CYB. Assignment 2. Toy Android Malware analyzer

CYB. Assignment 2. Toy Android Malware analyzer

- Description and Objectives
  - Objectives
- Documentation and deadline
- Task 1. Train malware classifiers
  - Training Dataset
  - Steps
- Task 2. Apply trained malware classifiers on real APKs
  - Steps
- Appendix A. Androguard installation and usage
  - Androguard installation
  - Androguard usage
    - Analysis using standalone commands
    - Analysis from Python code
  - EXTRA: Retrieving API calls (for Androguard 3.4.x only)
  - EXTRA: Retrieving disassembled instructions

Description and Objectives

Use static analysis and ML to develop a simple prototype malware detector for Android APKs.

Objectives

- Training of ''classical'' ML models
- Use of Androguard as a static analysis tool for feature extraction
- Qualitative evaluation of resulting models on a reduced set of APKs

Documentation and deadline

- **Groups:** 1 or 2 students
- **Deadline:** 18th December 2024 (send by e-mail to ribadas@uvigo.gal)
- **Report structure:**
  - Description of the task
  - Description of model training on the dataset: steps followed and results (Task 1 documentation)
  - Extraction of features with Androguard and evaluation of models on the set of APKs: steps followed and results (Task 2 documentation)
- **Code** developed code for Task 1 and Task 2.

Task 1. Train malware classifiers

Training Dataset

- Yerima, Suleiman (2018). Figshare Dataset:
  - Android malware dataset for machine learning 2 [Alternative source: Android Malware Dataset for Machine Learning (Kaggle version)]
- S. Y. Yerima and S. Sezer. Paper:
  - "DroidFusion: A Novel Multilevel Classifier Fusion Approach for Android Malware Detection" in IEEE Transactions on Cybernetics, vol. 49, no. 2, pp. 453-466, Feb. 2019, doi: 10.1109/TCYB.2017.2777960.

Dataset info.

- 215 attributes (described at `dataset-features-categories.csv`)
  - **Manifest Permission (113)**
  - **API call signature (73)**
  - Intent (23)
  - Commands signature (6)
  - 2 classes: B=Benign, S=Malware
- 15,036 instances (`drebin-215-dataset-5560malware-9476-benign.csv`)
  - 5,560 malware apps from Drebin Dataset
  - 9,476 benign apps

Dataset feature/column names

- permisson.names
- api_call.names
- intents.names
- commands.names

Steps

1. Randomly split dataset into training (80%) and testing (20%)
2. Select two "classical" machine learning algorithms of your choice (kNN, decision tree, SVM, ...) among the ones available in `scikit-learn` (or any other ML library)
3. Train **two models** for each selected algorithm
   1. Using only *Manifest Permissions* features (features in `permission.names`)
   2. Using both *Manifest Permissions* (features in `permission.names`) and *API call signature* features (features in `api_call.names`)
4. Report the obtained performance measures (accuracy, precision, recall, F-score)

Notes

1. It is not necessary to carry out an extensive study of the hyperparameters of the ML algorithms used (default or other reasonable values are sufficient).
   - In this case we were dealing with a very small and limited dataset.
   - A more realistic malware detector would require more information and samples.
   - The aim of this assignment is just to see how features extracted by static analysis could be used in an Android APK classifier.
2. You can get the name of available *Manifest Permissions* and *API call signature* features using GNU/Linux command line

```
$ grep "API call signature" dataset-features-categories.csv | sort -u | cut -d , -f 1 >
api_calls.txt
abortBroadcast
ACCOUNT_MANAGER
```

```
android.content.pm.PackageInfo
android.content.pm.Signature
android.intent.action.SEND
android.os.Binder
android.os.IBinde
...

$ grep "Manifest Permission" dataset-features-categories.csv| sort -u  | cut -d , -f 1 >
permissions.txt
ACCESS_COARSE_LOCATION
ACCESS_FINE_LOCATION
ACCESS_LOCATION_EXTRA_COMMANDS
ACCESS_MOCK_LOCATION
ACCESS_NETWORK_STATE
...
```

Task 2. Apply trained malware classifiers on real APKs

The best models from the previous Task will be applied to a small collection of malicious and benign APK samples to perform a manual qualitative evaluation.

