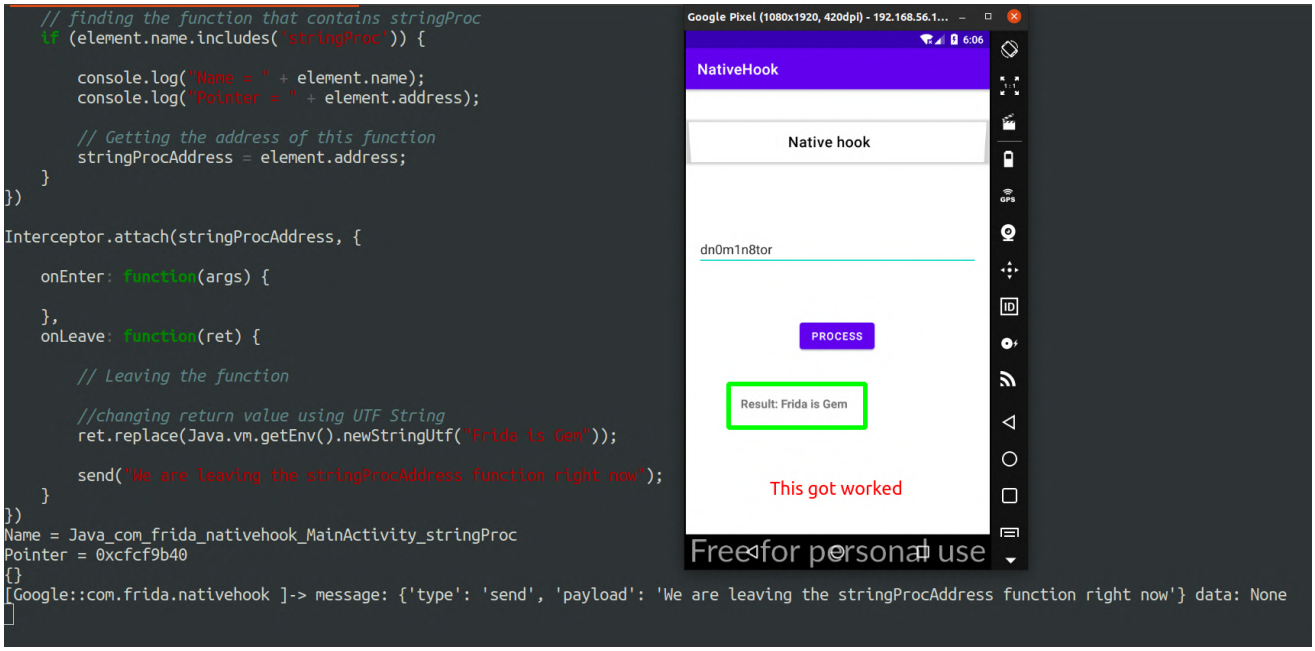
But the error is still the same



F. You've observed that it requested a pointer, and we supplied it, yet we still encounter an error. If you aim to alter the return value, this method isn't effective. Instead, we need to generate a new string, but not a Java string as that won't function in this context. Thus, we must utilize a UTF string created through the Java Environment, which will function properly.

```
// Address of the function we want to hook
var stringProcAddress = 0x0;

// Getting all exports out of the libnative-lib.so
var exportedFunctions = Module.enumerateExports("libnativehook.so")

// Iterating over this array trying to find stringProc
exportedFunctions.forEach(function(element, index){

    // finding the function that contains stringProc
    if(element.name.includes('stringProc')) {

        console.log("Name = " + element.name);
        console.log("Pointer = " + element.address);

        // Getting the address of this function
        stringProcAddress = element.address;
    }
})

Interceptor.attach(stringProcAddress, {

    onEnter: function(args) {

    },
    onLeave: function(ret) {

        // Leaving the function

        //changing return value using UTF String
          ret.replace(Java.vm.getEnv().newStringUtf("Frida is Gem"));

        send("We are leaving the stringProcAddress function right now");
    }
})
```

Executing the above script



As you can see above we are able to change the return value using Frida.

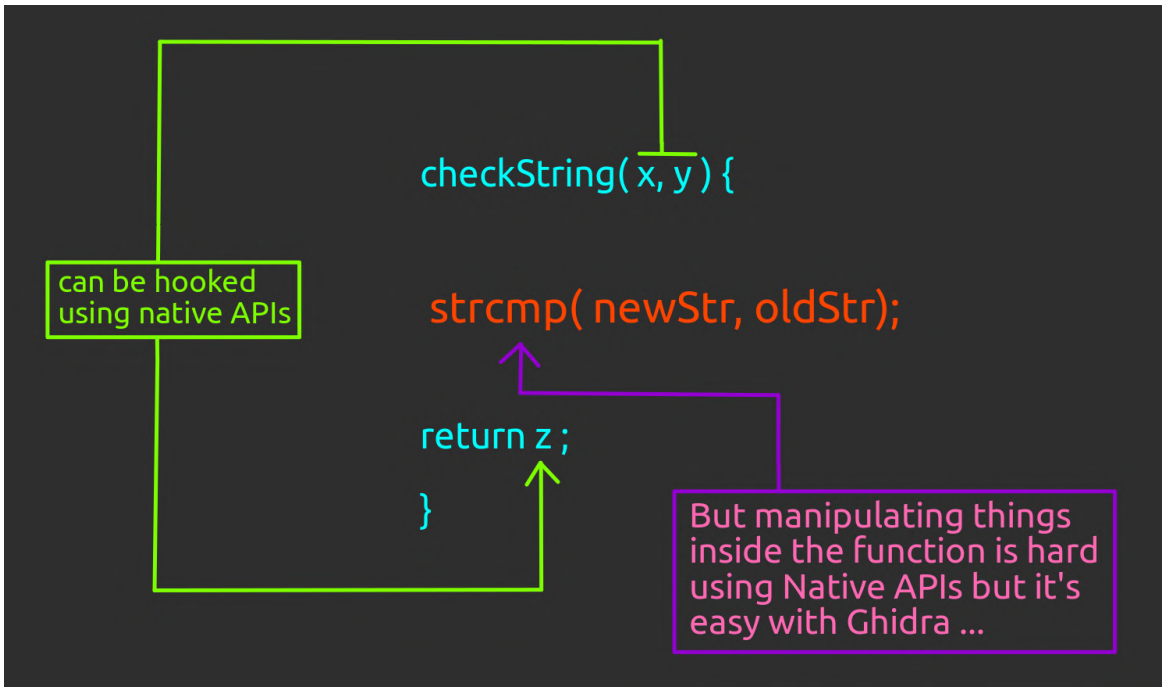
I hope this section has now trained you guys to hook native functions using Frida with proper understanding.

