POLITECNICO DI MILANO

Corso di Laurea MAGISTRALE in Ingegneria Informatica

Dipartimento di Elettronica e Informazione

Static Detection and Automatic Exploitation of Intent Message Vulnerabilities in Android Applications

Relatore: Prof. Stefano Zanero

Tesi di Laurea di:

Daniele Gallingani, matricola 784235

Anno Accademico 2012-2013

Acknowledgements

I firstly want to thank Prof. Rigel Gjomemo for all the time he dedicated to me, helping and driving me through the development of this thesis.

Then, I want to thank Prof. V.N. Venkatakrishnan for the precious advice and for giving me the opportunity to work with his team.

Thanks to Henddher Pedroza for advising me while sharing common problems working with Soot and Heros.

I also want to thank all the people that shared with me these last years of study. Being part of such a great group, significantly improved my technical and collaboration skills.

Another thanks goes to Stefano Maggi for giving me sporadic but precious feedbacks.

Finally, a special thanks goes to Stefano Zanero for supporting me growing this research and for helping me concluding and improving this work.

DG

Ampio estratto

Negli ultimi anni molti ricercatori hanno evidenziato vulnerabilità derivanti dalla scarsa comprensione delle linee guida messe a disposizione da Android o, piú in generale, dal suo modello di sicurezza. Android, nello specifico, pone la comunicazione inter processo come uno dei suoi paradigmi principali. Nonostante ciò, la progettazione della sicurezza di questa comunicazione viene lasciata agli sviluppatori, ai quali viene richiesto di seguire precise linee guida. Android non impone esplicitamente nessuna linea guida per aiutare gli sviluppatori nell'utilizzare della vasta gamma di paradigmi di sicurezza che Android stesso mette loro a disposizione. Questo lavoro vuole evidenziare che in molti casi reali, queste linee guida non vengono seguite e che questo "cattivo comportamento" può provocare reali minacce per gli utenti finali.

In questa tesi presento la progettazione e l'implementazione di uno strumento per l'analisi di sicurezza delle applicazioni Android. Questo lavoro concentra la propria attenzione sulle informazioni scambiate tra diversi componenti interni alle applicazioni attraverso il meccanismo di "message passing" di Android. Il lavoro mira a scoprire automaticamente vulnerabilità che possono scaturire in seguito a pratiche non consone, o semplice distrazione, nell' uso dei meccanismi di scambio di informazioni fra sotto componenti interni o esterni alle applicazioni.

L'obbiettivo del lavoro, in pratica, è quello di produrre un tool indirizzato a qualsiasi sviluppatore che voglia testare la propria appli-

cazione. Lo strumento si propone di essere intuitivo e facile da usare.

Il tool consiste in un analizzatore statico del codice, capace di acquisire pacchetti compilati di applicazioni Android e produrre all'utente precise relazioni circa lo stato di sicurezza dell'applicazione presa in esame.

In aggiunta, viene utilizzato un approccio formale e corretto per produrre contenuti maligni che dimostrino la reale esistenza e efficacia di un possibile attacco.

Questo lavoro è stato originariamente pensato in seguito alla scoperta di alcune vulnerabilità, tramite una analisi manuale di applicazioni presenti sul Google Play Store. L'esplorazione ha confermato che in molti casi, anche i più semplici principi di isolamento dei sotto componenti sono violate, anche in applicazioni popolari e largamente utilizzate.

La mancanza di sanitizzazione degli input, che preverrebbe questa classe di attacchi, è la dimostrazione evidente della sottostima di questo problema nello sviluppo di applicazioni.

Nel primo capitolo introduco le caratteristiche del sistema operativo Android, con particolare attenzione alle sue opzioni di sicurezza e suo suoi meccanismi di "message passing".

Nel capitolo 3 descrivo nello specifico il "threat model" analizzato nella tesi con particolare riferimento ai tre modelli principali di risorse prese di mira da un possibile attaccante: server remoti, storage locali o interfacce utente.

Il capitolo 4 contiene le idee di base e le scelte progettuali che stanno dietro all'analizzatore.

Il capitolo 5 contiene una trattazione riguardante gli strumenti e le tecnologie utilizzate che permettono di raggiungere l'obiettivo di ricerca.

Il capitolo 6 è una spiegazione di come l'analizzatore sia stato us-

ato in esempi di applicazioni reali che ne hanno dimostrano l’efficacia.

Infine, il capitolo 7 contiene una trattazione dei benefici pratici derivanti dall’utilizzo dell’analizzatore e spiega come questo strumento possa essere esteso ed integrato in futuro per analizzare una più vasta area di problemi di sicurezza.

Summary

In this thesis I present the design and the implementation of a tool to analyze the paths that information exchanged by different process on the phone takes, in order to automatically detect vulnerabilities that may come out from bad programming practices or simple distractions while implementing Android applications.

The goal of this work is to produce a tool, targeted to every developer who wishes to test his application. It aims to be intuitive and simple to use.

The tool implements a static code analyzer able to take in input Android application packages and produce as output precise security reports on the analyzed application. A formal and sound approach is then adopted to produce malicious exploits demonstrating the actual feasibility of a possible attack.

The analysis performed focuses its attention to the facilities provided by Android to easily enable applications subprocesses to communicate.

This work was originally ideated after some manual exploration in real Android applications, downloaded from the store. This exploration confirmed that in several cases even the most simple isolation principles were violated, even by popular and broadly used applications.

