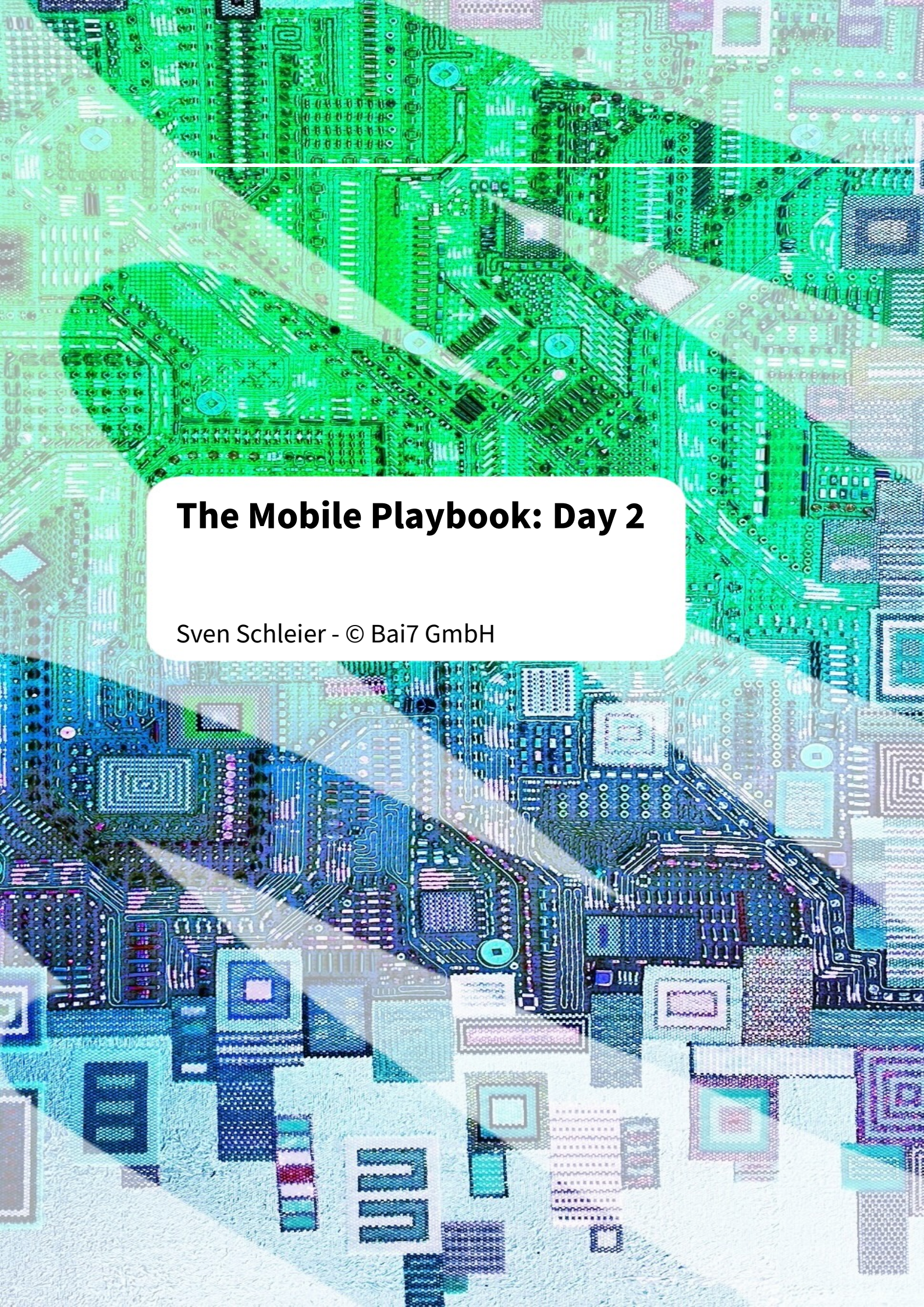


The Mobile Playbook: Day 2

Sven Schleier - © Bai7 GmbH

Contents

Lab - Fork and setup Repo

Time to finish lab	5 minutes
App used for this exercise	None

Training Objectives

In the following exercise we will fork a Github repository that we will use for some of the Android exercises.

Preparation

Github

- Fork the following repo: https://github.com/bai7-at/mobile-playbook-dev-ios

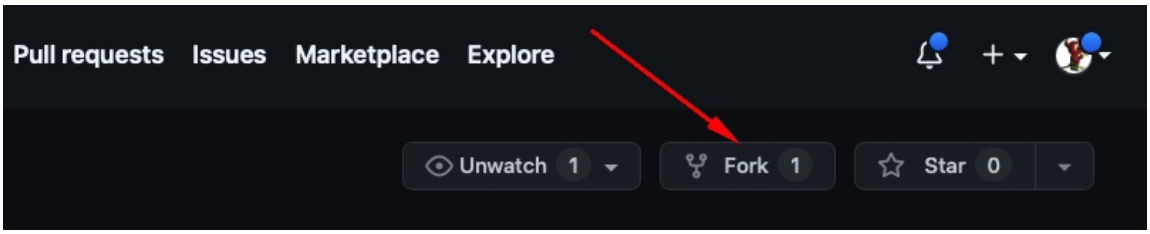


Figure 1: Fork repo

- In the next screen simply click “Create fork” and it will be forked to your Github account.
- Go to “Actions” in your newly forked repo and click on the green button to enable Github workflows.

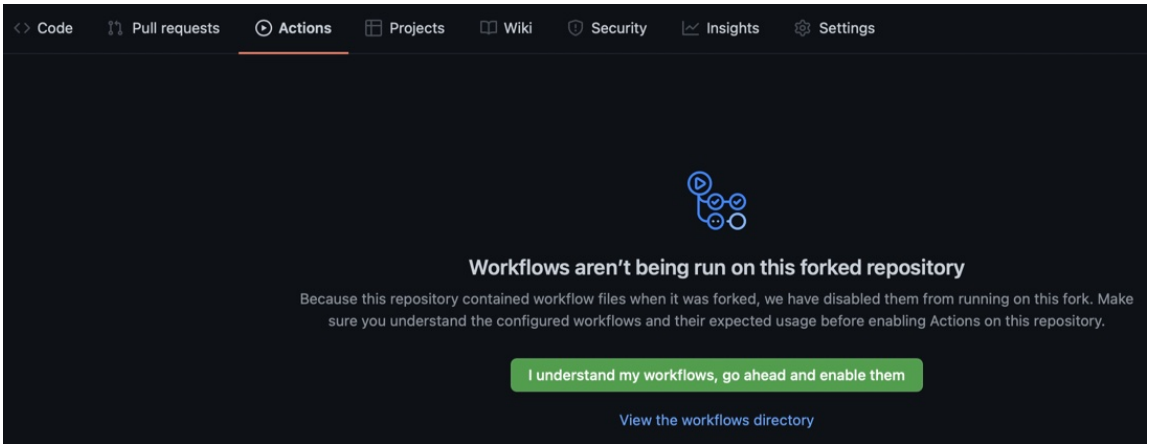


Figure 2: Enable Workflows

- Enable “Issues” in your forked repo. You can find it in the “Settings” tab under “Features”.

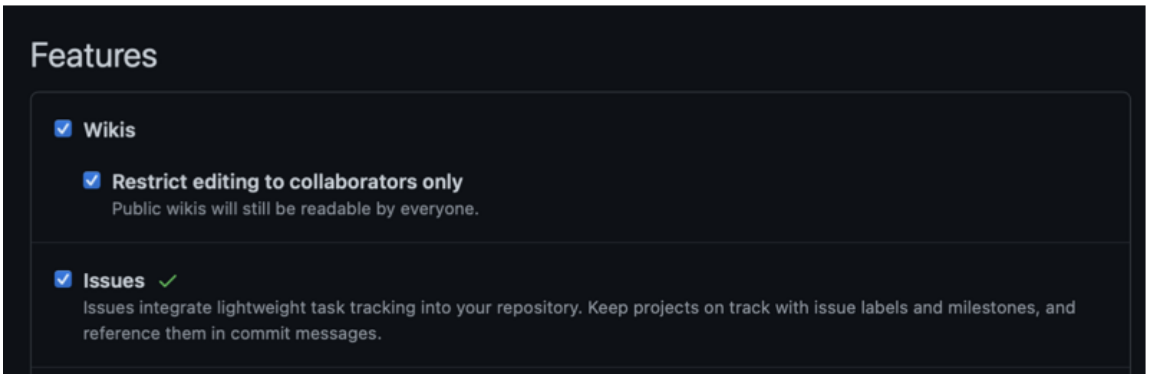


Figure 3: Enable Issues

Congratulations, you forked the repo! Later we will be activating some Github Actions to execute automated security checks.

Lab - Static Scanning with mobsfscan

Time to finish lab	20 minutes

Training Objectives

In this lab we will be scanning the source code of an iOS app for vulnerabilities and misconfiguration with `mobsfscan`.

Tools used in this section

- `mobsfscan` - https://github.com/MobSF/mobsfscan

Preparation

- Go to your forked repo and to the main directory in your browser. Select the `.github/workflows` folder and the file `03-mobsfscan.yml`.
- In the next screen press either the key “e” on your keybord or press the edit button top right. In the editor mode uncomment everything by selecting the whole text and pressing “Command” + “/” (on macOS).

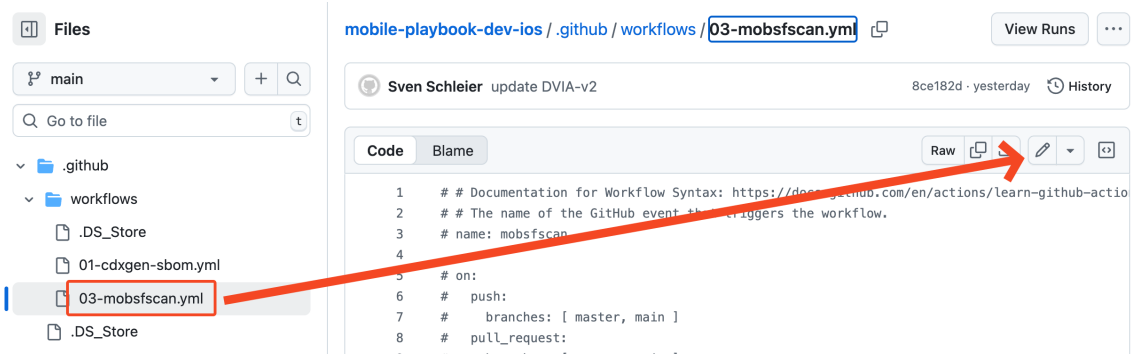


Figure 4: Edit mobsfscan workflow

- Commit the changes directly into the main branch.

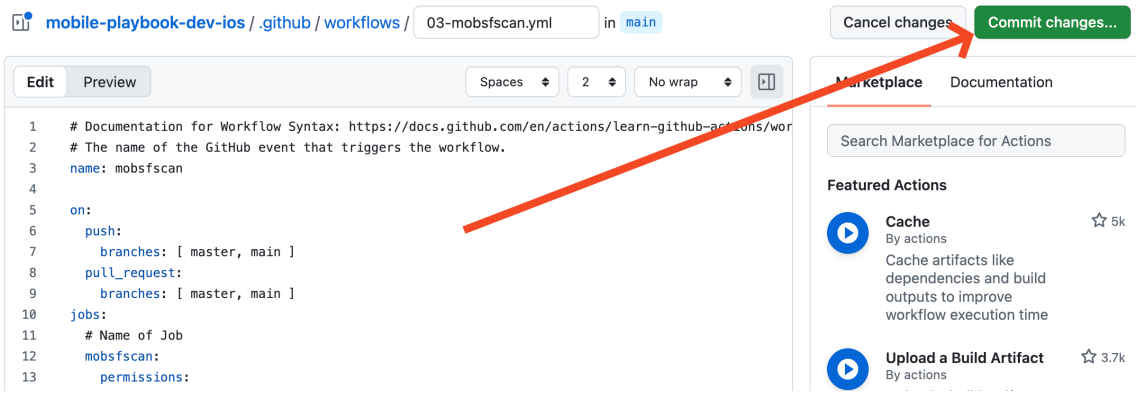


Figure 5: Commit changes

Github Action

- Click on “Actions” and you will see the workflows that were just triggered due to the commit. Click on the `mobsfscan` worflow run. The scan might still be running.

