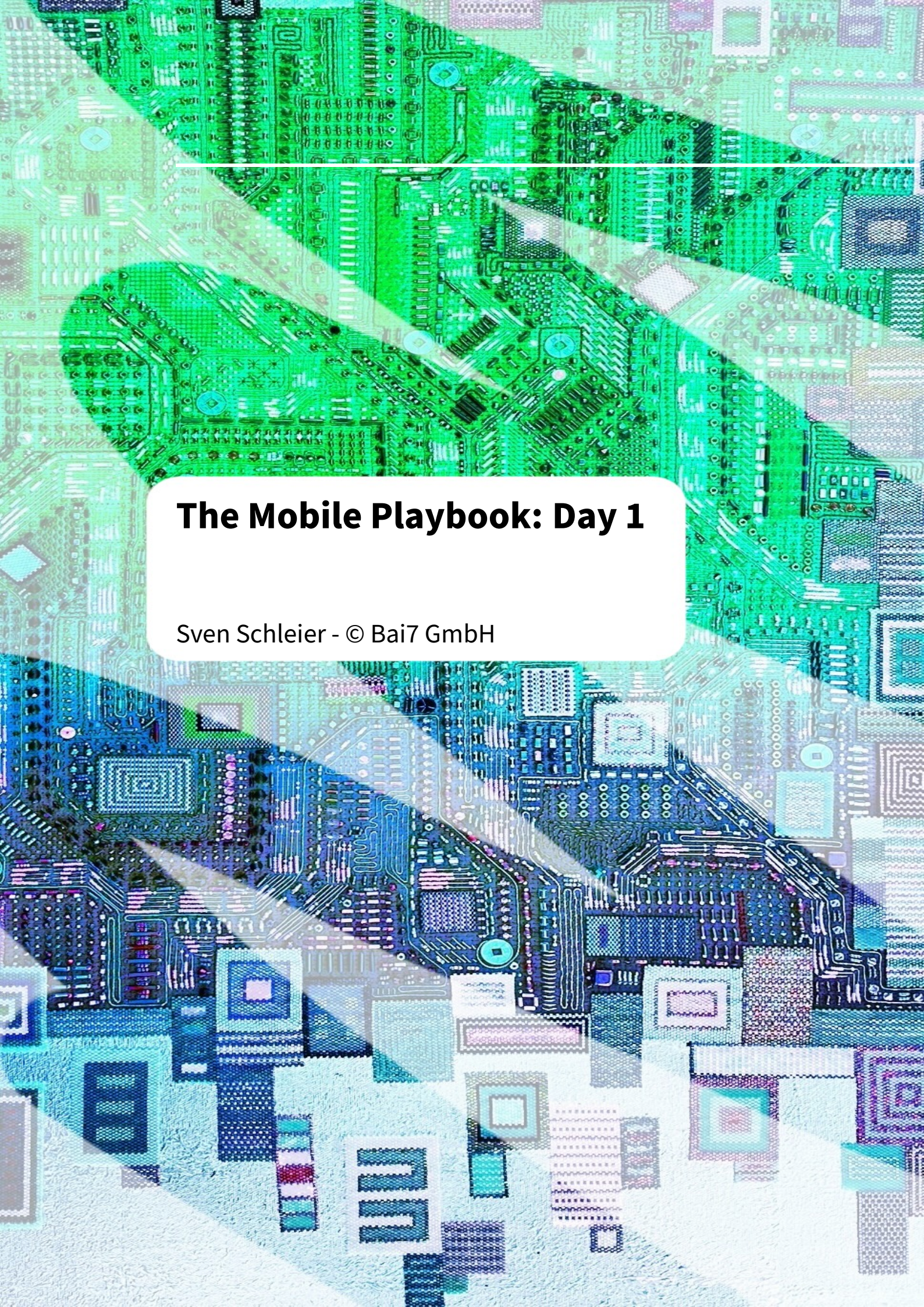

The Mobile Playbook: Day 1

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

Clone the repo

Clone the repo https://github.com/bai7-at/mobile-playbook-dev-android/ to your local machine if you haven't done it yet.

Fork the repo

- Login into Github with your account. A personal account using "GitHub Free" is sufficient for the training. If you don't have an account, create one now: https://github.com/signup
- Fork the following repo: https://github.com/bai7-at/mobile-playbook-dev-android

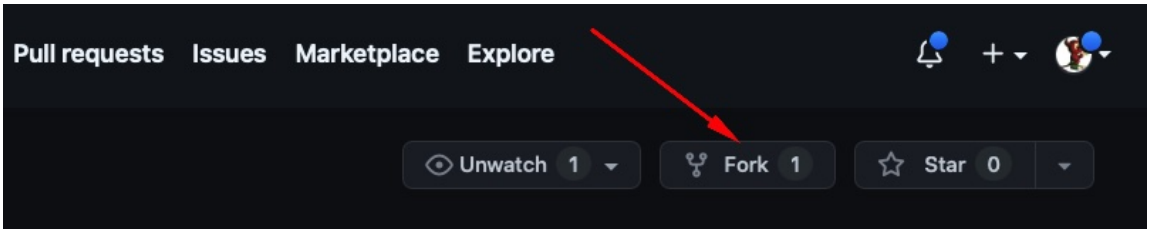

Figure 1: Fork repo

- 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.

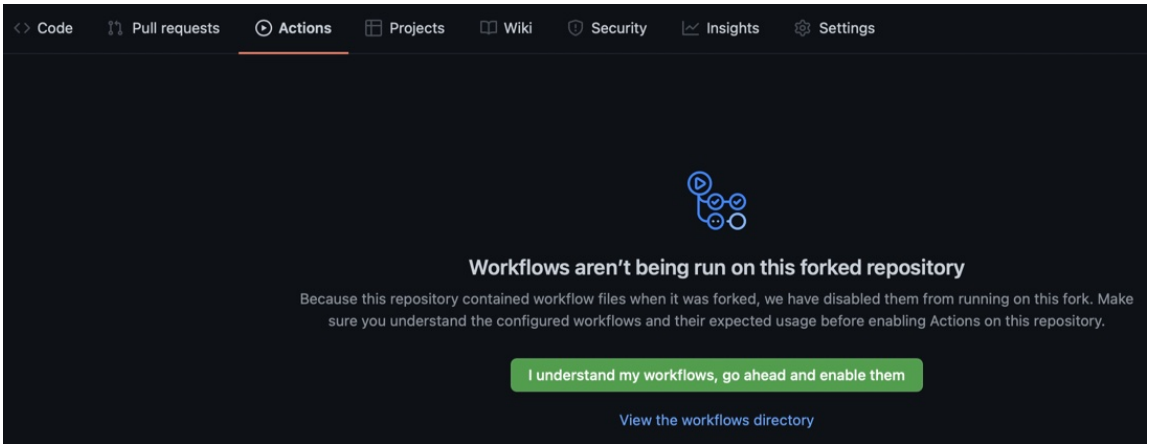

Figure 2: Enable Workflows

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

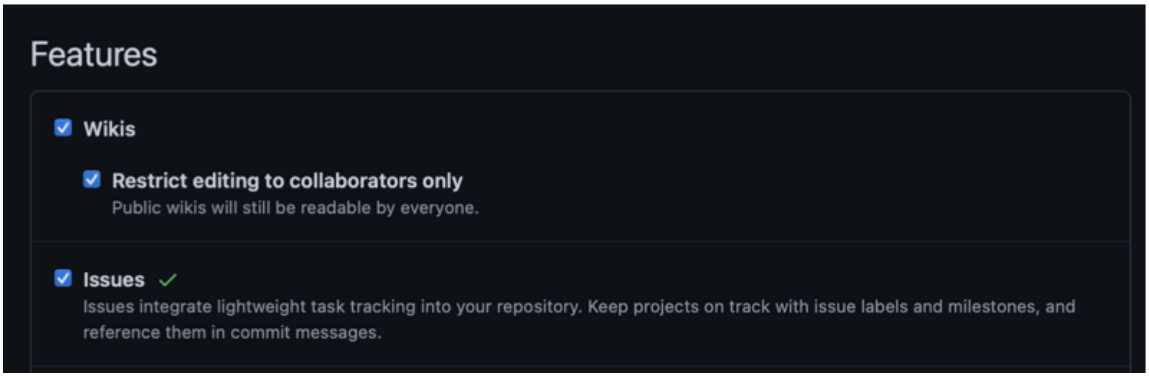

Figure 3: Enable Issues

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

Lab - Secret Scanning

Time to finish lab	15 minutes
App used for this exercise	NYT decompiled

Training Objectives

Learn what an attacker would do after decompiling an APK and use the decompiled code to scan it for secrets with the tool `gitleaks`. In our lab I already decompiled the New York Times Android App for you and we simplified it by executing gitleaks through a Github Action.

Tools used in this section

- `gitleaks` - https://github.com/gitleaks/gitleaks

Preparation

- Go to your forked repo and click on “Actions”. Select the Github workflow “Gitleaks‘. On the right side you have a dropdown menu called”Run workflow”. Open it and click on “Run workflow”:

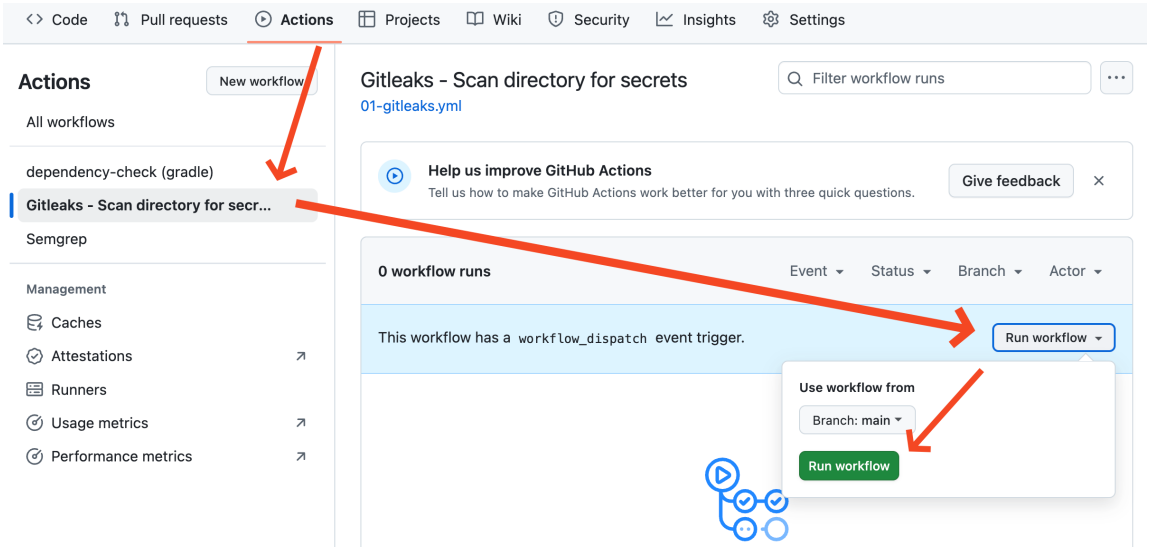

Figure 4: Gitleaks Github Action

- Click on “Actions” and “Gitleaks”, you will see the workflows that we just triggered. Click on the `Gitleaks` worflow run.

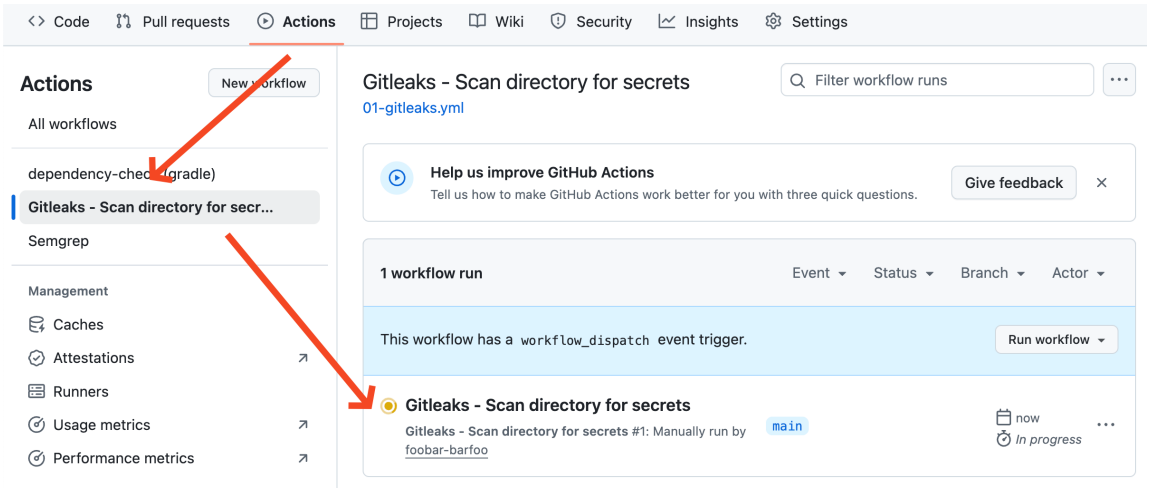

Figure 5: Actions running

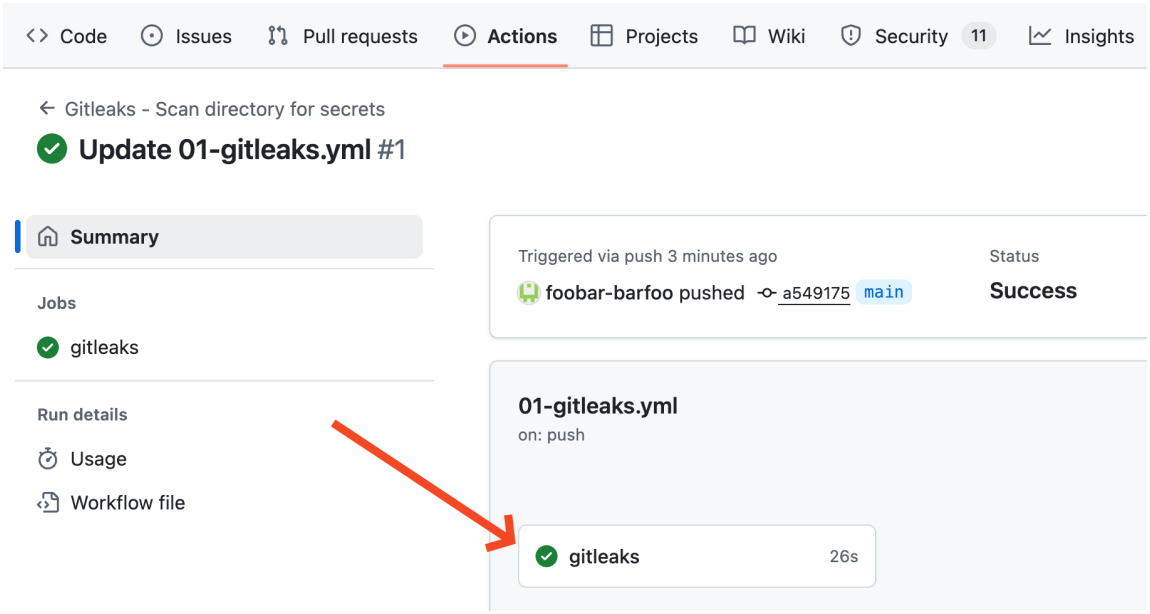

Figure 6: Actions running

- Click on the step “Run Gitleaks over nyt_decompiled directory”.