The lack of exhaustive sanity checks preventing this class of attacks is an evidence of the underestimation of this problem in real world application development.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the last few years many researches have highlighted several possible vulnerabilities deriving from poor understanding of the official Android security guidelines[1] and of the Android security model in general. Android specifically makes interprocess communication and collaboration one of its main paradigms. Although this communication must be carefully designed by developers, that need to follow some basic design guidelines, such as Activity state isolation and so on. Android does not take explicit measures to enforce the design of applications to be intrinsically secure. This means that the decision of using or not the large amount of security features embedded in Android is left to the final application developer. This philosophy optimistically, assumes that every developer has a reasonably good security background, to understand and not to underestimate the problems that a non-defensive programming approach can cause.

This work highlights how in several real-world cases, the design guidelines are not properly followed, and this “bad behavior” can lead to real threats for the final users.

The goal of this thesis is to design and implement a tool capable of automatically detect programming choices, when developing Android applications, that can lead into security risks for the final user. The tool, after having demonstrated the existance of such risks, should eventually be able to warn the developer about these risks, and suggest a set of possible solutions. As described in Chapter 3, the subset of programming choices taken into consideration are the ones concerning messages exchanged, in particular accepted, by application’s Activities.

The rest of the thesis is organized as follows. Chapter 2 presents the state of the art of the Android OS. Chapter 3 describes more precisely which set of vulnerabilities this thesis deals with. Chapter 4 contains the core idea behind the analyzer. Chapter 5 is an overview of the tools and the techniques, along with design choices made in order to meet the research goal. Chapter 6 is a description of a real-world example of use of the tool. Chapter 7 contains a discussion of the practical benefits deriving by the usage of the tool and about how such tool can be extended to analyze

a broader circle of security implications.

Chapter 2

State of the art

2.1 Android OS Overview

In the past years Android OS had became the most widespread mobile operative system in the world, surpassing all the competitors by market share.[14]

One of the reasons for Android success is its *open source* nature. It has to be noticed that each phone manufacturer can easily customize and adapt to their needs the operating system, without developing a new one from scratch, determining drop in development costs, and a subsequent drop in the final product price. Of course, this possibility is not restricted to manufacturing companies, but everyone can change and build the code for various purposes such as bug fixing, development contributions or personal curiosity.

This exposition of the code, moreover, may lead to security problems: a clear sketch of the security features of the operative system can easily be delineated, leading to a broader knowledge of the security issues the operative system could have. This knowledge might be exploited for benign purposes such as fixing, but also maliciously, endangering the final phones users.

When a security issue is detected and fixed by the community, the propagation of this patch is, in practice, difficult for two main reasons: first, due to the high fragmentation in Android versions, it might become complicated to detect all the versions in which the bug is present. Second the modification performed by the manufacturers on the actual deployed versions of the operating system might be significant, leading into substantial differences with respect to the official AOSP branch. This misalignment usually makes it difficult for the manufacturer to keep their deployed versions up to date.

2.1.1 Android OS Architecture

Android OS is a comprehensive operative system built on the top of the Linux Kernel. It is structured as a stack consisting of four logical layers, composed by different modules written in Java, C++ and C.

- **Applications**: these are the actual programs the user interacts with. Applications can be written either in *Java*, with the support of the official Android SDK, or in *C++* over the official Android NDK.

- **Application Framework**: this is a set of services handling applications lifecycle, message passing and content sharing between applications and managing application specific resources such as images and localized strings.

- **Libraries and Runtime**: Libraries are a collections of facilities available for both application framework components and applications. They provide a unified way to access database storage, web browsing components, media players and so on. The runtime enables each application to run in its own instance of Dalvik VM.

- **Linux Kernel**: this is the lower layer of the infrastructure. In addition to the Linux Kernel, in this layer are also present hardware drivers, in charge of communicating directly to the phone's physical components. These drivers are clearly device specific and there exist phone specific build types in AOSP, containing specific driver binaries (sources are usually not available).

Figure 2.1 shows the complete OS architecture, along with all of the macro-components.

Java code of Applications and libraries is compiled into Dalvik bytecode, after being compiled into the classical Java bytecode. This bytecode is specifically designed to run on mobile devices (ARM processors), carrying some specific optimizations. Its main characteristic is to be register-based, unlike the classical Java bytecode which is stack-based. The Dalvik VM was designed to run efficiently multiple VM instances so to guarantee isolation among processes. From Android 2.2 a built-in JIT compiler has also been added to the Dalvik virtual machine, in order to speed up the runtime execution of Android applications.

This thesis focuses on the first two layers of the described hierarchy, i.e. the *application layer* and the *application framework layer*.