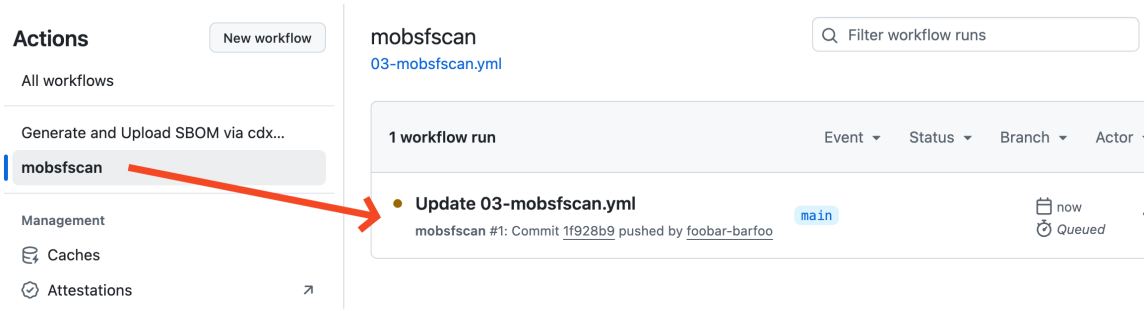


Figure 6: mobsfscan Workflow

- Once the mobsfscan job is done you will have identified new vulnerabilities that were added to the “Security” tab, click on it.

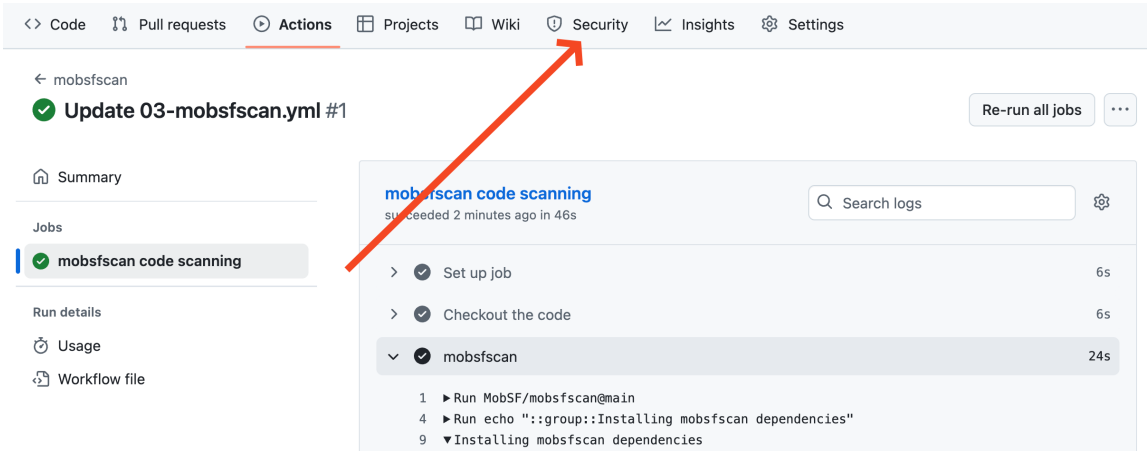


Figure 7: Security Tab

- The `mobsfscan` Github Action will now be executed every time we are doing a commit or pull request and will raise now an alert if sensitive information is being detected. But detecting is only half the job, as we need to manage the amount of detected vulnerabilities in an efficient way. Luckily this was already being taken care of automatically through our Github action and the SARIF file that was being created and imported into Github Code Scanning. This is the relevant snippet from the `mobsfscan.yml` file:

```
    # Upload the SARIF file to Github, so the findings show up in "
       Security / Code Scanning alerts"
    - name: Upload mobsfscan report
      uses: github/codeql-action/upload-sarif@v1
      with:
        sarif_file: results.sarif
```

- Go to "Code Scanning Alerts". This allows us to manage the identified vulnerabilities. Github takes care of de-duplication of findings automatically through fingerprinting, so Github will not flag out the same finding again in consecutive scans.

We have a large number of findings and to start with we should focus on specific rules from `mobsfscan` and start with `ios_hardcoded_secret`:

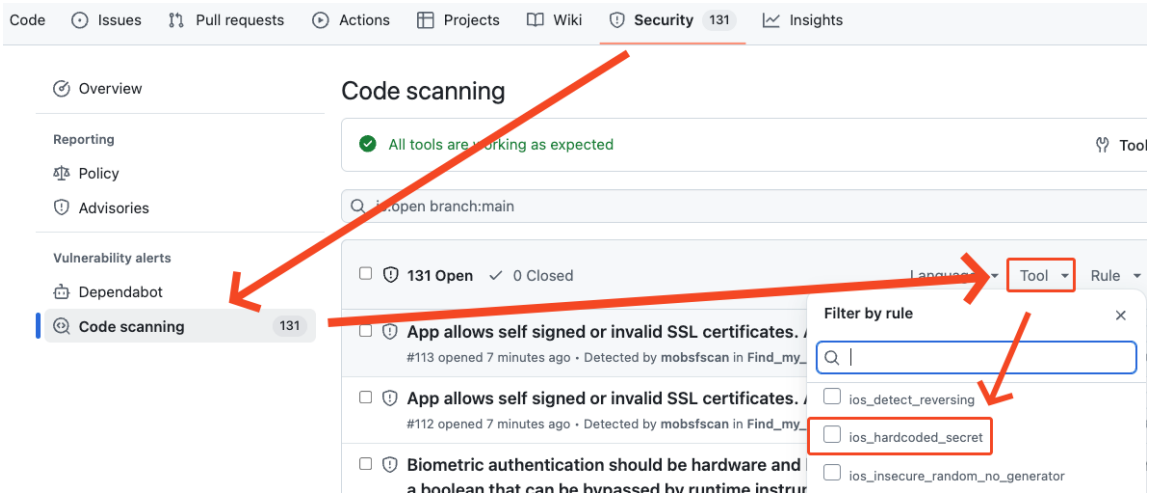


Figure 8: Findings for specific rule

- Now the work starts! Your job is to triage the identified findings, go through them one by one and decide if they are either a:
  - valid finding (then create an issue),
  - false positive,
  - used in tests, or
  - if you won't fix it (and why).

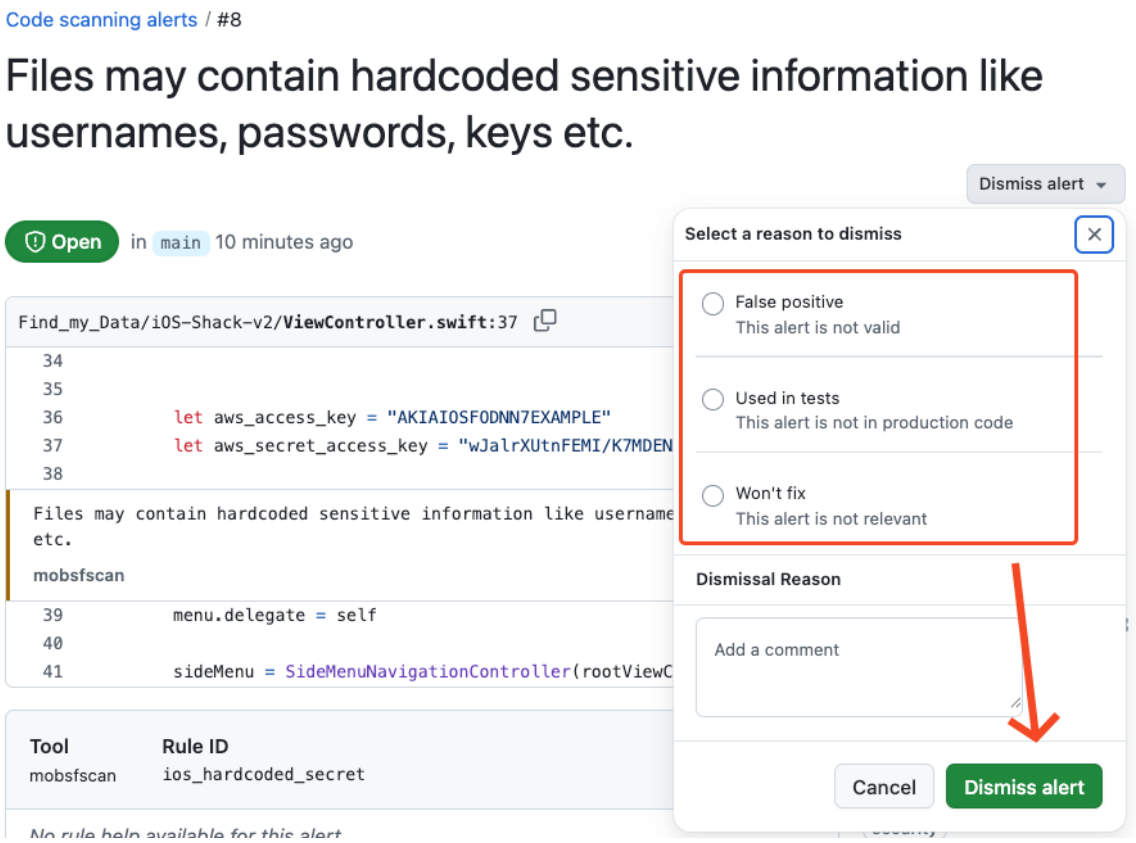


Figure 9: Triage findings

> You can exclude for now the rules `ios_log` and `ios_app_logging` as they have almost 90 findings.

Go through each finding and make a decision! Once done with the first rule, choose another one.

> In case a finding is identified but no code is highlighted, this means the absence of code patterns might be a vulnerability.

Congratulations, you are now able to scan vulnerabilities with `mobsfscan` and detect and triage vulnerabilities and manage code scanning alerts with Github Code Scanning!

Appendix

Github Actions are defined in Yaml files and have a very simple structure and are self explanatory; you might want to revisit the Gitleaks yaml file after you've done the lab to understand the flow in more depth:

```
# Documentation for Workflow Syntax: https://docs.github.com/en/
   actions/learn-github-actions/workflow-syntax-for-github-actions
```

```yaml
# The name of the GitHub event that triggers the workflow.
name: mobsfscan

on:
  push:
    branches: [ master, main ]
  pull_request:
    branches: [ master, main ]
jobs:
  # Name of Job
  mobsfscan:
    permissions:
       id-token: write
       contents: read
       security-events: write
    runs-on: ubuntu-latest
    name: mobsfscan code scanning
    steps:
    # This action checks-out your repository under
       $GITHUB_WORKSPACE, so your workflow can access it.
    - name: Checkout the code
      uses: actions/checkout@v2
    # Execute the MobSF scan and create a SARIF file
    - name: mobsfscan
      uses: MobSF/mobsfscan@main
      with:
        args: 'Find_my_Data --sarif --output results.sarif || true'
    # Upload the SARIF file to Github, so the findings show up in "
       Security / Code Scanning alerts"
    - name: Upload mobsfscan report
      uses: github/codeql-action/upload-sarif@v2
      with:
        sarif_file: results.sarif
```

Lab - Frida 101 (Frida-Server)