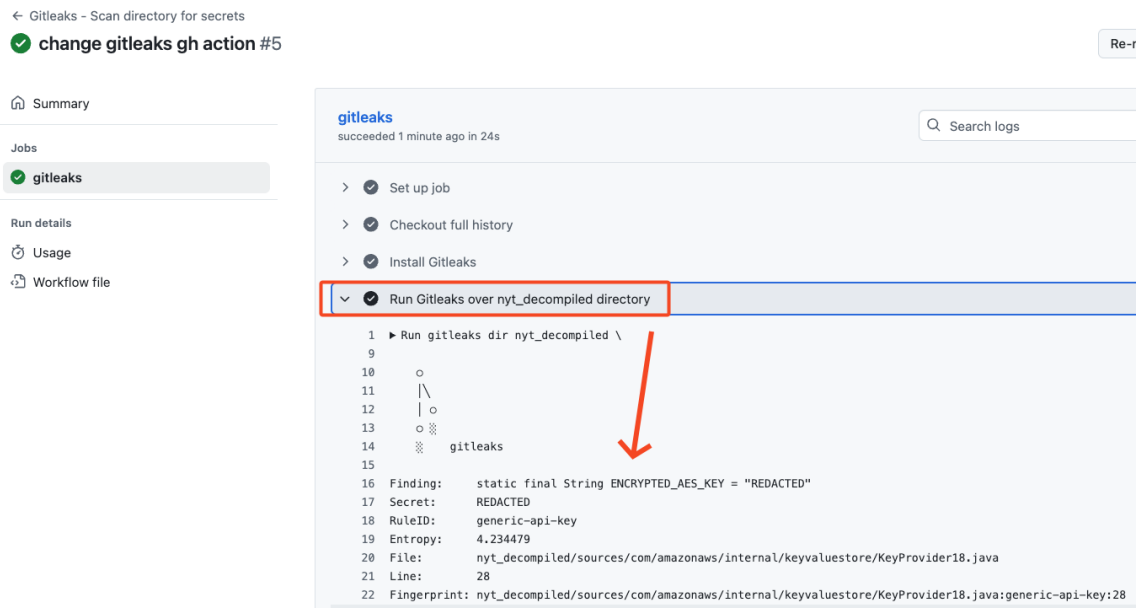

Figure 7: Gitleaks run completed

- You will get the detailed output of the whole `gitleaks` job including all steps.
- The `gitleaks` Github Action will now be executed every time we are doing a commit or pull request to change the gitleaks Github action or if we trigger it manually, like we just did now. This is achieved through the `workflow_dispatch` command in the Github action.
- But detecting is only half the job, as we need to manage the amount of detected secrets 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 `01-gitleaks.yml` file:

```
      - name: Upload SARIF to GitHub Security
        uses: github/codeql-action/upload-sarif@v3
        with:
          sarif_file: gitleaks.sarif
```

- Go to the “Security” tab in your repo and click on “Code scanning” on the left side. All secrets that could be identified by `gitleaks` are all automatically imported into the code scanning alerts.

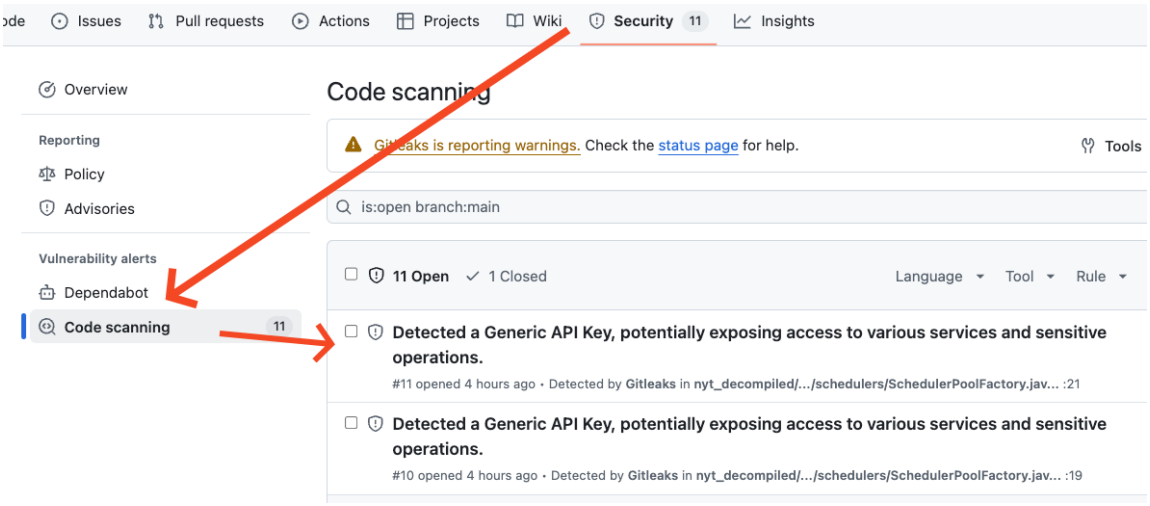

Figure 8: Gitleaks results

- This allows us to manage the identified alerts. Github takes care of de-duplication of findings automatically through fingerprinting, so Github will not flag out the same finding again in consecutive scans. Click on the first finding and you can get all the details around it.

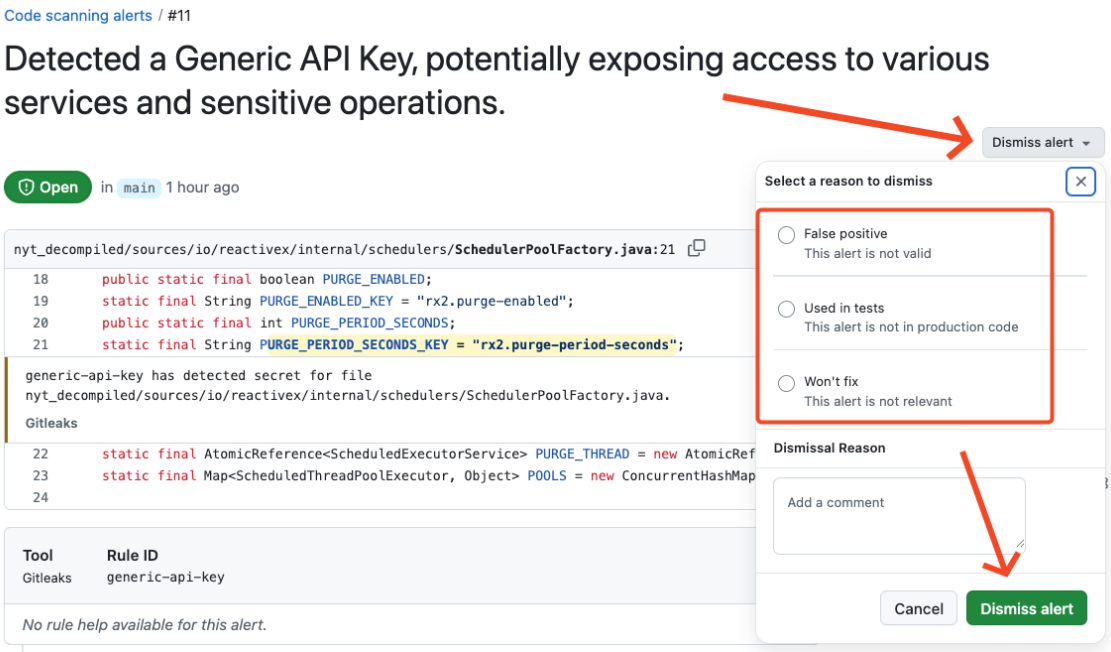

Figure 9: Gitleaks Triage

- Your job is to triage the identified findings and decide if they are a:
  - valid finding (then keep the finding),
  - False Positive or
  - if you won't fix it.

Once you are done with it: Congratulations, you are now able to detect secrets!

Appendix

Rules

The rules in `giteaks` are defined as regular expressions, which can always trigger false positives. Here is an example for AWS keys.

You can add your own configuration and rules, or ignore secets.

Lab - Scan an APK for Secrets

Time to finish lab	15 minutes
App used for this exercise	com.nytimes.android.apk

Training Objectives

In this exercise you reviewing the results of an `apkleaks` scan of the New York Times app (the APK). Your goal is to check if you would report them or if they are a false positive.

Tools used in this section

- `apkleaks` - https://github.com/dwisiswant0/apkleaks

Exercise - Android Analyse Local Storage

The scan with `apkleaks` on the New York Times app has already been done for you and was stored in the file `nyt-apkleaks.txt`.

Your job is now to validate the secrets detected by `apkleaks` by opening the text file `nyt-apkleaks.txt` in your editor of choice. The file is available in the root directory of the repo you cloned to your local machine.

Make a decision if they are a false positive or a finding that you would report and share with the developer of the app!

Open in another terminal the decompiled Java Code in your editor of choice (like VS Studio Code). It is available in the directory of the repo you cloned earlier:

```
$ cd ~/mobile-playbook-dev-android/nyt_decompiled
```

> With the command `jadx -d com.nytimes.android.rebuild.apk` it was already decompiled for you to save you some time.

In the decompiled code that you opened in your editor you can search for the identified secrets (e.g.an API key) and try to identify it's context of use.

In case you find a S3 or GCS Bucket, you can try the following to check if it's publicly accessible:

```
$ curl <bucket-URL>
```

Lab - Software Composition Analysis

Time to finish lab	15 minutes
App used for this exercise	Tasky

Training Objectives

Learn how to scan Android dependencies with the tool `dependency-check` by using Github Actions.

Tools used in this section

- `dependency-check` - https://jeremylong.github.io/DependencyCheck/
- `dependency-check` Gradle Plugin - https://jeremylong.github.io/DependencyCheck/dependency-check-gradle/index.html

Preparation

- In your forked repo click on “Actions” and you will see the workflow that was triggered earlier. Click on the “dependency-check (gradle)” workflow run. The scan might still be running.

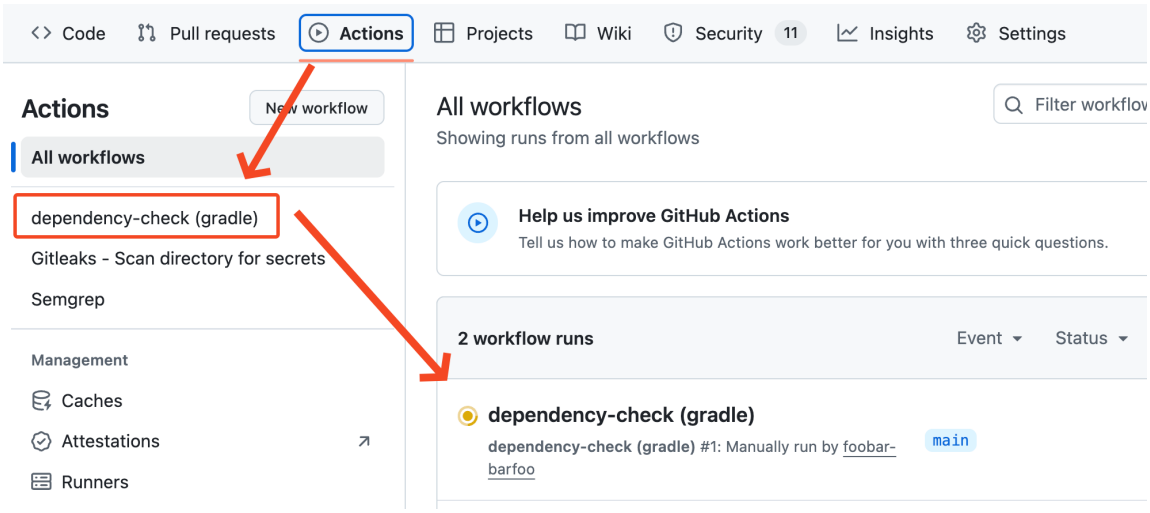

Figure 10: Worklow run

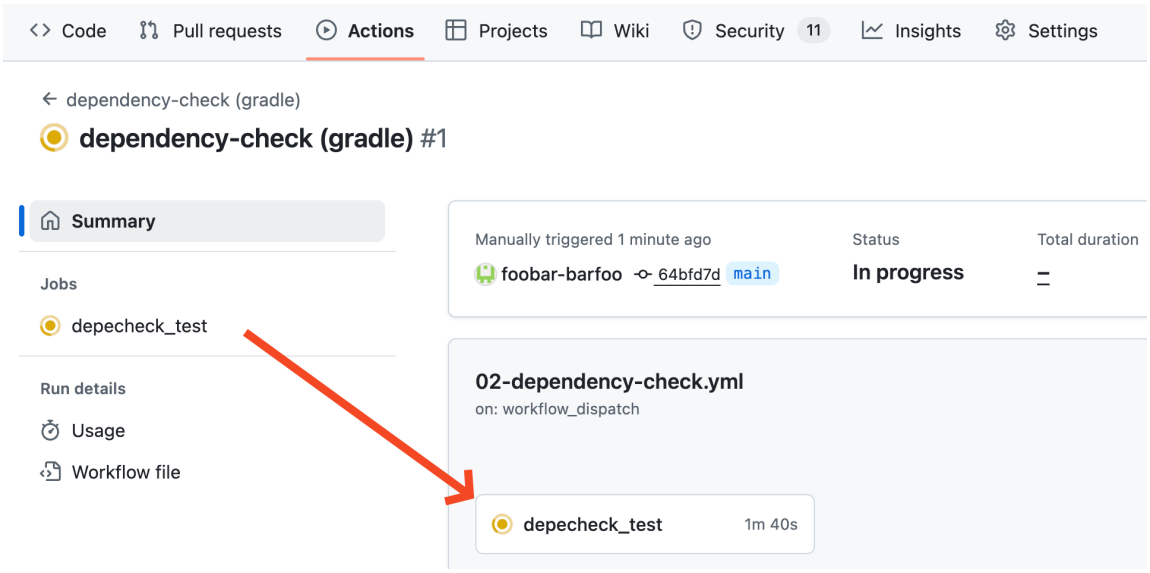

Figure 11: Worklow run

> The first scan can take over 10 minutes, as dependency check need to build up a vulnerability database and download data. That's why you triggered this scan earlier. In consecutive runs this database is cached in your fork and the workflow will take around 2-3 minutes only.

