Behavior-based Mobile Malware Analysis and Detection

The development of this document is funded by the following grants from the US National Science Foundation: No. 1723707 and 1623713, Michigan Space Grant Consortium (MSGC) Research Seed Grant, and Oakland University Research Seed Grant.

1 Overview

In this lab, you will learn how to use program analysis tools to analyze Mobile Apps and report any malicious activities and/or behaviors. The analysis includes both static and dynamically analysis. Some existing program analysis tools, such as Soot[1], FlowDroid[2] and VirusTotal[3], will be introduced. The learning objectives of this lab are listed below:

1. Understand technologies that analyze software.
2. Be able to use specific tools to perform analysis against mobile malicious Apps.

2 Background

2.1 Static Code Analysis

Static code analysis is a method of examining source code before a program is run. It's done by analyzing a set of code against a set (or multiple sets) of coding rules. In contrast with dynamic analysis, which is an analysis performed on programs while they are executing, in most cases, the static analysis is performed on some version of the source code, and in the other cases, some form of the binary code[4]. Figure 1 illustrates the working process of the static analyses.

[1]https://github.com/soot-oss/soot
[2]https://github.com/secure-software-engineering/FlowDroid
[3]https://www.virustotal.com/gui/home/upload
[4]https://en.wikipedia.org/wiki/Static_program_analysis

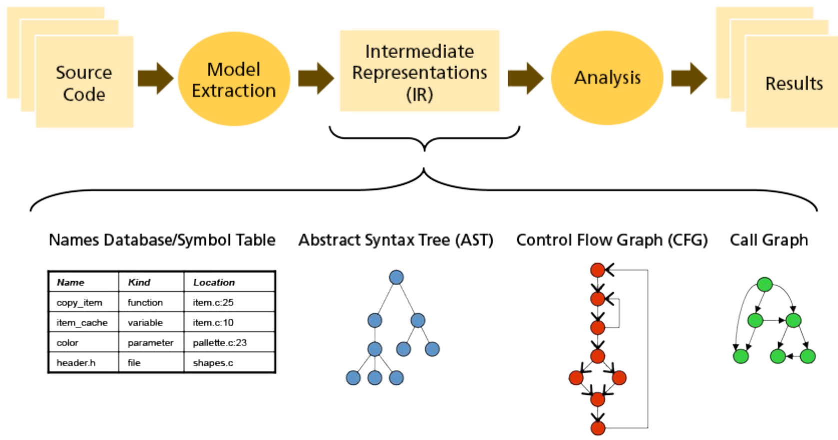

Figure 1: How Static Analysis Works.

2.2 Dynamic Analysis

Dynamic code analysis is the analysis of computer software performed by executing programs on a real or virtual processor. For dynamic program analysis to be practical, the target program must be executed with sufficient test inputs to cover almost all possible outputs. The use of software testing measures such as code coverage helps ensure that a good slice of the program's set of possible behaviors has been observed. Also, care must be taken to minimize the effect that instrumentation has on the execution (including temporal properties) of the target program. Dynamic analysis is in contrast to static program analysis. Unit tests, integration tests, system tests, and acceptance tests use dynamic testing[5]. Figure 2 illustrates the working process of the dynamic analyses.

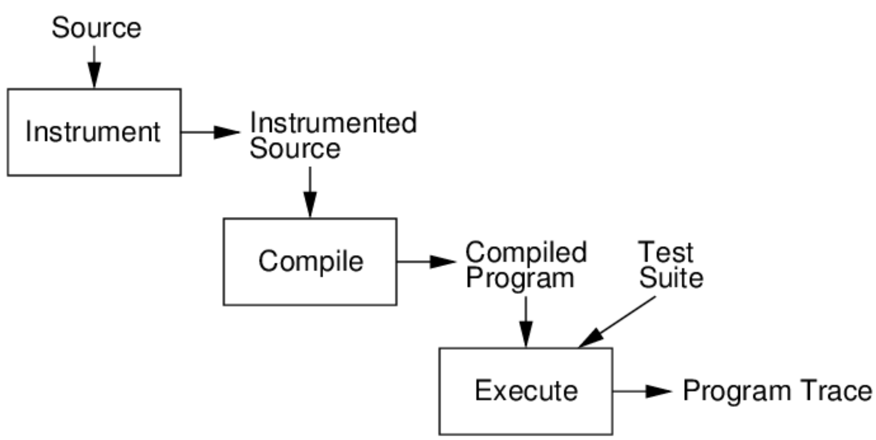

Figure 2: How Dynamic Analysis Works.

