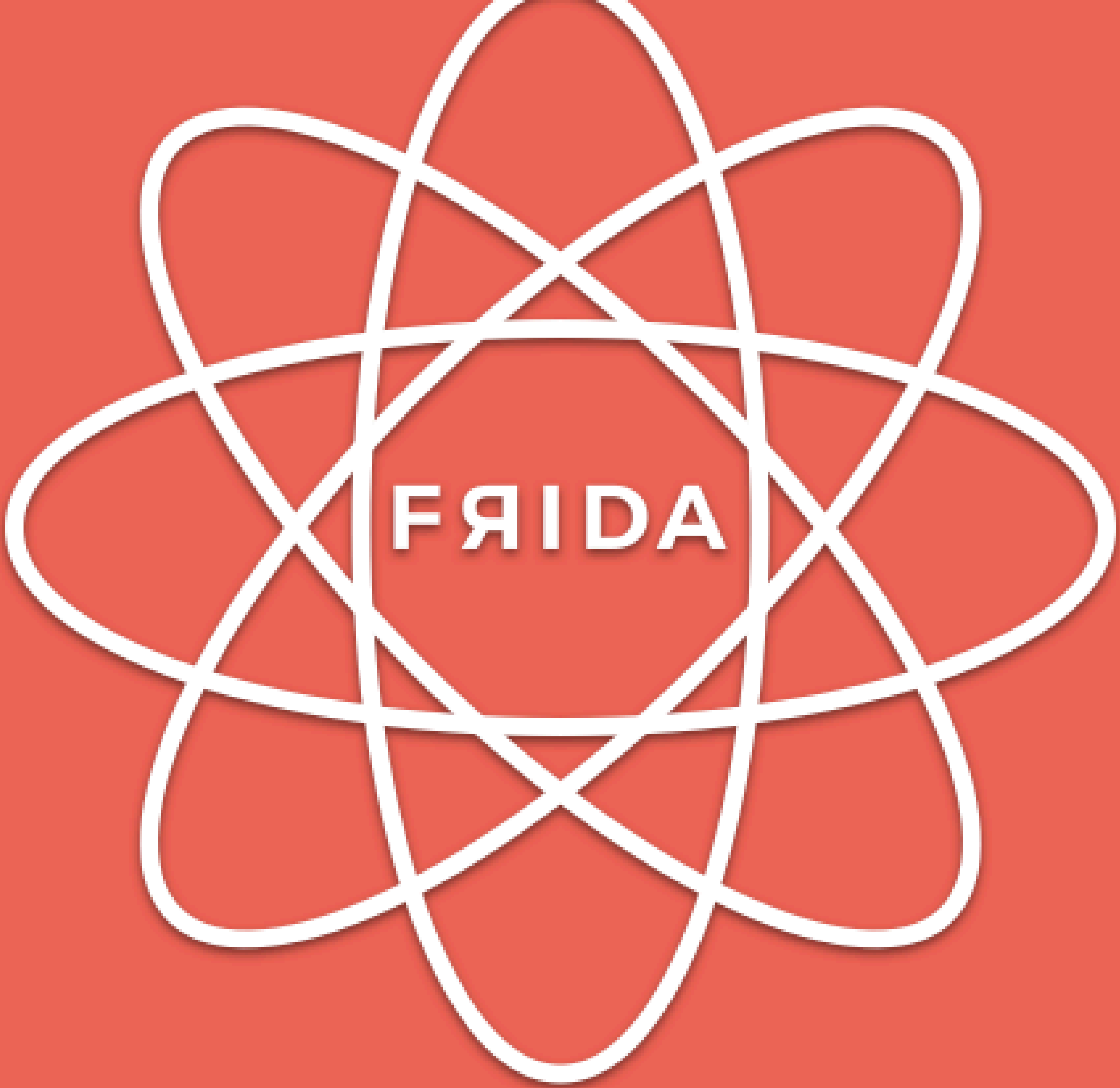
howest
hogeschool

Mobile Security

Week 5: Dynamic tooling - Frida



FRIDA

Static analysis

Static analysis (also known as static code analysis or source code analysis) employs tools to examine program code in search of application coding vulnerabilities, back doors, or other malicious code that might provide hackers access to sensitive corporate data or consumer data.