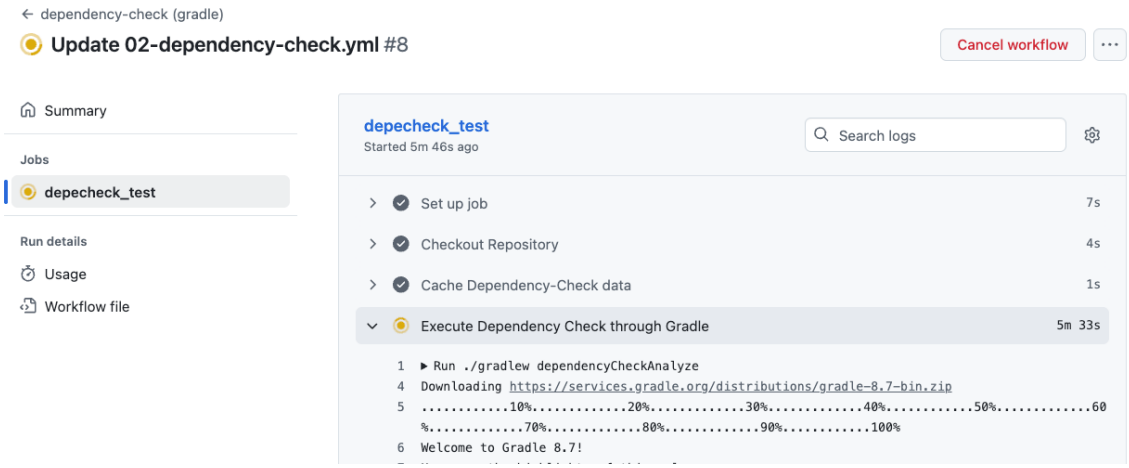

Figure 12: Worklow run

- Go to the "Security" tab in your repo and click on "Code scanning" on the left side. New vulnerabilities could be identified by `dependency check` and are all automatically imported into the code scanning alerts. Filter the new vulnerabilities by selecting the tool `dependency-check`:

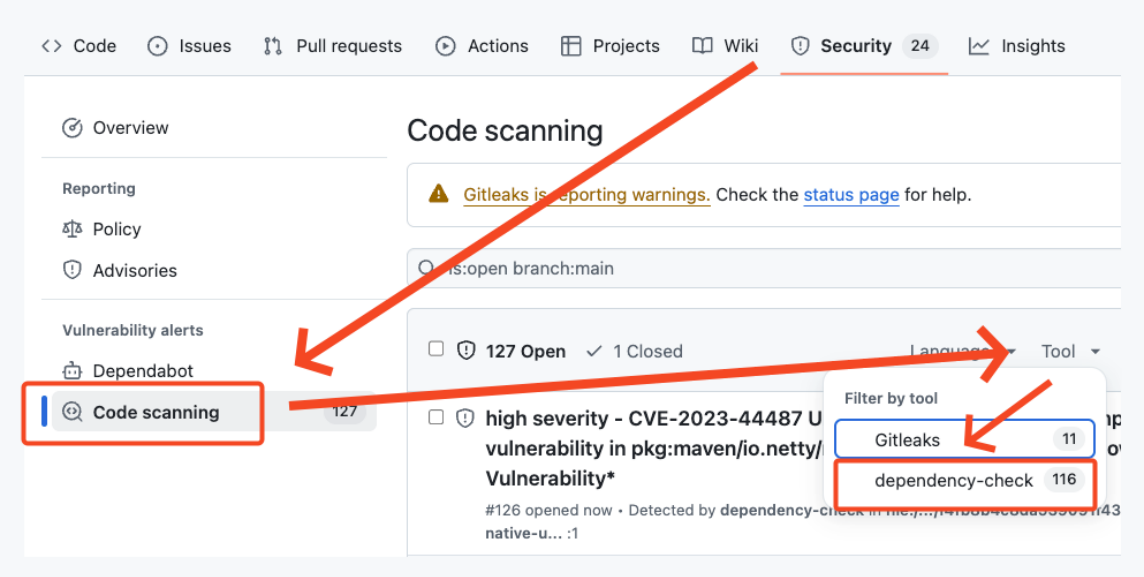

Figure 13: Findings

- We have now a lot of false positives through `netty` or `grpc` libraries, that are used to build the project and for testing but are not ending up in our Android app. Therefore we need to find a way to “suppress” them.
- Let’s include the file `suppresssion.xml` for `dependency-check` to remove these false positives from our scan result. Open the file `tasky/app/build.gradle.kts` and edit it:

Figure 14: Edit build.gradle.kts

- Remove the comments in line 22 to include the `suppressionFile`.

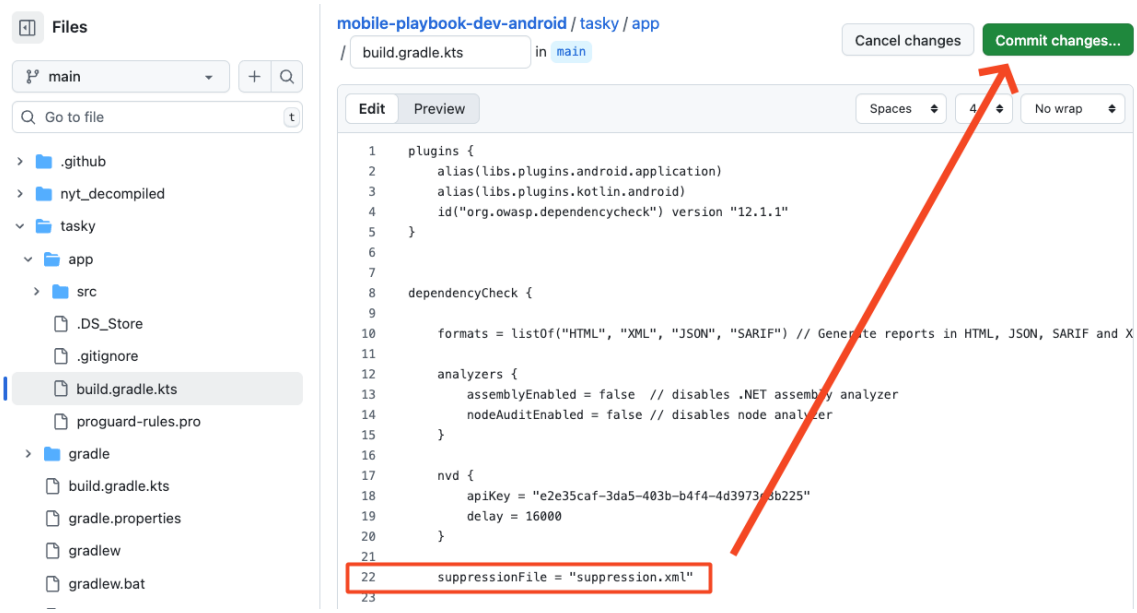

Figure 15: Include suppression.xml

- The Github Action is triggered again and the dependency check scan should be done in a few minutes. Now all the false positives are automatically removed. Select the tool dependency-check and you will have 13 open findings.

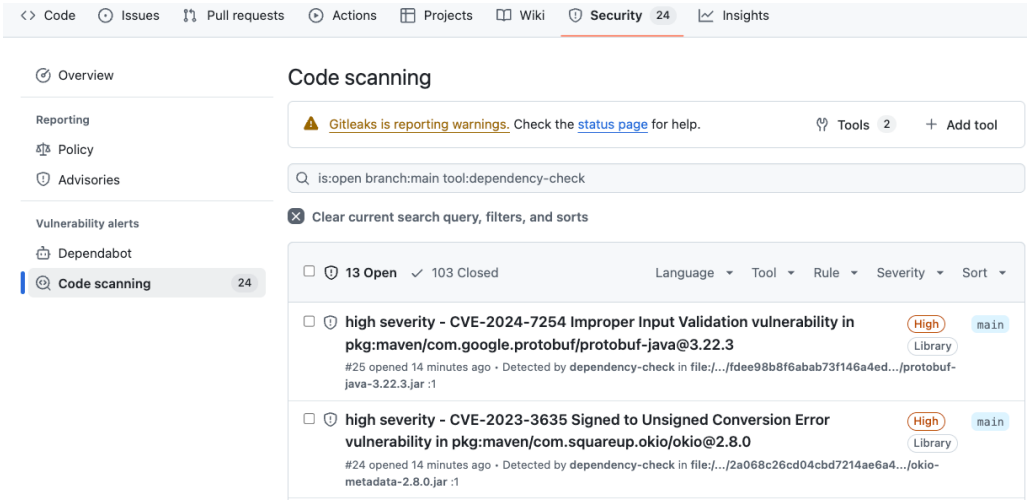

Figure 16: Include suppression.xml

- Your job is to triage the identified findings and decide if they are a:
  - valid finding,
  - false positive,
  - used in tests, or
  - if you won't fix it (and why).

Go through each finding and make a decision!

You can look into the HTML report that is being generated for each workflow run.

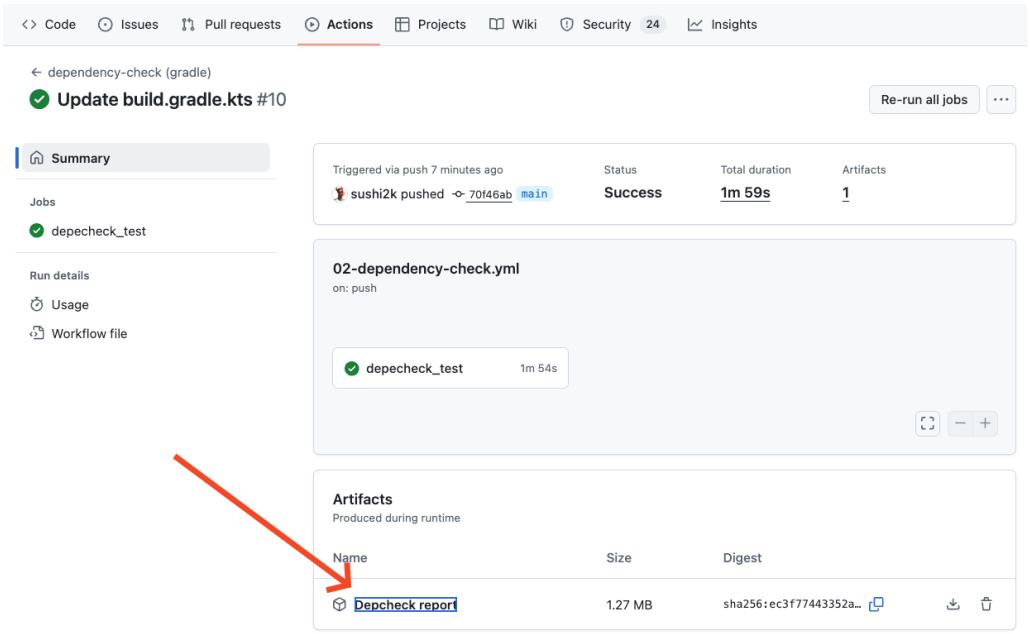

Figure 17: Include suppression.xml

Congratulations, you are now able to scan, detect and triage known vulnerabilities in your libraries and manage them with code scanning alerts!

Lab - Static Scanning with semgrep

Time to finish lab	15 minutes
App used for this exercise	Finstergram.apk and Kotlin-Shack.apk

Training Objectives

In this lab we will be scanning the decompiled code base of two Android apps for vulnerabilities and misconfiguration with `semgrep` by using a Github Action.

Tools used in this section

- `jadx` - https://github.com/skylot/jadx
- `semgrep` - https://github.com/semgrep/semgrep

Preparation

- Go to your forked repo and click on “Actions”. Select the Github workflows “Semgrep‘. On the right side you have a dropdown menu called”Run workflow”. Open it and click on “Run workflow”:

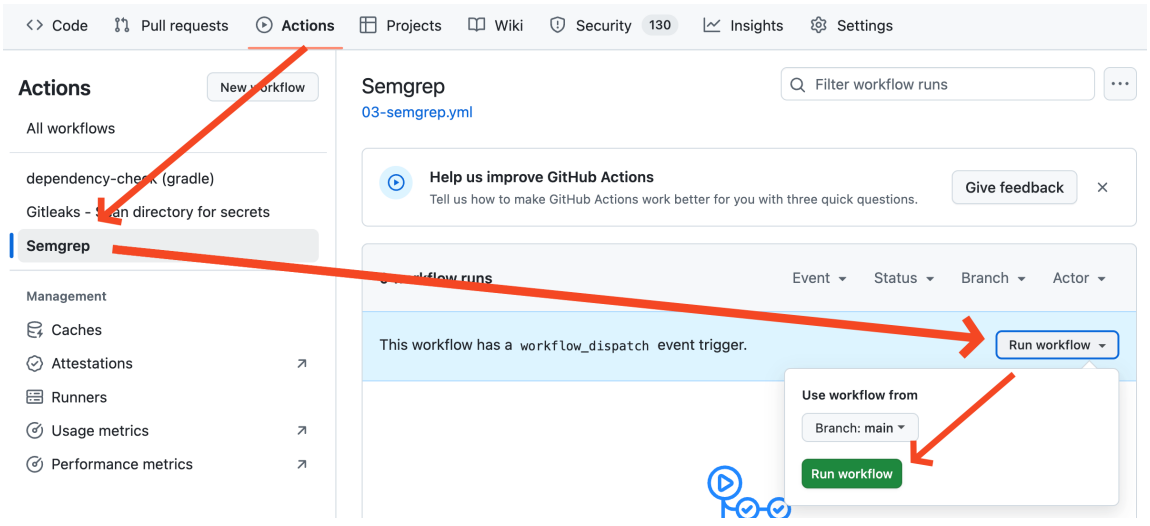

Figure 18: Gitleaks Github Action

- Click on “Actions” and “Semgrep”, you will see the workflows that we just triggered. Click on the `Semgrep` worflow run. While it’s scanning, continue to read, this will take 1-2 minutes.

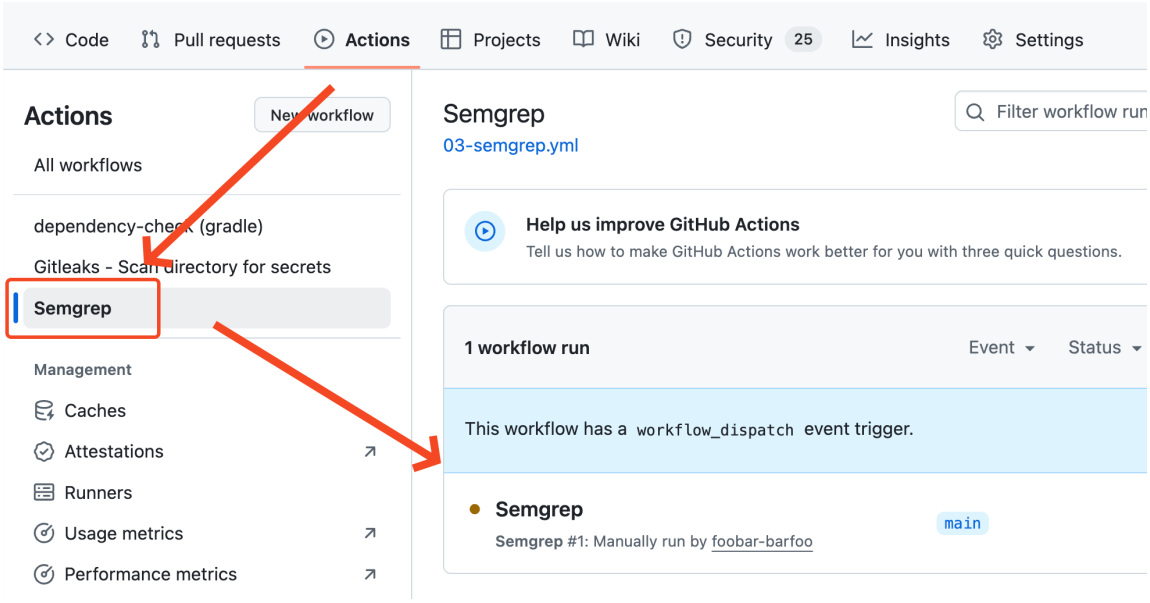

Figure 19: Running semgrep workflow

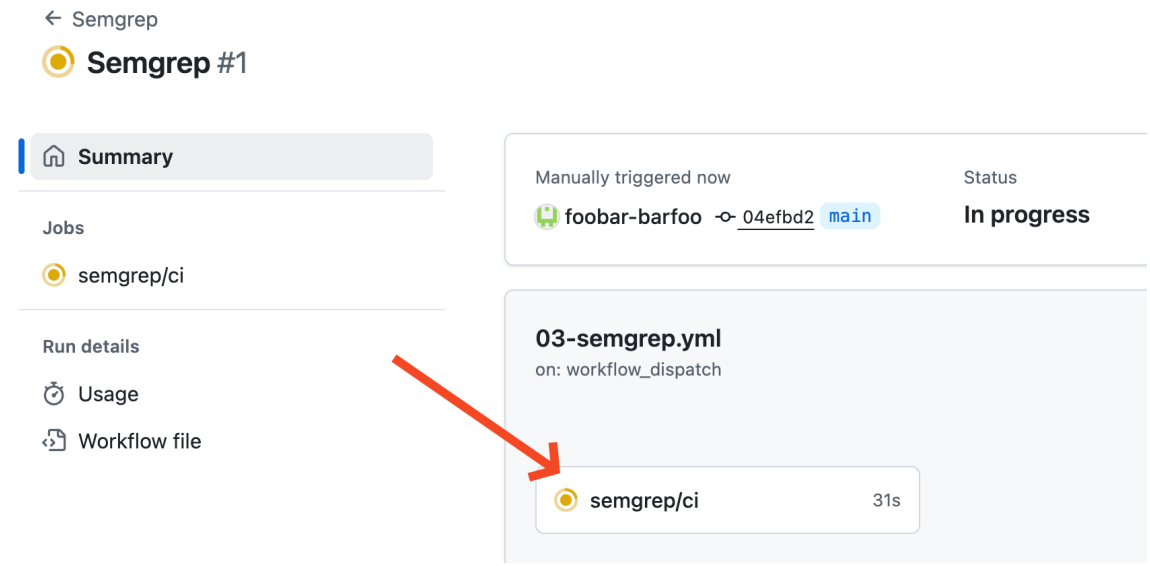

Figure 20: Running semgrep workflow

Semgrep Github Action

- The app `Finstergram` and `Kotlin-Shack` was already decompiled for you with `jadx` and saved into the repo.
- Once the semgrep job is done you will have identified new vulnerabilities that were added to the “Security” tab, click on it.

