Introducing A S E F - Android Security Evaluation Framework

An idea making a journey through different paths of Android Security to reach a destination that may only be pursued with the help of a community.

Weakness in a Strength

The Android OS available as open source is widely popular among developers and various smart phone manufacturers as it enables them to add to it and create interesting new apps. We already see its prominence with a vast amount of variation providing users a range of choices. However, this strength becomes a challenge. As a security researcher, I'm interested in taking a microscopic look into an issue with a very specific set of requirements and coming to a generic conclusion which considers all of its possible variations.

Edge of smartness & Elements of a Security Model

To add the extra elements of smartness, various categories of applications are developed and distributed for Android. Although security was called 'a bit relaxed' in initial versions, a more reliable and regulated way of downloading these applications would be a well managed Android Market (e.g. Google Play). However, these markets are growing, while in the past, we have seen attackers managing to make malicious apps available affecting a major set of users. Initiatives have been taken to protect these markets by a security service (e.g. Bouncer from Google Play) but there is evidence which makes us doubt if these security measures are completely bulletproof.

If you are a user who religiously sticks to downloading apps from such markets, your initial security model becomes having faith in the security model of the market itself. But many users want to root their devices and unlock hidden features. Also, there are adventurous users who want to download applications that are not necessarily available from a regulated marketplace. In either case, how can a user make that final judgement call of whether an app is safe or malicious?

The Judgement Call and the Dilemma of Permissions

One straightforward answer from Google to this question is "Permissions". Agreed! A set of declared permissions are available to any user before installing any given application on a device to make a decision based on what type of access it needs to function.

But the difficulty arises because there is an inclination to ignore it due to an inconvenience to analyze each permission. After all, these apps are meant to make your life easier! Also, the temptation to try a new app often overrides a safer way of making a choice.

From a Smoke Screen to a Layer of Transparency

Irrespective of the above mentioned subjective way of making a decision, and while looking at these apps in a more objective way through its own security model of permission, the real question to answer is “Are these permissions a convenient, comprehensive and a conclusive way of making this decision?” Along with my experiments during the initial stage of research and past known evidence of bypassing this permission model, my answer to this is “No”. In fact, it is generally difficult to perform a security analysis purely based on asset information associated with these Android applications.

So, let’s take a different approach to this, and while not entirely ignoring these permissions, let’s find out how they are exercised as well as how the user data is managed, handled and communicated by an app. Looking at it from a broader perspective, let’s attempt to find that layer of transparency between a user and an app by exposing how exactly an app is behaving.

Functionality Aspects vs Behavioral Analysis

This means not only being aware of the functionality of an app by knowing what features it has to offer but focusing on its ability to access your data with or without your permission. And if it is accessing your data, is this data sensitive, personal and private? How is it handled? Is it being sent anywhere and if so, where and how? Is it communicating with a server that is already blacklisted? Is there malware in it? Is there any aggressive adware in it? How much bandwidth does it consume? Does it contain security flaws that can be exploited during run time?

In order to answer questions like this, I thought of developing a tactic which I call the “Behavioral Analysis” of an app. The technique is to simulate the entire lifecycle of an Android app on an Android device (virtual/physical) and collect data while triggering behavioral aspects of it. In simple words, download an Android app from the Internet, install it on an Android device, launch it and mess with it (e.g. by clicking different buttons, scrolling up/down, swiping, etc.) While doing so, collect an activity log using adb (Android debug bridge utility, which is available as a part of an Android SDK) and network traffic using tcpdump (a widely used packet capturing tool).

During such a simple yet thorough approach of performing a behavioral analysis for various apps, interesting results were found about apps leaking sensitive information like IMIE, IMSI, SIM card or a phone number of a device. Some malicious apps might just send this data in clear text over the Internet and are much easier to be caught by analyzing collected behavioral data. However, some malicious apps can be sophisticated enough to detect the default settings of a virtual Android device and might behave differently in such settings. In order to overcome such limitations, a virtual device can be custom built by fine-tuning the kernel and also altering default settings to emulate a real device or it can be replaced by a physical Android device.