3. Using Ghidra

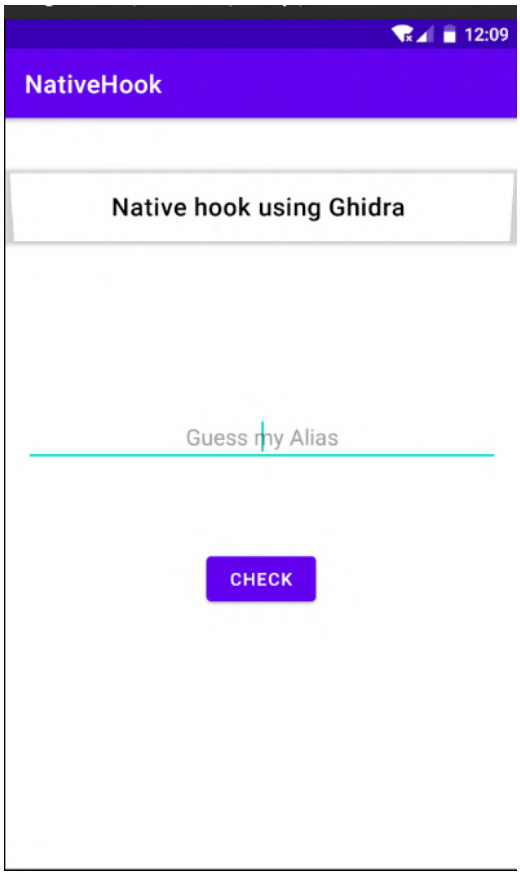
Ghidra is a software reverse engineering (SRE) framework created and maintained by the National Security Agency Research Directorate. This framework includes a suite of full-featured, high-end software analysis tools that enable users to analyze compiled code on a variety of platforms including Windows, macOS, and Linux. Capabilities include disassembly, assembly, decompilation, graphing, and scripting, along with hundreds of other features. Ghidra supports a wide variety of processor instruction sets and executable formats and can be run in both user-interactive and automated modes.

But why is there a need to use Ghidra when we have Native APIs?

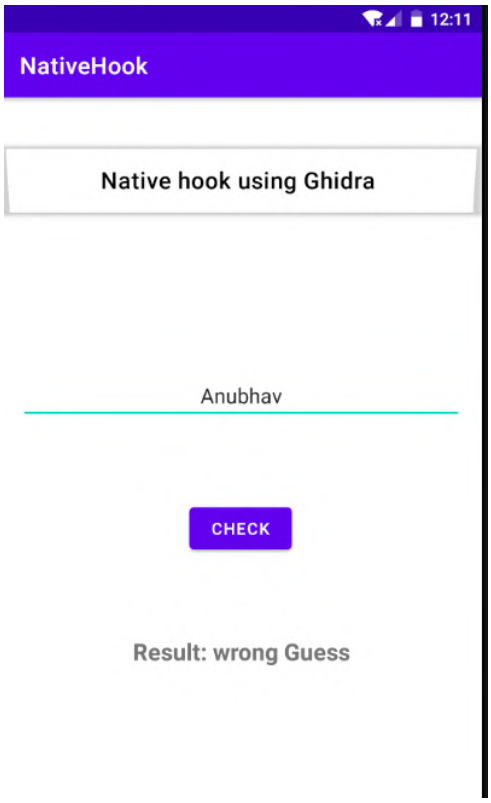
Suppose `checkString` is a function that takes some parameters, performs string comparison with those parameters, and returns the result. Suppose you want to hook the `strcmp` function which is inside the `checkString` function. Hooking this function is challenging using Native APIs. The difficulty lies in obtaining the address of `strcmp` in the below function; otherwise, the process is similar to using native APIs to hook.



To demonstrate this hooking using Ghidra, I have created an application that prompts the user to "Guess my alias."



If you provide an incorrect name and click on "check," it will display "wrong guess."



So here, we need to find the correct value using Frida.