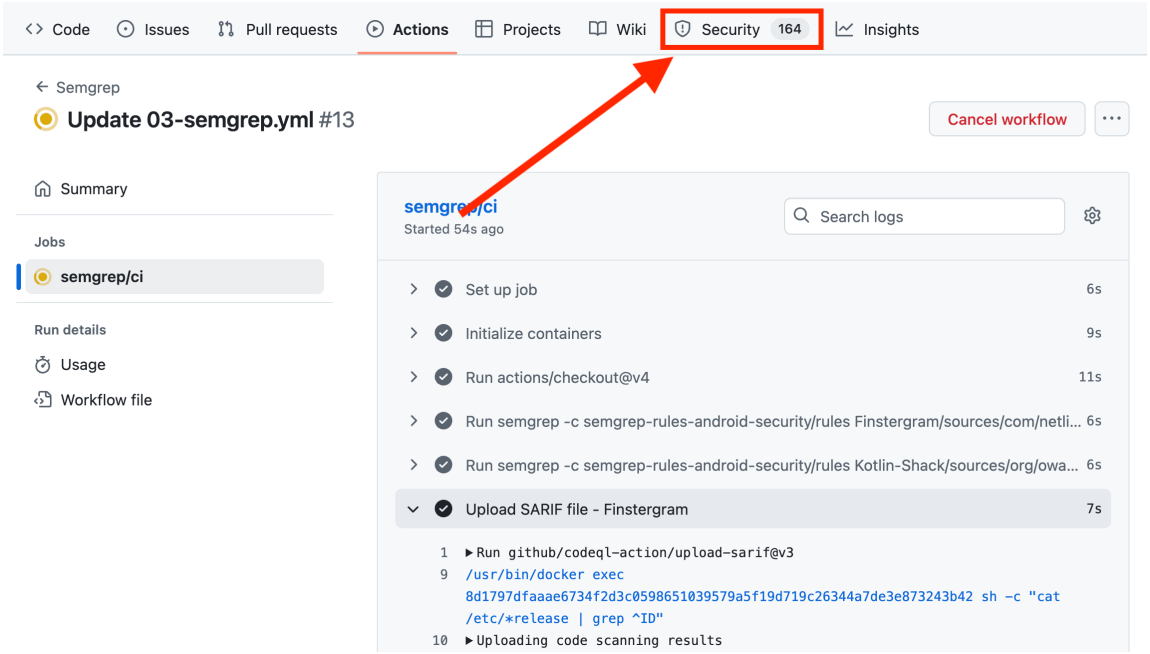

Figure 21: Security Tab

- The `semgrep` Github Action will now be executed every time we are doing a commit or pull request to change the gitleaks Github action or if we trigger it manually, like we just did now. This is achieved through the `workflow_dispatch` command in the Github action.
- 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 `semgrep.yml` file:

```
    # Upload the SARIF file to Github, so the findings show up in "
       Security / Code Scanning alerts"
    - name: Upload semgrep report
      uses: github/codeql-action/upload-sarif@v3
      with:
        sarif_file: results.sarif
```

- Go to “Code Scanning Alerts” and filter for the “Semgrep OSS” tool. This allows us to manage only the identified vulnerabilities of `semgrep`. Github takes care of de-duplication of findings automatically through fingerprinting, so Github will not flag out the same finding again in consecutive scans.

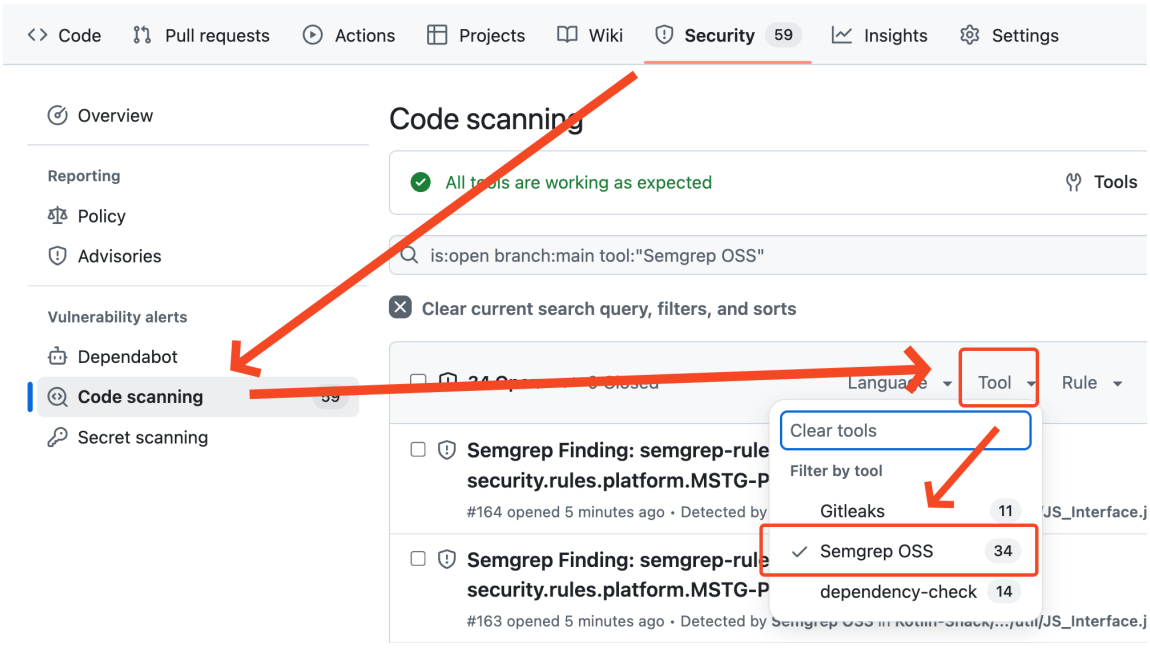

Figure 22: Triage findings

- 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).
- Go through each finding and make a decision! You can select rule by rule, to filter and group the results. Once done choose another one.

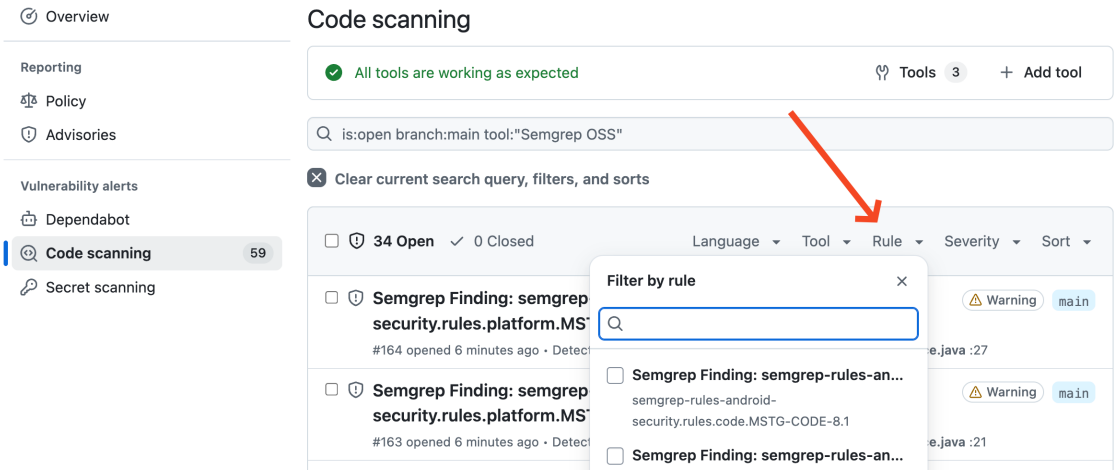

Figure 23: Select rule

Congratulations, you are now able to scan vulnerabilities with `semgrep` and detect and

triage vulnerabilities and manage code scanning alerts with Github Code Scanning!

Resources

- Mobile App Finstergram - https://github.com/netlight/finstergram

Lab - Frida 101 (Frida-Server)

Time to finish lab	15 minutes
App used for this exercise	Kotlin-Shack.apk

Training Objectives

1. Learn how to use Frida
2. Use a Frida script that is bypassing root detection

Tools used in this section

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

Exercise - Frida Usage

Preparation

- Login into Corellium: https://bsides.enterprise.corellium.com/ and open your Android device instance.
- If you haven't done it yet, install the Kotlin-Shack app: Click on "Apps" on the menu on the left side and then "Install App". Select the app "Kotlin-Shack.apk" from the repo you cloned earlier, its in the "apps" folder. Then install the app and follow the workflow on the screen.

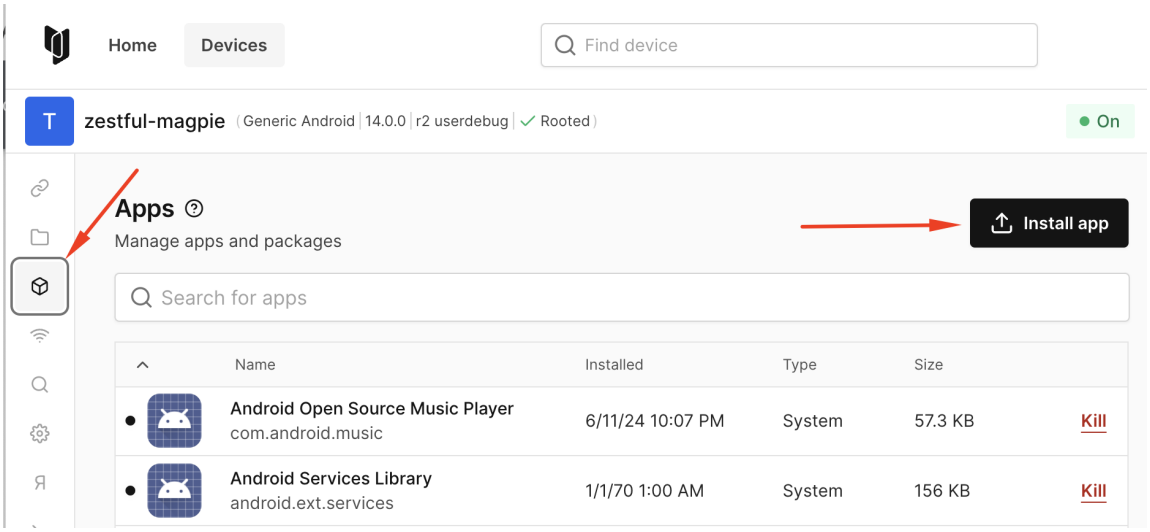

Figure 24: Install Android app

- Once installed, start the app `Kotlin-Shack` on the Android device.

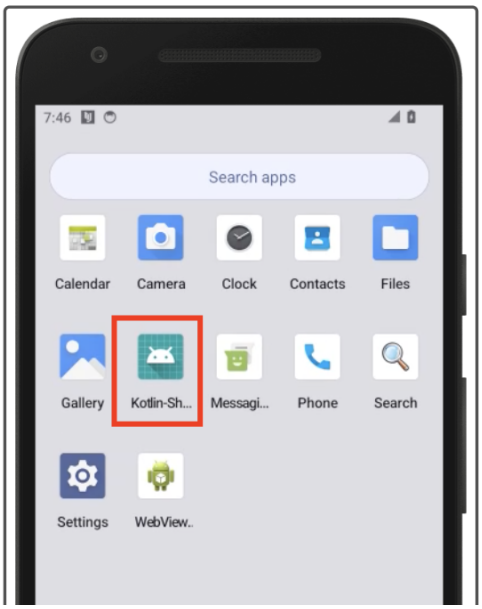

Figure 25: Start the app

- Open the Frida tab in Corellium.