Limitations of Manual Research

Manual research facilitates deeper analysis of security aspects of Android apps while performing such behavioral analysis. But once the reliable tactics are well defined, it becomes overwhelming to execute them repeatedly over a larger set of Apps. The amount of time and effort it takes to download, install, launch, trigger behaviors and collect data for one App is significant. And we are just talking about collecting such data, but how about organizing it for large number of apps - and more importantly - interpreting it?

The rate at which new apps are being developed and becoming available is very high compared to the rate at which they are evaluated through such manual research. And we are talking about getting ahead of the curve, so automating this entire process becomes an absolute necessity.

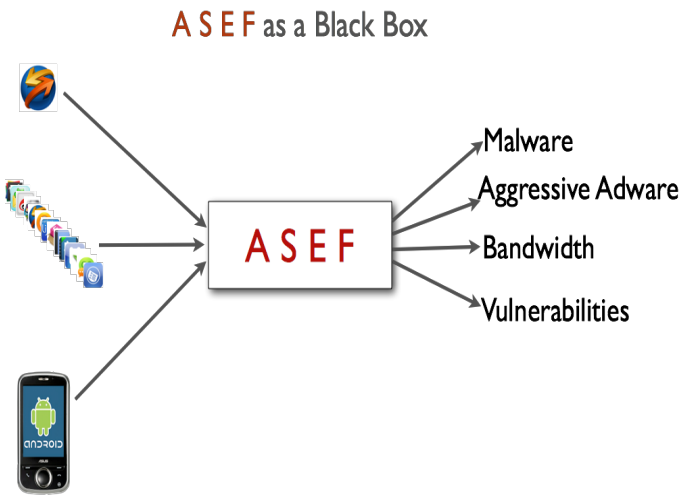
Automated Virtual Environment - The Creation of ASEF

While automating this manual process of behavioral analysis, I came up with the idea of the 'Android Security Evaluation Framework'.

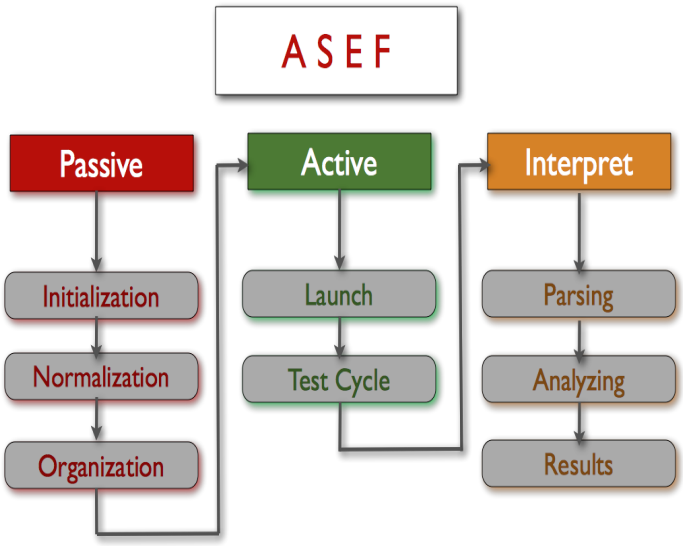
ASEF is designed and developed to simulate the entire lifecycle of an Android application in an automated virtual environment to collect behavioral data and perform security evaluations automatically over 'n' number of apps.