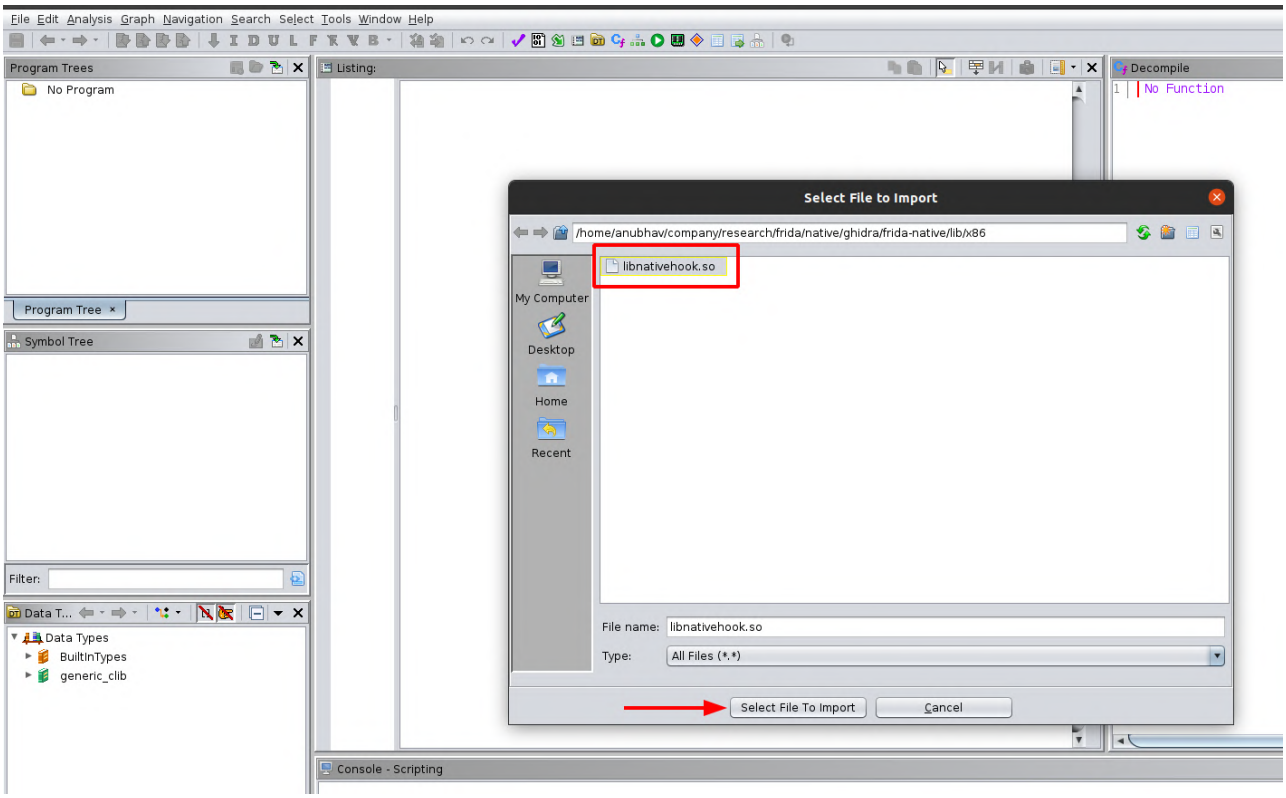
A. Decompile the application using the apktool

```
apktool d <apkname.apk>
```

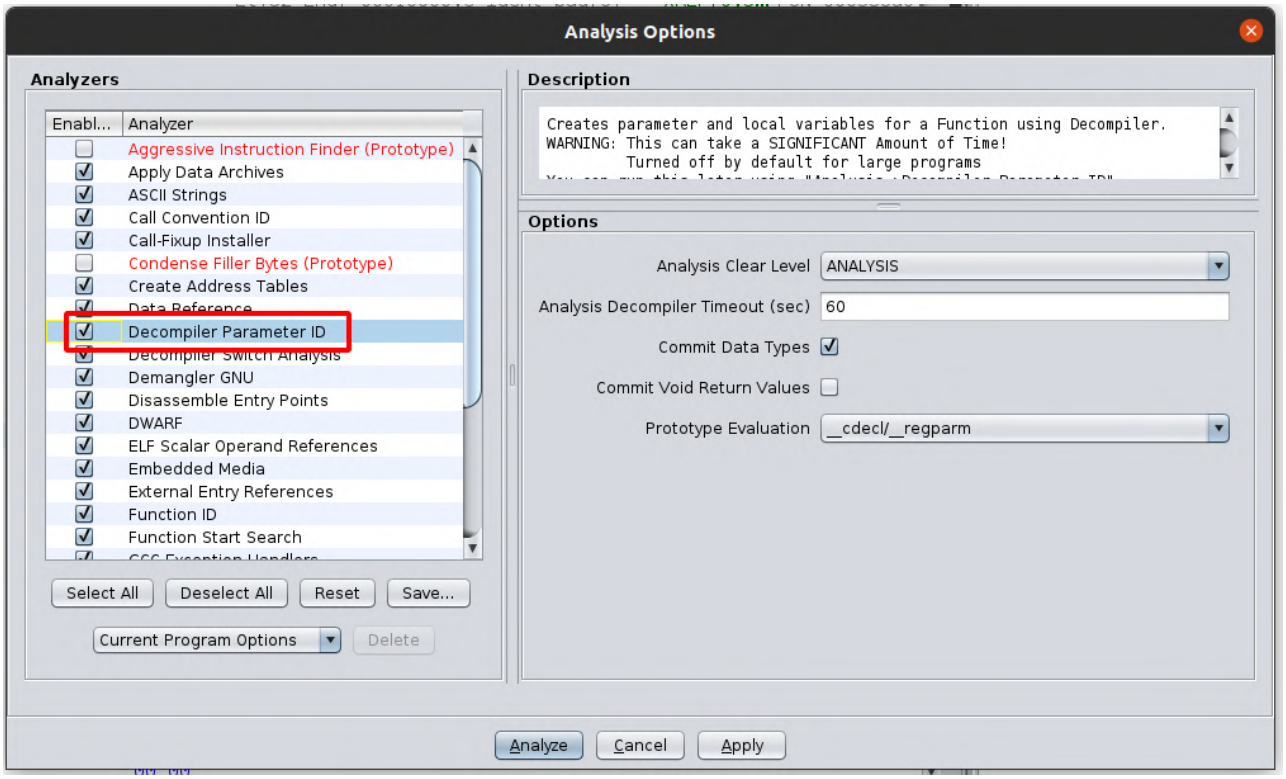
B. Go to the "lib" folder and retrieve the libnativehook.so file. Here, we are using the x86 architecture file since I am using an emulator with x86 architecture.

```
❯ pwd
                                    frida/native/ghidra/frida-native/lib/x86
❯ ls
libnativehook.so
```