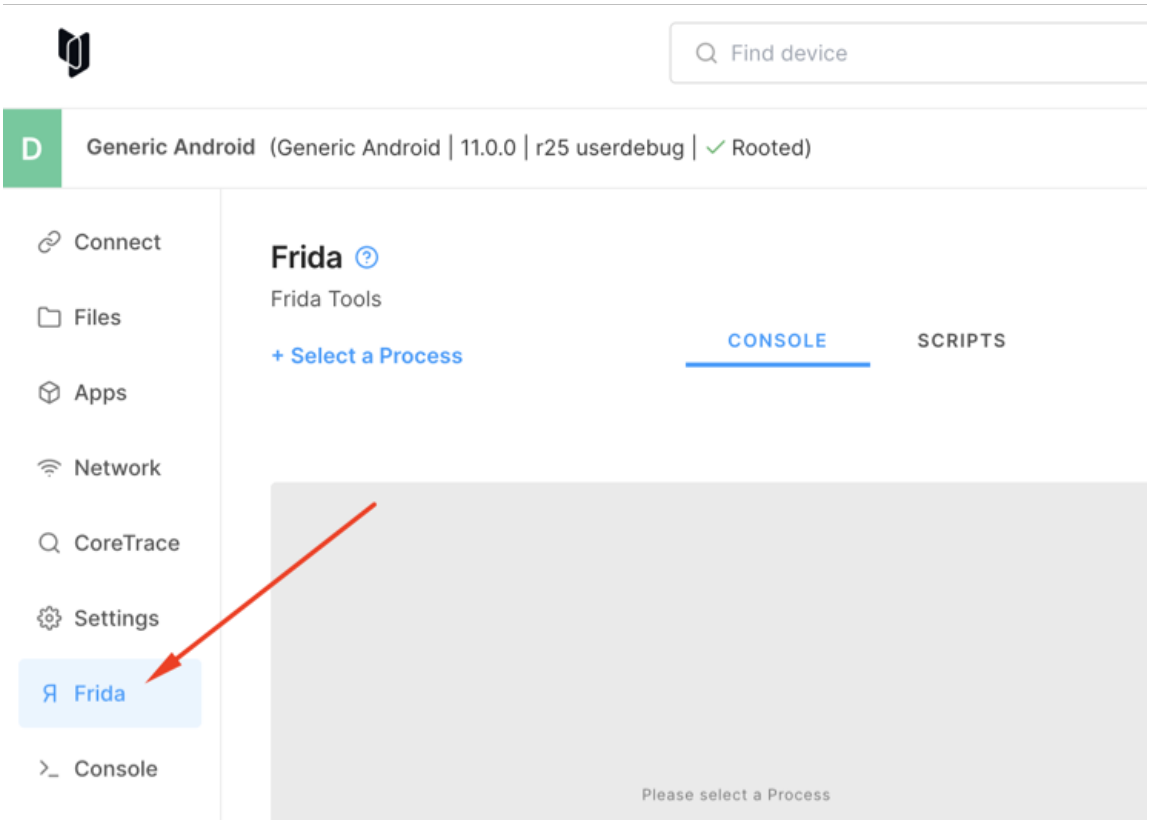

Figure 26: Frida functions in Corellium

- Click on “Select a Process” in Frida and search for “Kotlin” and you will find the running app (your process id, the PID, will be different in your case):

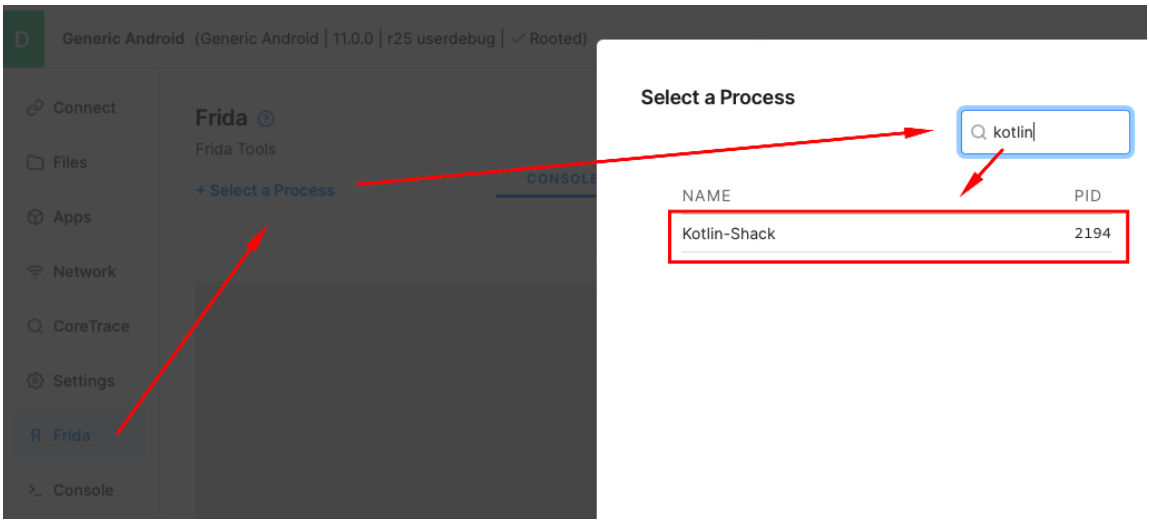

Figure 27: Select a process in Frida

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

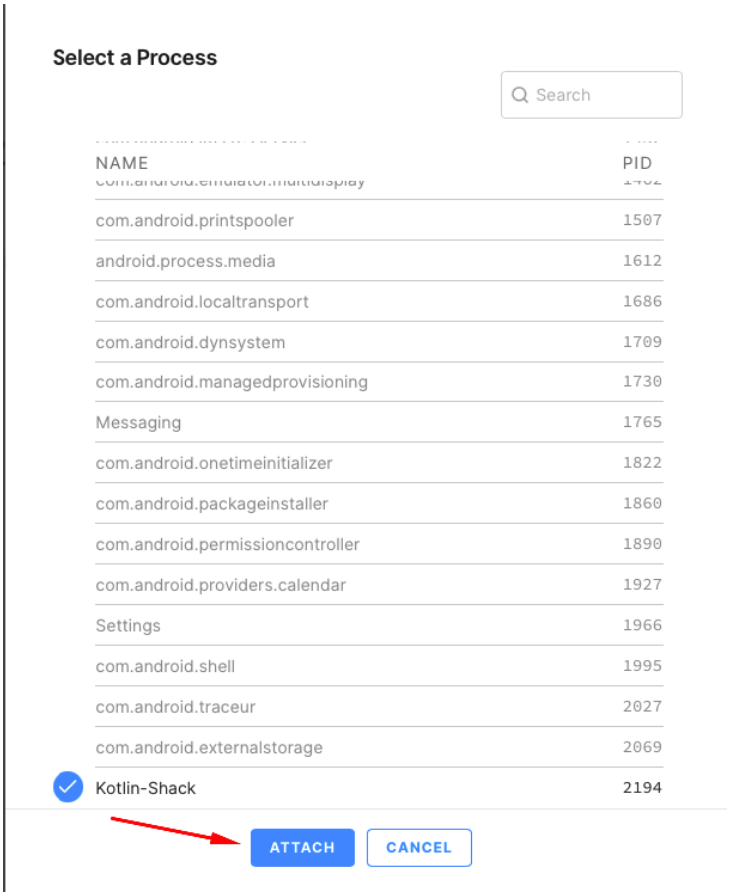

Figure 28: Attache to the Kotlin-Shack app process

- You can see now the Frida console.

```
     ____
    / _  |   Frida 16.3.3 - A world-class dynamic instrumentation
       toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to 127.0.0.1:27042 (id=socket@127
      .0.0.1:27042)

[Remote::PID::2438 ]->
```

Frida CLI (console) is now connecting to the Frida-Server running on the Android instance and the Frida Core Framework is injecting the Frida-Agent into the specified identifier (`Kotlin-Shack` or the package name `org.owasp.mstgkotlin`).

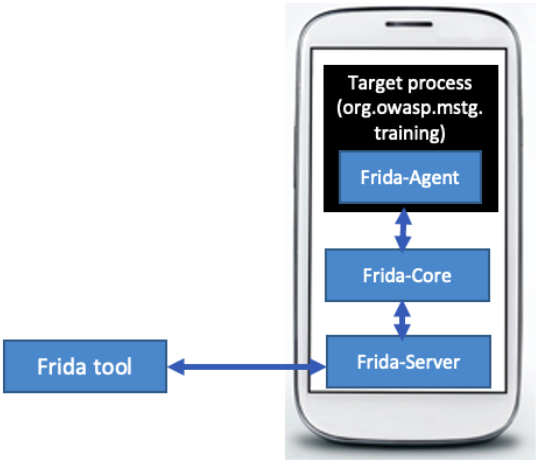

Figure 29: Frida Server

Click on the multiple root detection button in the app. This will give you a Toast message on the bottom of the UI that the app has detected that the Android instance is rooted.

Our mission is to bypass this check during runtime with Frida and make the app think that this Android phone is NOT rooted.

So what can we do with the interactive Frida console? You can write commands to Frida using it's JavaScript API, just press the `tab` key on your keyboard in the CLI to see the available commands. For example you can query if Java is available or what Android OS version is used:

```
[Remote::PID::2194 ]-> Java.available
true
[Remote::PID::2194 ]-> Java.androidVersion
"13"
```

The Frida CLI allows you to do rapid prototyping to create Frida scripts and also debugging.

Frida CLI + Scripts

Besides using the CLI of `frida`, we can also load scripts. This will load pre-defined JavaScript calls as script that Frida will execute in the context of the targeted app. Click in Corellium on the top right on “Scripts” in the Frida window.

You can find the `Frida-Scripts` directory in the repo you cloned earlier. Upload the script `test.js` from this directory and execute the script:

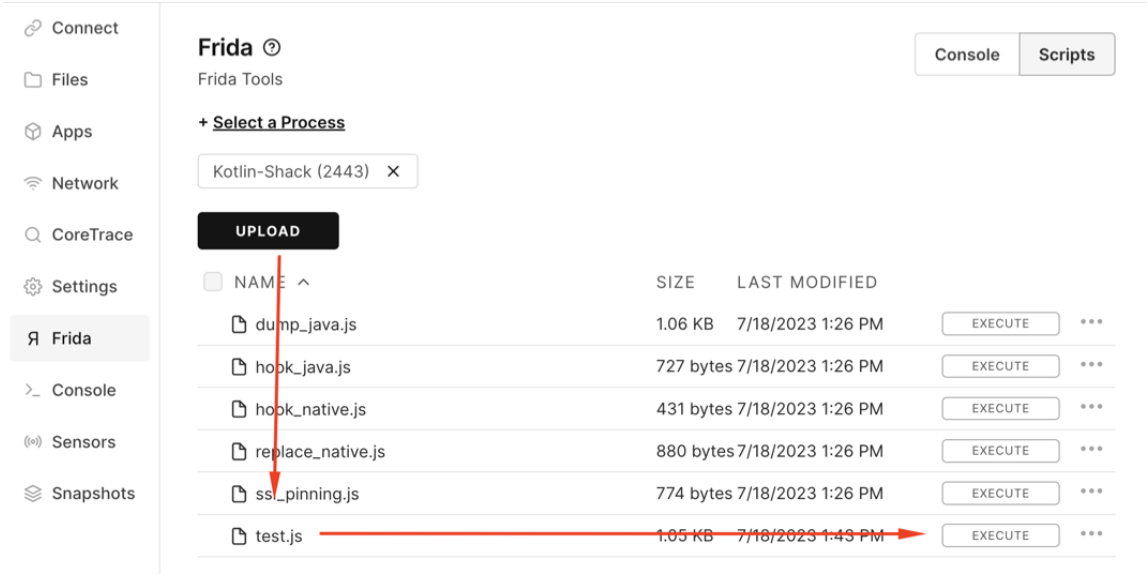

Figure 30: Upload and execute test.js

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

Figure 31: Confirm execution of script

In the Frida console, type in `makeToast()` and trigger the function. You can also only type `make` and wait for a second and you should see the dropdown menu with the available functions. You can just select it and then by pressing Enter, but need to add the round brackets.

```
Remote::PID::2194 ]-> makeToast()
```

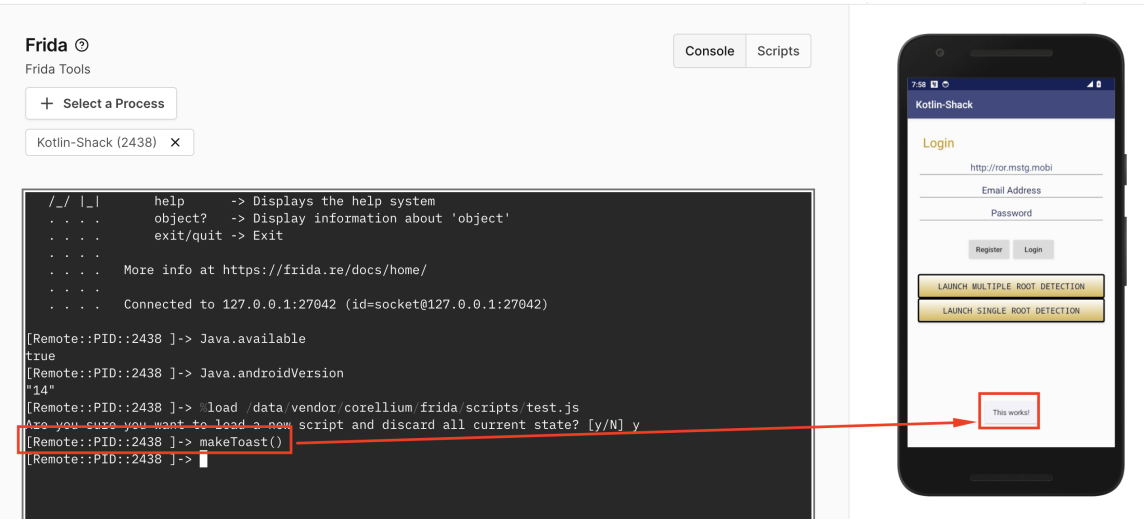

Figure 32: Toast message appears

Look what happened in your Android Instance! Type in `makeToast()` again and trigger the

function again. You will see a so called “Toast” message in the bottom of the screen for a few seconds.

So what is happening here (you can see the code also when selecting the script and click on the three doted lines and then “edit”)?

```
1  /*
2     Simple Android Toast
3     https://www.yodiw.com/frida-android-make-toast-non-rooted-device/
4  */
5
6  function makeToast() {
7      // Check if frida has located the JNI
8      if (Java.available) {
9
10         Java.perform(function () {
11             var context = Java.use('android.app.ActivityThread').currentApplication().getApplicationContext();
12
13             Java.scheduleOnMainThread(function() {
14                 var toast = Java.use("android.widget.Toast");
15                 toast.makeText(Java.use("android.app.ActivityThread").currentApplication
                       ().getApplicationContext(), Java.use("java.lang.String").$new("This works!"), 1).show
                       ();
16             })
17         })
18     }
19 }
```

Figure 33: Frida script

- Line 6 is defining a function called `makeToast`, which we just called in Frida.
- Line 8 is checking if the Java runtime is available, so this script will only work for Android.
- Line 10 contains the `Java.perform` wrapper, which is used to attach this function to the current thread, in our case of the Kotlin Shack App.
- Line 11 is getting the application `context` of the app we are hooking into.
- Line 13 is running the function specified on the main thread of the VM by using `Java.scheduleOnMainThread`
- Line 14 is specifying the class (`android.widget.Toast`) Frida should be using and is assigning it to the variable `toast`.
- Line 15 is calling the function `makeText` that will show the string “This works!” as Toast message in the app.

This proofs that we basically can do whatever we want with the Android app by using Frida, including changing the behavior during runtime by injecting code!

Bypass “Multiple Root Detection Check”

In order to bypass the multiple root detection check, we need to know now the class and function that we should hook into. Usually we would decompile the APK with `jadx-gui` to have a look at the decompiled classes. This was already done for you. In the cloned repo, go to `Kotlin-Shack/sources/org/owasp/mstgkotlin/` and open the class

`MainActivity.java` in your favourite editor. This file contains a method that is combining several root detection checks.

Your task to find the multiple root detection method and it's name in this class!

Once you have the method name upload another Frida script called `bypass-multi-root.js`. Click on the 3 dots and edit the file.

