Traffic analysis

This module captures and analyses network traffic generated by a targeted Android application in order to detect the leakage of personally identifiable information (PII).

Getting Started

Prerequisites:

An internal network between the host machine and the emulator needs to be created. Follow this procedure or run the script config.sh as follows:

```
sh config.sh
```

Installing:

1. Login to Docker Hub:

```
docker login
```

2. Pull the image of traffic analysis:

```
docker pull cliip/platform:traffic
```

Configuration:

1. Set up the **executor.config** file (See Figure 1 for details):

```
phase-one_timeout = 350
permissions = False
reboot = False
[testing_env]
testing_server_ip = 192.168.1.50   # Server control IP
testing_server_port = 4000   # Server control port
testing_terminal = 4fa3f19c  # Mobile device identifier or IP

[testing]
phase-one_timeout = 350 # Time in seconds that traffic will be captured in Phase-One
permissions = False  # If True, all requested permissions will be granted before the
test
reboot = False  # If True, the mobile device will reboot before the test
phase-two_timeout = 30 # Time in seconds that traffic will be captured in Phase-One
monkey = True  # If True, a monkey will generate pseudo-random events/inputs
testing_label = int_transfers   # Any label to identify the test

[rabbitmq]
username = privapp   #RabbitMQ username
password = ElCieloEsAzul    #RabbitMQ password
server_ip = 192.168.1.50   #RabbitMQ server IP
queue = testing  #RabbitMQ queue name
exchange = int_transfer   #RabbitMQ exchange name
```

```
[base]
base_path = /privapp/app/  #Base directory containing the project files
results_output = /privapp/app/logs/    #Directory to save results
```

2. Set up the file **info.device** by defining the PIIs to be searched:

```
<PII type 1>: <PII 1>;<PII 2>;<PII 3>;...
<PII type 2>: <PII 4>
<PII type 3>: <PII 5>;<PII 6>
...
```

The following example defines the type "aaid" and two specific values to look for in the traffic:

```
  aaid: 784df00d-95ca-47f7-99cf-9dfe3a2ddcb2;2b00b940-8163-4364-b65b-a056353f24a5:
```

Run:

The following command will run the container:

```
docker run --network host -e LANG=C.UTF-8 cliip/platform:traffic
```

After the command has finished the control server will be listening in port 4000 of the local machine and the proxy will listen in port 8080.

API reference:

1. Configure the targeted device (ip) and the targeted Android application (app):

```
URL: /config
Method:  Get
URL params:
   -   ip=[alphanumeric] -> [required]
```

2. Install the CA certificate in the target device configured in 1:

```
URL: /cert
Method:  Get
URL params:
    -   nothing
```

3. Install the Frida server in the target device configured in 1:

```
URL: /hook
Method:  Get
URL params:
    -   nothing
```

4. Upload a targeted APK (apk):

```
URL: /upload
Method:  Post
URL params:
   -   apk=[alphanumeric] -> [required]
```

5. Start the phase one of the capture (idle phase) for a specific number of seconds (timeout), after providing requested permissions (if permissions is True) and after rebooting the device (if reboot is True):

```
URL: /phase-one
Method:  Get
URL params:
   -   timeout=[numeric] -> [required]
   -   permissions=[True|False] -> [required]
   -   reboot=[True|False] -> [required]
```

6. Start the phase two of the capture for a specific number of seconds (timeout). It will use a Monkey to automate the inputs and events generation (if monkey is True):

```
URL: /phase-two
Method:  Get
URL params:
   -   timeout=[numeric] -> [required]
   -   monkey=[True|False] -> [required]
```

7. Analyse the raw network traffic captured:

```
URL: /analyse
Method:  Get
URL params:
   -   nothing
```

8. Retrieve the results of the analysis:

```
URL: /result
Method:  Get
URL params:
   -   nothing
```

9. Sanitize the testing environment to start a new evaluation.

```
URL: /sanitize
Method:  Get
URL params:
   -   nothing
```

Usage/Test:

1. Make sure the mobile device is connected, enter in the host terminal:

```
adb devices
```

It should list the device identifiers or IPs.