[5]https://en.wikipedia.org/wiki/Dynamic_program_analysis

2.3 FlowDroid [1]

FlowDroid[6] is a context-sensitive, flow-sensitive, field-sensitive, object-sensitive, and lifecycle-aware static analysis tool for Android applications. It is built upon Soot[7] and Heros[8]. A very precise call-graph is used to ensure flow sensitivity and context sensitivity. For the purpose of malware detection, FlowDroid statically computes data-flows in Android apps and Java programs, which is utilized to find out data leaks.

3 Task 1: Lab Set-up

In this lab, you need to use four malware samples for analysis. Their locations are listed below:

- Three malware are located under the folder `lab6/apks` on our pre-built Ubuntu 20.04 VM. They are: 1) `Claco.A.apk`; 2) `Dropdialer.apk`; an 3) `Obad.A.apk`.
- The fourth malware, namely `reverse_tcp.apk`, which you created in the "*Developing Mobile Malware*" lab is located at `lab7/volume`, if you have done that lab successfully.

The environment for this lab has been pre-built in the Docker image yangzhou301/lab6 [9], on which `/root/apks` is a shared folder mapping to `lab6/apks` on host.

First, run the following commands to check `lab6/apks` folder to see whether the `.apk` files are there.

```
//Make sure the malwares are there in the folder.
$ cd $HOME/lab6
$ ls apks
Claco.A.apk  Dropdialer.apk  Obad.A.apk
```

You should also copy the `reverse_tcp.apk` to the `apks` folder.

```
//Copy the malware from the shared folder to the malware folder.
$ cp $HOME/lab7/volume/reverse_tcp.apk $HOME/lab6/apks
```

Then, pull the Docker container's image and start it.

```
//Pull (download) the container for this lab.
$ docker pull yangzhou301/lab6
```

[6]https://security-summer-labs.readthedocs.io/en/latest/lab6/readme.html#flowdroid-static-analysis
[7]http://www.sable.mcgill.ca/soot/
[8]http://sable.github.io/heros/
[9]https://hub.docker.com/r/yangzhou301/lab6

Once you start the container, you should see a screen with shell at `/root` directory as illustrated in Figure 3.

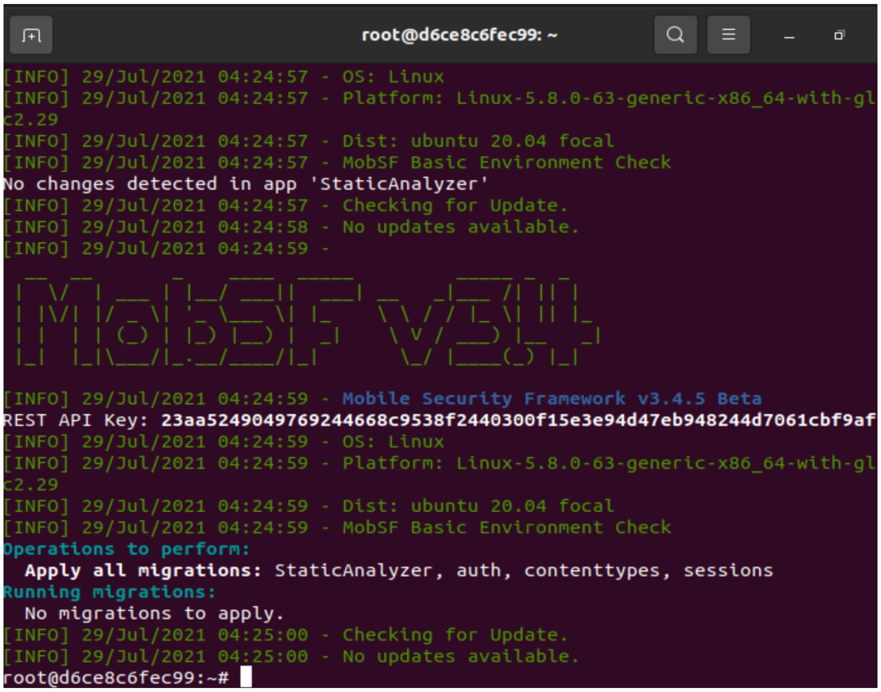

Figure 3: A screenshot of root shell in the container.