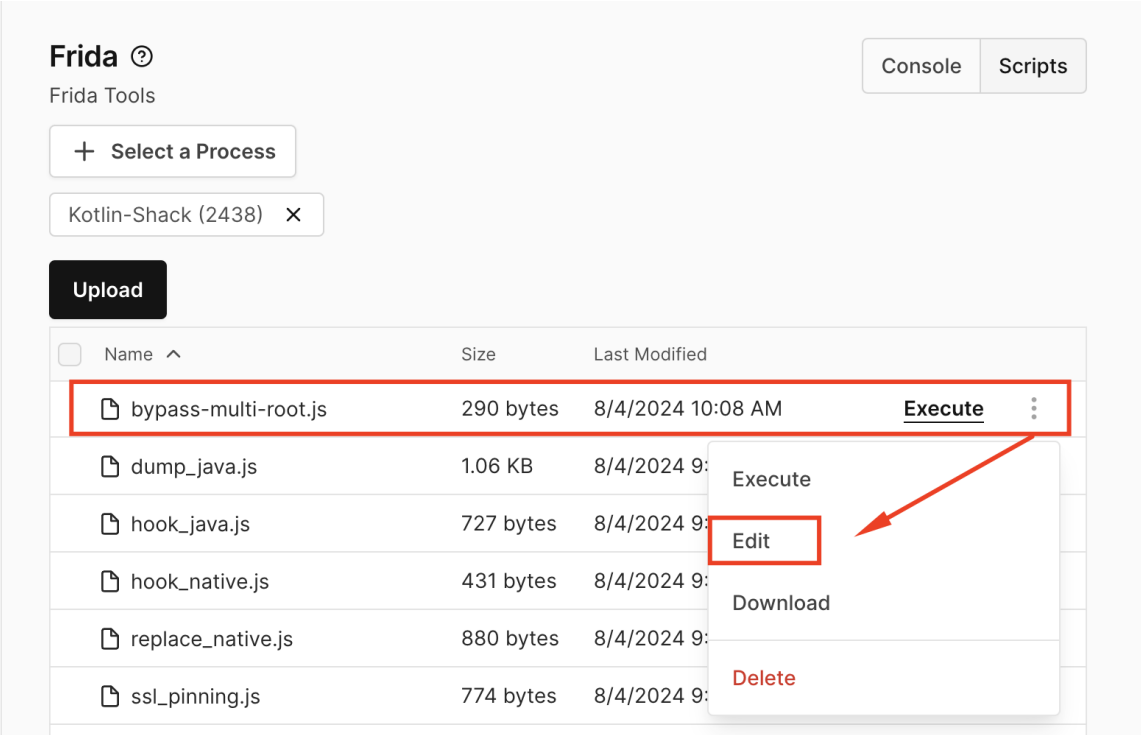

Figure 34: Upload byass-multi-root.js and edit it

In line 7 you will see "FUNCTION". Replace this with the function name you identified for the multiple root detection check.

```
1  setTimeout(function(){
2      Java.perform(function (){
3          console.log("[*] Script loaded")
4
5          var MainActivity = Java.use("org.owasp.mstgkotlin.MainActivity")
6
7          MainActivity.FUNCTION.overload().implementation = function() {
8              console.log("Root detection script triggered")
9
10         }
11
12     });
13
14 });
15
```

Figure 35: Edit byass-multi-root.js

Once done, save it and execute the script and confirm it in the console.

```
[Remote::PID::2196 ]-> %load /data/corellium/frida/scripts/bypass-
   multi-root.js
Are you sure you want to load a new script and discard all current
   state? [y/N]
y
[*] Script loaded
```

Go back to the app, and click on the multiple root detection button. The Frida script will be triggered but with an error:

```
[*] Script loaded
Root detection script triggered
Error: expected an unsigned integer
Error: Implementation for isDeviceRooted expected return value compatible with boolean
    at ne (frida/node_modules/frida-java-bridge/lib/class-factory.js:621)
    at <anonymous> (frida/node_modules/frida-java-bridge/lib/class-factory.js:598)
```

Figure 36: Error in execution of script

Add the `return` command with the right boolean value into the Frida Script after the `console.log` command, so that the App thinks the device is not rooted and that the error vanishes.

You can edit the script as follows. Once you edit it, click save and the script will be automatically reloaded.

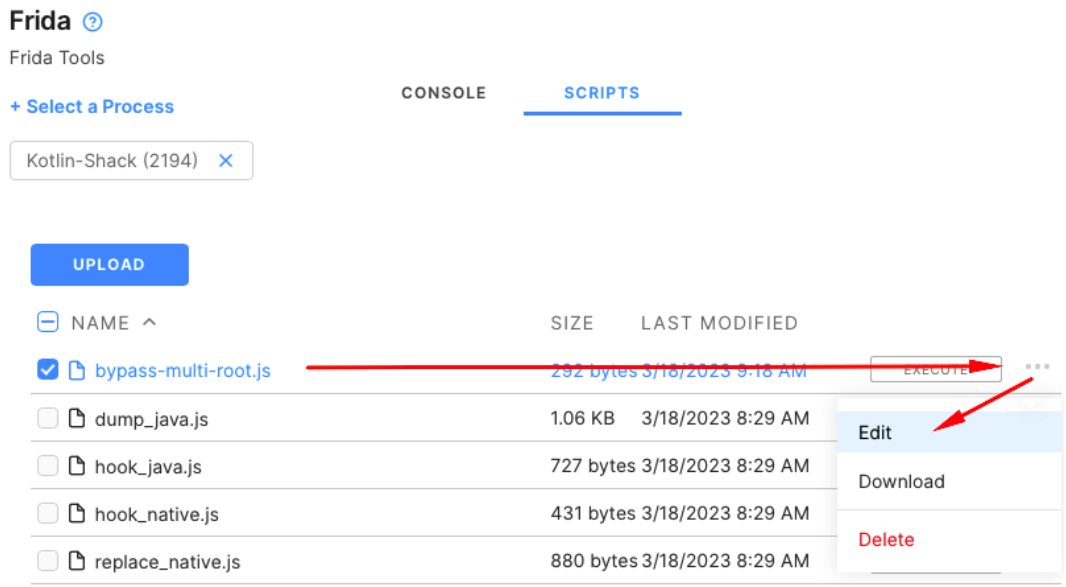

Figure 37: Edit script

Once you added the right return value, the output in the app should look like this:

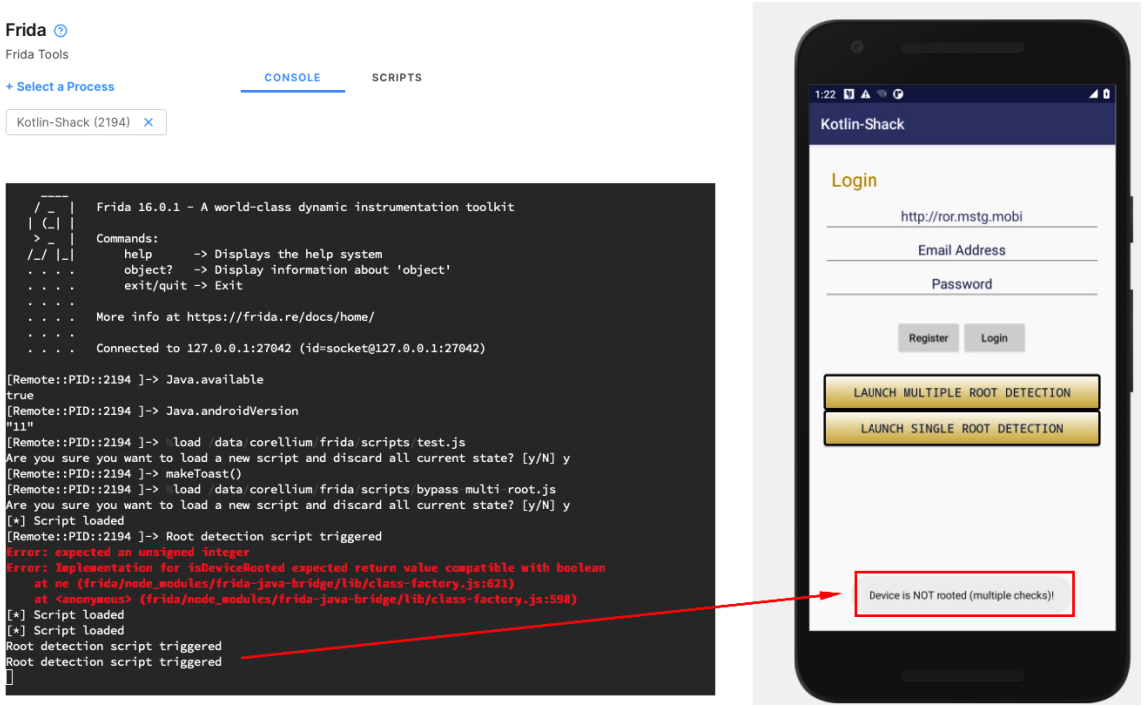

Figure 38: Root detection bypassed

How did we trick now the app into thinking that Android is not rooted? The script contains the following `overload` function:

```
MainActivity.isDeviceRooted.overload().implementation = function()
    {
    console.log("Root detection script triggered")
    return false
}
```

This script is now telling Frida that once the identified function is called from the `MainActivity` class, to overwrite the function with the **`return false`** value. So every time the function is called, it will now only return the boolean value **`false`**.

Congratulations: You bypassed the multiple root detection check with Frida!

References

- Install Frida on Android - https://www.frida.re/docs/
- Frida Example Script - https://www.frida.re/docs/examples/
- Frida Codeshare - https://codeshare.frida.re/browse
- Frida JavaScript API (Java) - https://www.frida.re/docs/javascript-api/#java

Lab - Bypassing Certificate (aka SSL) Pinning

Time to finish lab	15 minutes
App used for this exercise	MSTG-Hands-on.apk

Training Objectives

Bypass different SSL Pinning implementations with Frida

Exercise 1 - Bypassing SSL Pinning with Frida

Preparation - Frida

- Install the app `MSTG Hands-on.apk` in the Android device and start it.

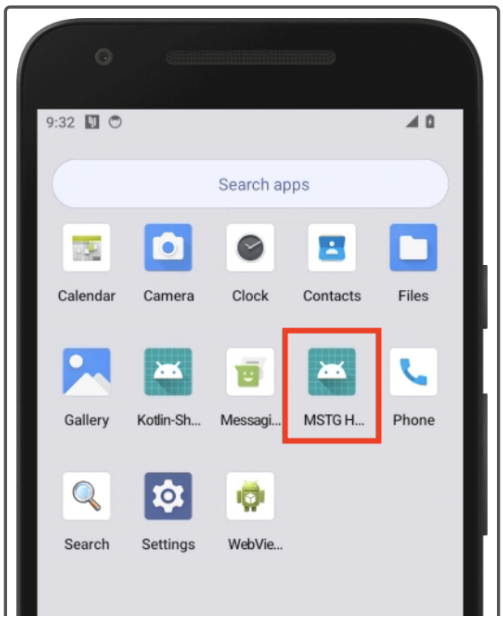

Figure 39: Install Android app

- Open the Frida tab in Corellium.

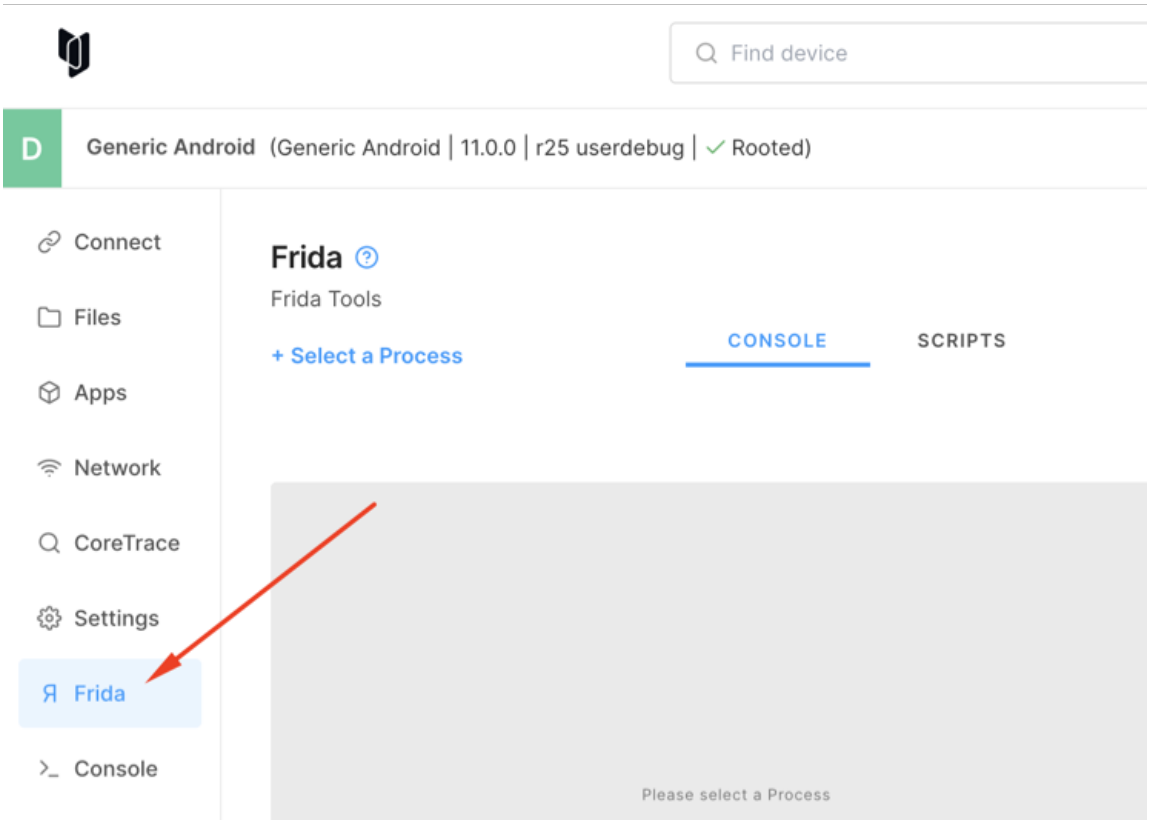

Figure 40: Frida functions in Corellium

- Click on “Select a Process” in Frida and search for “MSTG” and you will find the running app (your process id, the PID, will be different in your case):