Static analysis

Search for interesting data (passwords, URLs, API endpoints, API keys, encryption, tokens, ...)

- Check if the application is in debug mode and try to "exploit" it
  - Can I start the postLogin screen without logging in?
- Is the application saving data locally or external?
  - What is actually stored when the data is locally?
  - Can you bypass the data checked online?

Dynamic Analysis

Dynamic Analysis employs tools to examine running programs.

Instead of putting code offline, vulnerabilities and program behavior may be monitored while it’s running, giving insight into how it behaves in the real world.

Dynamic Analysis

There are 2 types of Dynamic Analysis

1. Sniffing traffic
2. Code instrumentation

Dynamic Analysis

Code instrumentation is done with Frida.

Powerful introspection tool that allows to interact with the runtime of a Android process.

Frida

Frida gives the possibility of injecting snippets of JavaScript into native apps on Windows, macOS, GNU/Linux, iOS, watchOS, tvOS, Android, FreeBSD, and QNX.

Frida enables live code injection without source code access

Frida

Frida supports interaction with the Android Java runtime though the Java API.

Frida is able to hook and call both Java and native functions inside the process and its native libraries.

The JavaScript snippets have full access to memory, e.g. to read and/or write any structured data.

Frida

- Instantiate Java objects and call static and non-static class methods (Java API).
- Replace Java method implementations (Java API).
- Enumerate live instances of specific classes by scanning the Java heap (Java API).
- Scan process memory for occurrences of a string (Memory API).
- Intercept native function calls to run your own code at function entry and exit (Interceptor API).

Frida

Frida offers three different modes:

- Injected
- Embedded
- Preloaded

Frida - injected

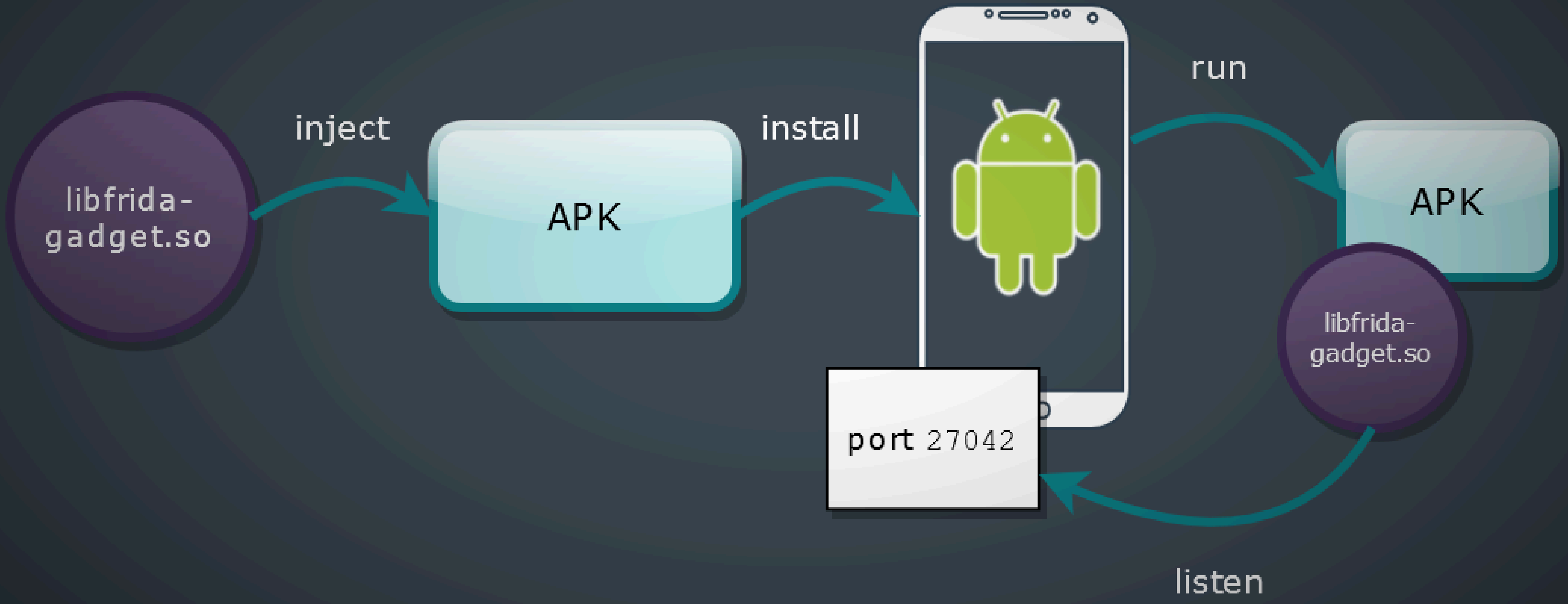
The most common mode and includes the functionality of attaching or hooking into an existing app at startup and embedding additional logic/code. Frida-Core provides the necessary functionality by integrating.