Here is a top level design overview of ASEF :-

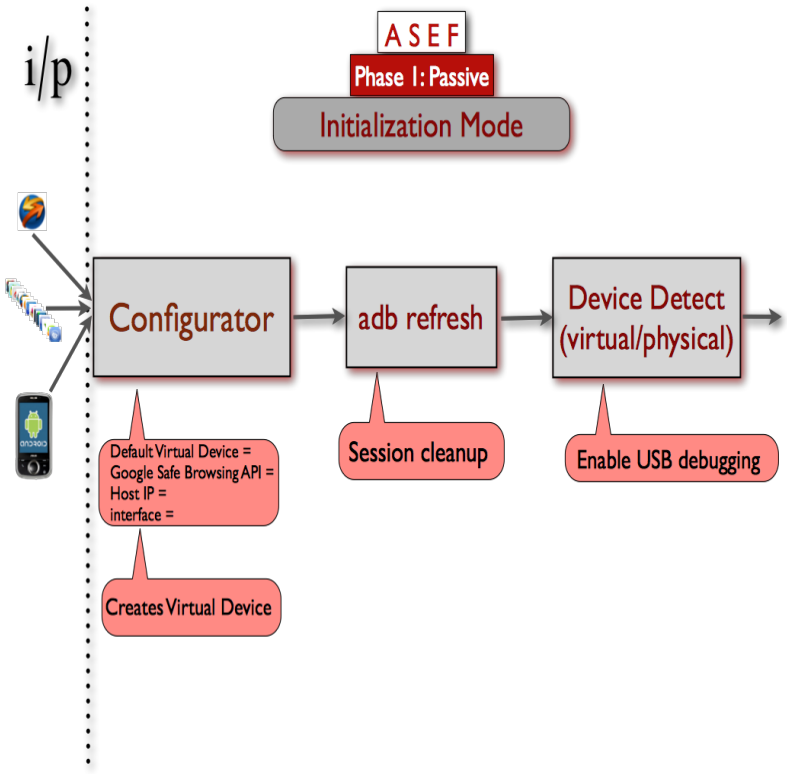
These applications can be in the form of an individual .apk file, a collection of such .apk files or more commonly installed directly on a device.



For either .apk files or apps pre-installed on a device, ASEF will extract and/or migrate these applications to its test suite to perform an automated behavioral analysis.

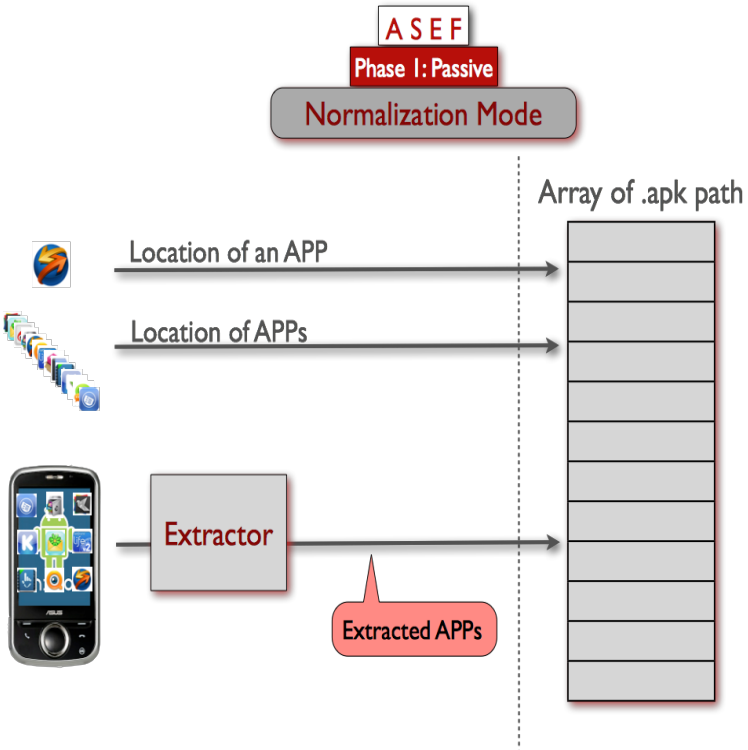


The framework has three phases:
Passive phase tries to collect all the necessary data required to run a test cycle.
Active phase is where apps are being run through a test cycle one by one and behavioral data is collected.
Interpret phase is where parsers are trying to analyze all this data and generate results.



Initialization mode :