2. Invoke the aforementioned APIs, for example:

```
http//ip:port/config?ip=”X.X.X.X”
```

It should return ...

Build

```
```

```
docker build --build-arg USERNAME=<gitlab username> --build-arg PASSWORD=<gitlab
password> -t cliip/platform:tagname .
```

Upload:

1. Login to Docker Hub:

```
docker login
```

2. Push the image of traffic analysis:

```
docker push cliip/platform:traffic
```

Documentation (details)

This section explains in more detail the functions and interfaces of each traffic analysis component, as well as the organization of all necessary files in their respective directories.

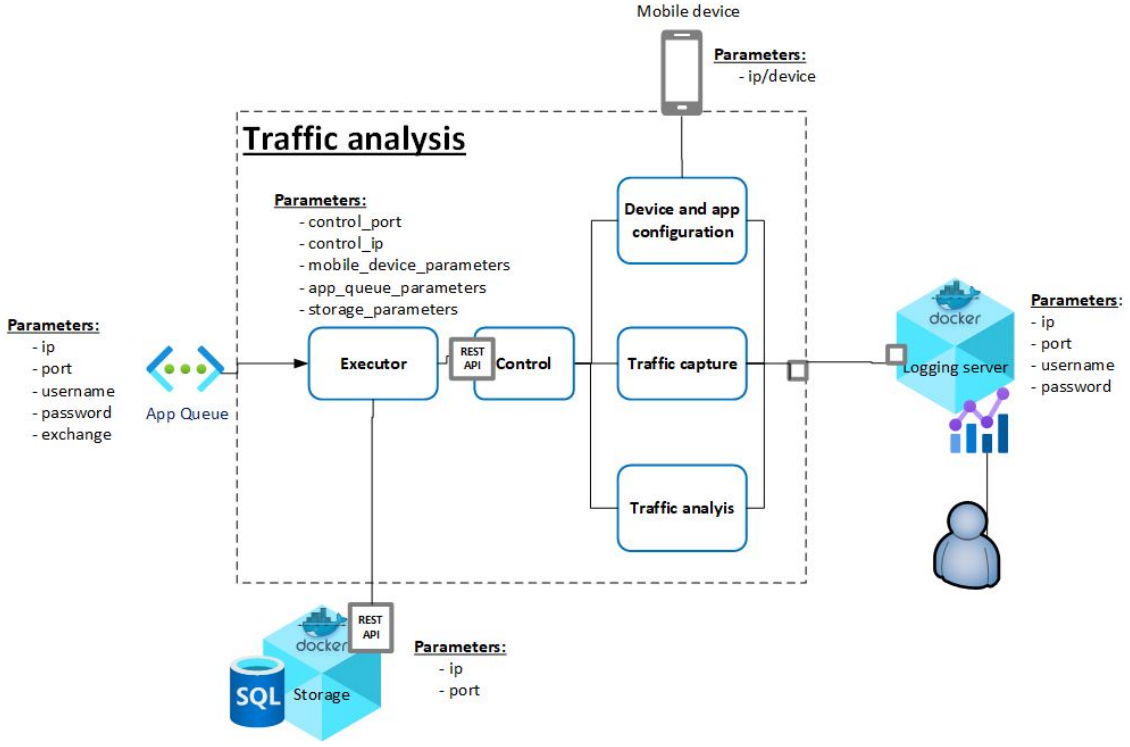

Figure 1. Components of the traffic analysis

1. executor/

This folder contains the files of the ***Executor*** component shown in Figure 1. It implements a Rabbit MQ consumer that launch the traffic analysis by using the REST API of the ***Control*** server, every time an Android App arrives at the ***App Queue***.

1.1. executor/executor.config

This file contains all the configuration parameters of the ***Executor*** component.