Time to finish lab	20 minutes
App used for this exercise	Find_my_Data.ipa

Training Objectives

1. Learn how to use the Frida Server
2. Use a Frida script

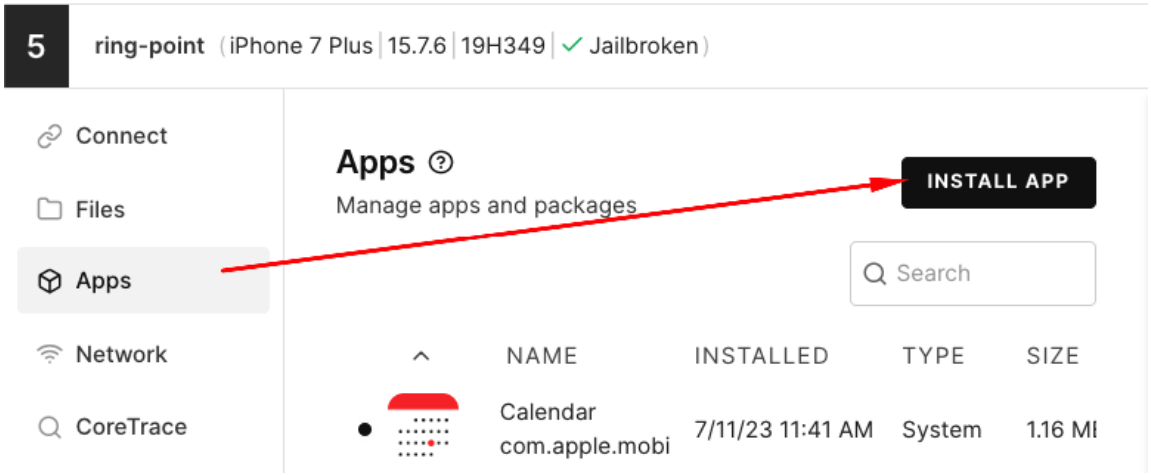
Tools used in this section

- Frida - https://www.frida.re/

Exercise - Frida Usage

Preparation

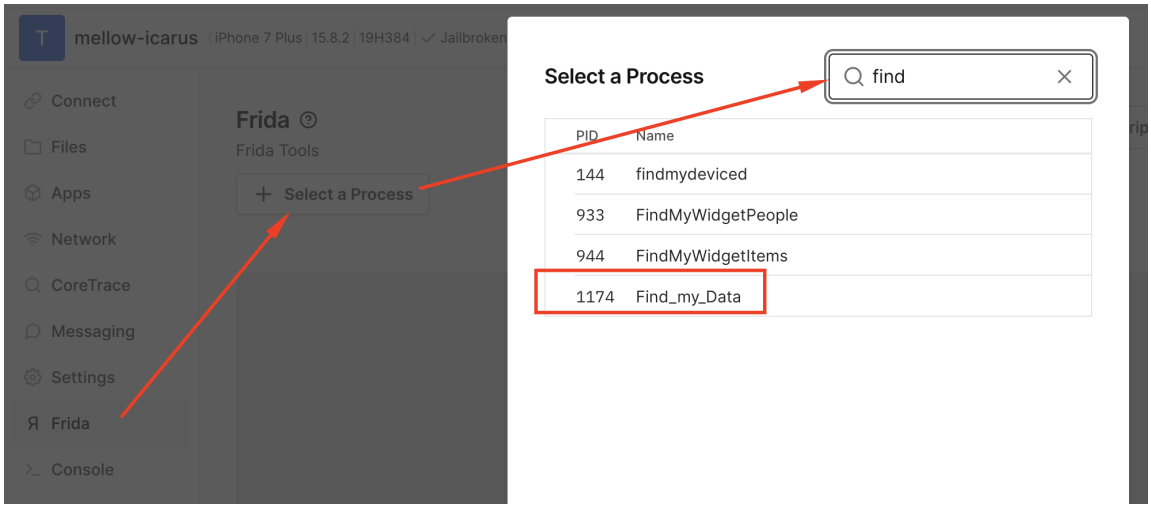
- Go to the “Apps” menu in Corellium and install the app `Find_my_Data.ipa` by selecting “Install App”. You downloaded the IPA as part of the preparation pack today, it’s in the “Apps” folder. This might take a minute to install.



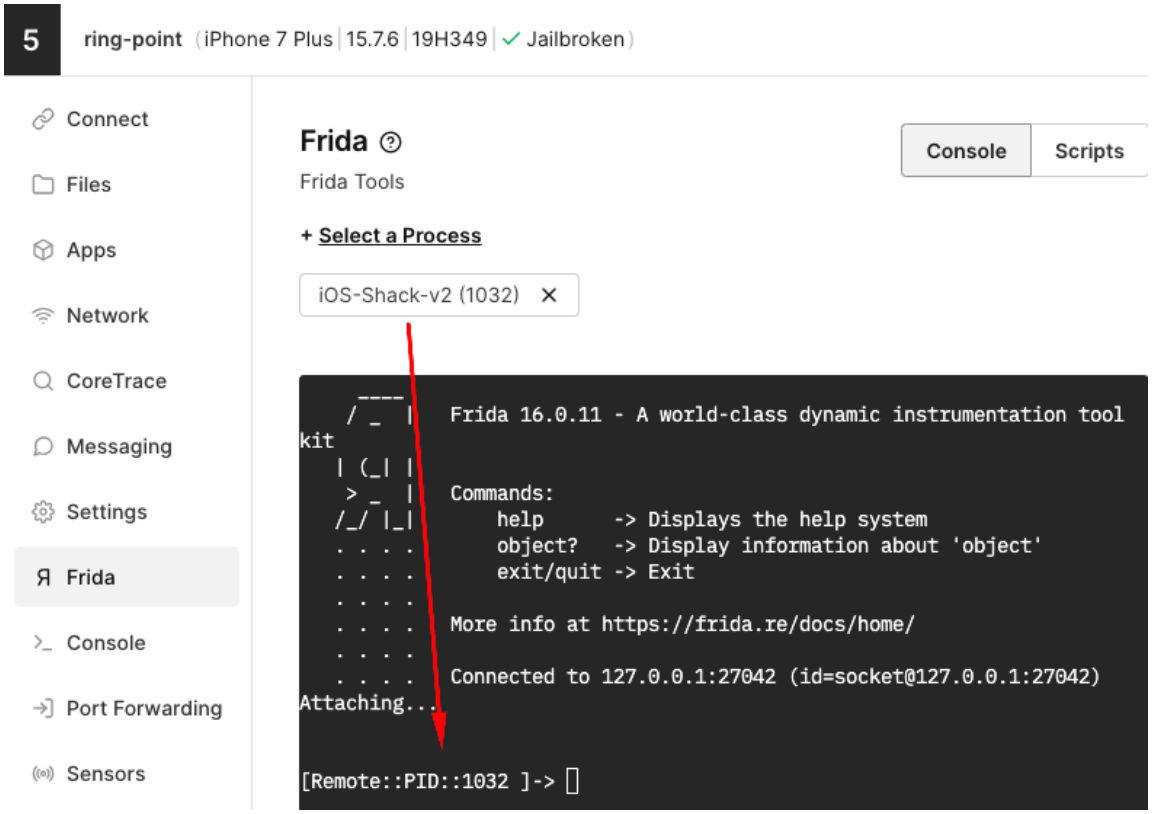
- Run the app “Find_my_data”:

Frida

- Start the Frida Server by clicking on the Frida tab on the left in Corellium. Click on “Select a Process” and search for “find” and you will find the running app (your process id, the PID, will be different in your case):



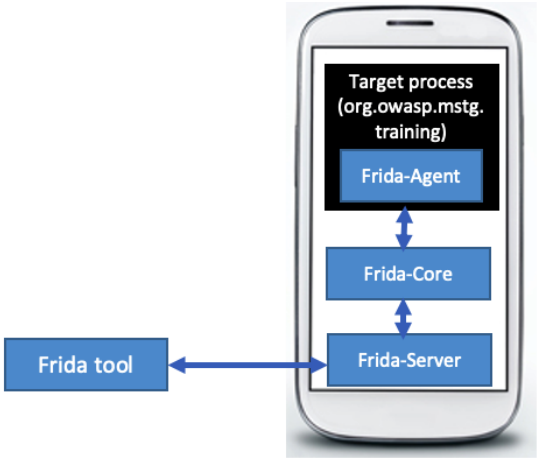
- Once selected, attach to the process and make sure that the app is running in the foreground on the iPhone.



- You can see now the Frida console.

```
     ____
    / _  |   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 127.0.0.1:27042 (id=socket@127
      .0.0.1:27042)
Attaching...
[Remote::PID::1032 ]->
```

Frida CLI (console) is now connecting to the Frida-Server running on the iOS instance and the Frida Core Framework is injecting the Frida-Agent into the specified identifier (`iOS-Shack-v2`).



So what can we do now with the interactive Frida console? You can write commands to Frida using it's ObjC API, just press Tab in the CLI to see the available commands. For example you can execute `ObjC.available`, which returns a boolean value specifying whether the current process has an Objective-C runtime loaded.

```
[Remote::PID::2194 ]-> ObjC.available
true
```

We can query for information of the app (e.g. class and method names) and the Frida CLI allows you to do rapid prototyping and also debugging. With the following command we can query for the bundle-id of the app:

```
[Remote::Find_my_Data]-> ObjC.classes.NSBundle.mainBundle().
   bundleIdentifier().toString()
"info.s7ven.ios.data"
```

> The Frida CLI is case-sensitive. So in case you are facing any errors this might be the issue.

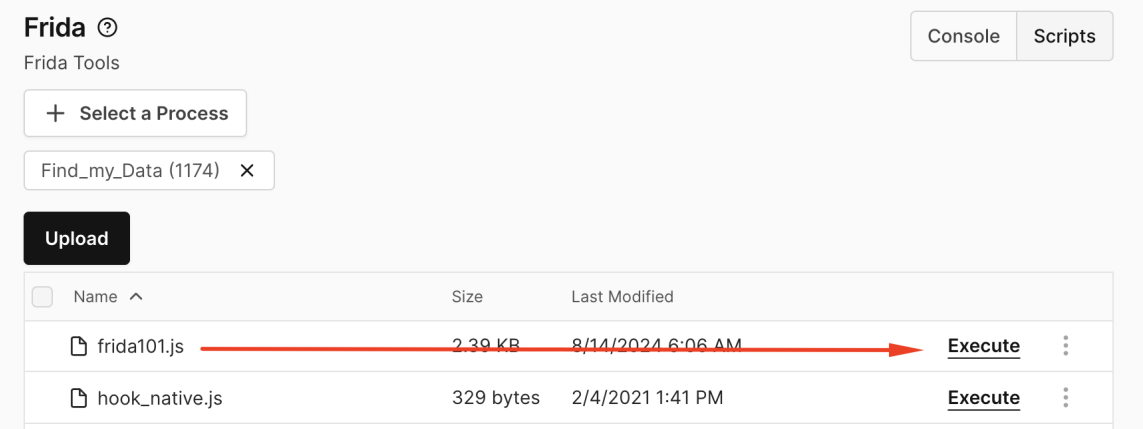
Frida Scripts

Besides using the CLI of `frida`, we can also load scripts. This will load pre-defined JavaScript commands as script that Frida will execute in the context of the targeted app.