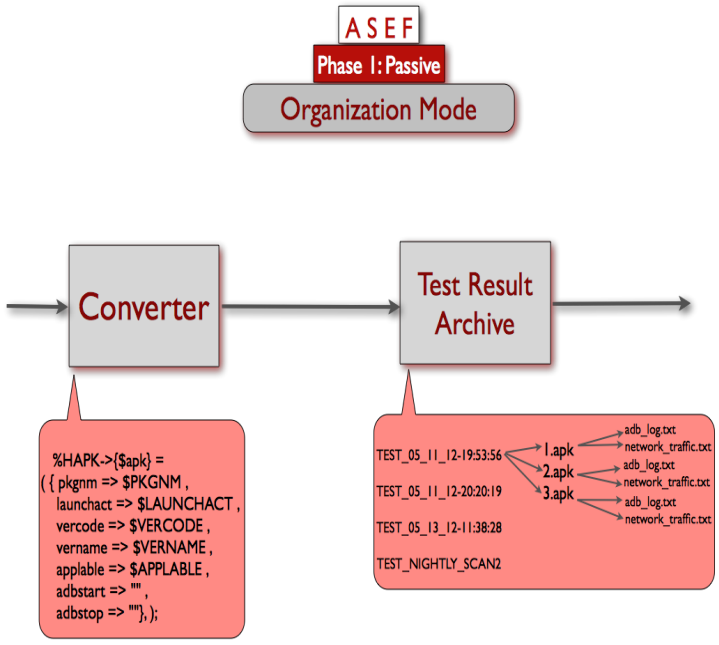
1) Configurator module will configure ASEF with user defined settings (e.g name of the Android Virtual Device - AVD, Google's safe browsing API, Host IP and interface on which to sniff the next traffic.)
2) Session clean up will be performed by adb refresh module. (adb - Android debug bridge)
3) Device Detect module will detect the connected virtual or physical Android devices and inform the user if it has detected them successfully. (Enable USB debugging settings on your Android device before connecting it to the machine running framework.)



Normalization mode :

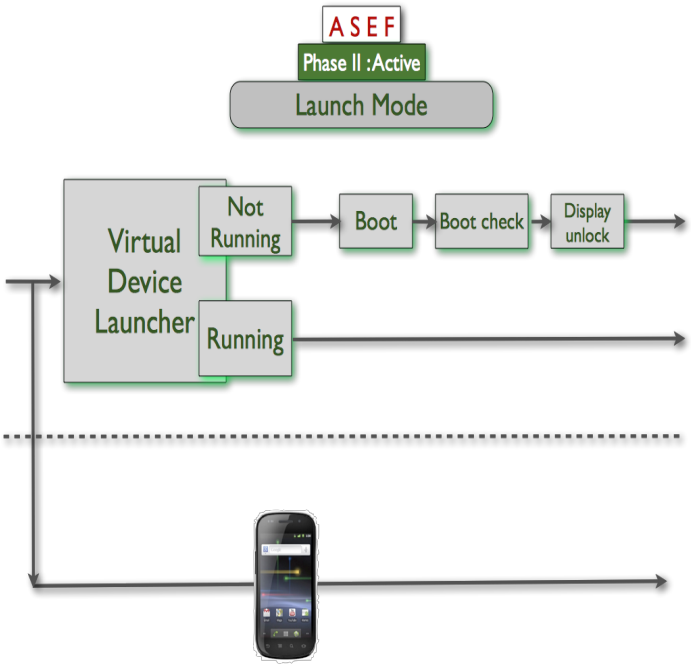
1) In the first two use-cases where a single app or multiple apps are available in .apk format, the location of these apps will be stored in an array and will be used for further processing.
2) In the use-case where the user has all the apps installed right away on a device through an app store, the extractor module will extract all these .apk files from the device to the local machine running ASEF and then store the path of extracted .apk files to the array. All you have to do is connect your Android device with a 'usb debugging mode enabled' to a machine running ASEF via a USB cable and let the extractor module take care of its business. I do want to emphasize though that at no point will ASEF run any passive or active test on a user's device. The extractor module will query the list of installed packages from user's device and filter our user-installed apps from system apps. As such our focus here is to evaluate the security of user-installed apps. During each iteration of filtering user-installed apps, it will also try to find the location of the actual .apk file on a device. Once it's found, it will pull that .apk file on a local machine and populate the array with a correct local path of each .apk file.

From a design point of view, irrespective of any type of future input, if a developer manages to get to this stage of having an .apk file on a local machine, the rest of the functionality of ASEF remains the same from this mode onwards.



