Poster Abstract: Towards Automatic Triggering of Android Malware

Adrien Abraham, Radoniaina Andriatsimandefitra, Nicolas Kiss, Jean-François Lalande, and Valérie Viet Triem Tong[1]

INRIA – CENTRALESUPELEC – INSA CVL, France
[1]Contact author: valerie.viettriemtong@centralesupelec.fr

An important part of malware analysis is dynamic analysis. Dynamic analysis try to defeat the techniques used by malware developers that hide their malicious code using obfuscation, ciphering, stealth techniques, etc. For example, a malicious developer can simply delay the execution of his malicious code for a certain period of time. His goal is to be sure that the malware runs on a device of a real user and not on an analysis platform. Thus, the major constraint with dynamic approaches is that their efficiency relies on the effective observation of the malicious behavior.

For automating application executions, a first framework [1] proposes to stress applications by sending pseudo-random streams of user events such as clicks, touches, or gestures, and system-level events. Better than a monkey, Dynodroid [2] generates more relevant UI and system inputs. Nevertheless, for a lot of malware we are far from triggering their behavior.

Android malware are regularly a repackaged version of regular applications: the malicious code is dissimulated inside the initial code. From a quantitative point of view, an android application is a collection of bytecode instructions that can represent a lot of possible execution paths. The previous cited approaches mainly focus on the test of the application and cannot cover all possible execution paths: using these techniques will certainly not reveal interesting observations for the malicious behavior.

We are currently working on a solution to automatically identify suspicious parts of the code and then to trigger its execution. Our approach is divided into three steps. The first step resorts to static analysis: we define a scoring function that computes an indicator of risk for each method in the bytecode. The second step consists in computing an execution path that leads to the code identified as the most dangerous. The third step enables to modify the bytecode in order to force this particular execution path. This last step is the most tricky: it requires to change the control flow and to generate the right UI events in order to succeed in executing the suspicious code.

References

1. Google: UI/Application Exerciser Monkey. http://developer.android.com/tools/help/monkey.html
2. Machiry, A., Tahiliani, R., Naik, M.: Dynodroid: An input generation system for android apps. In: 9th Joint Meeting on Foundations of Software Engineering, Saint Petersburg, Russia (2013) 224–234

Towards Automatic Triggering of Android Malware

A. Abraham R. Andriatsimandefitra N. Kiss J.-F. Lalande V. Viet Triem Tong

INRIA/CENTRALESUPELEC research project Cidre – INSA CVL
France

The problem: malware hiding techniques

Malware wait before running to evade dynamic analysis

- a fixed or dynamic period of time
- a user input
- a system event
- an order from a remote server
- a particular state of their hosted application
- something else ?

Existing solutions

Some frameworks proposed to test the infected application

1. using random inputs
2. running a maximal branches of code

BUT

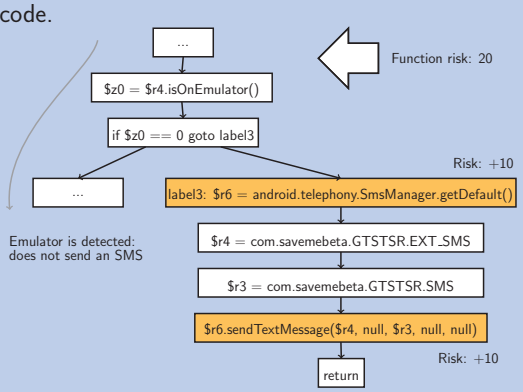
1. it is uncertain
2. it is unnecessarily expensive

First step: static identification of malicious code

A scoring function computes an indicator of risk for each instruction in the bytecode.

The score increases with calls to specific Java methods such as:

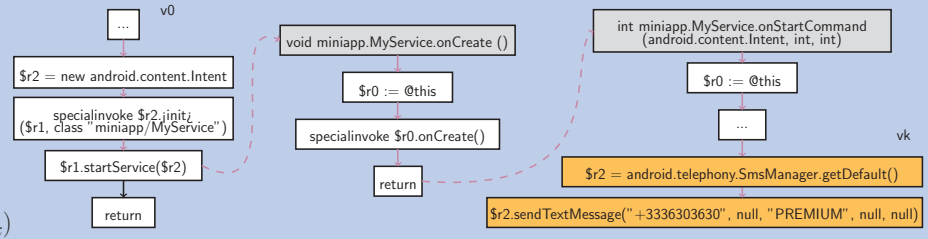
- `android.telephony.SmsManager` for sending SMS
- `android.telephony.TelephonyManager` for getting device infos
- `android.context.pm.PackageManager` for installing/removing apps
- `java.util.Timer,TimerTask` for the implementation of *timebombs*
- `java.lang.Runtime,Process` for executing native binaries
- `dalvik.system.DexClassLoader` for loading code dynamically



Second step: recomputing an execution path to the identified malicious code

To compute an execution path to the most scored unit of code:

- $\forall$ f, functions of the malware, compute: $G_f = (V_f, A_f)$. Let $G = \cup_f G_f = (V, A)$
- $\forall$ intents, events from $v_i \in V_i$ to $v_j \in V_j$: Add (v_i, v_j) to A
- Let v_k the scored unit of code
  Let v_0 the entry point (onCreate())
  Compute $path = shortest\_path(G, v_0, v_k)$



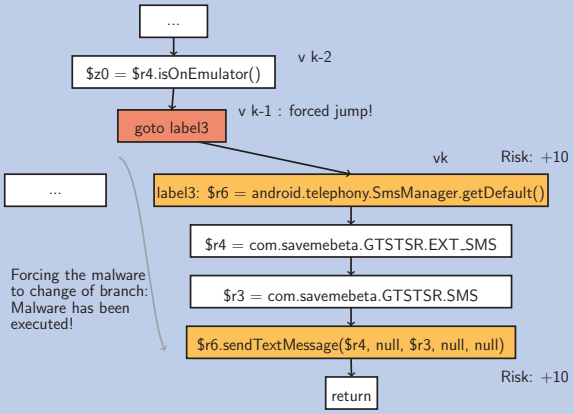
Third step: forcing the execution path

To force the execution of the most scored unit of code $path$:

- Make a *standard* execution
- Let $path = (v_1, ..., v_e, ..., v_k)$:
  v_e is the last unit of code executed.
- $\forall i > e > k$, if v_i is a condition, $Force(v_i)$
- Execute the malware again.

Benefits:

- a malware that is executed shows its effects
- detection tools can be trained or evaluated



Contact: valerie.viettriemtong@centralesupelec.fr