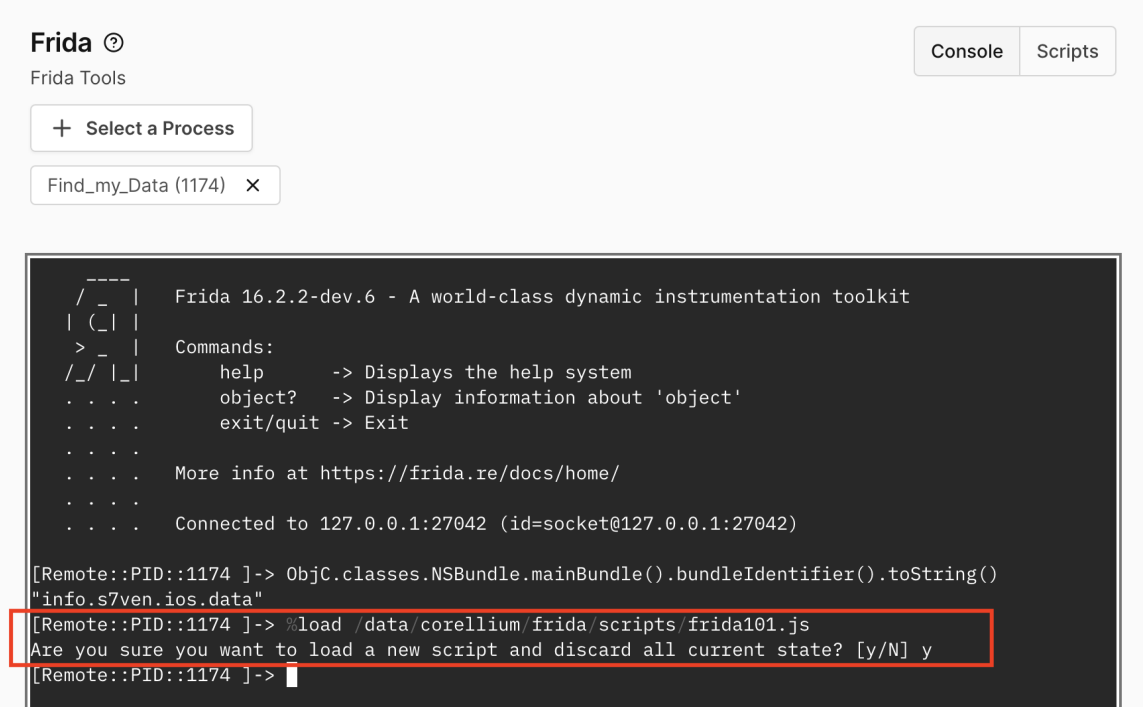
You can find the `Frida-Scripts` directory in the directory where you unzipped the preparation file.

Click on the top right on "Scripts" and upload the script `frida101.js`:

Execute the script:



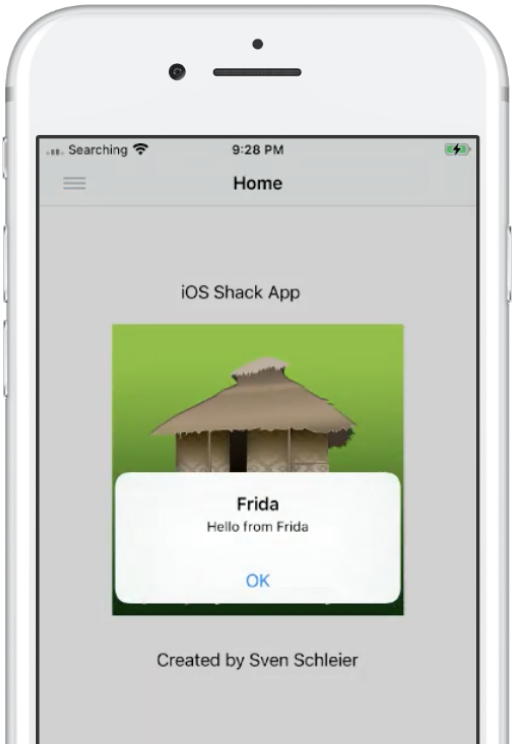
Go back to the console and type in `y` and confirm.



Let's call the function `helloWorld()` in the Frida console:

```
[Remote::PID::1032]-> helloWorld()
```

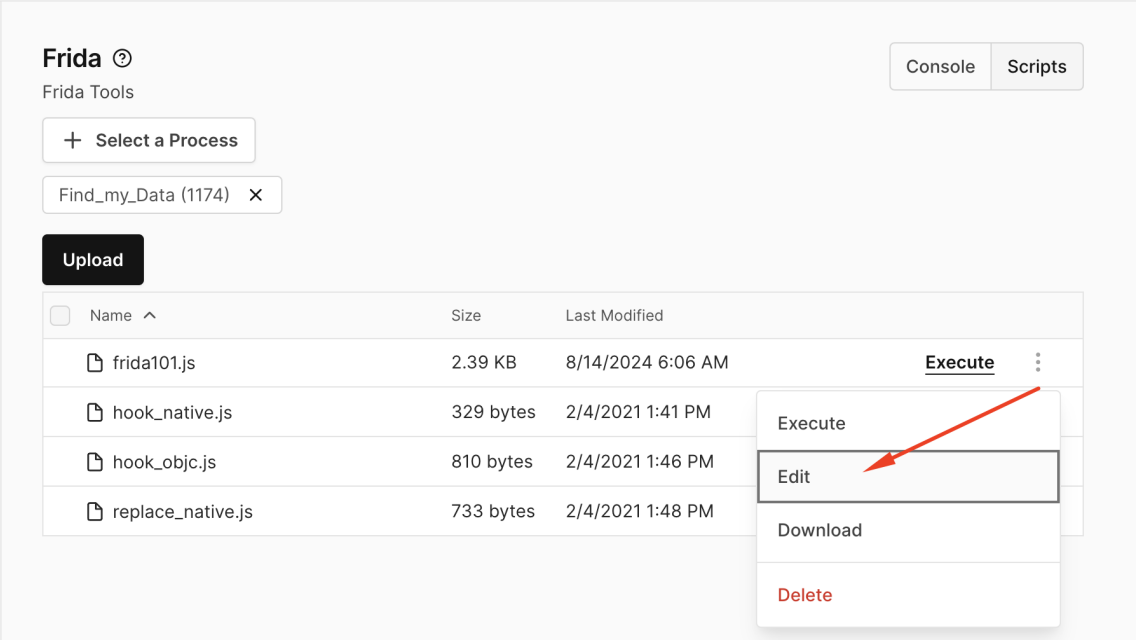
This function is implemented in the Frida script that we just loaded. You will not see any output in the Frida console, but look at your phone!



Through Frida we are in full control of the app and can modify the app during runtime in any way we want.

Modify a Frida Script

Edit the file `Frida-Scripts/frida101.js` in Corellium. You can see in the script the function `helloWorld()` that we just called.



In the Frida-Scripts folder in the repo that you cloned earlier today, open the file `extension.js` in an editor of your choice. You will see two functions. Copy both functions from `extension.js` into the `frida101.js` file, before the `helloWorld()` function in Line 5. Save the script.

```
1  /*  https://frida.re/docs/examples/ios/
2   *  Sent a Hello World to your app:
3   *      helloWorld()
4   */
5
6  function appInfo() {
7    var output = {};
8    output["Name"] = infoLookup("CFBundleName");
9    output["Bundle ID"] = ObjC.classes.NSBundle.mainBundle().bundleIdentifier().toString();
10   output["Version"] = infoLookup("CFBundleVersion");
11   output["Bundle"] = ObjC.classes.NSBundle.mainBundle().bundlePath().toString();
12   output["Data"] = ObjC.classes.NSProcessInfo.processInfo().environment().objectForKey_("HOME").toString(); output["Binary"]
         = ObjC.classes.NSBundle.mainBundle().executablePath().toString();
13   return output;
14 }
15
16 function infoLookup(key) {
17   if (ObjC.available && "NSBundle" in ObjC.classes) {
18   var info = ObjC.classes.NSBundle.mainBundle().infoDictionary(); var value = info.objectForKey_(key);
19   if (value === null) {
20   return value;
21   } else if (value.class().toString() === "__NSCFArray") {
22   return arrayFromNSArray(value);
23   } else if (value.class().toString() === "__NSCFDictionary") {
24   return dictFromNSDictionary(value); } else {
25   return value.toString(); }
26   }
27   return null;
28 }
29
30 function helloWorld()
```

SAVE CLOSE

This Frida script offers us now 3 different functions:

- `appInfo()` - Dump key app paths and metadata
- `infoLookup()` - Helper function for appInfo()
- `helloWorld()` - Print a Hello World message into an iOS app

Once you save the file, the script will automatically be reloaded in the Frida CLI, so you can call now `appInfo()`. Go back to the Frida console and give it a try!

```
[Remote::PID::1032 ]-> appInfo()
{
    "Binary": "/private/var/containers/Bundle/Application/D68ED06F
       -3746-4CC1-9F0C-F772CAAE16CE/Find_my_Data.app/Find_my_Data",
    "Bundle": "/private/var/containers/Bundle/Application/D68ED06F
       -3746-4CC1-9F0C-F772CAAE16CE/Find_my_Data.app",
    "Bundle ID": "info.s7ven.ios.data",
    "Data": "/private/var/mobile/Containers/Data/Application/
       C57CA8B9-D0CE-4DBC-AC25-2B77565391FB",
    "Name": "Find_my_Data",
    "Version": "1"
}
```

This will print a lot of useful meta-information for a penetration tester about the app, including the data and bundle paths.

Congratulations you are able to use the Frida server and can modify and load scripts!

References

- Install Frida on iOS - https://www.frida.re/docs/ios/
- Frida JavaScript API (Objective-C) - https://frida.re/docs/javascript-api/#objc

Lab - Sensitive Data in Local Storage

Time to finish lab	20 minutes
App used for this exercise	Find_my_Data.ipa

Training Objectives

Identify sensitive information stored in the iOS App

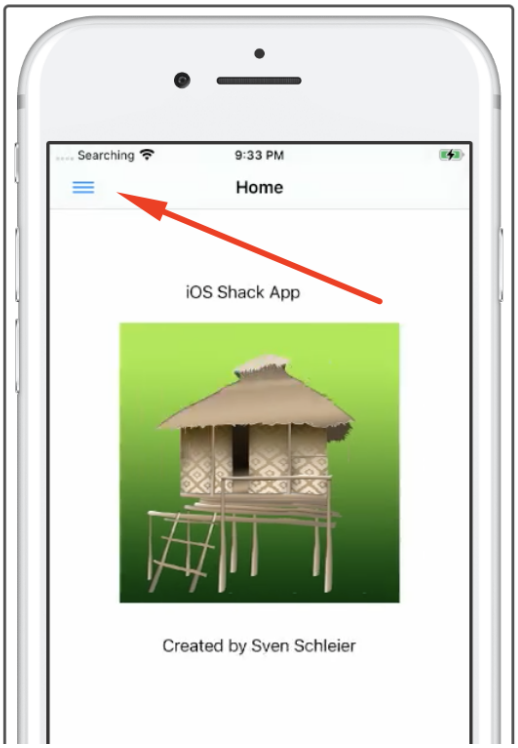
Tools used in this section

- Corellium

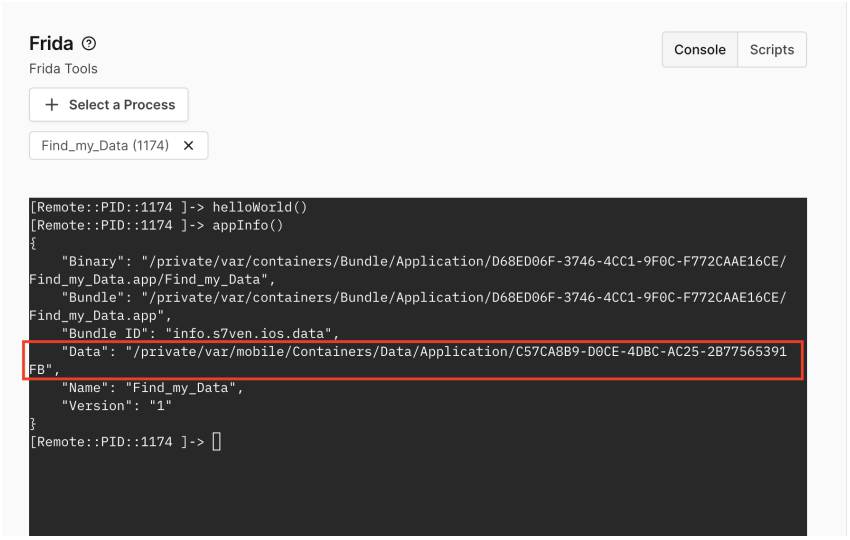
Exercise

Identify Sensitive Data

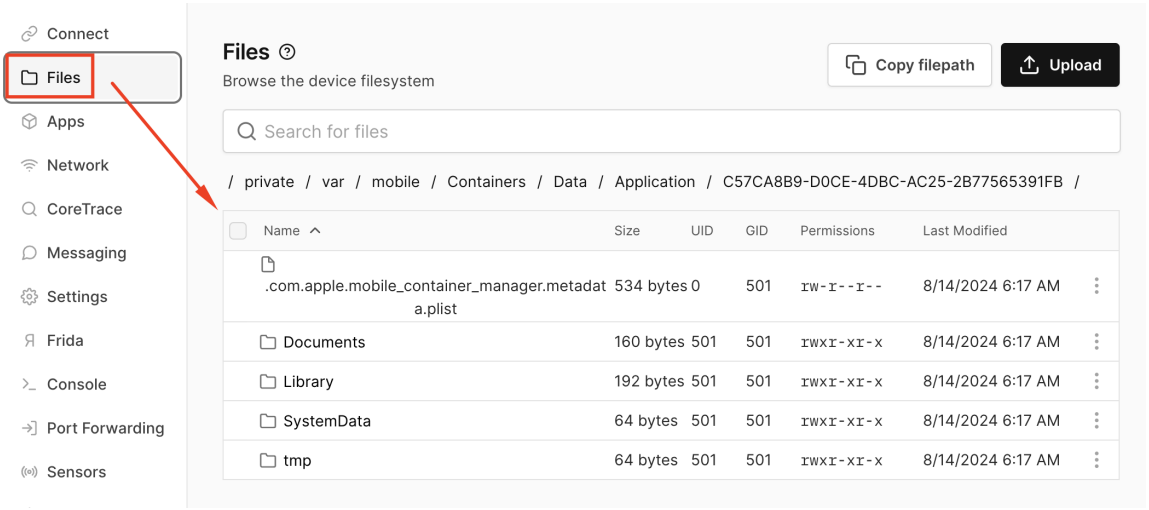
- Go to the app you installed in the previous lab and click on the menu button top left and on “Sensitive Data”. This will create files in the App’s sandbox.