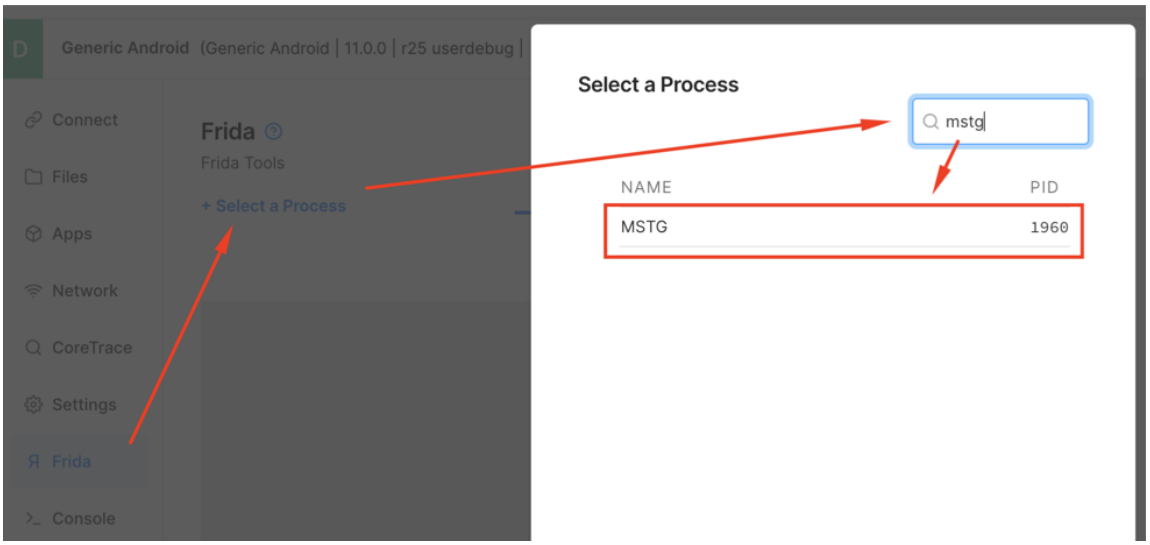

Figure 41: Select a process in Frida

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

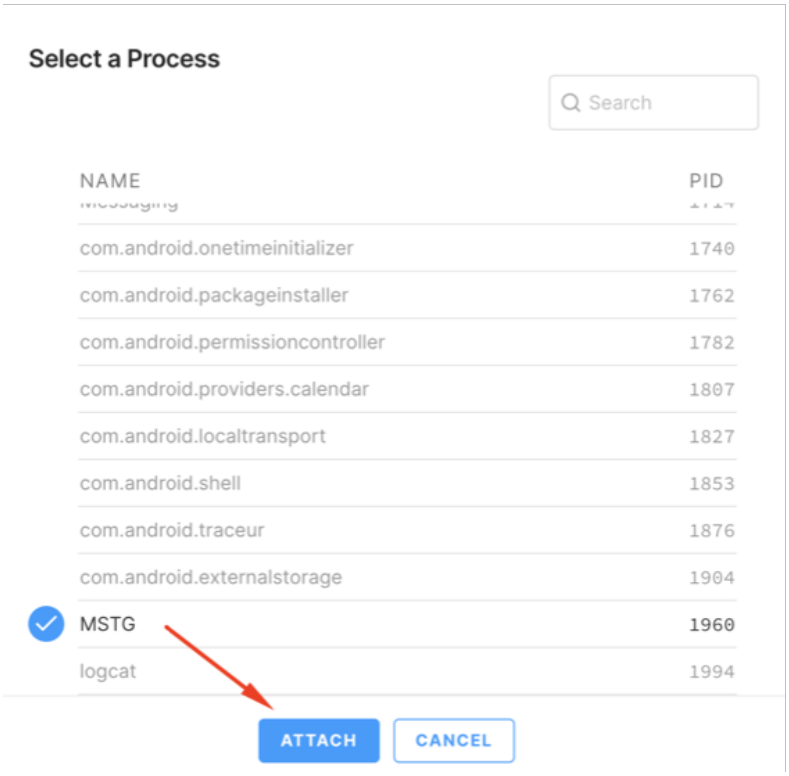

Figure 42: Attache to the MSTG app process

- You can see now the Frida console.

```
     ____
    / _  |   Frida 16.3.3 - A world-class dynamic instrumentation
       toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to 127.0.0.1:27042 (id=socket@127
      .0.0.1:27042)

[Remote::PID::10277 ]->
```

Frida CLI (console) is now connecting to the Frida-Server running on the Android instance and the Frida Core Framework is injecting the Frida-Agent into the specified identifier (`MSTG` or the package name `org.owasp.mstg.android`).

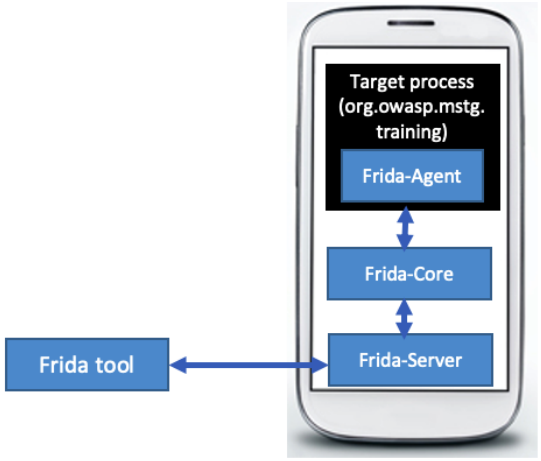

Figure 43: Frida Server

MSTG Android App - TrustManager

- Open the Network Tab and start monitoring.

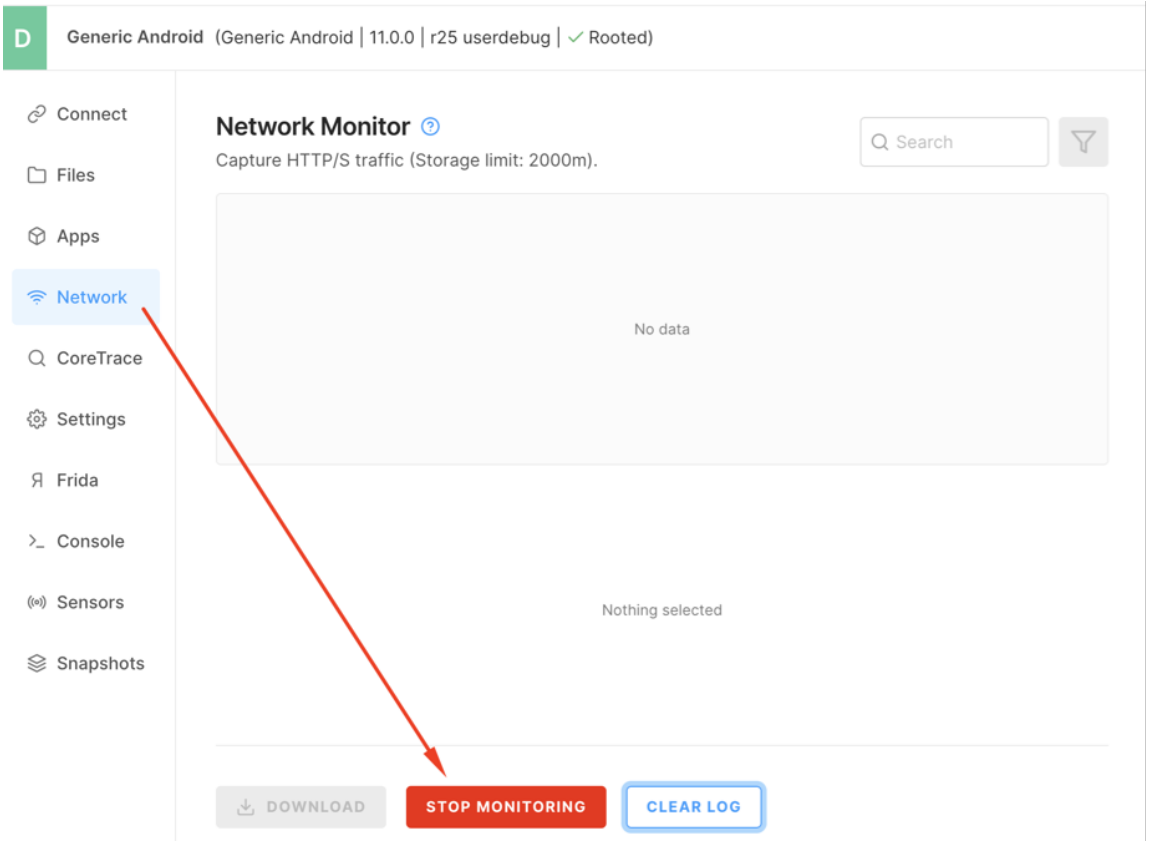

Figure 44: Network monitoring

- Click on “HTTPS Request with SSL Pinning TM”.

Figure 45: SSL Pinning Trust Manager

- Due to SSL Pinning we can not intercept and don't see the request in Network Monitor and it will remain empty, but we can try to bypass it with Frida now!
- Select the Frida script `ssl_pinning.js` and click on execute.

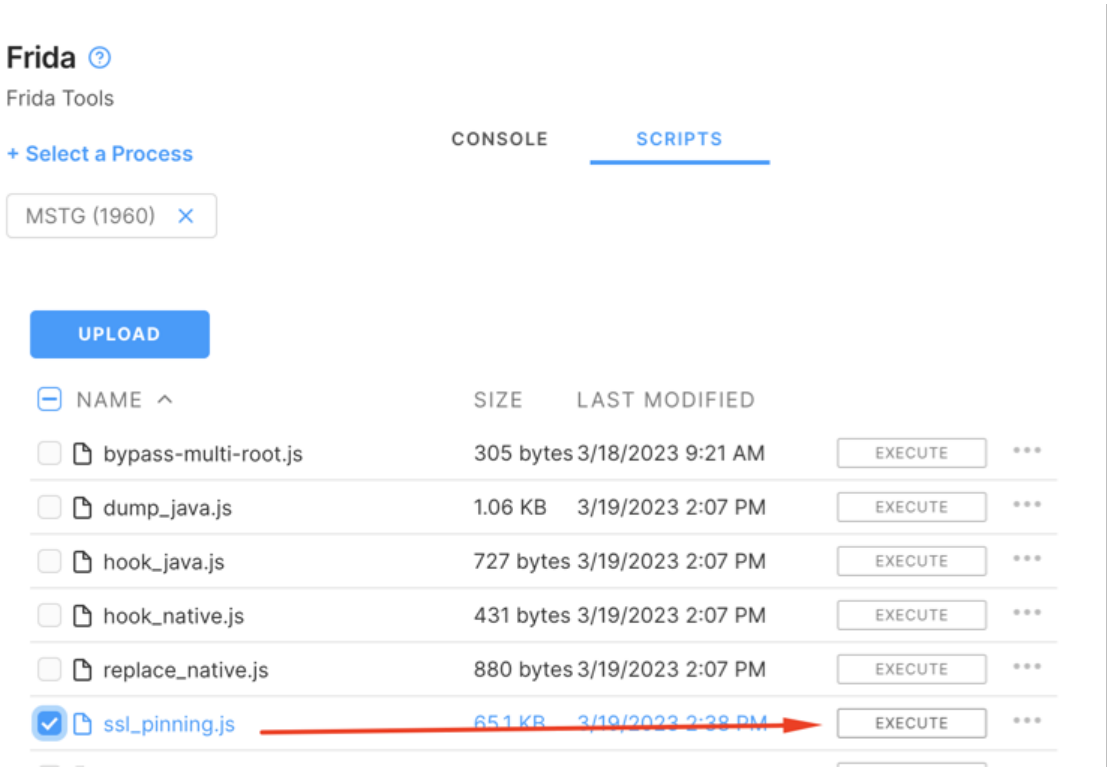

Figure 46: Select ssl_pinning script

- Acknowledge the loading of the script in the “Console” Tab:

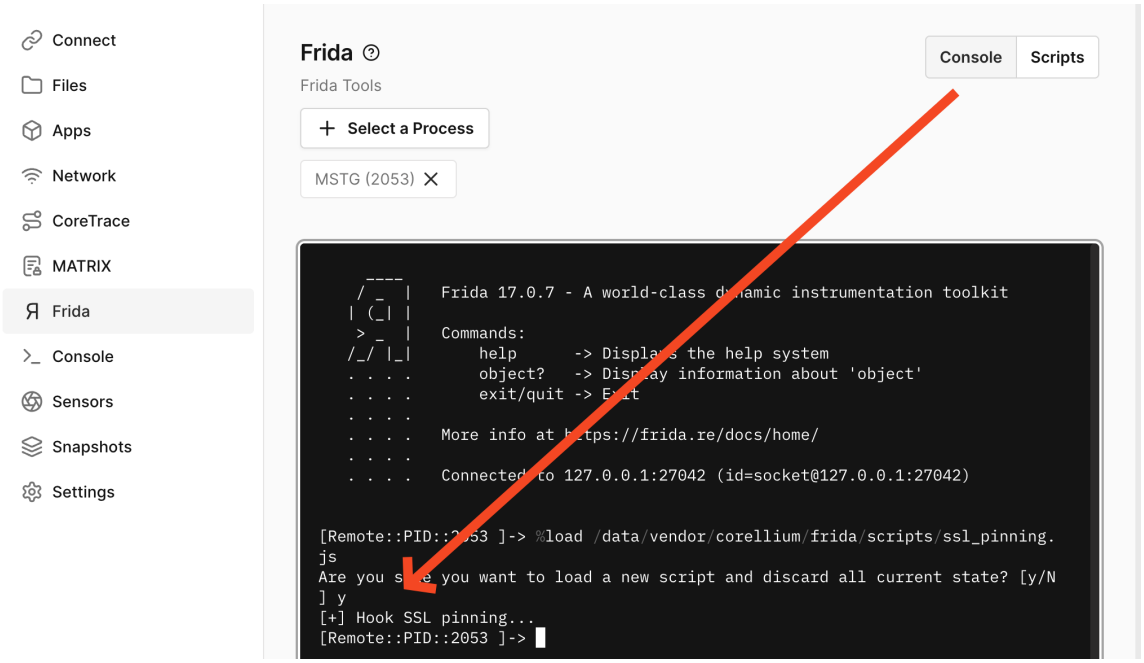

Figure 47: Select ssl_pinning script

- Click on the “Back” button and click again on “HTTPS Request with SSL Pinning TM”.

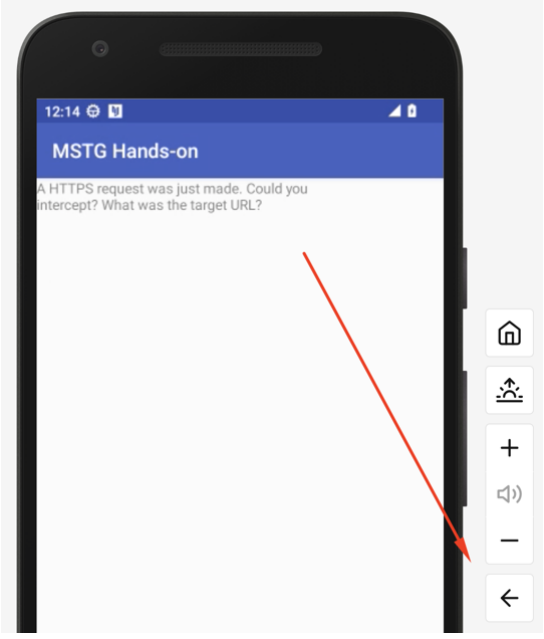

Figure 48: Send request again

- Switch to the networking tab. Which domain was the request sent to?