- The APK samples available in the Quark-engine project (a platform for static and dynamic pentesting and analysis of mobile applications) will be used.
- Tools from the Androguard project will be used to extract permissions and API calls (see details in Appendix A).

Steps

1. Download APK samples from Quark-engine repository (https://github.com/quark-engine/apk-samples) [from command line using `git` command or from the GitHub web page, using the `[CODE]` link to get a ZIP file]

   ```
   $ git clone https://github.com/quark-engine/apk-samples.git
   ```

   There are two subfolders: `malware-samples` with 7 malware APK files, and `vulnerable-samples` with 11 vulnerable educational Android Apps.

   ```
   $ tree apk-samples
   apk-samples
   ├── malware-samples
   │   ├── 0c2494c03f07f891c67bb31390c12c84b0bb5eea132821c0873db7a87f27ccef.apk
   │   ├── 13667fe3b0ad496a0cd157f34b7e0c991d72a4db.apk
   │   ├── 14d9f1a92dd984d6040cc41ed06e273e.apk
   │   ├── Ahmyth.apk
   │   ├── com.cdim.driver.core.apk
   │   ├── f9bfec4403b573581c4d3807fb1bb3d2.apk
   │   ├── Roaming_Mantis.apk
   │   └── Roaming_Mantis.dex
   ├── README.md
   └── vulnerable-samples
       ├── allsafe.apk
       ├── AndroGoat.apk
       ├── diva.apk
       ├── dvba.apk
       ├── InjuredAndroid.apk
   ```

```
├── InsecureBankv2.apk
├── insecureShop.apk
├── MSTG-Android-Java.apk
├── ovaa.apk
├── pivaa.apk
└── Vuldroid.apk
```

2. Extract the permissions and API calls defined in the training dataset from the APKs in this collection using the Androguard parser (see Appendix A).
   Importan: It should be noted that the format in which Andorguard returns the permission and API call information is not exactly the same as that used in the training dataset. This mismatch must be resolved and in some cases such a match may not be possible.
3. Apply the models trained in Task 1 to the features extracted from the APKs in the collection and report the results.

Appendix A. Androguard installation and usage

Androguard is a Pyhton library for analysing APK files of Android applications. It includes functions to extract the APK metadata (`AndroidManifest.xml` content, resources), disassemble the Dalvik bytecode and access the information about the classes and methods that make up the App code.

- Androguard at Github (version 4.1.x)
- Official Androguard documentation (version 3.4.x)

Androguard can be used in Pyhton code as an external library, but it also provides a set of standalone commands to perform typical tasks (`androguard`, `androlyze`, `andoaxml`, `andorsign`, ...)

Important: The Androguard project is currently undergoing an update and restructuring.

- The official documentation corresponds to the previous stable versions (3.x.x branch), although it is reasonably valid for the use that will be given to this tool in this Assignment,
- The new stable version (branch 4.x.x) is under development and the (scarce) available documentation is in the project Wiki at GitHub.

Androguard installation

We have different options to install Androguard (see Androguard Docs > Installation):

1. Using `pip` [it will install version 4.1.x]

   ```
   $ pip install -U install androguard
   ```

   This option will install the Androguard standalone commands (`androguard`, `androlyze`, `andoaxml`, `andorsign`, ...) at `$HOME/.local/bin`
   With `export PATH=$HOME/.local/bin:$PATH` this folder can be included in the default search path.
2. Using Debian packages (Debian and Ubuntu) [it will install version 3.4.x]

   ```
   $ sudo apt install androguard
   ```

   This option will install the Androguard standalone commands (`androguard`, `androlyze`, `andoaxml`, `andorsign`, ...) at the default search path.
3. Download and install source code from Github [it will install version 4.1.x] (see install from source)

```
$ git clone --recursive https://github.com/androguard/androguard.git
```

Note: In case of getting an error about STDIN use in `IPyhton`, a workarround is to downgrade `IPython` to version 7.34.0

```
$ pip install ipython==7.34.0
```

Androguard usage

The analysis of APK files with Androguard can be carried out from Pyhon code (using Androguard as a library) or from the standalone commands included in the Androguard distribution.