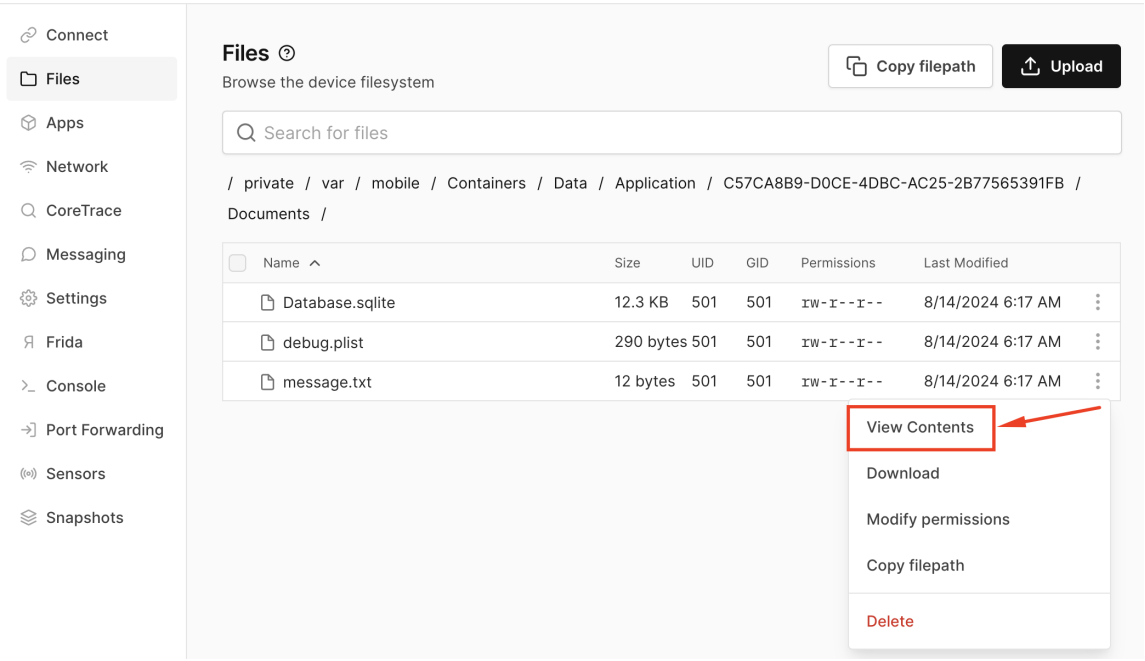
- From the previous lab you have all the information you need to access the apps’s sandbox. Check the output of `appinfo()` and search for the “Data” key, the value will the be the directory of this app on your device.



- Go to the files menu on the left in Corellium and navigate to the Data path of your app.



- Once you identified files in the apps' sandbox, you can view the content when clicking on the menu on the right:



> Your goal is to find all data created by the app and identify if they contain sensitive information or not.

If you find a sqlite database, download it and you can analyze it with Sqlitebrowser.

What sensitive information could you find? Do you need further protection mechanisms for the sensitive information? If so, how would you do it and implement it?

Lab - Runtime Manipulation

Time to finish lab	15 minutes
App used for this exercise	DVIA-v2.ipa

Training Objectives

1. Use existing Frida scripts
2. Bypass a client side security check in an iOS app.

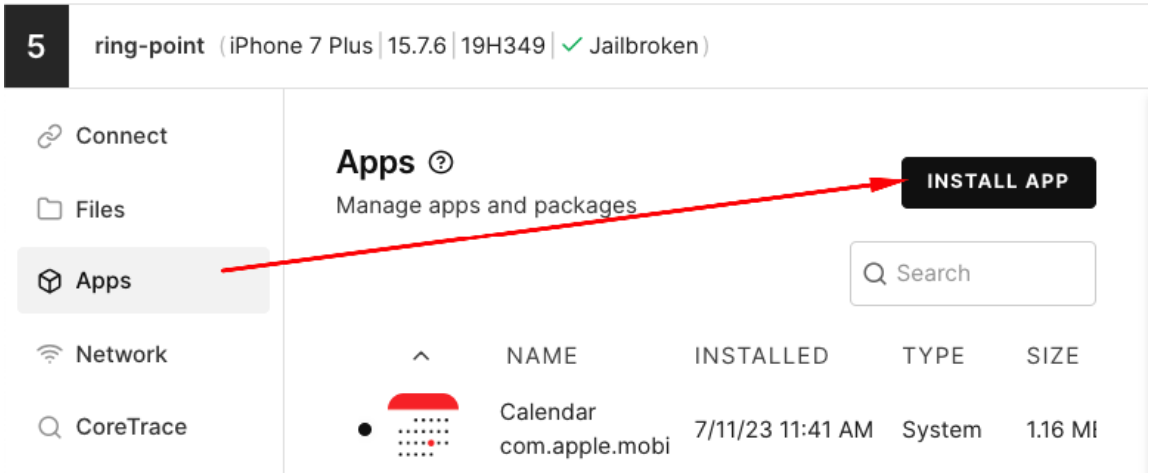
Tools used in this section

- Frida - https://www.frida.re/

Exercise

Preparation

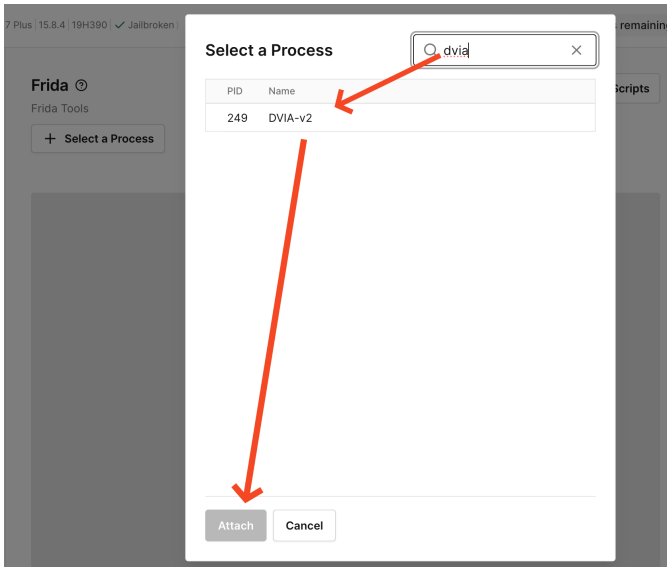
- Go to the “Apps” menu in Corellium and install the app `DVIA-v2.ipa` by selecting “Install App”. You downloaded the IPA as part of the preparation pack today, it’s in the “Apps” folder. This might take a minute.



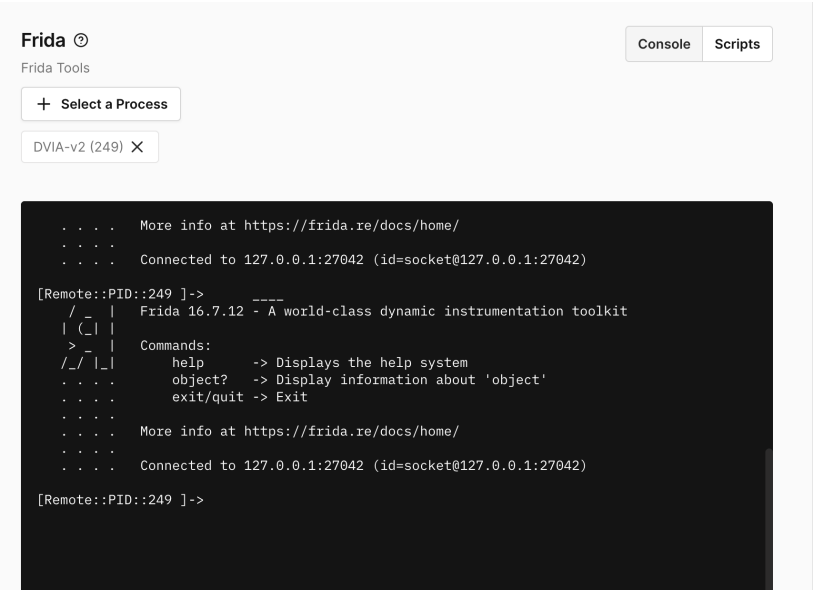
- Run the app “DVIA-v2”:

Frida

- Start the Frida Server by clicking on the Frida tab on the left in Corellium. Click on “Select a Process” and search for “dvia” and you will find the running app (your process id, the PID, will be different in your case).



- Once selected, attach to the process and make sure that the app is running in the foreground on the iPhone.

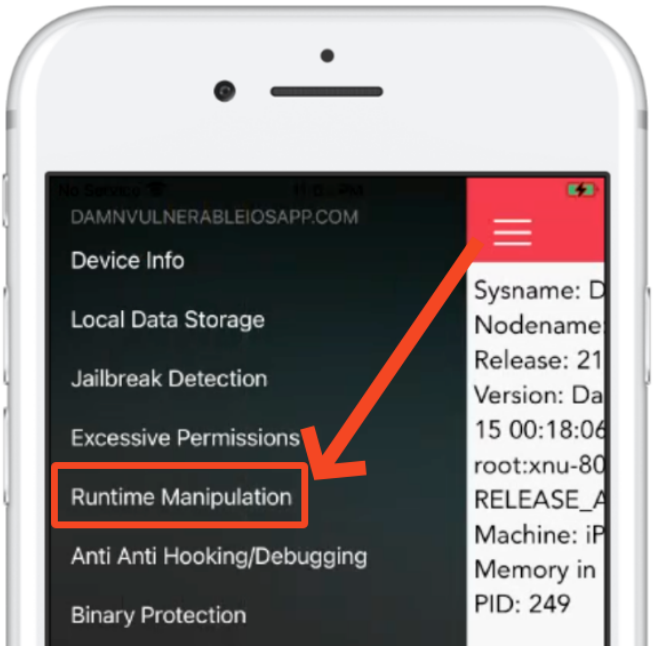


- You can see now the Frida console.