Exercise 2 - Bypassing SSL Pinning when using OkHttp3

MSTG Hands-on App - OkHttp3

- Go back and click on the button `HTTPS Request with SSL Pinning OkHttp3`
- Verify in Network Monitor if you can intercept the request.
- Your new skill you just learned in Example 1 to bypass SSL Pinning will not be working here, as this function is not using the TrustManager, but a common network library for Android called OkHttp3.
- Find another Frida script on https://codeshare.frida.re/browse that is able to bypass SSLPinning for `OkHTTPv3` and that allows you to intercept the request. Once you found a script (there are quite a few) you can edit the file `ssl_pinning.js`, you can either replace the existing content with your new script that you found and save it, or comment the existing content and append your new script:

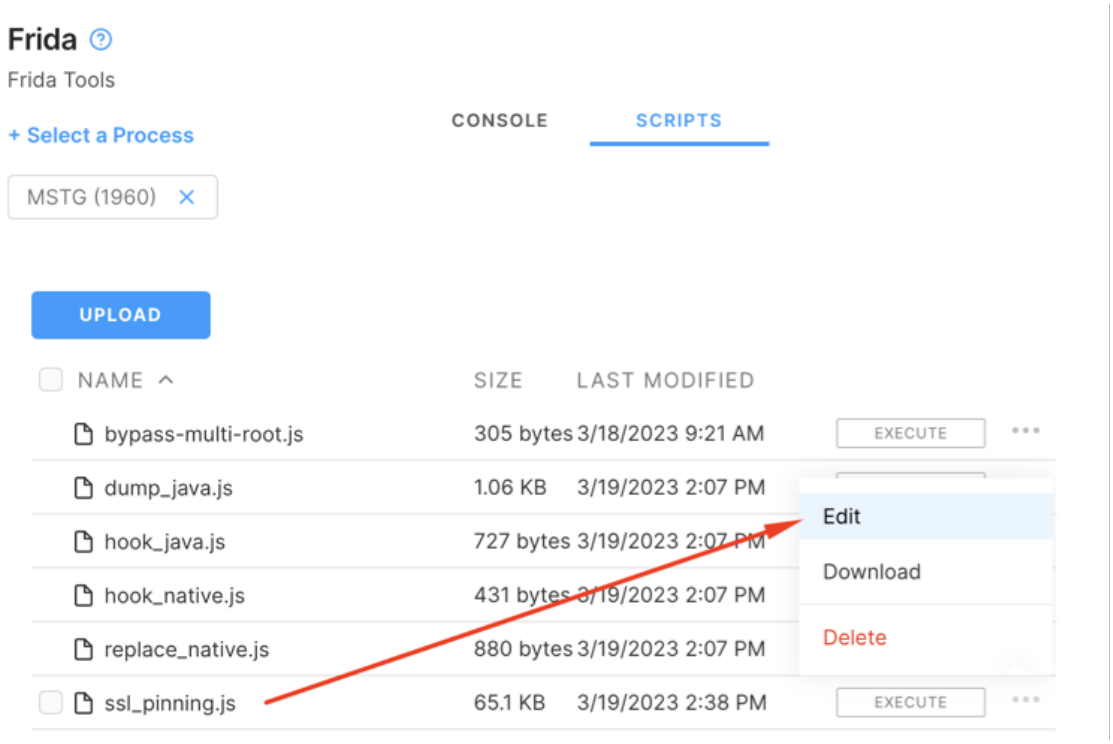

Figure 49: Edit ssl_pinning script

- The script will be automatically reloaded once saved. Go back to the network monitor and click again on the button `HTTPS Request with SSL Pinning OkHttp3`, which domain was the request sent to? If you don't see a HTTP request, try another script.

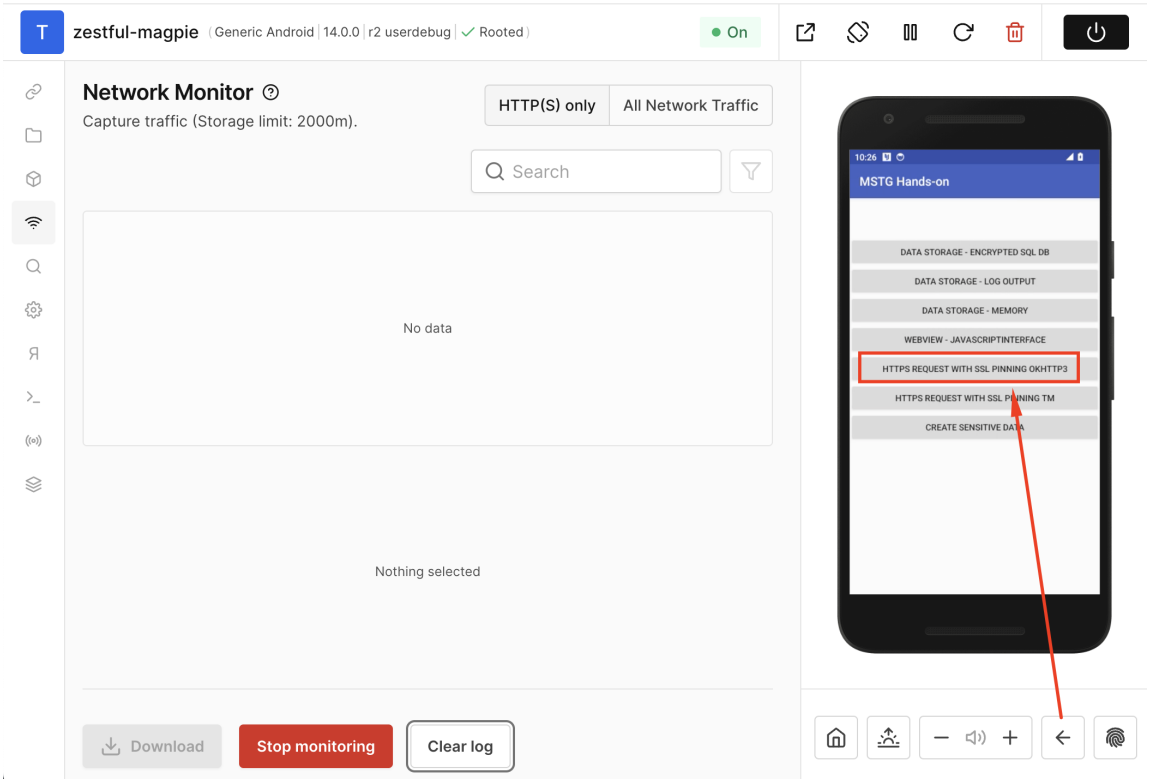

Figure 50: Send request again

> Is it possible to make the SSL Pinning implementation more secure, so it cannot (easily) be bypassed?

References

- Various SSL Unpinning Frida Scripts - https://codeshare.frida.re
- Bypass SSL Pinning on Android in the MSTG - https://mas.owasp.org/MASTG/techniques/android/MASTG-TECH-0012/#bypass-custom-certificate-pinning-statically
- Bypass SSL Pinning in a Flutter App - https://blog.nviso.be/2019/08/13/intercepting-traffic-from-android-flutter-applications/

Lab - Anti-Frida

Time to finish lab	15 minutes
App used for this exercise	Anti-Frida.apk

Training Objectives

Use a Frida script that is bypassing client side controls

Time for completing this lab: 20 min

Tools used in this section

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

Preparation - Installation

Install the app `Anti-Frida.apk` in Corellium by going to the apps menu and clicking on "Install App". Afterwards start the app. You will see the following screen:

Figure 51: Start screen of Anti-Frida app

You can see this app has three different checks available two of them are green. The Frida server should still be running in the background on your Android instance, therefore the "Check Frida Server" will be red. We will explore the checks later in more detail.

Frida CLI (REPL)

- Make sure the app "Anti-Frida" is running in the foreground.
- Click on "Select a Process" in Frida and search for "Frida" and you will find the running app (your process id, the PID, will be different in your case).

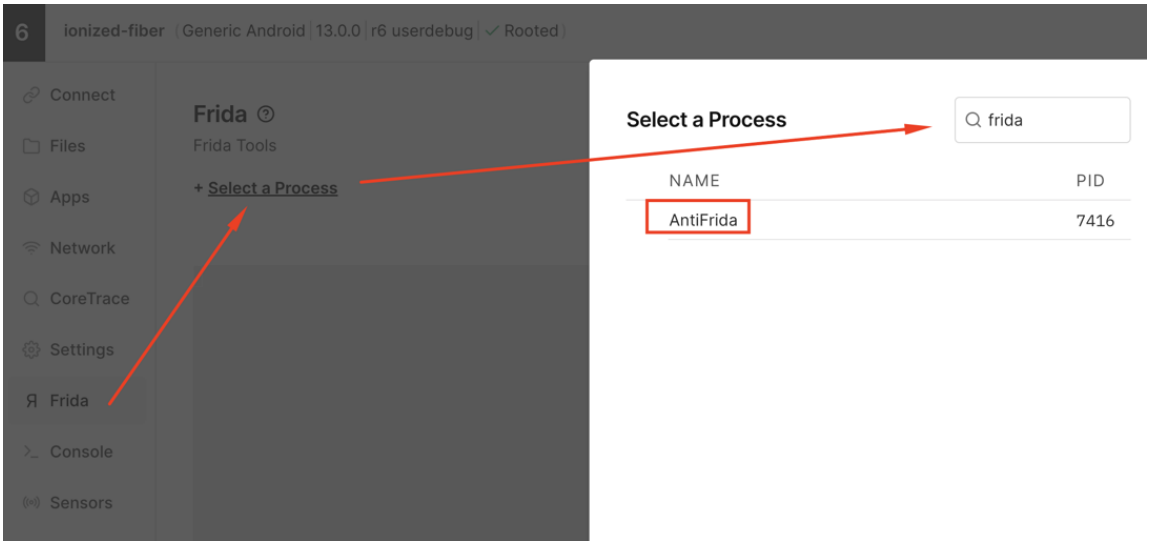

Figure 52: Attach to Anti-Frida app

- Attach to the process and you can see now the Frida console.

```
     ____
    / _  |   Frida 16.3.3 - A world-class dynamic instrumentation
       toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to 127.0.0.1:27042 (id=socket@127
      .0.0.1:27042)

[Remote::PID::11533 ]->
```

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

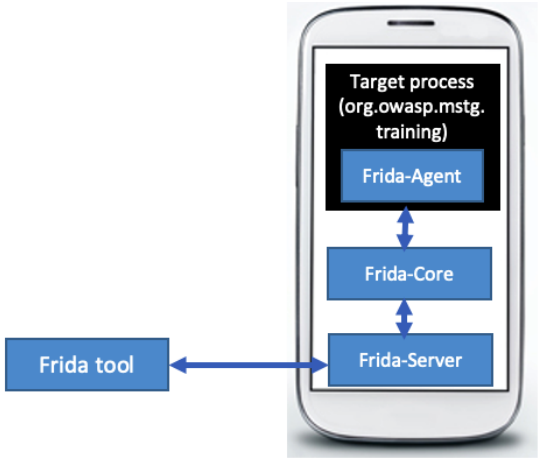

Figure 53: Frida server

If you go back to Corellium in the Anti-Frida app and click the button “Check Frida in memory” the app could now also detect that we were injecting the Frida-Agent into the app and it turns now also red. No worries, we will turn this around soon!

Figure 54: Screen of Anti-Frida app

Your mission is now to hock into this function that is checking the memory for Frida and turn it green again, even though that Frida is injected into the memory.

Identify Classes and Functions

Before we can bypass any of the checks for Frida and inject code, we need to identify the classes and functions we want to manipulate!

In order to identify the classes and methods of the app, it is usually the easiest to decompile the APK to it’s Java Source Code and browse through it.

The app `Anti-Frida.apk` was already decompiled by using the tool `jadx-gui`. Open the folder `Apps/Anti-Frida/sources/org/owasp/mstg/antifirda` which is the directory you downloaded earlier today. Open the file `MainActivity.java` in `VS Code` or your editor of choice.

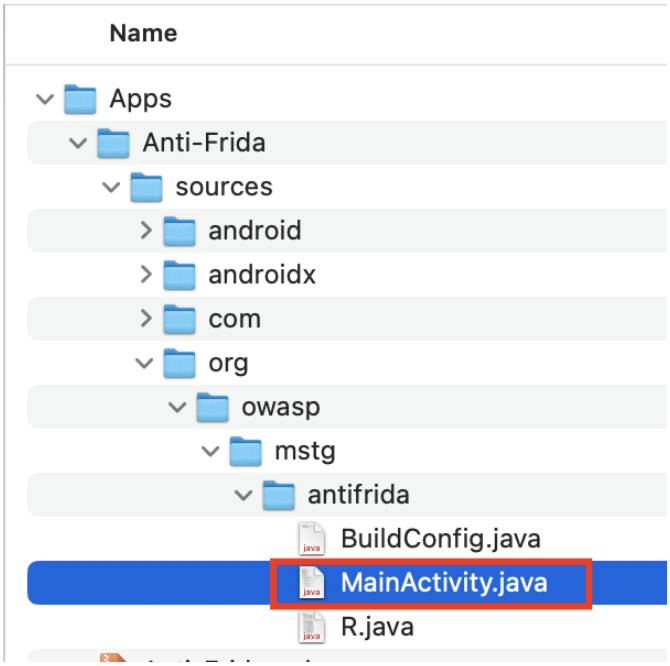

Figure 55: Classes of Anti-Frida app

Go through the source code of `MainActivity.java` and you should be able to see three different functions that are the implementation of the three different checks in the app.

Bypass "Check Frida in Memory"

In the folder `Frida-Scripts` of the preparation pack you downloaded earlier is a script called `frida-bypass-AntiFrida.js`.

Find the function name that is executing the memory check for Frida and replace `FUNCTION-NAME` in line 5 of the Frida-script `frida-bypass-AntiFrida.js` with it.

```
    MainActivity.FUNCTION-NAME.overload().implementation = function
       () {
```

Safe the script and upload `frida-bypass-AntiFrida.js` to Corellium and execute it. Switch to the Console and accept the script execution.

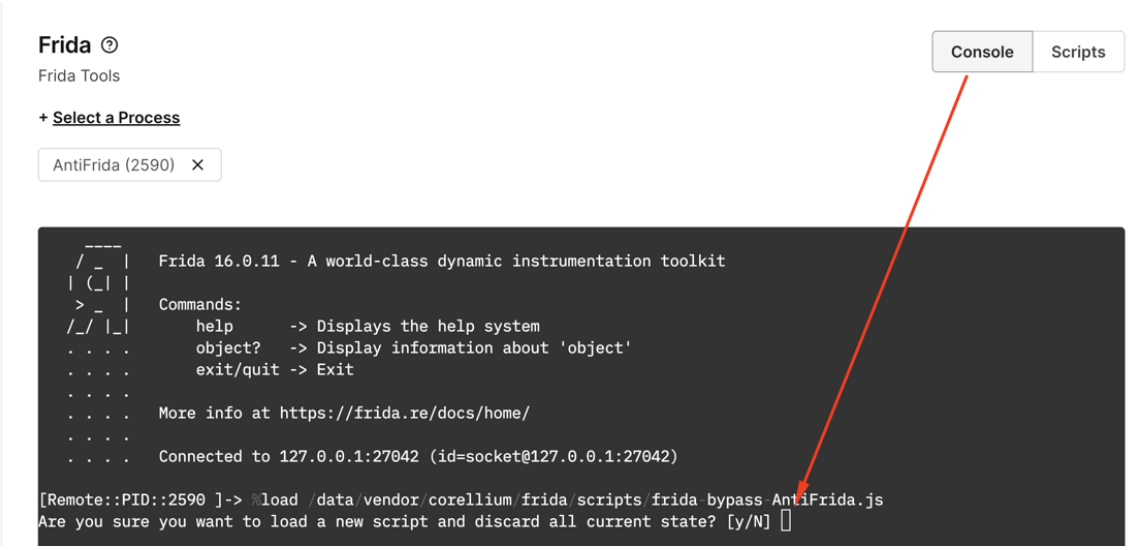

Figure 56: Execute Frida script

Press the button “Check Frida in Memory”. The script is not throwing an error, but the check is still red.

```
[Remote::PID::11533 ]-> [*] Script loaded
[*] Script loaded
```

You’re mission is now to bypass the memory check! So it looks like this again:

Figure 57: Screen of Anti-Frida app

For this, you need to modify the script `frida-bypass-AntiFrida.js`. Check the return value in the script and make the app and Frida work without any errors.

> You can open a 2nd tab in your browser to have the console and scripts screen always open. The display of the Android device can ONLY be open in one tab!

The script is automatically reloaded in the Frida console. Then you can just click the “Check Frida in Memory” button again to see if the bypass is working, there is no need restart Frida or the app. This feature is very handy for creation of Frida scripts.

References

- Install Frida on Android - https://www.frida.re/docs/
- Frida Codeshare - https://codeshare.frida.re/browse
- Frida JavaScript API (Java) - https://www.frida.re/docs/javascript-api/#java
- Frida - https://frida.re
  - `frida-ps` - https://www.frida.re/docs/frida-ps
  - `frida` - https://www.frida.re/docs/frida-cli