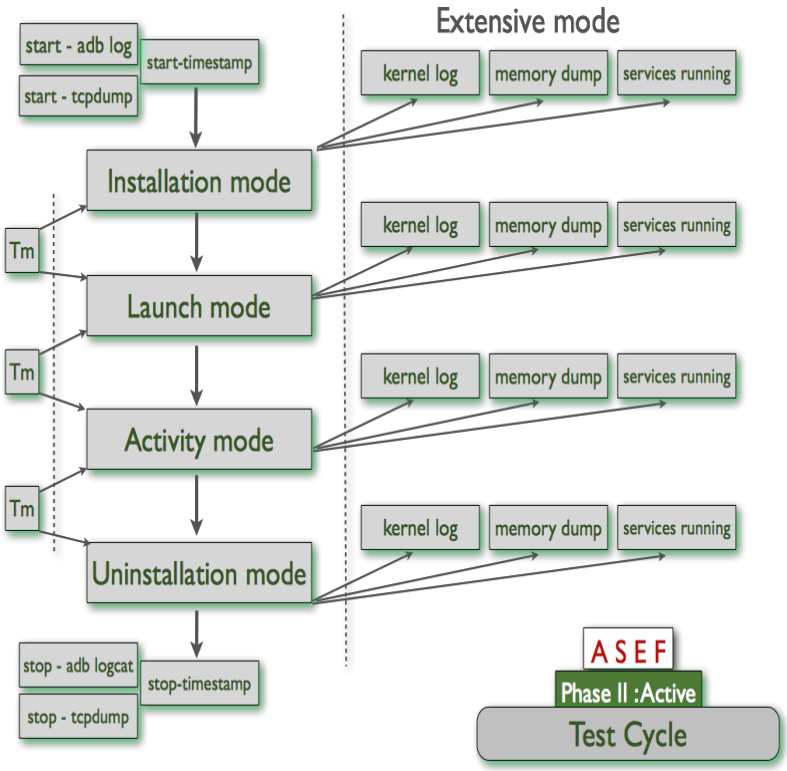
Organization mode :
1) Converter module will extract all the necessary asset information from an .apk file and will populate the hash data structure with required information. (e.g. application label, package name, version number, launcher activity etc..)
2) Test result archive module will create a hierarchy of directories where all the collected behavioral data will be well organized and later used by parsers.

After this mode, ASEF will switch gears to the active phase.



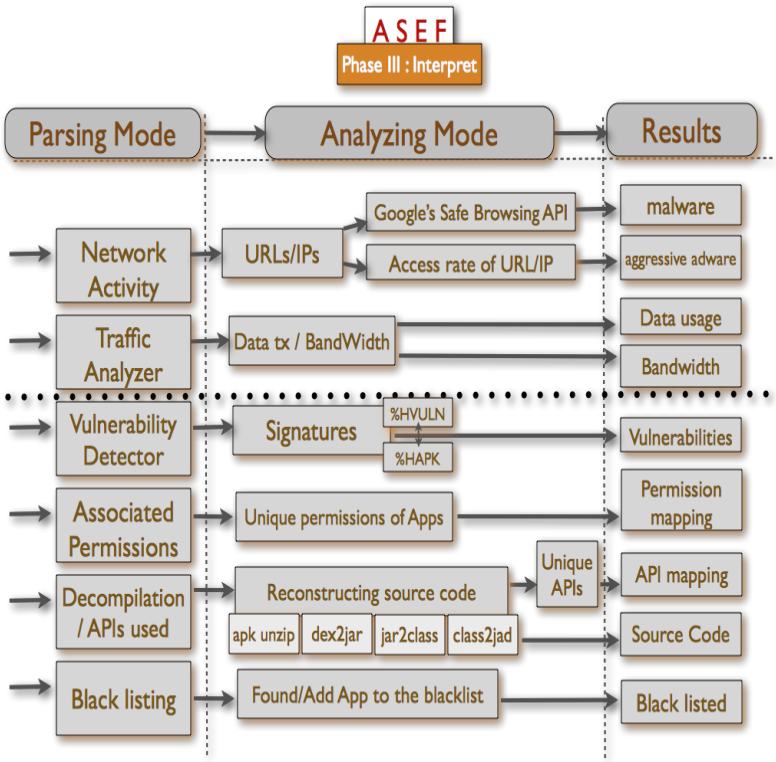
1) If the user defined virtual device is already running, it will be detected and used for running tests. But if it is not detected, the device launcher module will boot the virtual device and actively track the boot process using a background process to detect if the boot was completed successfully. Once booted, it will unlock the display by remotely sending a swipe gesture so a user can see the tests performed during the test cycle. However, having a display locked or unlocked won't make a difference to the actual functionality aspect of it.