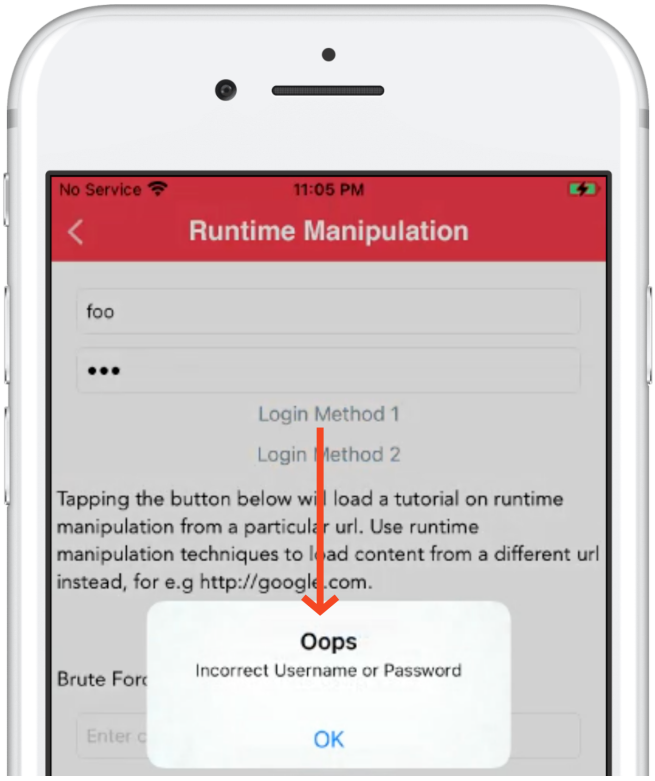
```
     ____
    / _  |   Frida 16.7.12 - A world-class dynamic instrumentation
       toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to 127.0.0.1:27042 (id=socket@127
      .0.0.1:27042)

[Remote::PID::249 ]->
```

Damn Vulnerable iOS App v2 (DVIA-v2)

- In the DVIA-v2 App, click on the top left corner on the menu and select `Runtime Manipulation`.



- Click on “Start challenge”, type in any string in the username and password and click on “Login Method 1” and you will get the following message.

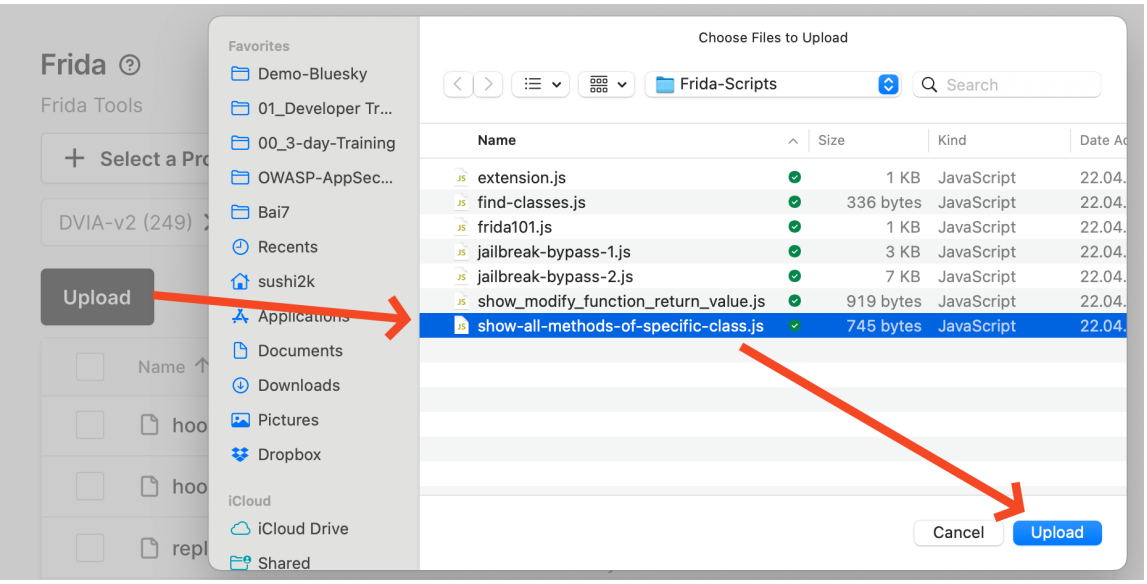


Our mission is to bypass this login check!

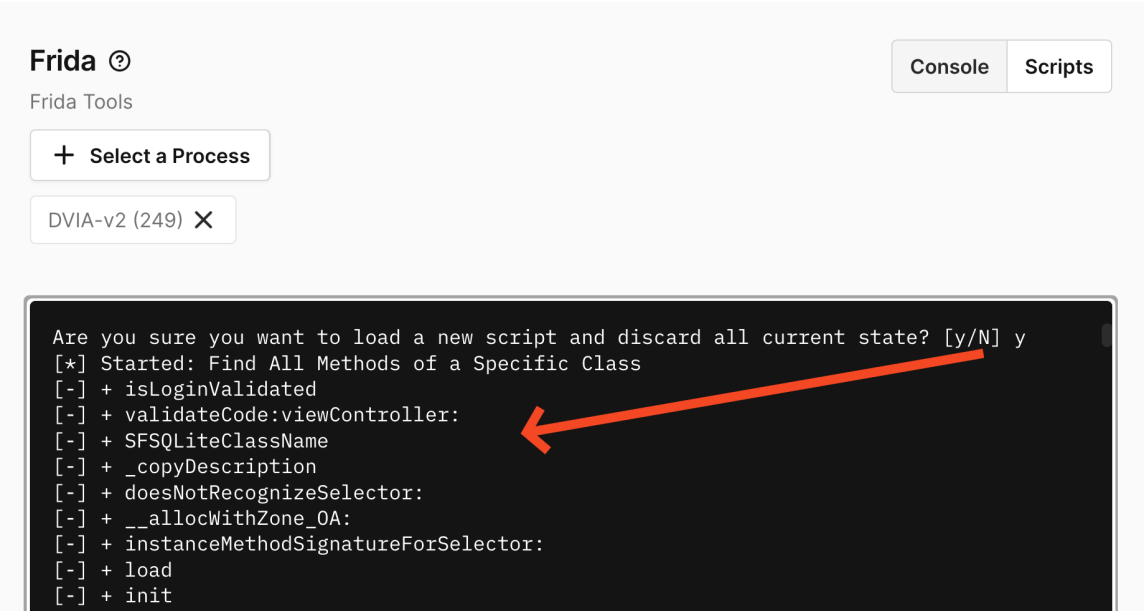
Frida CLI + Scripts

We already know that we can use Frida scripts to inject them into the target app.

- Upload the script `show-all-methods-of-specific-class.js`. This will list all methods of a class. The script already contains the class name `LoginValidate` and will show all methods of it.



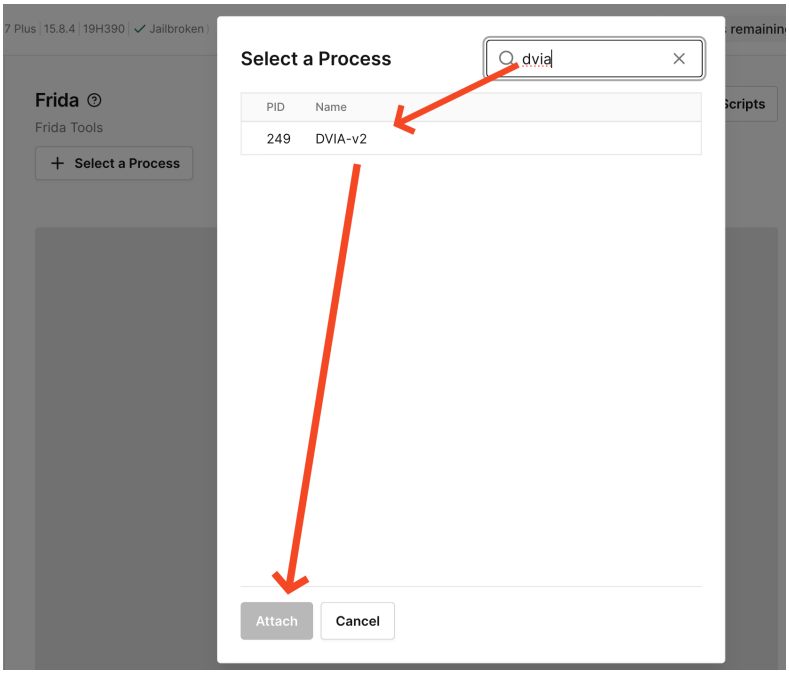
- Click on the button “Execute” next to the script you just uploaded.
- Switch to the console tab and confirm the script execution with “y” and press enter. This script will dump now all methods of the class we specified in the script and this might take a few seconds.



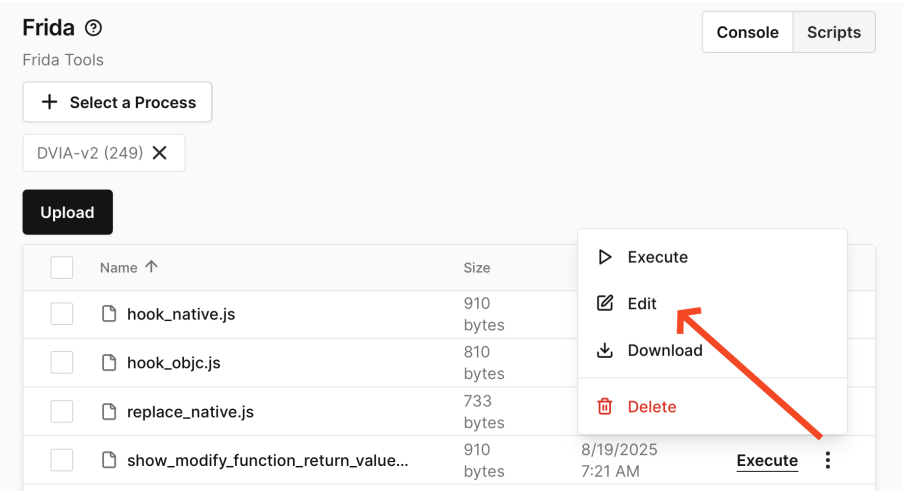
> Note: In the output you can find plus and minus signs in front of the methods. (+) stands for class method and (-) for instance method.

- When it’s finished search with “Ctrl+F” the output in the Frida console for the keyword “isLoginValidated”. You should find this method name.

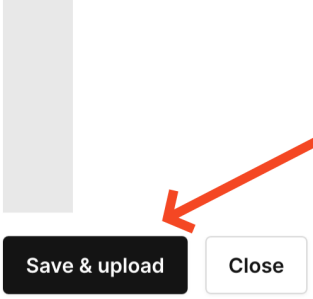
- To avoid that this script is now executed all the time, we will stop Frida on the DVIA-v2 app and re-attach it.