4 Task 2: Static Analysis with FlowDroid

For this task, you need to analyze an Android malware, namely `Claco.A.apk`, which steals text messages, contacts, and all SD Card files. It also automatically downloads `svchosts.exe` when the Android phone is connected to the PC in the emulation mode. The malware `svchosts.exe` will sequentially record sounds around the infected PC and upload them to remote servers.

Before running `FlowDroid` with `Claco.A.apk`, you must specify a definition file for the **sources** and **sinks**. In general, the **sources** define the statements/locations that takes a source of the sensitive data, while the **sinks** define the statements/locations that can possibly leak the sensitive data to the outside world. In this lab, you can use the `SourcesAndSinks.txt` file provided by `FlowDroid`, which targets on looking for privacy issues. Let's apply it as an example to analyze the data-flow in `Claco.A.apk` with the command in Figure 4.

```
//Run Flowdroid to perform static analysis for Claco.A.apk.
$ java -jar soot-infoflow-cmd-jar-with-dependencies.jar -a apks/Claco.A.apk
-p $ANDROID_SDK/platforms/ -s SourcesAndSinks.txt
```

Figure 4: The command used to analyze `Claco.A.apk`.

The above command will produce a long report as the analysis result (Figure 5).

```
...
[main] INFO soot.jimple.infoflow.android.SetupApplication - Collecting callbacks and building a callgraph took 1 seconds
[main] INFO soot.jimple.infoflow.android.SetupApplication - Running data flow analysis on Claco.A.apk with 68 sources and 194 sinks...
...
[main] INFO soot.jimple.infoflow.android.SetupApplication$InPlaceInfoflow - Callgraph construction took 0 seconds
...
[main] INFO soot.jimple.infoflow.android.SetupApplication$InPlaceInfoflow - IFDS problem with 10212 forward and 4505 backward edges solved in 0 seconds, processing 14 results...
[main] INFO soot.jimple.infoflow.android.SetupApplication$InPlaceInfoflow - Current memory consumption: 249 MB
[main] INFO soot.jimple.infoflow.android.SetupApplication$InPlaceInfoflow - Memory consumption after cleanup: 35 MB
[main] INFO soot.jimple.infoflow.data.pathBuilders.BatchPathBuilder - Running path reconstruction batch 1 with 5 elements
[main] INFO soot.jimple.infoflow.data.pathBuilders.ContextSensitivePathBuilder - Obtainted 5 connections between sources and sinks
...
[main] INFO soot.jimple.infoflow.android.SetupApplication$InPlaceInfoflow - The sink virtualinvoke $r7.<java.io.FileOutputStream: void write(byte[])>($r8) in method <smart.apps.droidcleaner.Tools: boolean GetContacts(android.content.Context)> was called with values from the following sources:
...
[main] INFO soot.jimple.infoflow.android.SetupApplication$InPlaceInfoflow - - r5 = interfaceinvoke $r4.<android.database.Cursor: java.lang.String getString(int)>($i0) in method <smart.apps.droidcleaner.Tools: boolean GetContacts(android.content.Context)>
...
<smart.apps.droidcleaner.Tools: boolean GetAllSMS(android.content.Context)> was called with values from the following sources:
...
[main] INFO soot.jimple.infoflow.android.SetupApplication$InPlaceInfoflow - - $r9 = interfaceinvoke $r4.<android.database.Cursor: java.lang.String getString(int)>($i1) in method <smart.apps.droidcleaner.Tools: boolean GetAllSMS(android.content.Context)>
[main] INFO soot.jimple.infoflow.android.SetupApplication$InPlaceInfoflow - The sink virtualinvoke $r13.<java.io.DataOutputStream: void write(byte[],int,int)>(r5, 0, $i0) in method <smart.apps.droidcleaner.Tools: boolean UploadFile(java.lang.String,java.lang.String,java.lang.String,java.lang.String,android.content.Context)> was called with values from the following sources:
[main] INFO soot.jimple.infoflow.android.SetupApplication$InPlaceInfoflow - Data flow solver took 1 seconds. Maximum memory consumption: 249 MB
[main] INFO soot.jimple.infoflow.android.SetupApplication - Found 11 leaks
```

Figure 5: The output report of `FlowDroid` for `Claco.A.apk`.