2) In order to collect more real life data, it would be important to consider the limitations of a virtual device as it has predefined default settings. A malicious app may be smart enough to detect such settings and may behave differently knowing it to be virtual device. This can be overcome by configuring virtual device to emulate a real physical device or in fact, there is a much simpler option of using a physical Android device itself if available. If the Android device is also attached with the machine running this framework to run all test on it, the virtual device detect module will be automatically bypassed.



Once the virtual or physical device is booted/detected, all the apps will go through the test cycle serially. A test cycle will start with capturing a time stamp of a start of an event while launching the adb log and tcpdump in the background process to collect the data for a period of each test cycle.

Each app will be installed, launched and triggered for various behaviors and later uninstalled. There are advanced settings to capture kernel logs, memory dump and running services at each stage and called extensive mode. Tm is the time duration between each stage which can be fine tuned to prolong the time duration of each test cycle in order to collect more data.



ASEF will go into interpret mode after having collected all the behavioral data.

During interpret mode, various parser modules will try to determine the interaction with any command and control (C&C) servers using Google's safe browsing API, the aggressive bandwidth usage permission mappings. There more than 50 signatures written addressing known security flaws in widely available apps on Android markets.

For more design details, a live demo and statistical results, please feel free to attend my presentation at an upcoming BSides Las Vegas Security Conference on Wednesday, July 25 at 5pm PST at the Artisan Hotel : http://bsideslv.com/talks.php#bg107

Link to the video of Demo :- http://www.youtube.com/watch?v=1nQgD4PUiy0&feature=youtu.be

Forensic Analysis and Behavioral Patterns

As mentioned before, in extensive mode, snapshots are taken of kernel logs, memory dump and running services at different stages of test cycle. Currently parsers are not interpreting this information, but as a work in progress, this feature will be helpful to conduct a forensic analysis or even find certain behavioral patterns among them.

Data Harvesting and Statistical Analysis :-

All the data collected during the test cycle is well categorized and organized under a test result hierarchy and is used by different parsers to interpret. This data can be used to show the

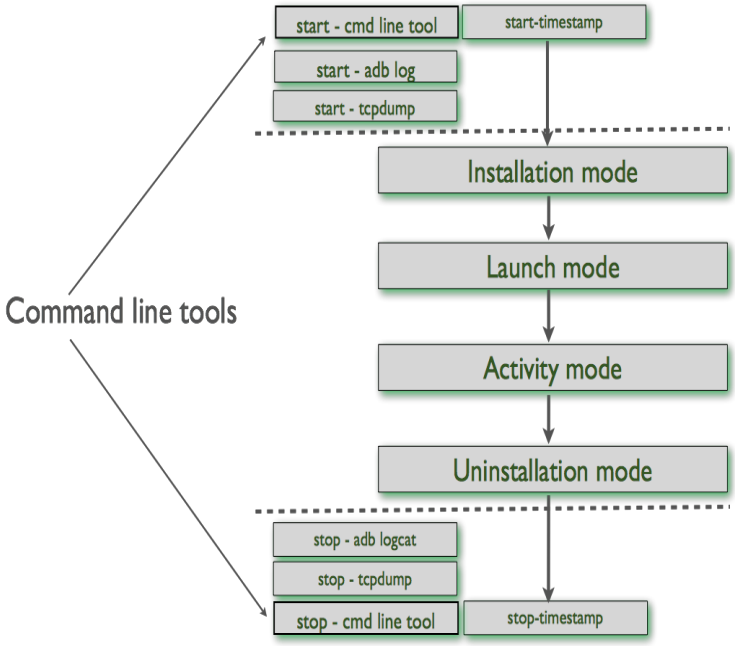
immediate results of the test. This data harvesting is not only useful for a short term gain, but as we keep learning more about what information to look for and how to interpret it, adding more parsers to analyze this harvested data will make it much easier to reuse it moving forward.

At the same time, it will enable us to create statistics that can be helpful to narrow down where to focus our further efforts and understand the bigger picture from different perspectives.

Please refer presentation slides for already collected statistical data.