Frida - embedded

This mode is selected for non-jailbroken iOS devices or non-rooted Android devices. In this case, the frida-gadget is integrated into the app to be analysed. It can then be interacted with using Frida-based tools such as `frida-trace`.

Frida - preloaded

This mode includes the autonomous execution of scripts from the file system using frida-gadget without external communication.



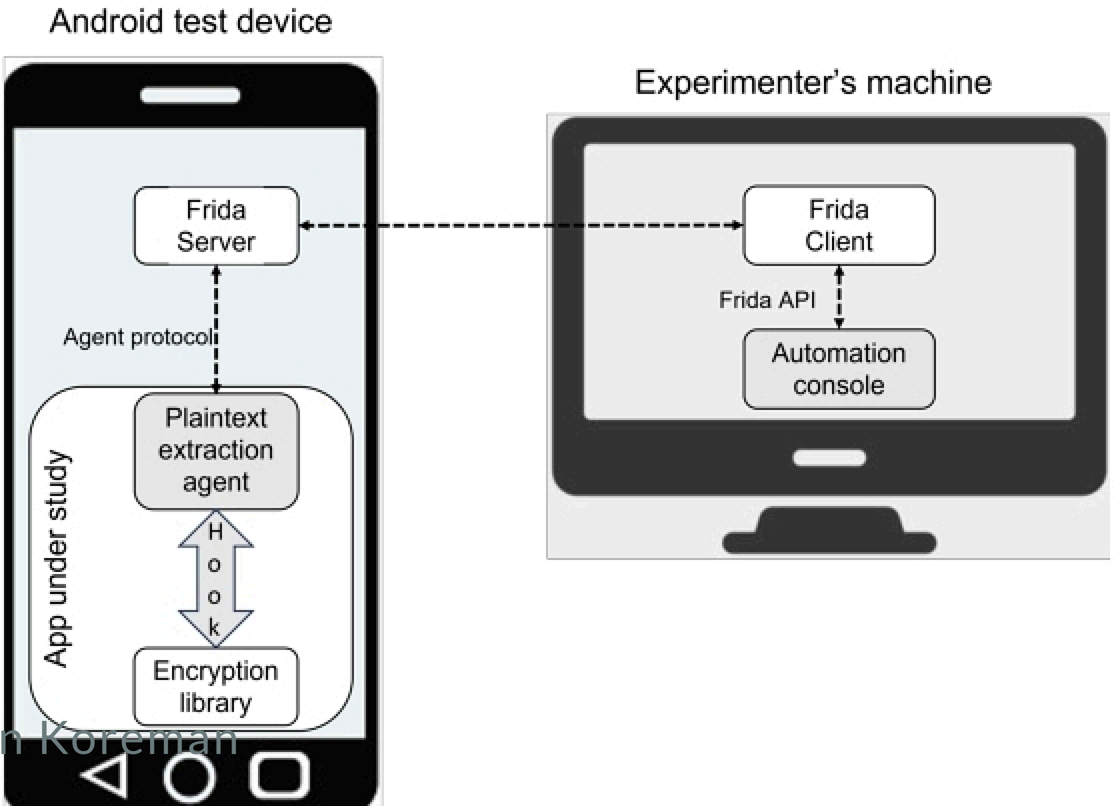
Koen Koreman

Frida tools

Build-in tools provided when installing Frida, that includes the Frida CLI (frida). For example: `frida-ps`, `frida-ls-devices` and `frida-trace`

Frida

Frida is standalone, all you need is to run the frida-server binary from a known location in your target Android device.



Installing Frida

The Android device needs to be rooted before Frida can be fully used.