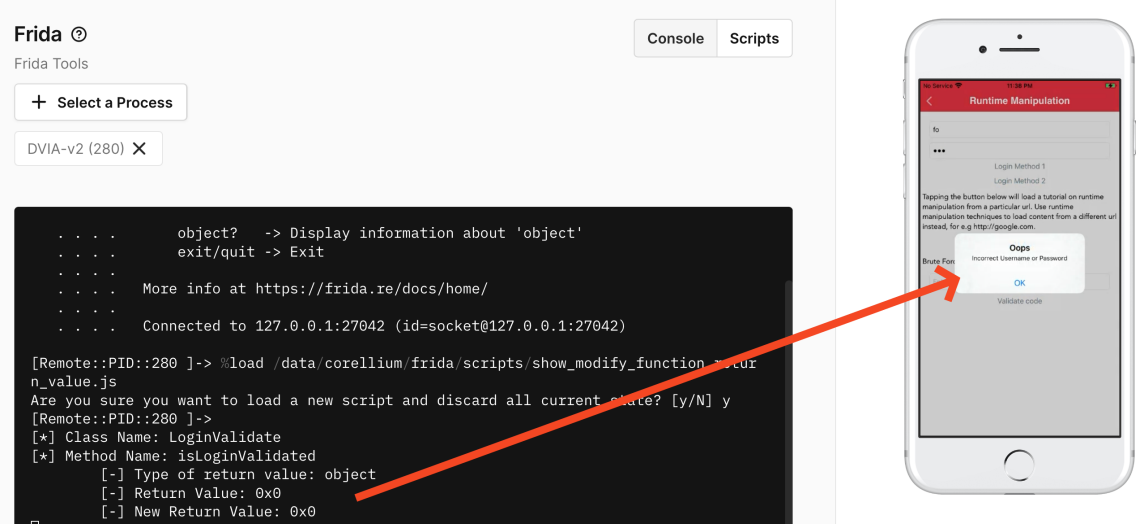
- Once you have identified the method upload the script `show-modify-function-return-value.js`. Edit the script by replacing the placeholder "YOUR_EXACT_FUNC_NAME_HERE" with the method you found in line 21 and save it.



```
1  function show_modify_function_return_value(className_arg, funcName_arg)
2  {
3      var className = className_arg;
4      var funcName = funcName_arg;
5      var hook = eval('ObjC.classes.' + className + '["' + funcName + '"]');
6      Interceptor.attach(hook.implementation, {
7        onLeave: function(retval) {
8          console.log("\n[*] Class Name: " + className);
9          console.log("[*] Method Name: " + funcName);
10         console.log("\t[-] Type of return value: " + typeof retval);
11         //console.log(retval.toString());
12         console.log("\t[-] Return Value: " + retval);
13         //For modifying the return value
14         var newretval = ptr("0x0") //your new return value here
15         retval.replace(newretval)
16         console.log("\t[-] New Return Value: " + newretval)
17       }
18     });
19 }
20 //YOUR_CLASS_NAME_HERE and YOUR_EXACT_FUNC_NAME_HERE
21 show_modify_function_return_value("LoginValidate","isLoginValidated")
```



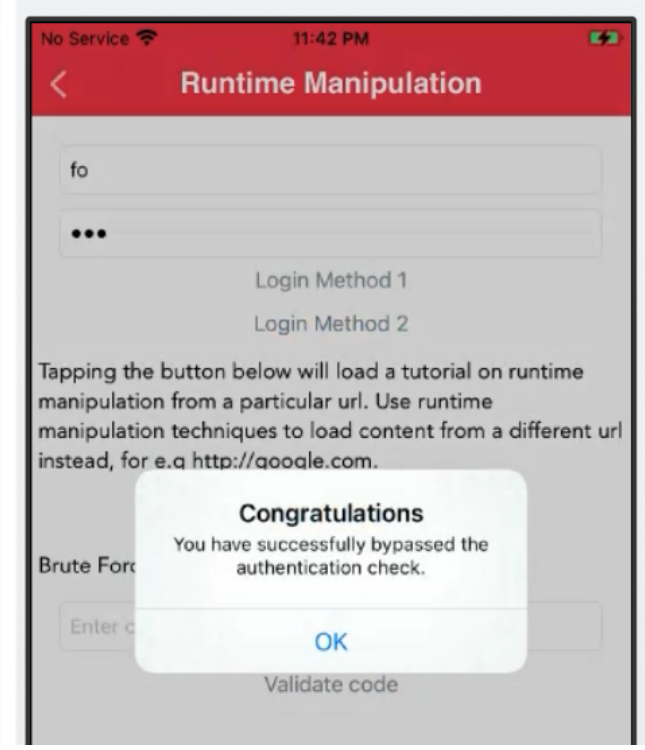
- Execute the script and confirm the script execution in the Frida tab and press the button “Login Method 1” again. The Frida script is triggered and we see the return value of the method we hooked into in the Frida console. But we still cannot login.



- We need to change the return value from false to true by changing 0 to 1. Edit the script and change the return value from `0x0` to `0x1` in line 14.

```
1  function show_modify_function_return_value(className_arg, funcName_arg)
2  {
3      var className = className_arg;
4      var funcName = funcName_arg;
5      var hook = eval('ObjC.classes.' + className + '["' + funcName + '"]');
6      Interceptor.attach(hook.implementation, {
7        onLeave: function(retval) {
8          console.log("\n[*] Class Name: " + className);
9          console.log("[*] Method Name: " + funcName);
10         console.log("\t[-] Type of return value: " + typeof retval);
11         //console.log(retval.toString());
12         console.log("\t[-] Return Value: " + retval);
13         //For modifying the return value
14         var newretval = ptr("0x1") //your new return value here
15         retval.replace(newretval)
16         console.log("\t[-] New Return Value: " + newretval)
17       }
18     });
19 }
20 //YOUR_CLASS_NAME_HERE and YOUR_EXACT_FUNC_NAME_HERE
21 show_modify_function_return_value("LoginValidate" ,"isLoginValidated")
```

- Press the button “Login Method 1” again.



If you couldn’t login, you might have the wrong class or method name in the script or a typo. Edit it and try again! Check the output in the Frida console or ask the trainer.

What is the script `show-modify-method-return-value.js` doing?

Lab - Bypass Jailbreak Detection

Time to finish lab	15 minutes
App used for this exercise	Jailbreak-1.ipa

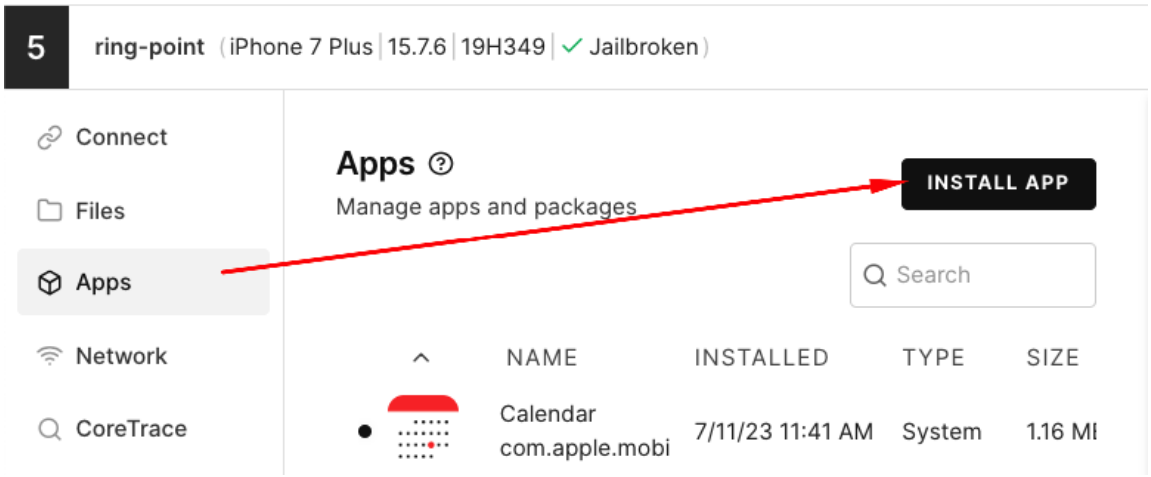
Training Objectives

Bypass Jailbreak Detection Mechanisms

Exercise

Preparation

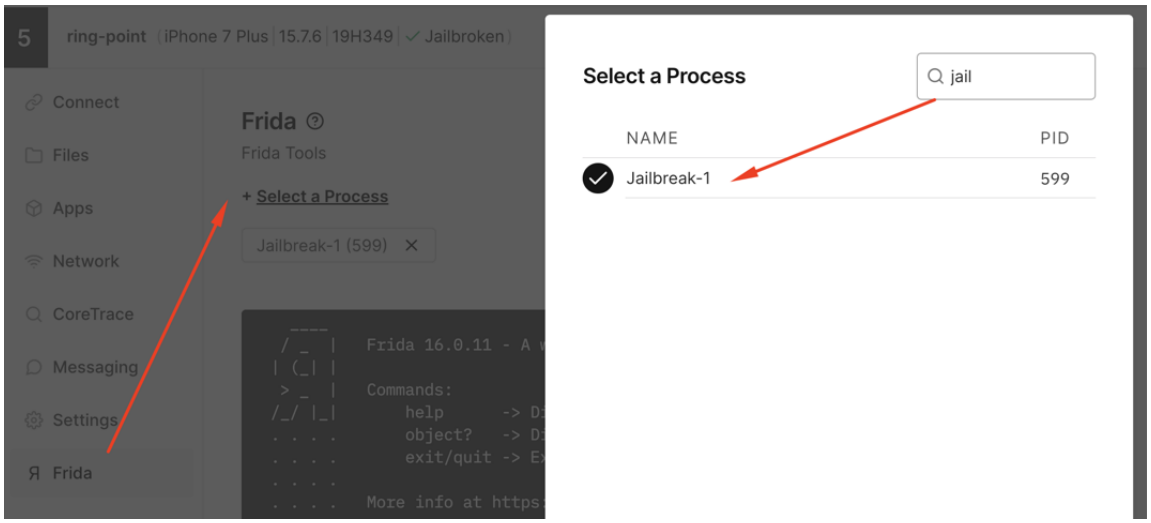
- Go to the “Apps” menu in Corellium and install the app `Jailbreak-1.ipa` by selecting “Install App”. You downloaded the IPA as part of the preparation pack today, it’s in the “Apps” folder. This might take a minute.



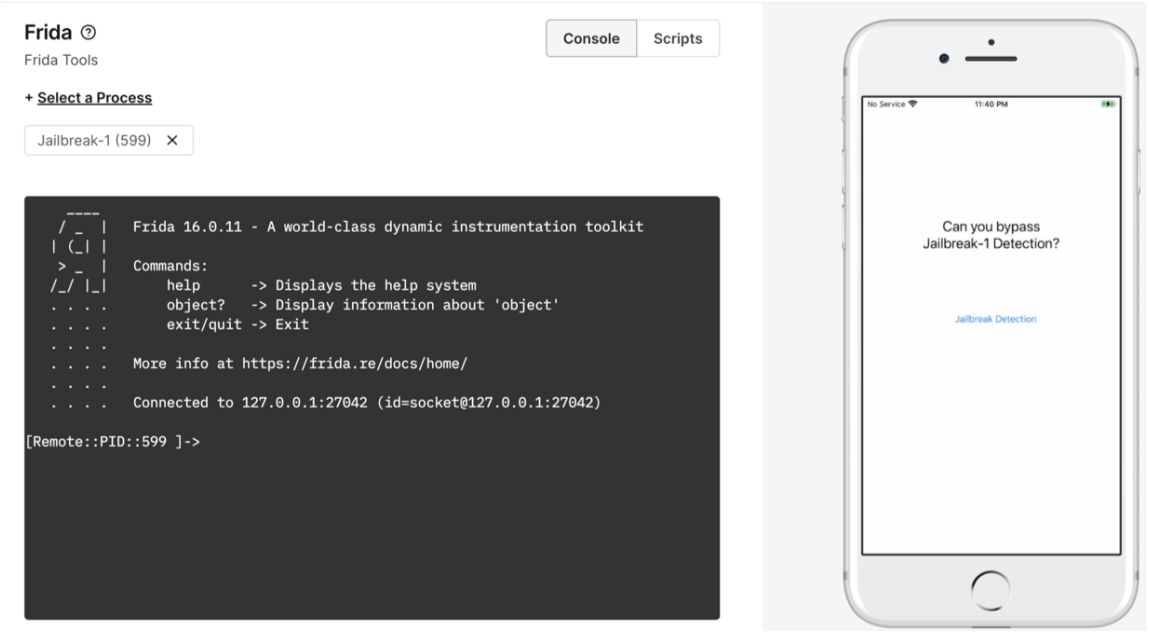
- Run the app “Jailbreak-1”:

Jailbreak App

- Start the Frida Server by clicking on the Frida tab on the left in Corellium. Click on “Select a Process” and search for “jailbreak” and you will find the running app (your process id, the PID, will be different in your case):



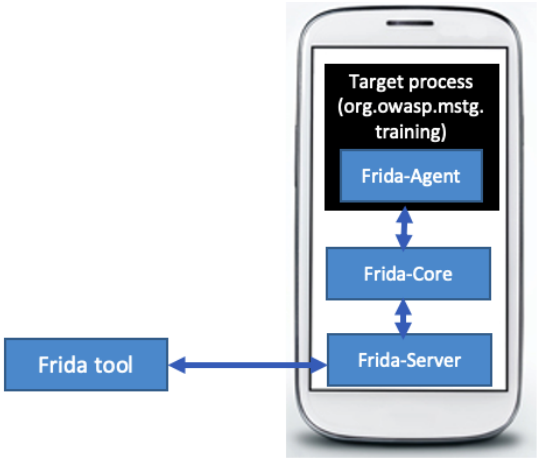
- Once selected, attach to the process and make sure that the app is running in the foreground on the iPhone.