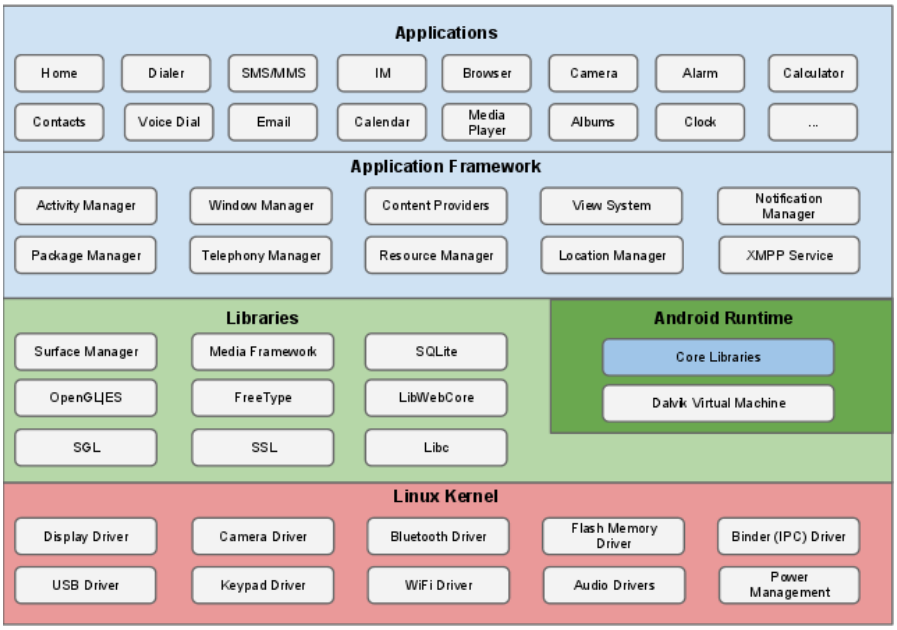

Figure 2.1: Android OS architecture. Source: `http://source.android.com/tech/security/index.html#android-platform-security-architecture`

2.1.2 Android OS Security Model

Android claims to be designed to be the most secure and usable mobile operating system, providing automatic mechanisms to guarantee user data and resources protection along with application isolation.

The OS strongly relies on the consolidated Linux security model in which process isolation is guaranteed by design, except for a (reduced) set of privileged code fragments that run with root privileges. Linux policies on user resources ensure that *user A* will not access CPU resources, memory, files or devices belonging to *user B*. The concept of user isolation is exploited by Android in order to provide a complete application isolation (*application sandboxing*). Differently from the traditional Linux implementation, in Android, each application is assigned to a unique user ID. Each application will then run into a separate process belonging to its user. This ensures that no unauthorized data access can occur at native code level.

As mentioned above application sandboxing is also ensured by the fact that com-

pletely separated instances of Dalvik VMs are started for each new application that is launched. This guarantees complete isolation also at Java code level.

Together, these two approaches should prevent memory corruptions to compromise the security of the device. Nevertheless application Sandboxing is breakable on a non properly configured device.

Another higher level security feature that Android OS provides is the so called *Application Permission Model*. By default, applications can access a quite limited number of resources. Most of those can only be accessed through OS calls. Applications, in order to access non-default features, must declare their intention of usage: a misdeclaration of a requested resource causes a security exception, raised by the application framework, leading to a failure of serving such request. The *Application Permission Model* contains a fine grained, hierarchical list of capabilities applications can declare. The choice between allowing or not applications to access the declared resources is demanded to the final user at installation time. Unfortunately the capabilities cannot be partially accepted/unaccepted, so the only way a user has to prevent an application not to access a specific capability is not to install the application at all. This choice is presented to the user when installing applications, whether they are downloaded from the official Google Play Store or from unknown sources.

2.1.3 Android OS interprocess communication

A single Android application package can contain several (independent) components. Each of those components, along with its type has to be declared in a *manifest* file. Components can belong to one of the following categories:

- **Activities**: Activities generally contain the code for one single user-focused task. Activities have associated UI components the user can interact with. Typically, the main entry point for an application is a distinguished activity (declared in the manifest file).

- **Services**: Services are pieces of code that run in the background independently from the current context presented to the user. Services communicate with the external environment via interfaces available through remote procedure calls.

Due to sandboxing, application components cannot directly share information or send messages one to an other. In order to let such isolated processes to communicate, the OS has to provide a controlled mechanism. Android interprocess communication is made possible by the following three elements:

- **Binder**: It is a lightweight procedure call mechanism, mostly used when the communication demands high performance communication. It is implemented using a custom Linux driver. Exploiting Binder, the *Services* discussed above can efficiently exchange information.

- **Intents**: Intents are objects representing simple messages that can be exchanged by different processes. The name Intent was chosen because of the fact that these messages usually are sent when a process has an *intention* to interact with another process. Intent messages either carry information on a specific destination, or express a generic request that needs to be served by some process able to manage it. Intents can also act as broadcast messages informing a set of interested processes that some status change has occurred (i.e. a network status change information may be interesting for both a browser and a chat application that could notify the user in case of a connection drop, or resume a paused operation after the network status returns available).

- **Content Providers** Content providers is the name given to the interfaces used to expose some particular data, generally directly available for all the processes. An application may want to expose some data it controls to other applications; this can be done by the exposer application by implementing a

content provider interface, in which the process specifies its behavior in response to queries from the outer world.

Intents are the fundamental communication element in the Android interprocess communication system.

2.2 Related Work

In this section we present three main relevant research areas in Android security.

- **Permission analysis**

  When installing an Android application, the user is requested to accept and allow the application to access user and system resources (personal informations, photo camera, GPS location, ...). This is done presenting by to the user a list of all the resources required by the application to work. It has been proven that such a choice is not optimal for two main reasons: first, users might not have an adequate understanding of the meaning of the entries in such list and the security issues that accepting a given permission can cause. Second, even experienced users were found not to pay the required attention to this screen, mainly because is often long and verbose. Other research areas are interested in analyzing the risks and the conceptual flaws of this permission model.[10][8]

  Other problems when dealing with Android permission arise because of the high usage of inter application communication. An application that has not declared the usage of the system contacts can use another application, enabled to read user's contacts, as a proxy for this information, as long as one is installed on the phone. This means that the permission list that the user need to accept might not be exhaustive.[6,9]

- **Automatic malware detection**

  Android by default only allows to install applications downloaded from trusted sources (i.e. the Google Play store). Anyway, users can easily install applica-

tions coming from arbitrary sources (even email attachments) just by disabling an option in the OS settings. Unfortunately, even applications installed from the Google Play Store are inadequately controlled for the presence of malicious code. This has pushed a large part of researches in the Android security world to come up with techniques to automatically detect malicious pieces of code in applications. In this area, which is the broadest in Android security, many approaches have been proposed, from the more formal to the more empirical ones.[2,5,15,17,18]

- **General security toolkits** Another broad area of research concerns the study of automated systems to automatically detect vulnerabilities in applications that could be exploited by malicious applications to access user's personal data, fake user's behavior or damage the system itself. The toolkits basically can be used to test applications against a given subset of vulnerabilities.[12][11]

  Android ICC (Inter Component Communication) security has experienced an increasing concern in the past years. The current state of the art is presented in[13].