Installing Frida

1. Download Frida-server from https://github.com/frida/frida/releases
    1. Make sure to choose the correct version: `android-x86_64.xz` for the emulator
2. uncompress the file and copy it to the device using adb

Installing Frida

```
$ unxz frida-server.xz

$ adb root # might be required
$ adb push frida-server /data/local/tmp/
$ adb shell "chmod 755 /data/local/tmp/frida-server"
$ adb shell "/data/local/tmp/frida-server &"
```

Frida is now running on the device.

Installing Frida

On the host frida is a Python library.

```
pip3 install frida-tools
```

```
pip3 install frida
```

Remember best practices from SACA: create a `venv`

Installing Frida

Frida can be tested using the following command:

`frida-ps -U`

This shows all the running processes on the device (like the `ps` command in Linux)

```
  PID NAME
...
15901 com.koenk.mobilesecurity.labs
13194 com.android.chrome
13282 com.twitter.android
...
```

Frida commands `frida-ls-devices`

List all connected devices using Frida

```
$ frida-ls-devices

# example output

Id                                        Type    Name
----------------------------------------  ------  ----------------
local                                     local   Local System
0216027d1d6d3a03                          tether  Samsung SM-G920F
1d07b5f6a7a72552aca8ab0e6b706f3f3958f63e  tether  iOS Device
tcp                                       remote  Local TCP
```

Frida commands - `frida-ps`

Connect Frida to an device over USB and list running processes

```
$ frida-ps -U

# List running applications
$ frida-ps -Ua

# List installed applications
$ frida-ps -Uai

# Connect Frida to the specific device
$ frida-ps -D 0216027d1d6d3a03
```

howest
hogeschool

Demonstration

Details demonstration

1. Show the installer and installation process
    1. `adb shell ls -l /data/local/tmp`
2. Start up Frida using `adb shell "/data/local/tmp/frida-server &"`
3. Use basic Frida commands
    1. `frida-ls-devices`
    2. `frida-ps -U`
    3. `frida-ps -Ua`
    4. `frida-ps -Uai`

Frida commands - `frida-trace`

`frida-trace` is a tool for dynamically tracing function calls.

It automates the creation of JavaScript hooks for tracing method and function calls in Android applications, both Java and native.

Valuable for inspecting app behavior, debugging, reverse engineering, and security testing.

Frida commands - `frida-trace`

The core syntax for targeting an Android app is:

```
frida-trace -U -j '<class>!<method>' -N <app_id>
```

`-U` : Target USB device.
`-N <app_id>` : Specify the unique application identifier.
`-j` : Specify Java methods using patterns or wildcards (e.g., *!*submit* matches all methods containing "submit").

Can trace individual functions, all functions in a class, or use wildcards to trace many methods at once.

Frida commands - `frida-trace`

Trace all methods named checkPassword in any class:

```
frida-trace -U -j '*!checkPassword' -N com.example.app
```

Trace all methods in the class com.bank.auth.Login:

```
frida-trace -U -j 'com.bank.auth.Login!*' -N com.example.app
```

Trace every method whose name contains "submit":

```
frida-trace -U -j '*!*submit*' -N com.example.app
```

Trace the specific method doLogin in com.bank.auth.Login:

```
frida-trace -U -j 'com.bank.auth.Login!doLogin' -N com.example.app
```

Frida commands - `frida-trace`

Example `frida-trace` on application.

1. Find the application PID `frida-ps -Ua`

```
 PID  Name              Identifier
----  ----------------  ---------------------------------------
2661  Camera            com.android.camera2
3151  Chrome            com.android.chrome
6482  Clock             com.google.android.deskclock
1590  Google            com.google.android.googlequicksearchbox
7024  Jetchat           com.example.compose.jetchat
3081  Settings          com.android.settings
...
```

Frida commands - `frida-trace`

2. Find the classes and methods in that package

```
frida-trace -U -j '*profile*!*' -p 7024
```

```
Started tracing 25 functions.
 10425 ms  ProfileScreenState.getUserId()
 10425 ms  <= "me"
 10452 ms  ProfileScreenState.getPhoto()
 10453 ms  <= "<instance: java.lang.Integer>"
 10454 ms  ProfileScreenState.getUserId()
 10454 ms  <= "me"
 10456 ms  ProfileScreenState.getPhoto()
 10456 ms  <= "<instance: java.lang.Integer>"
...
```