- Click on “Console” button on the top right. You can see now the Frida console.

```
     ____
    / _  |   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 127.0.0.1:27042 (id=socket@127
      .0.0.1:27042)

[Remote::PID::599 ]->
```

Frida CLI is now connecting to the running Frida server and is injecting the Frida Agent into the `Jailbreak-1` app.

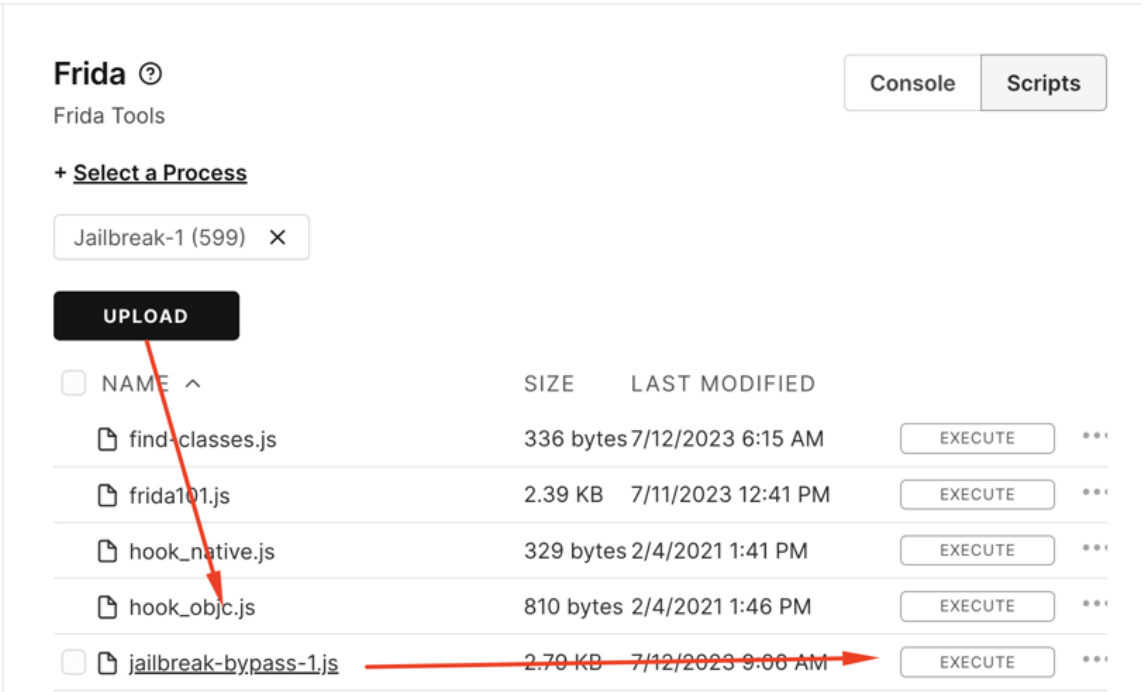


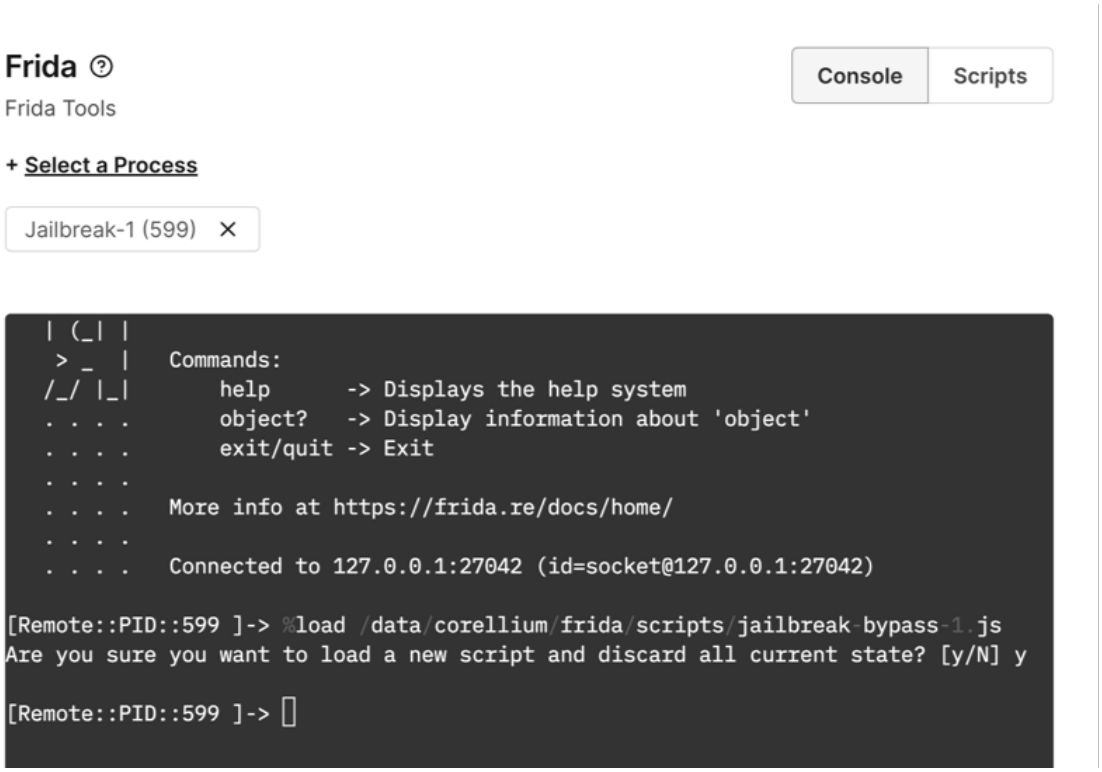
- In the App, click on the button to trigger the `Jailbreak Detection`. A pop-up will appear saying the device is jailbroken. This check we want to bypass.

Frida CLI + Scripts

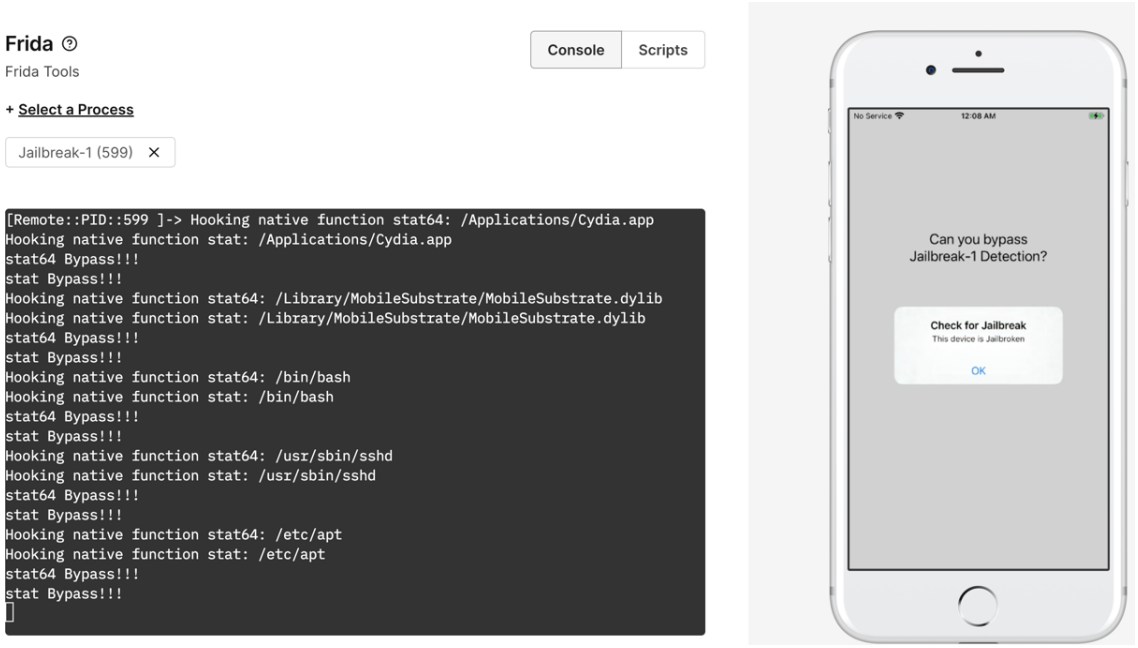
We already know that we can use Frida scripts to inject them into the target app.

- Upload the script `jailbreak-bypass-1.js` and execute it and go back to the console and confirm the execution with "y".



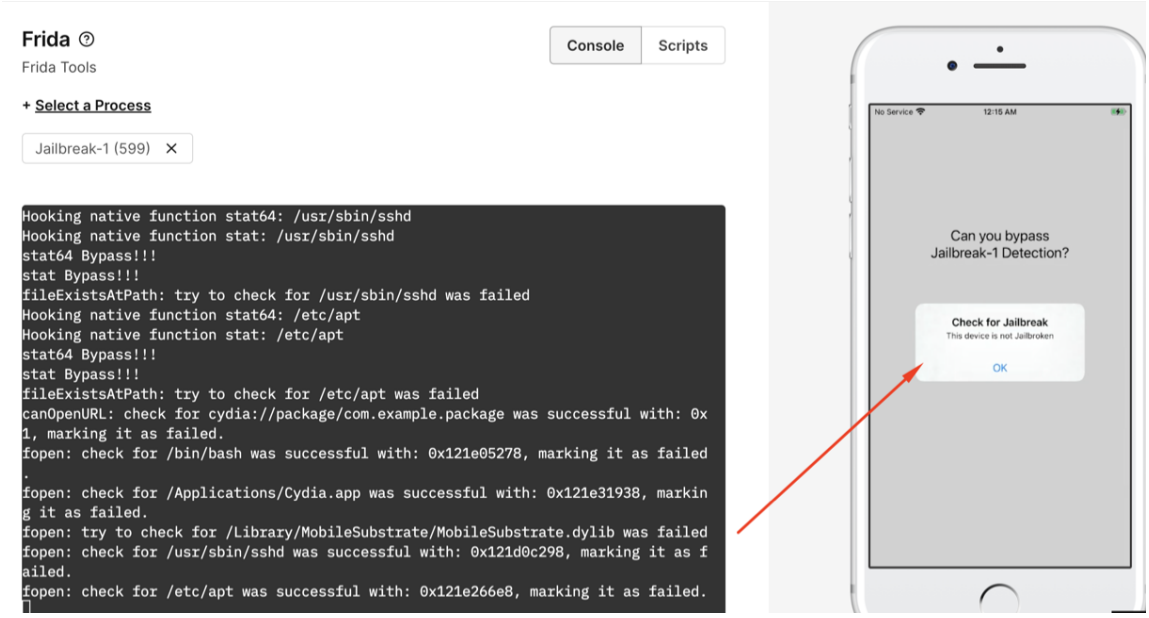


- Trigger again the jailbreak detection button. You can see in the console output that some checks were detected and hooked and therefore bypassed, but the app still detects the jailbreak.



- This script was therefore not fully working and couldn't bypass some checks. Upload the script `jailbreak-bypass-2.js` and execute it and go back to the console and confirm the execution with "y".
- Trigger again the jailbreak detection button. You can see in the console output that some checks were detected and hooked and therefore bypassed, and this time the app

is not detecting the jailbreak anymore.



- How many different jailbreak detection mechanisms that are being bypassed can you find? You can also have a look at the source code of `jailbreak-bypass-2.js`.

Congratulations, you are able to bypass Jailbreak detection like a penetration tester!