  One remarkable research that inspired this work is Quire[7]. Quire is a lightweight framework to enhance Android IPC mechanism by adding message provenance. The implementation relies on a signature scheme that allows message recipient validation before delivering. The class of attacks they want to prevent is closely related to the ours: *confused deputy attacks*. Their approach needs to modify the Android OS in order to guarantee the described security features, our approach aims to obtain a similar result by preventing behaviors that can potentially introduce vulnerabilities in applications.

Chapter 3

Problem Definition

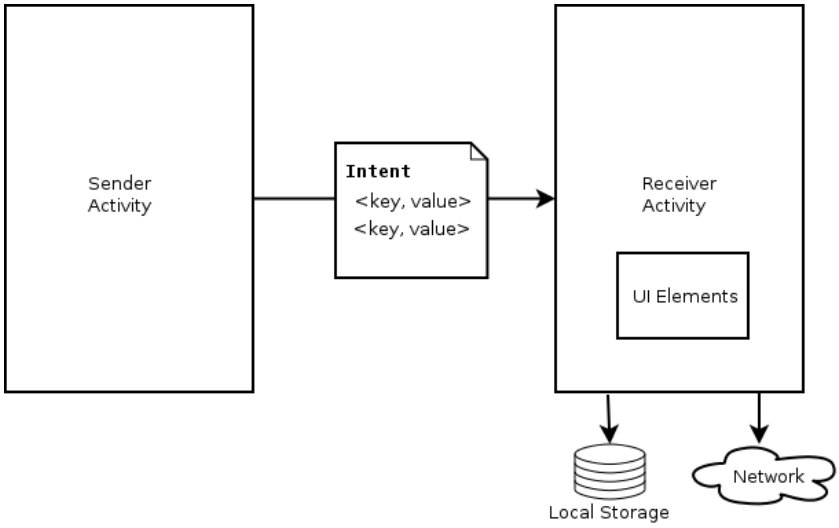

Figure 3.1: Activity data exchange model

3.1 Overview

In order to define the problem we will adopt a model consisting of two activities: the first, identified as *sender*, produces as output an Intent carrying a set of messages targeted to another Activity, identified as *receiver*. The *receiver*, after being notified that it is a suitable recipient of the Intent, extracts the information in the messages in order to complete the operation specified by the logic of the Activity itself.

3.1.1 Messages as Collections of Parameters

Without loss of generality, we can think that an exchanged message consists of a collection of named parameters (*Extras*). The names (keys) uniquely identify a piece of information inside the Intent, therefore the *receiver* usually relies on its prior knowledge about these keys in order to retrieve a particular information. This can be viewed as a protocol that both the *sender* and the *receiver* should implement in order to let the communication succeed.

Given a key, its corresponding payload can have an arbitrary type: the Intent Java API accept all the primitives Java types, but also a generic *Serializable* object.

Table 3.1: INTENT API METHODS FOR EXTRA MANIPULATION

Setters	Getters
$getBoolean(String\ key)$	$putBoolean(String\ key,\ boolean\ value)$
$getByte(String\ key)$	$putByte(String\ key,\ byte\ value)$
$getChar(String\ key)$	$putChar(String\ key,\ char\ value)$
$getDouble(String\ key)$	$putDouble(String\ key,\ boolean\ value)$
$getFloat(String\ key)$	$putFloat(String\ key,\ float\ value)$
$getInt(String\ key)$	$putInt(String\ key,\ int\ value)$
$getLong(String\ key)$	$putLong(String\ key,\ long\ value)$
$getSerializable(String\ key)$	$putSerializable(String\ key,\ Serializable\ value)$
$getShort(String\ key)$	$putShort(String\ key,\ Short\ value)$
$getString(String\ key)$	$putString(String\ key,\ String\ value)$

To retrieve these variables, then, the Activity willing to use them also needs to know their exact type. The API provides specific utility methods for each of the primitive types plus a generic $getSerializableExtra(String\ name)$ method that returns an object implementing the *Serializable* interface: the access to the object's informations will be available only after a cast to its original class.

Table 3.1 lists the main API methods. These functions are also available in their arrayed fashion, allowing extras to carry multiple values under a single key.

The Intent APIs provide reflective interfaces to retrieve all the *Extra*s a message includes. The use cases of this approach are however limited mainly to test purposes. It is usually convenient to explicitly name the parameters, providing them an explicit semantic expressed by the name itself.

The proposed analysis deals only with primitive types, leaving out more complex structures such as Objects. In practice, this is not a real limitation, since strings represent the largest amount of data exchanged.

3.1.2 Activity Lifecycle

Android Activities are managed by the help of a stack structure: the Activity currently running is the one at the top of the stack and it is in a *running state*. When a new Activity is launched, the one previously running is moved to either a *paused state* or a *stopped state*.