<u>Deliverable 1</u>: Run `FlowDroid` with the same source-and-sink configuration and explore the privacy issue in the malware `reverse_tcp.apk` (This is the malware you constructed in the "*Developing Mobile Malware*" lab)? Is there any data leakage? If yes, point out the lines in the outputs, which are relevant to the potential data leakage?

5 Task 3: Dynamic Analysis with VirusTotal

In the previous task, you have learned how to analyze malware with static analysis. However, many malicious behaviors of malware remain undetected without actually running them. For this task, you should use `VirusTotal`[10], an online malware detection tool that aggregates the intelligence of many antivirus repositories, performs online scanning, and checks online behaviors of malware. More importantly, it performs *dynamic analysis* to detect malware inside Cuckoo sandbox[11].

⚠ Prerequisite

First, You should register an account on VirusTotal and log in. Otherwise, the dynamic analysis won't be functional.

For this lab, you need to analyze `Obad.A.apk`, a sophisticated malware, which

- sends SMS to premium-rate numbers;
- downloads other malicious programs and installs them on the infected device, and/or send them further via Bluetooth connection;

[10] https://www.virustotal.com/gui/
[11] https://cuckoosandbox.org/

- controls other machines remotely from a console; and
- Exploits several unpublished vulnerabilities (dated by the year 2014).

Open the official website of VirusTotal (www.virustotal.com) as Figure 6 and upload the `Obad.A.apk` file.

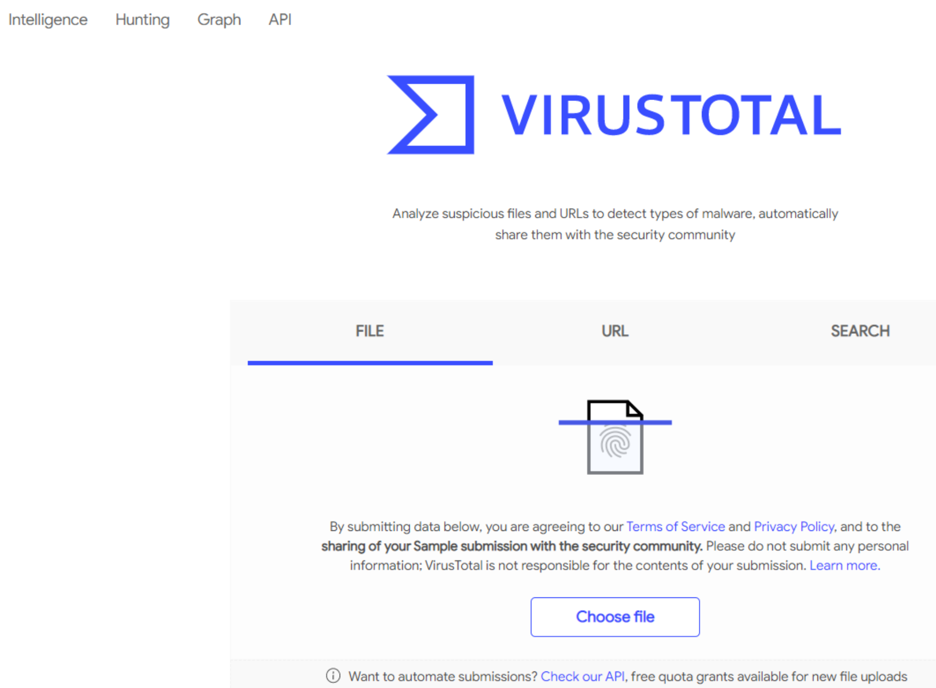

Figure 6: The web interface of `VirusTotal`.

The report should come out in a moment. Figure 7 illustrates the information and identifies the file as malware.