Extensibility through the Flexibility of Design :-

1) Command line utilities :-
The flexibility of the design allows enormous amount of relevant command line utilities to be integrated very easily in order to expand the functionality and a scope of this framework.



Wrapping such command line tools around the test cycle will help collect different types of data required to be gathered for analyzing apps. For all such command line tools added, writing a parser to extract relevant information won't be much harder than writing some regular expressions.

2) Custom built virtual devices
Another area where more functionality can be added in order to catch much more sophisticated apps is the virtual device on which we run our tests. Rather than using a default virtual device created from an Android SDK, it can be custom built by making contributions to the platform (source code) itself to track how sensitive data is handled and used.

Importance of Vulnerability Scanning :-

While the majority of the focus is still on intentionally written malicious apps by an attacker, the unintentionally introduced vulnerabilities in a genuine app by a developer may be equally harmful if exploited.

While performing Vulnerability Scans of my own Android Device using ASEF, I found some of my apps were vulnerable with a high severity and few of these apps were widely used ones like Mozilla Firefox and Adobe Flash Player. While I was surprised, I was also alerted by ASEF to apply a quick fix by updating all my apps to the most recent version available from the market and hoping it might make these vulnerabilities go away. After updating all my apps to the latest version available I felt safe. Well, not for too long, as scanning for vulnerabilities of fully updated Android devices came out to be more of a shock than a surprise as I still found a few of the apps being vulnerable in results. Investigating further I realized that some apps available from the market are still vulnerable even in the most recent versions, and they are hosted and downloaded by hundreds and thousands of users.

Once an app is found to be malicious, it gets blacklisted by a respective market. The same rule doesn't necessarily apply even in some proportion to an app which has a known security flaw and can be exploited to steal user's private data.

The vulnerability scanning feature of ASEF allows users to quickly update apps for which security flaws are known and indicates whether to use an app whose security flaws are left unresolved.

Solving Practical Real World Problems :-

The scope of ASEF isn't confined to research and statistical analysis. Ever since the idea of ASEF sprung up, it was meant to be unplugged to address real life, practical problems.

At two different layers where it can solve problems :-

1) Protect & Promote Android Markets

Developers who are eager to create their own app markets would benefit from integrating such an open source framework to protect and promote a market being targeted by an attacker. Before downloading any app, a user can see what this framework has to say about the behavioral analysis of the app along with what features it has to offer.

2) ASEF in a large organization

Adding a layer of security between app hosted on a market and app installed on a user's device can be simply achieved by integration of this framework with an app store.

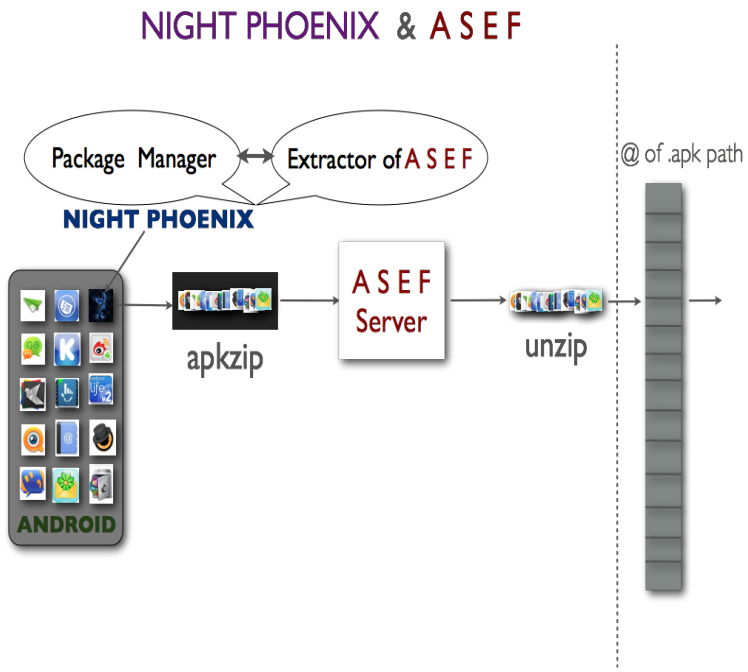
However what about the apps which are already installed on users' devices and also the ones that are downloaded and installed from a random place on an internet? How hard would it be to evaluate the security of hundreds of apps being used by number of Android users on various Android handset?