An Activity becomes paused when the new launched one partially covers it; it instead becomes stopped when it is completely covered. In this case, since the Activity is not visible at all, there is no need for it to keep running.

A running garbage collector can ask both paused and stopped applications to exit their execution, or it can simply kill them.

In Figure 3.2 is delineated this lifecycle, complete of all the methods that are invoked on the Activity by the system when a status change is requested.

Whenever there is a match between the Intent recipient and an Activity, this Activity is *always* created and pushed on the top of the stack. Thus the logic for handling the received data has to be encoded starting from the $onCreate()$ callback method. The Intent data will also be available after a $onResume()$ on a $onRestart()$ call since such data will be preserved when the activity is either paused or stopped. Since it is not reasonable for the Activity logic to wait until a stop or a pause request occurs to handle the received data, for the following analysis, it will be assumed that $onCreate()$ is the single *entry point* for the data, and the single point where the operations performed on the data start.

3.1.3 Activity Accessibility

Same-application Activities can always exchange data. Due to application isolation and sandboxing, Activities are not allowed, by default, to receive data by processes not belonging to the same application. This intention has to be explicitly declared in one of the ways listed below in the application manifest.

We can easily divide Activities in two macro groups:

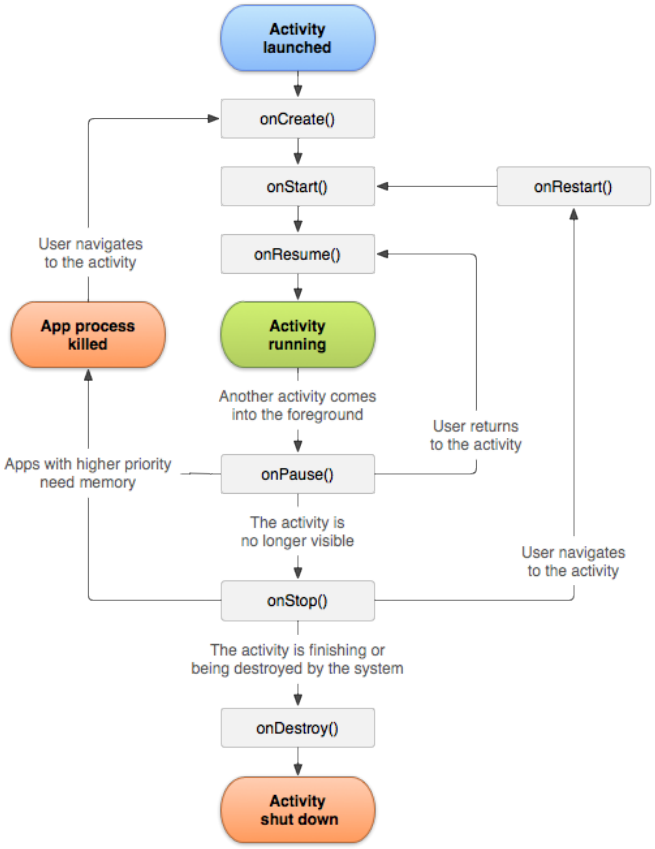

Figure 3.2: Android Activity Lifecycle Explained. Source: `http://developer.android.com/reference/android/app/Activity.html#ActivityLifecycle`

- **Intra Application accessible**: processes that respond and start only after receiving commands started from processes inside the same application.

- **Inter Application accessible**: applications that can be reached by Intents started by any other process in the system. Android provides two paradigms that activities must follow in order to be world accessible: *Intent Filter* and *Content Provider*.

  The paradigm that *Intent Filters* implements mainly consist in functionality delegation: processes delegate to a (set of) other processes a given operation that the processes responding to the Intent promise to accomplish.

  Content Providers, instead, as already discussed, are a paradigm for data exchange. Although Content Providers are public by default, they can be kept private (only accessible by the application) by setting the corresponding option in the application's manifest.

As specified in the *IPC* section of the "Security Tips" section in the Android developer guide, every Activity is responsible for the received data, and should perform input validation, since Intents and Intent filters cannot be considered security features.[1]

3.2 Android Root Access

As consequence of the Android's open-source fashion, many communities working around the project have raised. The work of these groups mainly focuses on modifications of the official project, mainly in terms of addings UI functionalities and specific settings. One of their claims, is manifested in the will of avoiding the control of the phone by the phone's manufacturer or by Google on the users' devices. They see the impossibility of the processes, other than the system ones, to run with root privilege as a fundamental lack of freedom. Beside this, they perceive the locking, by the manufacturers of the boot loaders as a constriction, since the phone

Table 3.2: ANDROID ROOT MARKET IMPACT

Name	Application/OS	Downloads (thousands)
Superuser	Application	10,000 - 50,000
Rom Manager	Application	5,000 - 10,000
Titanium Backup Root	Application	5,000 - 10,000
CyanogenMod	OS	4,200 *
Wireless Tether for Root Users	Application	1,000 - 5,000
Root Explorer	Application	500 - 1,000

** currently installed*

Data updated on March 9, 2013. Sources: Google Play Store, CyanogenMod Statistics.

users loose the capability of changing the built-in operative system with a desired one.

They started early, with respect to the Android birth, to produce tools able to "jail-break" (or "root") the phones (removing the constrains applied by the manufacturers) to let the users install any kind of software on their devices (either trusted or untrusted), having the possibility to escalate their privileges to root.

Understanding the penetration of such communities in the real phone market is hard. Unfortunately, there are no official reports on the number of rooted devices, or of devices mounting aftermarket Android versions.

We can anyway deduce that the impact is not marginal by taking a look at the number of downloads that both the most famous versions and their related applications received.