Frida commands - `frida-trace`

```
...
 18297 ms  ProfileScreenState.getTwitter()
 18297 ms  <= "twitter.com/taylorbrookscodes"
 18299 ms  ProfileScreenState.getTimeZone()
 18299 ms  <= "12:25 AM local time (Eastern Daylight Time)"
 18299 ms  ProfileScreenState.getTimeZone()
 18299 ms  <= "12:25 AM local time (Eastern Daylight Time)"
 20815 ms  ProfileScreenState.isMe()
 20815 ms  <= false
...
```

Frida commands - `frida-trace`

Hooks into selected methods at runtime, logging entry/exit and parameters.

Generates separate JavaScript handler files (e.g., /.js) with onEnter and onLeave callbacks.

These handlers can be edited to modify arguments, return values, or inject custom logic for advanced manipulation.

Real-time output is provided in the terminal during app usage for monitored methods.

Frida commands - `frida-trace`

Advanced Options and Scenarios

Use -f to spawn the application, or attach to a running process with `-p <pid>`

`-I/-i` options target native or JNI functions for deeper tracing (e.g., cryptographic libraries).

Useful for rapidly identifying sensitive logic, bypassing validation checks, or extracting secrets from protected code.

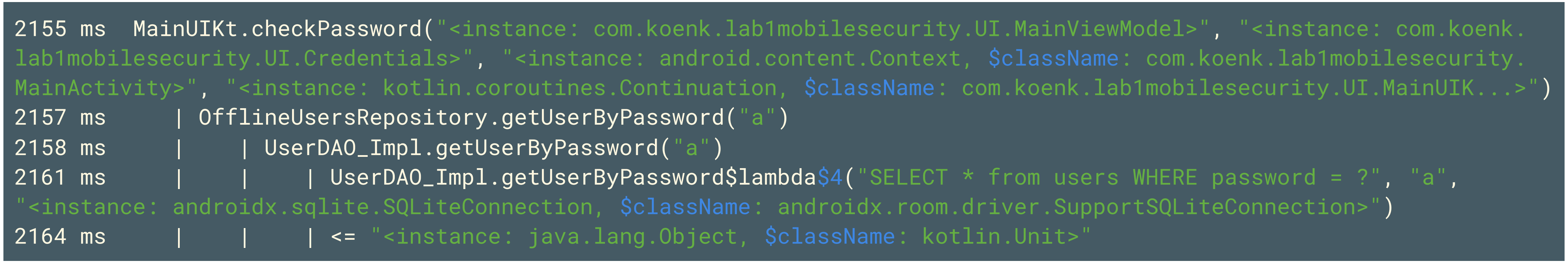
Complex scenarios allow modification of return values (e.g., forcing a validation method to always return true by editing the handler JS).

Demonstration

Details demonstration Lab1

1. Show running process `frida-ps -Ua` - with lab1 open
2. In the application notice `Password` let's see if there is some functionality with Password
3. `frida-trace -U -j '*!*Password* -p <pid>`
    1. `'*!*Password*'` ➡️ AllClasses!ContainingTheWordPassword

Details demonstration Lab1



```
2155 ms  MainUIKt.checkPassword("<instance: com.koenk.lab1mobilesecurity.UI.MainViewModel>", "<instance: com.koenk.
lab1mobilesecurity.UI.Credentials>", "<instance: android.content.Context, $className: com.koenk.lab1mobilesecurity.
MainActivity>", "<instance: kotlin.coroutines.Continuation, $className: com.koenk.lab1mobilesecurity.UI.MainUIK...>")
2157 ms     | OfflineUsersRepository.getUserByPassword("a")
2158 ms     |    | UserDAO_Impl.getUserByPassword("a")
2161 ms     |    |    | UserDAO_Impl.getUserByPassword$lambda$4("SELECT * from users WHERE password = ?", "a",
"<instance: androidx.sqlite.SQLiteConnection, $className: androidx.room.driver.SupportSQLiteConnection>")
2164 ms     |    |    | <= "<instance: java.lang.Object, $className: kotlin.Unit>"
```

- MainUI ➡ checkPassword ➡ MainViewModel.?
    - ➡ OfflineUsersRepository.getUserByPassword("a")
- UserDAO_Impl.getUserByPassword("a") ➡
    - UserDAO_Impl.getUserByPassword$lambda$4("SELECT * from users WHERE password = ?", "a",