My first thought is to let IT deploy this framework on a central server and have them deal with testing each Android device one by one. Not only would that be unfair and inefficient for IT to perform such a routine, but it would be impractical for employees or users to leave their devices even for few mins until the extractor does its magic.

Without much of a thought process, it becomes quite obvious to overcome that physical barrier of a USB cable which provides a connectivity and communication channel between the device and the framework.

In order to do so, as a proof of concept, I developed an Android app which will be installed locally on each device and will collect all these .apk files and send it to this server running ASEF.

It resembles the extractor module of ASEF as we discussed before and executes a similar workflow by using the package manager library.



Night Phoenix will iterate all installed packages, find the location of the apk files for each installed package, filter which ones are user-installed applications and create a zip file of it on an sd card and send it to the server program listening at a predefined port on a machine running ASEF.

As I mentioned before, no matter what type of input is used, once the way to get the .apk file on a local machine is figured out, the rest of the design remains the same after the normalization mode of ASEF.

The alarm manager can be easily integrated to this agent in order to schedule scans just like cronjobs, where the user won't even have to bother about constantly remembering to click a Night Phoenix app to stay safe.

This idea can be utilized to manage the security of Android devices in a large scale organization.

Who Watches the Watchmen :

During my talk, I briefly discuss the story behind Night Phoenix and also discuss the twist in the story: While performing a security evaluation of Android apps, we ended up introducing our own app.

Real Time Is Too Late in Android Security :

Imagine performing security in real time for all these apps. Either something bad is happening or has just happened. Even while we catch it while it's happening, at times it's too late to revert or recover.

The idea of having an automated virtual environment where all these apps are migrated, exercised and evaluated for security measures before even entering a real world of Android seems like the way to go in future.

The Next Generation of ASEF :

This can be achieved by having an Open Source Framework like ASEF as the very nature of such a framework allows us to envision the next generation of automation frameworks through the eyes of ASEF.

The next generation of ASEF will be scalable to perform such tasks over thousands of apps within the constraint of limited time. Also, automated signature development can be achieved by various feeds for security bulletins and a signature generator that can detect vulnerabilities based on package name and affected versions.

It would be interesting to distinguish security fixes from regular updates being pushed out to apps so that users can prioritize which updates to apply immediately. A lot more effort is needed for smart correlation of data, statistics and insightful UI reporting of it.

ASEF in the Cloud :

Making this open source framework available in the cloud can make it easily approachable and available to the large spectrum of users.

However, it will still require a decent financial investment to provide such functionality to multiple users. It can be temporarily overcome by maintaining the results data from ASEF in a central repository in the cloud. This can save the time of re-evaluating everything over and over again. At any given time, if the user is not satisfied with the already-collected results, the option of running an ASEF test on a local machine will always be there!

One Stop Solution for All Our Android Security Needs :-

For the most part, Android users' security model is somewhat fragmented as it can be based on the security model of a market from where the app is downloaded or software that performs security analysis using a predefined set of measures or even an individual's efforts to conduct insightful research.

While we are trying to absorb, process, observe, interpret, analyze and understand various security aspects of this rapidly growing yet dynamically changing field of Android security, it becomes very important to address all of these individual issues - looking at both the fine details as well as looking at them collectively in a much more comprehensive fashion on a larger scale.

While challenging and ambitious at the same time, the idea is to have a one-stop solution for the entire set of our Android security needs.

In line with this methodology, we've made some novice attempts to make a difference in the field of Android security with the help of automation, in form of ASEF.

Philosophically speaking, I see it as a silver lining on an edge of the dark cloud. However, on a futuristic note, I envision having an open source framework for security evaluation of Android applications which will encourage security enthusiasts and researchers to contribute, collaborate and integrate their ideas and efforts and channel it through such an open source framework and take it to the next level where it almost becomes the 'one stop solution' for all our Android Security needs. When we get there, this open source framework will still be called ASEF, but this time it will be 'Android Security Evolution Framework'. Until we get there, give ASEF a try and help improve this project with your comments, feedback and contributions.