C. Import the above file to Ghidra.



D. Keep the rest of the options as they are and simply enable `Decompiler Parameter ID`.



E. Once the analysis is complete, navigate to jadx-gui and examine the Java code to identify the target for hooking in order to obtain the correct answer.

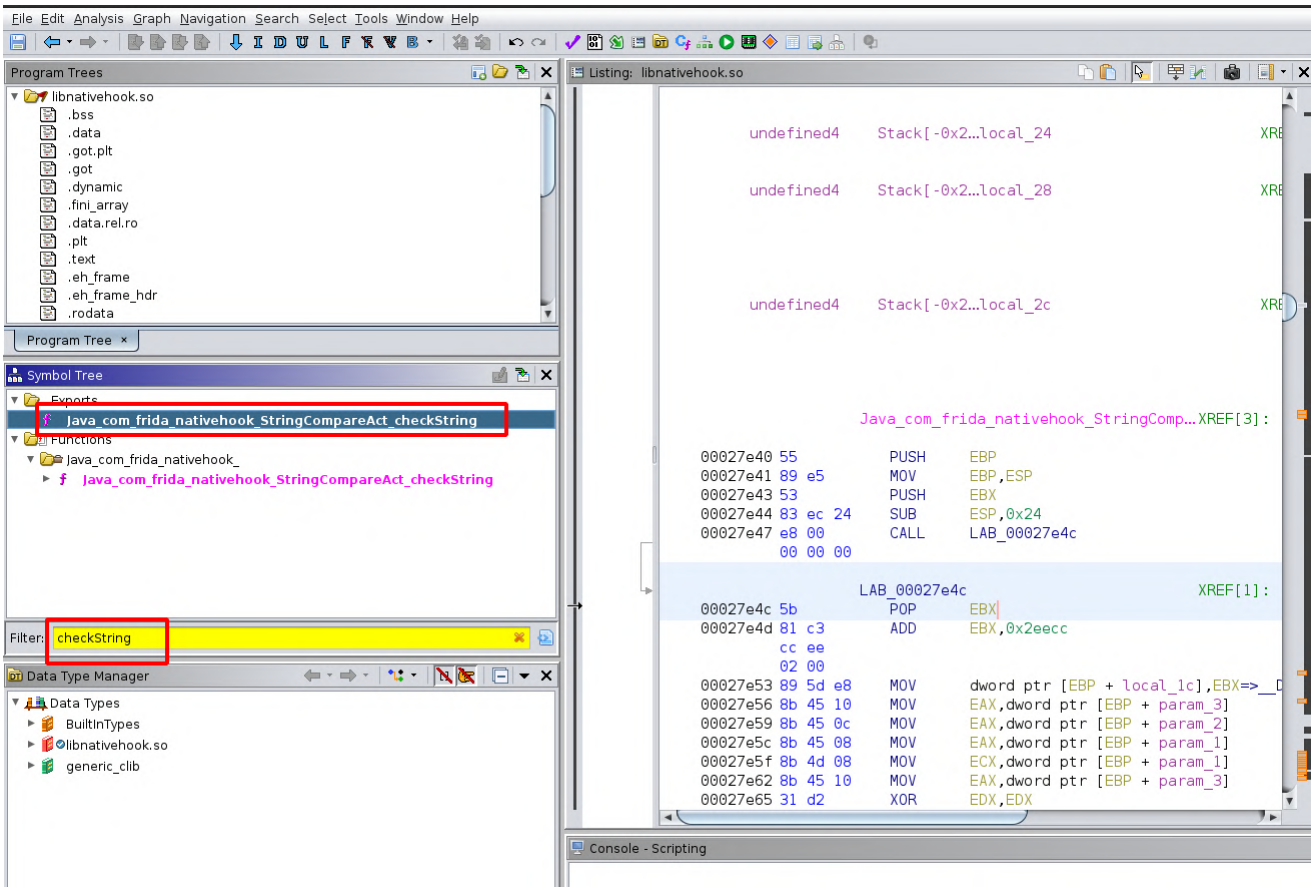
Viewing the AndroidManifest.xml file, we discovered an activity named `StringCompareAct`.