Details demonstration Frida-Demo

1. Show running process `frida-ps -Ua` - with lab1 open
2. In the application notice `secret` let's see if there is some functionality with secret
3. `frida-trace -U -j '*!*secret -j '*!*Secret* -p <pid>`

Details demonstration Frida-Demo

```
Started tracing 52 functions. Web UI available at http://localhost:40387/
           /* TID 0x115f */
  2210 ms  MainActivityKt.checkSecret("<instance: com.koenk.fridademo.MainViewModel>", "<instance: android.content.
  Context, $className: com.koenk.fridademo.MainActivity>", "")
  2211 ms     | MainViewModel.checkSecret("KoenK")
  2211 ms     | <= false
```

The function MainViewModel.checkSecret that returns a boolean

- Class: MainViewModel
- Function: checkSecret that needs a string input

Frida

Frida also provides a Java API, which is especially helpful for dealing with Android apps.
It lets you work with Java/Kotlin classes and objects directly.

Frida

This script overwrites the onResume function in the Activity class:

```
Java.perform(function () {
    var Activity = Java.use("android.app.Activity");
    Activity.onResume.implementation = function () {
        console.log("[*] onResume() got called!");
        this.onResume();
    };
});
```

It calls `Java.perform` to make sure that the code gets executed in the context of the Java VM.

Frida

It instantiates a wrapper for the `android.app.Activity` class via `Java.use` and overwrites the `onResume` function.

The new `onResume` function implementation prints a notice to the console and calls the original `onResume` method by invoking `this.onResume` every time an activity is resumed in the app.

Frida - scripting

Frida injects JavaScript into processes.

Run the Python script that connects to the device and loads the js file

```
device = frida.get_usb_device()

with open("script.js") as f:
    script = session.create_script(f.read())
script.load()
```

Frida - scripting

Opening an application and attach it to Frida from Python

```python
pid = device.spawn(["com.koenk.fridainject"])
device.resume(pid)
time.sleep(1)
session = device.attach(pid)
```

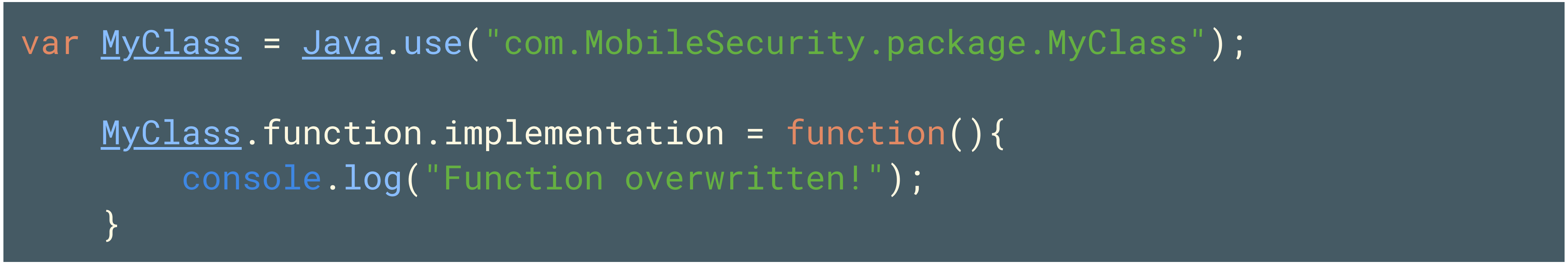
Frida - scripting

The JavaScript file

```
console.log("Script loaded successfully ");
Java.perform(function () {
    console.log("Starting implementation override.");
```

Frida - scripting

Overwriting functions



```
var MyClass = Java.use("com.MobileSecurity.package.MyClass");

    MyClass.function.implementation = function(){
        console.log("Function overwritten!");
    }
```