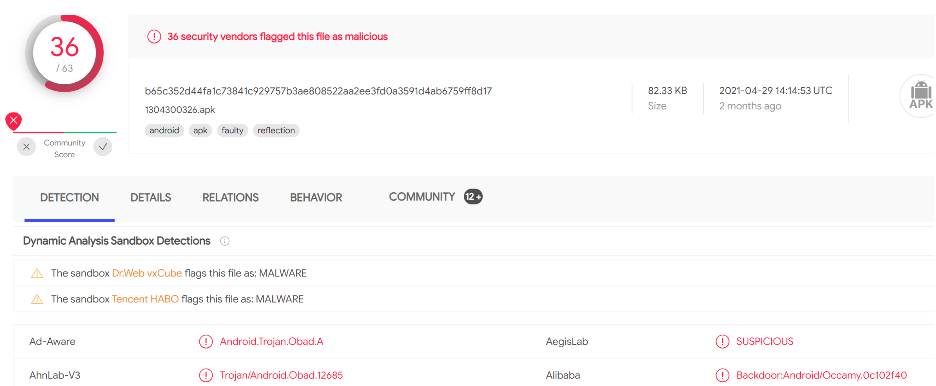

Figure 7: The output report of `VirusTotal` for `Obad.A.apk`.

Click the Details tab as Figure 8, you can read a detection report.

Contacted URLs

Scanned	Detections	URL
2015-05-19	2 / 63	http://www.androfox.com/load.php

Contacted Domains

Domain	Detections	Created	Registrar
www.google.com	1 / 86	1997-09-15	MarkMonitor Inc.
www.androfox.com	0 / 87	2018-03-20	NAMECHEAP INC
parkingpage.namecheap.com	1 / 87	2000-08-11	ENOM, INC.
mtalk4.google.com	0 / 85	1997-09-15	MarkMonitor Inc.
android.clients.google.com	0 / 86	1997-09-15	MarkMonitor Inc.

Figure 8: The Details report of `Obad.A.apk`.

Click the Relation tab as Figure 9, you can a more detailed report. In this report, you can see what domains or IP address this App tries to contact during its execution. It also gives a graphic summary about the files and addresses the App is related at run-time.

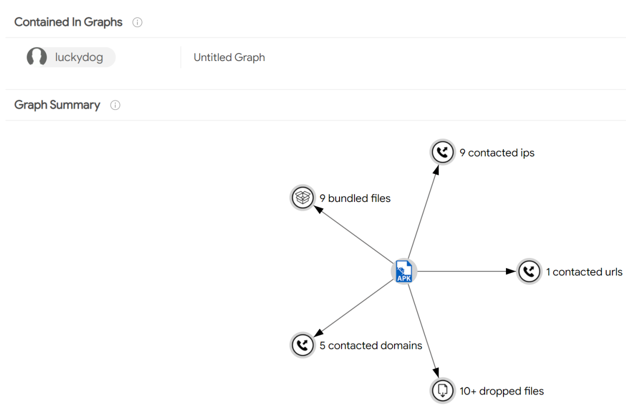

Figure 9: The Relations report of `Obad.A.apk`.

For more information about the run-time behaviors of this App, you can click the Behaviors tab as shown in Figure 10.

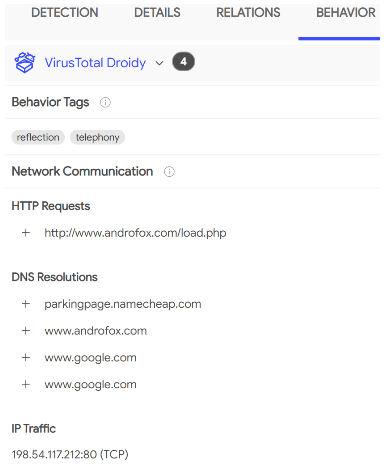

Figure 10: The Relations report of `Obad.A.apk`.

Of course, you can also read comments posted by other users in the Community panel, regarding this App.

Deliverable 2: Using the similar approach as illustrated above, analyze `reverse_tcp.apk` with VirusTotal. Answer the following questions: 1) Which IP address(es) the App can contact at run-time? 2) What are other behaviors you discovered that are relevant to the attack? Would you please provide screenshot(s) to support your answer?

References

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps, Acm Sigplan Notices 49 (6) (2014) 259–269.