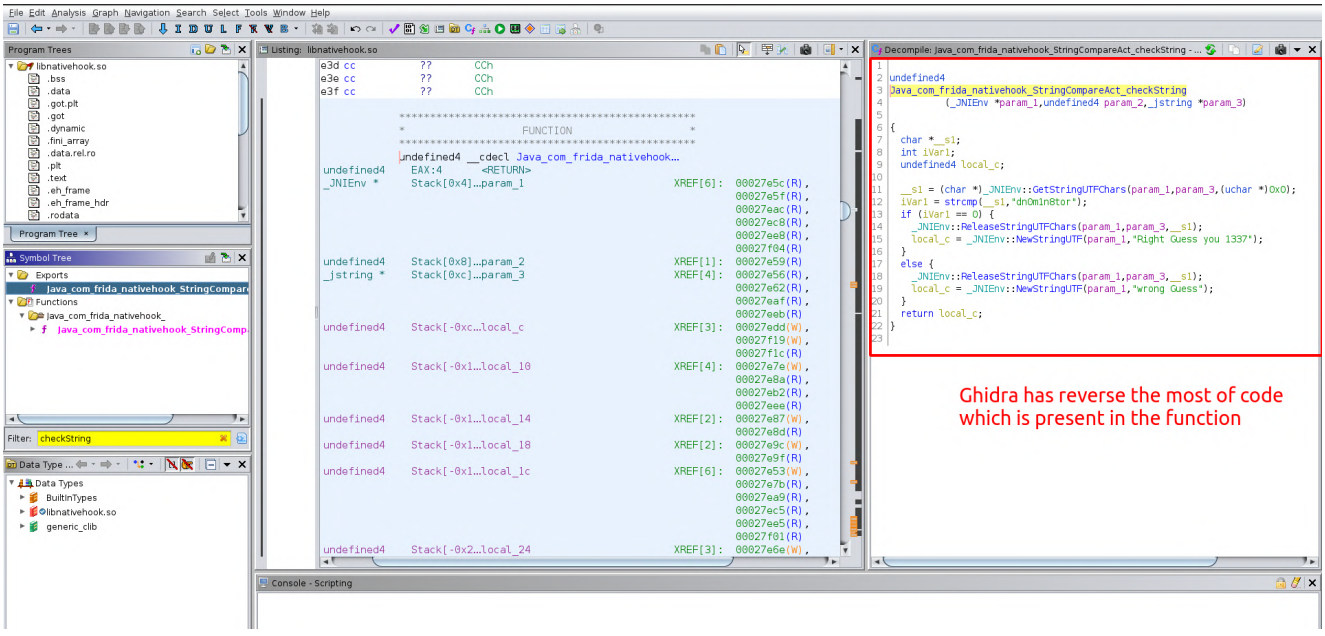
```
AndroidManifest.xml  MainActivity  StringCompareAct
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionCode="1" android:versionName="1.0"
android:compileSdkVersion="33" android:compileSdkVersionCodename="13" package="com.frida.nativehook" platformBuildVersionCode=
"33" platformBuildVersionName="13">
    <uses-sdk android:minSdkVersion="24" android:targetSdkVersion="32"/>
    <permission android:name="com.frida.nativehook.DYNAMIC_RECEIVER_NOT_EXPORTED_PERMISSION" android:protectionLevel=
"signature"/>
    <uses-permission android:name="com.frida.nativehook.DYNAMIC_RECEIVER_NOT_EXPORTED_PERMISSION"/>
    <application android:theme="@style/Theme.NativeHook" android:label="@string/app_name" android:icon="@mipmap/ic_launcher"
android:debuggable="true" android:testOnly="true" android:allowBackup="true" android:supportsRtl="true"
android:extractNativeLibs="false" android:fullBackupContent="@xml/backup_rules" android:roundIcon="@mipmap/ic_launcher_round"
android:appComponentFactory="androidx.core.app.CoreComponentFactory" android:dataExtractionRules="@xml/data_extraction_rules">
        <activity android:name="com.frida.nativehook.StringCompareAct" android:exported="false">
            <meta-data android:name="android.app.lib_name" android:value=""/>
        </activity>
        <activity android:name="com.frida.nativehook.MainActivity" android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
            <meta-data android:name="android.app.lib_name" android:value=""/>
        </activity>
        <provider android:name="androidx.startup.InitializationProvider" android:exported="false" android:authorities=
"com.frida.nativehook.androidx-startup">
            <meta-data android:name="androidx.emoji2.text.EmojiCompatInitializer" android:value="androidx.startup"/>
            <meta-data android:name="androidx.lifecycle.ProcessLifecycleInitializer" android:value="androidx.startup"/>
        </provider>
    </application>
</manifest>
```

After opening the activity, we can see that one native function is defined here. Therefore, we know we need to hook this.

```
package com.frida.nativehook;

import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;
import androidx.appcompat.app.AppCompatActivity;

/* loaded from: classes3.dex */
public class StringCompareAct extends AppCompatActivity {
    Button CheckButton;
    EditText inputBox2;
    TextView textView2;

    public native String checkString(String str);

    static {
        System.loadLibrary("nativehook");
    }
```

We will search this function in Ghidra and identify where it is defined.