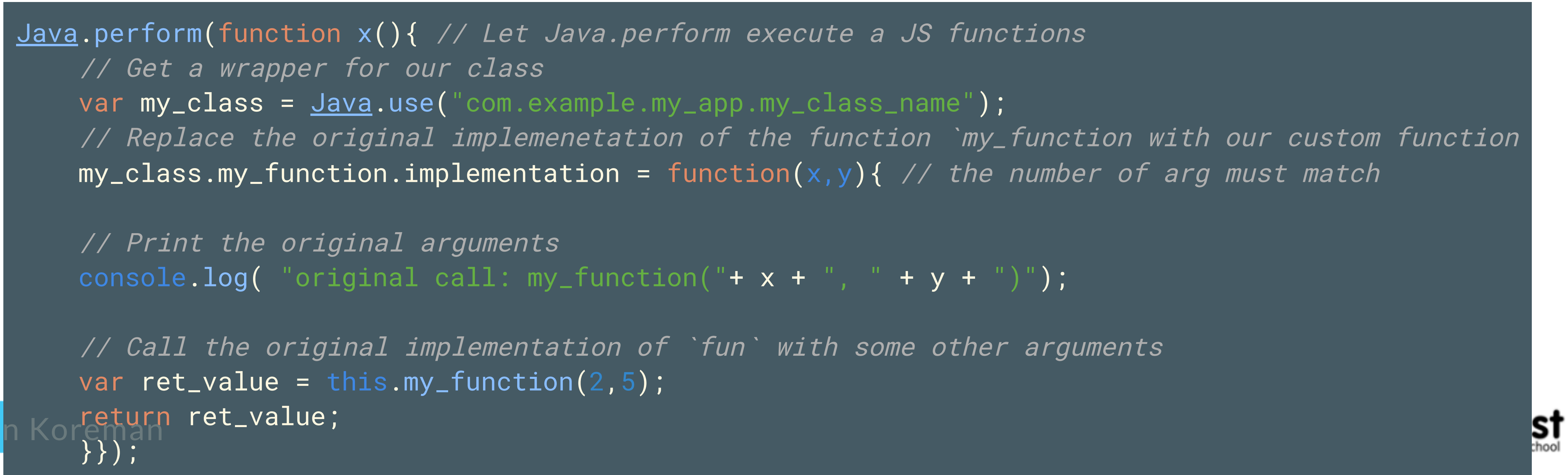
Frida - scripting - example

`frida -U -l script.js -p <pid>` ➡ use frida with the JS on the device

```
     ____
    / _  |   Frida 17.2.17 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .   Connected to Android Emulator 5554 (id=emulator-5554)
Attaching...
JavaScript loaded successfully
[Android Emulator 5554::PID::14036 ]-> Starting implementation override.
```

Frida - scripting - example

The JS part of the Frida scripting changes the applications behavior. In this part the names of classes and functions of the desired behavior should be known.



```
Java.perform(function x(){ // Let Java.perform execute a JS functions
    // Get a wrapper for our class
    var my_class = Java.use("com.example.my_app.my_class_name");
    // Replace the original implemenetation of the function `my_function with our custom function
    my_class.my_function.implementation = function(x,y){ // the number of arg must match

    // Print the original arguments
    console.log( "original call: my_function("+ x + ", " + y + ")");

    // Call the original implementation of `fun` with some other arguments
    var ret_value = this.my_function(2,5);
    return ret_value;
    }});
```

Frida - scripting - example

```python
# Extend part I python script with this functionality
def my_message_handler(message , payload): # This functions receives data from the JS script
    print message
    print payload

    if message["type"] == "send":
        data = message["payload"].split(":")[1].strip()
        script.post({"my_data": data}) # Send JSON object to the JS script

script.on("message" , my_message_handler) # Register our handler to be called
script.load()
```

Frida - scripting - example

```
// JS code
send(string_to_send); // send data to python code
recv(function (received_json_object) {
    string_to_recv = received_json_object.my_data
 }).wait(); // Receive data from the python code
```

Demonstration

Details demonstration Frida-Demo

1. Show running process `frida-ps -Ua` - with lab1 open
2. In the application notice `secret` let's see if there is some functionality with secret
3. `frida-trace -U -j '*!*secret -j '*!*Secret* -p <pid>`

Details demonstration Frida-Demo

```
Started tracing 52 functions. Web UI available at http://localhost:40387/
           /* TID 0x115f */
  2210 ms  MainActivityKt.checkSecret("<instance: com.koenk.fridademo.MainViewModel>", "<instance: android.content.
  Context, $className: com.koenk.fridademo.MainActivity>", "")
  2211 ms     | MainViewModel.checkSecret("KoenK")
  2211 ms     | <= false
```