In a rooted environment, since every candidate process can obtain root privileges, the security risks for the user increase dramatically, since the application sandboxing model is compromised. A root process, in fact, no longer obeys the sand box model. Such process can directly access and modify any local data, send to

any other process an arbitrary message, and launch any kind of command. When sending a message to another process, this data sent with root privileges will never be ignored, regardless of the security and isolation policies mentioned in the previous paragraph. As s consequence, developers simply cannot ignore the existence of rooted devices, but they should build applications in a defensive way. Intuitively, they should perform sanity checks on the data received even on Activities which access should not be public by specification.

3.3 Preliminary Analysis

The problem presented in the following sections was first noticed by monitoring inter-Activity Intent payload exchanges from some popular applications downloaded from the Google Play Store.

This simple monitoring was possible with the help of some instrumented classes in the Android libraries, in particular the ones concerning Activity lifecycle and general Activity management.

3.3.1 Instrumentation Class

The Android Java Core framework provides utility classes to instrument and monitor Activities lifecycle and its interaction with the system. When the instrumentation is enabled, the Instrumentation class is automatically instantiated before running the application code. This class is designed to be used by the code in the application itself, i.e. the monitoring logic has to be implemented inside the application.

Since for a preliminary analysis it may be hard and time consuming to inject code in the applications under consideration, we adopted a naïve approach. We modified the official Instrumentation class code, in order to make it log for us on a phone directory all of the Intent data traffic, regardless of the application or Activity currently running.[1] This was obtained enhancing $execStartActivity()$. The method is

[1]The experiment was done on a custom Android 4.0.2 build, targeted for the Android Emulator

called exactly before the Activity is created, and thus, exactly before the $onCreate()$ method is called on the Activity itself. In this way, we have the guarantee that the Intent cannot receive further manipulations, after we capture it.

It has to be noticed that the folder in which the data is saved must have world writable permissions. This is because the Instrumentation class code runs in the context of the currently running application, so it will become the application itself in charge of writing in the directory.

The recorded data then can be easily dumped off the phone and analyzed off-line.

3.3.2 Manual Testing

Android provides several testing and debugging facilities to automatize the testing process with a batch sequence of commands: in particular, am was found very useful to emulate sender Activities behavior. am is a shell tool to interact with the Android runtime, and it can be exploited to create and send Intent populated with Extra parameters.

```
am [start|startservice] -a <action> -n <component> -e <extra_key> <extra_value>
```

The command specified above, launches a generic component (can be either a Service or an Activity), specified in the $-n$ option, requesting it to perform an action ($-a$, typically VIEW), passing to it the set of Extra parameters specified with the $-e$ option (there exist several $-e$ type-specific options, $String$ is default).

This allowed us to manually, but easily, reproduce a crafted copy of the collected messages, and see how changes in message bodies were affecting the execution of the targeted activity, or the internal state of the whole application.

We were eventually able to change UI elements in the Activities, and to actually push text in the application local storage. This meant that no particular check

(*full-eng*).

Instrumentation class can be found under $< aosp\_root > /frameworks/base/core/java/android/app/$

was performed, but Activities were blindly accepting input strings. In the following section we describe the risks that such programming behavior creates.

3.4 Threat model

In this section are delineated the attacks and the threats to the final user that could come out from a non-defensive approach when dealing with received Intent payloads.

A malicious message could harm or change in an unexpected way the status of the application if one of the following conditions is satisfied.

Exposed Activity: Activities that have explicitly requested to be exposed, for data or functionality sharing, are, in a sense, world-accessible. Their access, can not of course exclude malicious requesters.

Unexposed Activity, Root access: Under this scenario, even not exposed Activities cannot be considered immune with respect to hostile messages. As discussed before, root applications can overcome the limits imposed by the sandboxing.

In particular, as shown in Figure 3.1, a malicious message can basically be targeted to one of the following resources:

- **Network**: a malicious message can be forwarded to a server. In this case the attack may not directly include the application, but use its code as a bridge to access and modify information on the server.

- **Local Storage**: the message may include parts able not only to store unwanted information on the application's local storage, but also to also to corrupt previously existing data.

- **UI Elements**: message parts may be used to compose notes aimed to notify the user of a certain situation. Crafted messages may lead the user to have a wrong perception of the current situation.

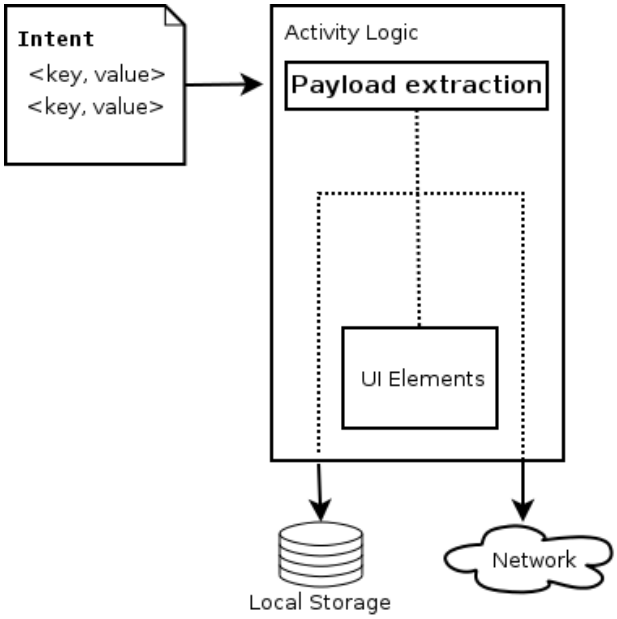

Figure 3.3: Attack target resources

Intuitively, the resources can be targeted by those kind of attacks, only if the the payload of the Intent is directly used as payload for new piece of information sent to the resource, or if there exist a flow in the code that connects the Intent payload to the resource, and such flow does not include any sanity check neither on the directly involved parameter nor on another related parameter.