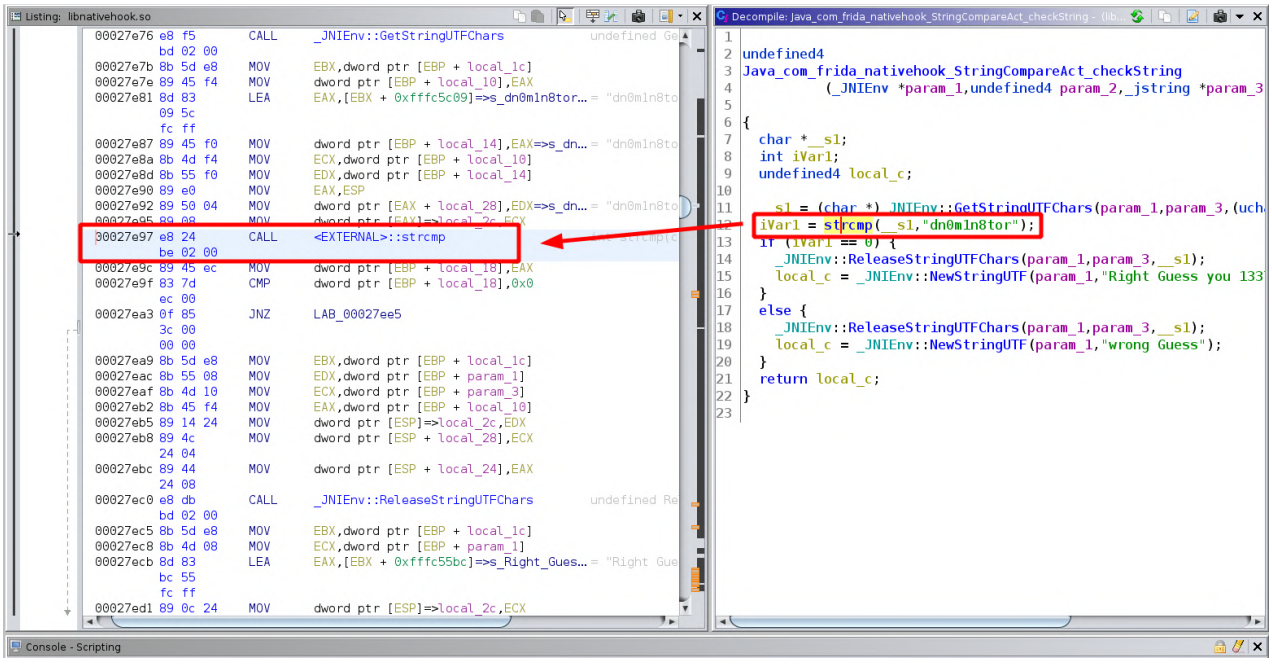


F. Now, upon analyzing the function, you will notice that there is an implementation of the `strcmp` function which compares the input string. However, at this point, Ghidra has provided us with the string necessary to solve this challenge. (Yet, we aim to solve it using Frida.)

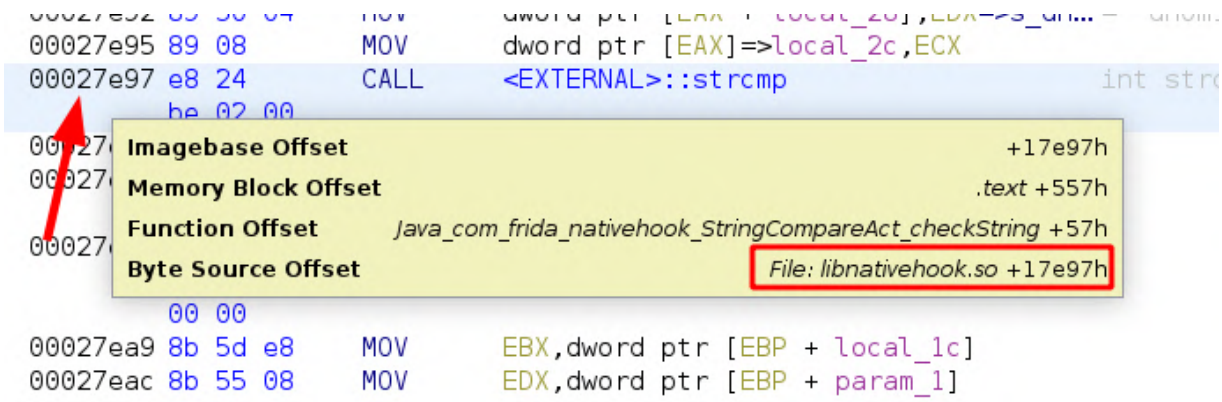
```
Decompile: Java_com_frida_nativehook_StringCompareAct_checkString - (libnativehook.so)
1
2  undefined4
3  Java_com_frida_nativehook_StringCompareAct_checkString
4            (_JNIEnv *param_1,undefined4 param_2,_jstring *param_3)
5
6  {
7    char *__s1;
8    int iVar1;
9    undefined4 local_c;
10
11     __s1 = (char *) _JNIEnv::GetStringUTFChars(param_1,param_3,(uchar *)0x0);
12   iVar1 = strcmp(__s1,"dn0m1n8tor");
13   if (iVar1 == 0) {
14     _JNIEnv::ReleaseStringUTFChars(param_1,param_3,__s1);
15     local_c = _JNIEnv::NewStringUTF(param_1,"Right Guess you 1337");
16   }
17   else {
18     _JNIEnv::ReleaseStringUTFChars(param_1,param_3,__s1);
19     local_c = _JNIEnv::NewStringUTF(param_1,"wrong Guess");
20   }
21   return local_c;
22 }
23
```

But suppose there is a scenario where the second parameter is passed as a variable rather than as a string as shown above. In that case, we will not be able to retrieve the value directly.