1.2. executor/queue_receive.py

This file allows to register a consumer in the ***App Queue*** and remains listening to the arrival of new messages. Once a message with the App name arrives:

- First, it retrieves the APK from the ***Storage*** component.
- Second, it creates a new Testing instance to perform the traffic analysis of the APK recovered by using the REST API of the ***Contro***l Server,

1.3. executor/testing.py

This file configures a new Testing instance in terms of the period of traffic capture, use of monkeys, granting permissions, and mandatory reboot before evaluation.

1.4. executor/traffic.py

This file is a python wrapper of the control server REST API.

2. control/

This folder contains all the files needed by the control server (**Control** and **device and app configuration** components in Figure 1)

2.1. control/traffic/routes.go

Golang file that defines the API endpoints of the Control server

2.2. control/traffic/handlers.go

Golang file that implements the handlers of the API endpoints of the Control server.

2.3. control/scripts/cert.sh

Shell script that installs the MITM proxy CA certificate in the mobile device enabling a man-in-the-middle attack. It is executed when the endpoint "/cert" of the control server is invoked.

2.4. control/scripts/hooker.sh

Shell script that installs the Frida Server in the mobile device. It is executed when the endpoint "/hook" of the control server is invoked.

2.5. control/scripts/start.py

Python script that launches the (1) Filebeat agent, (2) the MITM proxy, (3) the Frida Server CA, and the (4) the mobile app being evaluated. It is executed when the endpoint "/phase-one" of the control server is invoked.

2.6. control/scripts/monkey.py

Python script that launches a monkey to automate the generations of inputs or events in the mobile application being evaluated. It is executed when the endpoint "/phase-two" of the control server is invoked.

2.7. control/scripts/kill.sh

Shell script that kills the (1) the MITM proxy and (2) the Frida Server CA. It is executed when the endpoint "/phase-one" of the control server is invoked.

3. intercept/

This folder contains the files needed to capture the traffic generated by the mobile app being evaluated (**Traffic capture** component in Figure 1).

3.1. intercept/inspect_request.py

This python script captures network traffic only from the application being evaluated and stores it in files for later analysis. This script enables (1) to label the traffic infeasible to decrypt as for certificate pinning protection, and (2) to filter the traffic only from the application being

evaluated. It is executed when the endpoint "/phase-one" of the control server is invoked, specifically when the MITM proxy is launched.

3.2. intercept/pinning/`fridactl.py`

This python script communicates with the Frida server installed on the mobile device in order to (1) attach Frida to the mobile app process and (2) send the bypassing pinning cases. It is executed when the endpoint "/phase-one" of the control server is invoked

3.3. intercept/pinning/`pinning_cases.js`

This JavaScript file defines the hooks to be injected into common certificate pinning methods implemented in Android apps. New pinning bypass methods should be added here.

4. analyze/

This folder contains the files needed to analyze the captured traffic (**Traffic analysis** component in Figure 1).

4.1. analyse/static/domain_owners.json

This JSON file contains a dataset of domain ownerships (we rely on the dataset of webxray [LIBERT2018]). This enables mapping of individual destination domains to the owner company and even to parent companies. Each of them has a profile that includes the type of service, such as marketing, hosting, social media, design optimization, etc.

4.2. analyse/static/`info.device`

This file sets up the PII (Personal Identifiable Information) to be searched in the traffic, one per row. Each register (row) consists of a tuple "PII_type: [instances]". For example, the register "**Contact_E_Mail_Address: privtest88@gmail.com; privtest90@gmail.com**" sets up two email instances to be searched; when either of them is found, a **Contact_E_Mail_Address** exfiltration is alerted. New registers can be added, instances must be separated by a semicolon, with no spaces between them.

4.3. analyse/static/`categories`

This file sets up the categories of personal data, one per row. Each register (row) consists of a tuple "category: [PII_type_instances]". For example, this is a Contact register **"Contact: Contact_E_Mail_Address;Contact_Name;Contact_Phone_Number;Contact_Postal_Address;Contact_ZIP".** It is only for reporting purpose. New registers can be added, instances must be separated by a semicolon, with no spaces between them.

4.4. analyse/static/`api.conf`

We use the API ipstack to locate and identify the geolocation based on the IP address. This file contains the key to use the API and the config parameters.