Analysis using standalone commands

Using `androguard analyze [apk file]` (see getting started)

```
$ androguard analyze apk-samples/vulnerable-samples/ovaa.apk


>>> filename
apk-samples/vulnerable-samples/ovaa.apk
>>> a
<androguard.core.apk.APK object at 0x7fb5647b77c0>
>>> d
[<androguard.core.dex.DEX object at 0x7fb564739490>]
>>> dx
<analysis.Analysis VMs: 1, Classes: 2686, Methods: 21465, Strings: 22722>

Androguard version 4.1.2 started

In [1]: a.get_app_name()
Out[1]: 'Oversecured Vulnerable Android App'

In [2]: a.get_permissions()
Out[2]:
['android.permission.INTERNET',
 'android.permission.READ_EXTERNAL_STORAGE',
 'android.permission.WRITE_EXTERNAL_STORAGE']

In [3]: exit()
```

In both cases we will get an IPyhton interactive shell, with three variables (`a`, `d` `dx`) which carry the results of the APK analysis.

- `a` is an `APK` instance: see APK methods
- `d` is a list of `DalvikVMFormat` instances: see DalvikVMFormat methods
- `dx` is an `Analysis` instance: see Analysis mehtods

Note: `ovaa.apk` is and educational vulnerable Android App (see https://github.com/oversecured/ovaa)

Analysis from Python code

We can get the same result from Python code by using the `androguard.misc.AnalyzeAPK`, whick provide us with the same (`a`, `d` `dx`) variables.

```
from androguard.misc import AnalyzeAPK

from androguard.util import set_log
set_log("ERROR")   # disable full debug output

file = 'apk-samples/vulnerable-samples/ovaa.apk'
a,d,dx = AnalyzeAPK(file)

print(a.get_app_name())
print(a.get_permissions())

# IMPORTANT: Available only in Androguard v4.x
for api in dx.get_android_api_usage():
    print(api.class_name + " -> " + api.get_method().name)
```

EXTRA: Retrieving API calls (for Androguard 3.4.x only)

Version 4.0 of Androguard added to the `Analysis` class a `get_android_api_usage()` method.

To get those results in previous versions we have to provide a couple of auxiliary functions that replicate this feature.

```
from androguard.misc import AnalyzeAPK


def is_android_api(meth_analysis):
    """ Returns True if the method seems to be an Android API method.
        This method might be not very precise unless an list of known API methods   is given.
        :return: boolean """
    if not meth_analysis.is_external():
        # Method must be external to be an API
        return False

    # Packages found at https://developer.android.com/reference/packages.html
    api_candidates = ["Landroid/", "Lcom/android/internal/util", "Ldalvik/", "Ljava/",
"Ljavax/", "Lorg/apache/",
                      "Lorg/json/", "Lorg/w3c/dom/", "Lorg/xml/sax", "Lorg/xmlpull/v1/",
"Ljunit/"]

    for candidate in api_candidates:
        if meth_analysis.method.get_class_name().startswith(candidate):
            return True
    return False

def get_android_api_usage(dx):
    """ Get all usage of the Android APIs inside the Analysis.
        :return: yields :class:`MethodAnalysis` objects for all Android APIs methods """
    for cls in dx.get_external_classes():
        for meth_analysis in cls.get_methods():
            if is_android_api(meth_analysis):
                yield meth_analysis
```

```
file = 'apk-samples/vulnerable-samples/ovaa.apk'
a,d,dx = AnalyzeAPK(file)
for api in get_android_api_usage(dx):
    print(api.class_name + " -> " + api.get_method().name)
```

EXTRA: Retrieving disassembled instructions

See Parsing Instructions and Bytecode in Androguard documentation

Note: This feature is not necessary for this assignment, it is just an example to illustrate the low level information that can be extracted with Andorguard.

```
# Top 10 more frequent opcodes
from collections import defaultdict
from operator import itemgetter
c = defaultdict(int)

for method in dx.get_methods():
    if method.is_external():
        continue
    m = method.get_method()
    for ins in m.get_instructions():
        c[(ins.get_op_value(), ins.get_name())] += 1

for k, v in sorted(c.items(), key=itemgetter(1), reverse=True)[:10]:
    print(k, '-->',  v)
```

```
# Bytecode intructions for each method
for method in dx.get_methods():
    if method.is_external():
        continue
    m = method.get_method()
    print("> Method: ",m)
    for ins in m.get_instructions():
        print(ins.get_name(), ins.get_output())
    print("----")
```