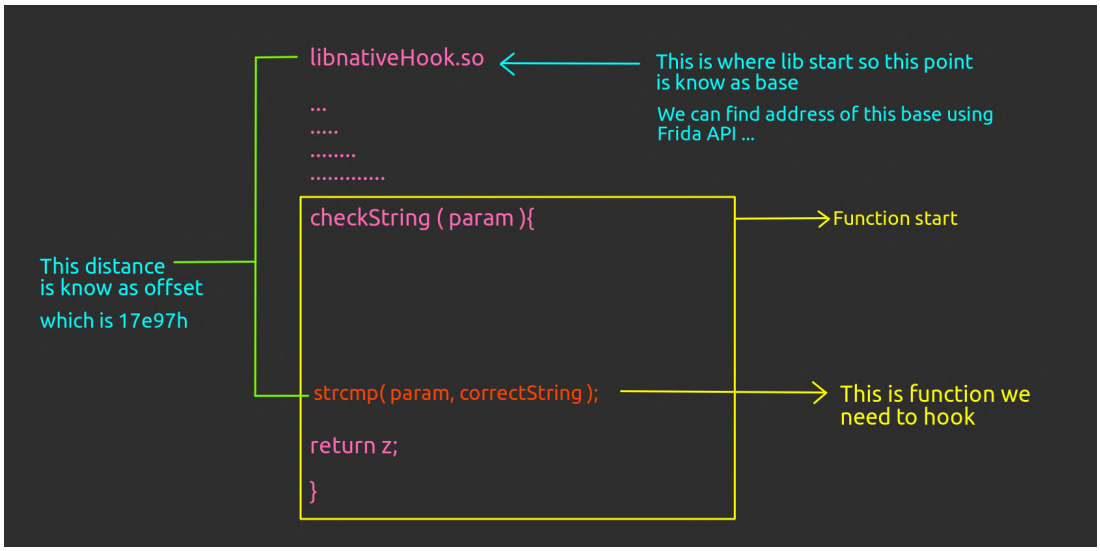
G. Click on the `strcmp` function



Click on the address "00027e97" to find the offset of that function, which is 17e97h.



The offset is simply the distance between where the library begins and where the declaration of the statement you're seeking to hook is located.



H. We need to find the baseAddress of libnativehook.so and add 17e97 to it to get the address of the strcmp function.

We can get baseAddress using the Frida API as Module.getBaseAddress("libnativehook.so") and can add 17e97 using .add in it

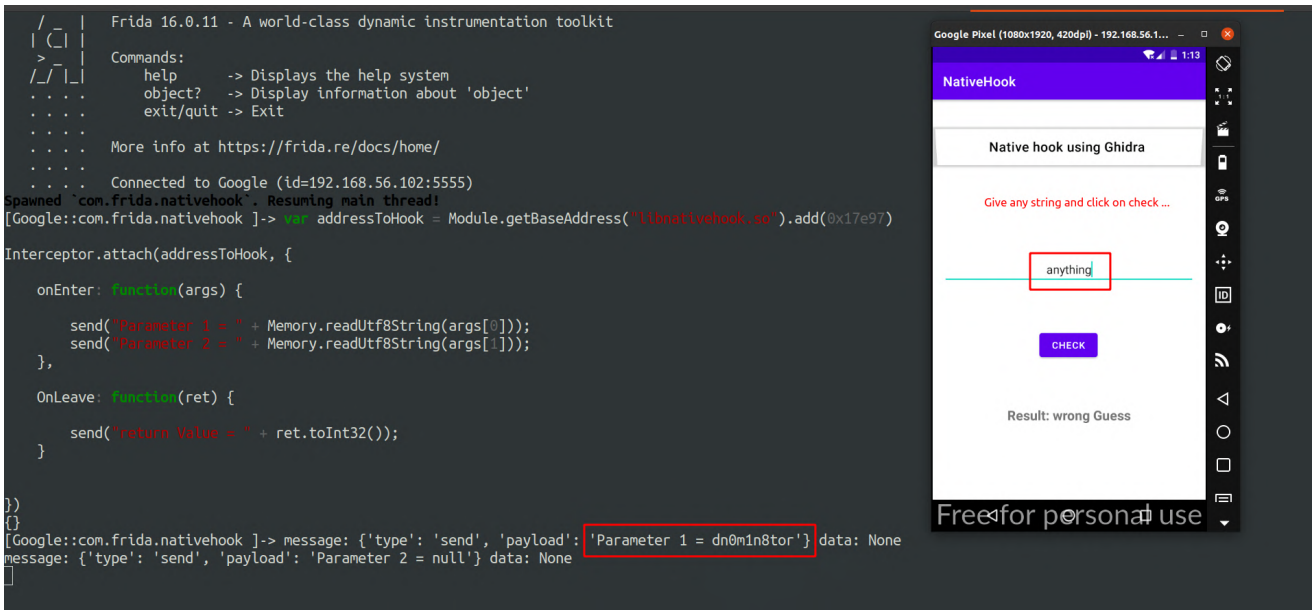
```
var addressToHook = Module.getBaseAddress("libnativehook.so").add(0x17e97)
```

Now that we have obtained the address, we can utilize `Interceptor.attach(address, call-backs)` to hook this function.