3.4.1 Attacker

In this scenario, we can think at an attacker as a normal application, installed on the user's phone, by the user itself, via either the Google Play Store or an another third party source. The application may carry some malicious code along with real functionalities, used for defacing purposes. The malicious code could remain silent until the user triggers a specific action on the malicious application, believing in the benignity of the used Application.

3.4.2 Victim

The victim, for this class of attacks, is the final phone user. The user act a fundamental role in this scenario, since he is the one eventually triggering the attack against himself and also that will perceive the changes or the anomalies on its own resources.

Users usually tend to trust applications downloaded from the Google Play Store, because the common sense suggest it to be a “safe” source. In reality, uploaded applications are not exhaustively tested and verified in actual environments by the Google Play Store staff, so there is not an actual guarantee that applications, even if they are distributed by a certified entity, do not embed malicious code fragments.

One may think that malicious applications could be considered suspicious because they may declare unusual or apparently not required permissions. It has been shown that, in practice, the effectiveness of such declarations is pretty low and that users, even if they are actually able to understand them, tend just to ignore the permission screen prompted on application installation.[8]

3.5 Characterization of the Attacks

The exploitation of this class of vulnerabilities result in a very broad set of use cases. The attacks can be classified in three categories, accordingly to the resource they target:

- **Server attacks**
- **Database attacks**
- **Phishing attacks**

3.5.1 Server Attacks

The attacks described in this section can be compared to Cross-Site Request Forgery web attacks. In CSRF attacks, the attacker makes use of a script or a crafted re-

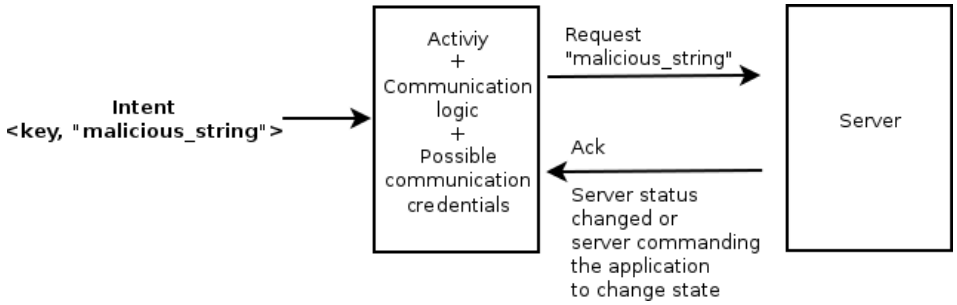

Figure 3.4: Server attack schema

source source URL in the page (typically an image) to induce the browser to directly or indirectly send a request to the server. The sent request is usually targeted to a route on the web server that accept GET parameters in input (usually to target of some form in the web application) to change the status of the server itself.

```
<img src="http://bank.com/transfer?from=Alice&amount=1000&to=Bob">
```

In this example, if we assume that the goal of the attack is to transfer money from Alice to Bob and the example image tag is somehow rendered by Alice's browser along with all other DOM elements of the Alice bank page, such browser will send the request to the server, believing to retrieve an image. Instead of responding with an image, the server will change its status if no extra check is performed on the logic in charge of serving the request: the attack will succeed.

The attack relies on the fact that since the resource is requested to be loaded in the same context (domain) of the rest of the application, in particular of the attacker target URL. In this scenario the user session identifier for the application will be also sent along with the request produced by the attacker. This implies that if the authorization security control in the target page only relies on the user session identifier, such page will have no way to distinguish between a request performed in the normal flow and another forged by an attacker.

We can think our Activity attack as a special CSRF attack where a malicious application is sending an Intent message to a target Activity he knows it will perform a request on the server. If this request will be created embedding some of the

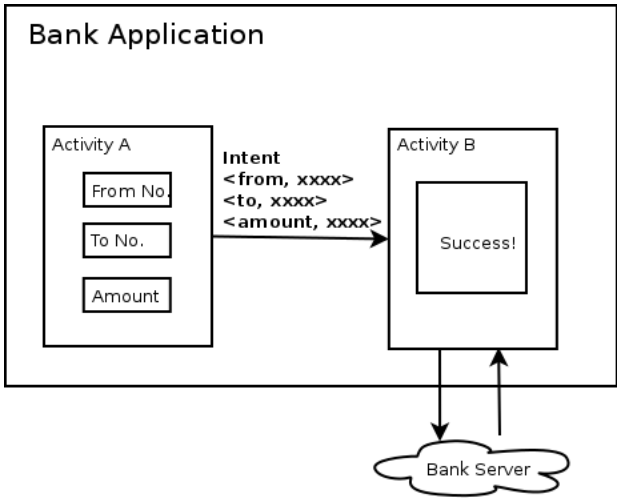

Figure 3.5: Server attack example

received Intent extras, an attacker may accurately change the extra values in order to manipulate the state of the server. Again, similarly to CSRF attacks, the request will produce the desired result, because it will be populated by the Activity, with all the session identifiers equivalent that the Activity use to attach when normally requested to perform this action. Also in this case, if no extra security mechanism is implemented, nor the Activity, nor the server will be able to distinguish between a normal and a malicious request.