The function MainViewModel.checkSecret that returns a boolean

- Class: MainViewModel
- Function: checkSecret that needs a string input

Details demonstration Frida-Demo

Start a Frida API console: `frida -U -p <pid`

```
setTimeout(() => { //Quick timeout so the Java environment is loaded
    Java.perform(() => { //Execute the implementation of the Java code
        try {
            //Create an instance of the needed class
            var MainViewModel = Java.use("com.koenk.fridademo.MainViewModel");
        } catch (e) {
            console.error('Exception caught:', e.message);
        }
        //Overwriting the original function so it always returns true
        MainViewModel.checkSecret.implementation = function(code){
            return true;
        }
    }); }, 0);
```

Frida - API

`frida -U -p [pid]`

```
frida -U -p 11275
     ____
    / _  |   Frida 17.2.17 - 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 Android Emulator 5554 (id=emulator-5554)

[Android Emulator 5554::PID::11275 ]->
```

Frida - API

```
Java.androidVersion
```

```
[Android Emulator 5554::PID::11275 ]-> Java.androidVersion "16"
```

```
Java.perform(() => console.log('Hello world!'))
```

```
[Android Emulator 5554::PID::11275 ]-> Java.perform(() => console.log('Hello world!'))
Hello world!
```

Frida - API

```
Java.enumerateMethods('');
```

```
Android Emulator 5554::PID::11275 ]-> Java.enumerateMethods('*jetchat.conversation*!*message*');
[
    {
        "classes": [
            {
                "methods": [
                    "messageFormatter"
                ],
                "name": "com.example.compose.jetchat.conversation.MessageFormatterKt"
            }
        ],
        "loader": "<instance: java.lang.ClassLoader, $className: dalvik.system.PathClassLoader>"
    }
]
```

Frida - API

- `Java.use("android.util.Log")` - Uses the provided class (in this case: android.util.Log)
- `my_method.implementation` - Overrides the default implementation
- `my_method.overload` - When polymorphism is used, this can be useful
- `Java.perform` - Used to execute Javascript code in the v8 engine on the main thread.

Frida - API

- `Java.choose` - Gets a wrapper to an already existing class instance!

```
Java.choose("com.example.my_app.my_wanted_class", {
    onMatch: function (instance) {
        console.log("Found instance: " + instance);
        console.log("Result of function: " + instance.some_function());
    },
    onComplete: function () { }

});
```

Frida - API

Thread Observation using `attachThreadObserver` allows researchers to monitor thread creation, termination, and renaming in real time, addressing a longstanding challenge in dynamic analysis.

```
const observer = Process.attachThreadObserver({
  onAdded(thread) { // Handle new thread
  },

  onRemoved(thread) {  // Handle thread termination
  },

  onRenamed(thread, previousName) { // Handle thread renaming
  }
});
```

howest
hogeschool

Frida scripts

Frida scripts

Frida has many scripts that helps pentesting applications. These scripts can be found here:

https://codeshare.frida.re/

Frida scripts

fridantiroot

This is a universal script for bypassing root detecting it contains all possible checks that are used in applications to detect root

Code: https://codeshare.frida.re/@dzonerzy/fridantiroot/
CMD:

```
frida -U --codeshare dzonerzy/fridantiroot -f APP_NAME --no-pause
```

Frida scripts

anti-frida-bypass

This is a script that bypasses Frida detection.

Code: https://codeshare.frida.re/@enovella/anti-frida-bypass/

CMD:
```
frida -U --codeshare dzonerzy/anti-frida-bypass -f APP_NAME
```

Frida scripts

aesinfo

This is a script that detects the used encryption techniques.

Code: https://codeshare.frida.re/@dzonerzy/aesinfo/

CMD:
```
frida -U --codeshare dzonerzy/aesinfo -f APP_NAME
```

Frida scripts

Due to the upgrade to Frida17, not all scripts work out of the box anymore, more information on these release notes:

https://frida.re/news/2025/05/17/frida-17-0-0-released/

Frida

Blogpost about usuage of Frida in a real life example:

https://labs.cognisys.group/posts/Breaking-Custom-Ecryption-Using-Frida-Mobile-Application-pentesting/

Lab time

See Lab 5 on LEHO