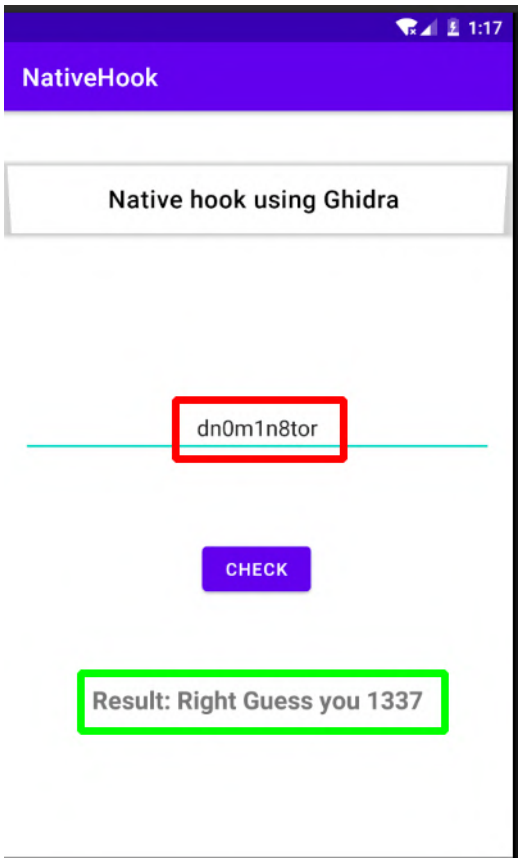
I. With the script below, we will hook the function.

```
            //Getting base address
            var addressToHook = Module.getBaseAddress("libnativehook.so").
    add(0x17e97)

            //Hooking into the function
            Interceptor.attach(addressToHook, {

              onEnter: function(args) {

                send("Parameter 1 = " + Memory.readUtf8String(args[0]));
                send("Parameter 2 = " + Memory.readUtf8String(args[1]));

                //Alternative to Memory.readUtf8String(args[0]) we can use Memory.
    readCString(args[0]) result will be same
                //send("Parameter 1 = " + Memory.readCString(args[0]));
                //send("Parameter 2 = " + Memory.readCString(args[1]));

              },

              OnLeave: function(ret) {

                send("return Value = " + ret.toInt32());
              }

            })
```

J. Executing the script against the application.



K. As you can see in the above output, we received the string `dn0m1n8tor`, which is being compared against our input. Let's provide this input to the application and see if we can solve this challenge or not.



We were able to guess the correct alias.

I hope you've grasped the concept of hooking any function with the help of Ghidra. You just need to follow simple steps.

1. Obtain the base address of the library using the Frida API getBaseAddress
2. Determine the offset of the statement to which we want to hook using Ghidra.
3. Add the offset to the base address.
4. Supply that address to the Interceptor.attach API and draft the callback according to your needs.

It's that simple.

About Payatu

Payatu is a Research-powered cybersecurity services and training company specialized in IoT, Embedded Web, Mobile, Cloud, & Infrastructure security assessments with a proven track record of securing software, hardware and infrastructure for customers across 20+ countries.

Mobile Security Testing

Detect complex vulnerabilities & security loopholes. Guard your mobile application and user's data against cyberattacks, by having Payatu test the security of your mobile application.

IoT Security Testing

IoT product security assessment is a complete security audit of embedded systems, network services, applications and firmware. Payatu uses its expertise in this domain to detect complex vulnerabilities & security loopholes to guard your IoT products against cyberattacks.

Cloud Security Assessment

As long as cloud servers live on, the need to protect them will not diminish. Both cloud providers and users have a shared. As long as cloud servers live on, the need to protect them will not diminish.

Both cloud providers and users have a shared responsibility to secure the information stored in their cloud Payatu's expertise in cloud protection helps you with the same. Its layered security review enables you to mitigate this by building scalable and secure applications & identifying potential vulnerabilities in your cloud environment.

Payatu

Web Security Testing

Internet attackers are everywhere. Sometimes they are evident. Many times, they are undetectable. Their motive is to attack web applications every day, stealing personal information and user data. With Payatu, you can spot complex vulnerabilities that are easy to miss and guard your website and user's data against cyberattacks.

DevSecOps Consulting

DevSecOps is DevOps done the right way. With security compromises and data breaches happening left, right & center, making security an integral part of the development workflow is more important than ever. With Payatu, you get an insight to security measures that can be taken in integration with the CI/CD pipeline to increase the visibility of security threats.

Code Review

Payatu's Secure Code Review includes inspecting, scanning and evaluating source code for defects and weaknesses. It includes the best secure coding practices that apply security consideration and defend the software from attacks.

Red Team Assessment

Red Team Assessment is a goal-directed, multidimensional & malicious threat emulation. Payatu uses offensive tactics, techniques, and procedures to access an organization's crown jewels and test its readiness to detect and withstand a targeted attack.

Product Security

Save time while still delivering a secure end-product with Payatu. Make sure that each component maintains a uniform level of security so that all the components "fit" together in your mega-product.

Critical Infrastructure Assessment

There are various security threats focusing on Critical Infrastructures like Oil and Gas, Chemical Plants, Pharmaceuticals, Electrical Grids, Manufacturing Plants, Transportation systems etc. and can significantly impact your production operations. With Payatu's OT security expertise you can get a thorough ICS Maturity, Risk and Compliance Assessment done to protect your critical infrastructure.

Cyber Threat Intelligence

The area of expertise in the wide arena of cybersecurity that is focused on collecting and analyzing the existing and potential threats is known as Cyber Threat Intelligence or CTI. Clients can benefit from Payatu's CTI by getting – social media monitoring, repository monitoring, darkweb monitoring, mobile app monitoring, domain monitoring, and document sharing platform monitoring done for their brand.

More Services Offered

- AI/ML Security Audit
- Trainings

More Products Offered

- EXPLIoT
- EXPLIoT Academy

Payatu

Payatu Security Consulting Pvt. Ltd.

www.payatu.com

info@payatu.com

+91 20 41207726