The example below shows the Activity equivalent of the previous bank example.

Suppose to have a bank application where the money transaction is implemented by the use of two Activities and they behave in the following way: the first Activity collects recipient and amount data from the users and sends these data to a second Activity via an Intent message. Suppose then that this second Activity (processing Activity) is the one in charge of sending the transaction to the bank server, showing a load indicator and eventually a success message.

If no extra security mechanism is provided (such as a request validation token), an attacker may send to the processing Activity a message commanding it to perform a transaction of an arbitrary amount to an arbitrary recipient. This transaction will be prepared by the Activity and successfully sent to the bank server. Of course

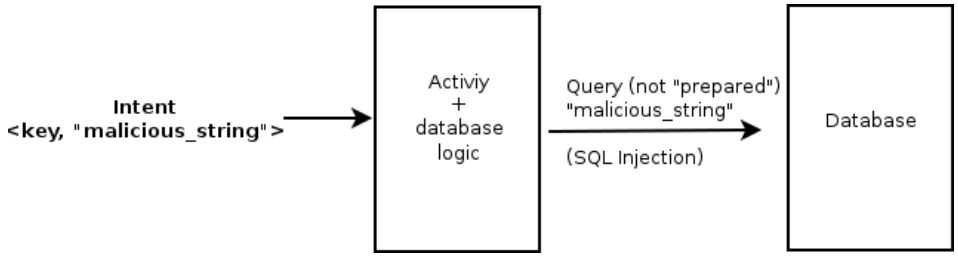

Figure 3.6: Database attack schema

this attack relies on the fact that there is an active user session in the bank application at the moment of the attack.

This class of attacks is anyway harder to perform, since their intrinsic architecture is more convoluted, even if they can directly exploit application local knowledge such as authentication tokens and so on.

3.5.2 Database Attacks

Under the usual assumption that no check is performed on the data extracted from the Intent, and that such data flows to the database, here classical SQL injection attacks can be performed.

Android offers APIs to deal with SQLite databases. The API are designed to strongly recommend the developer to use prepared statement (one for each insert, update, delete operations) so to explicitly prevent most of the SQL injections risks. It anyway offers a method to execute arbitrary SQL code: $execSQL(String\ sql)$. As shown in Figure 3.6, a malicious string that flows from an Intent message to the database, without passing through any prepared statement or extra sanity checks, could eventually damage or compromise user's local data.

I will now show with an example how not even prepared statements are enough to defend the application logic. Suppose an application storing sensitive data that requires a PIN to be accessed.

Suppose that, similarly to the previous example, the " change PIN" feature is implemented by two different Activities, the first requiring the user to insert the

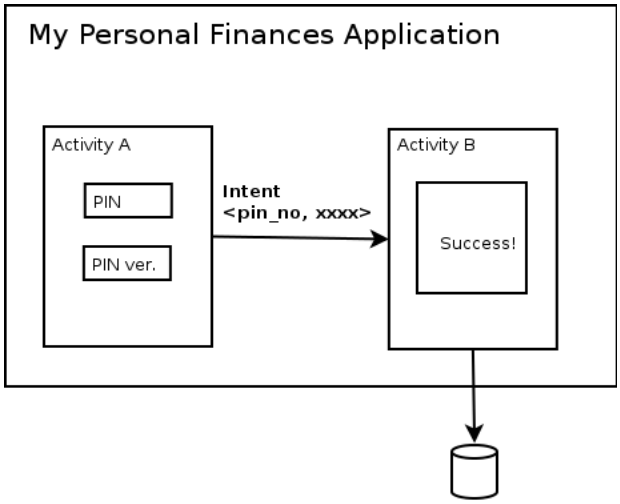

Figure 3.7: Database attack example

PIN and the second performing the update of the PIN on the local database and showing a success dialog. Of course there is no need for this Activity to be exposed to the outer world, since a PIN change is a typical internal operation. Suppose the application is installed on a rooted phone and the attacking application has obtained root rights. Under these assumptions, the attacker may send a message to the second activity that will change the local value of the PIN number into an arbitrary one. The application will then deny further user accesses to the application because she will not have any idea of what the new set PIN is.

One may say that there is no need to pass through an Activity to access local data, if the intentioned Activity has the rights to do so (root). This is generally true if the application's local data are not encrypted. An attack that flows trough the Activity, will easily overcome the encryption, because it will directly exploit the code of the Activity, executed in its specific context, that should have been designed to cope with encryption.

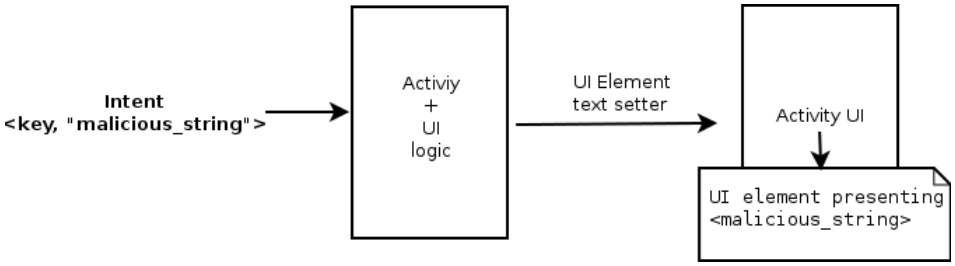

Figure 3.8: Phishing attack schema

3.5.3 Phishing Attacks

This class of attacks can be performed whereas the Activity presents some received payload data to the user.

In this an attacker may use application visive context to let the user believe some convenient (for the attacker) fact, or to give to the user a wrong perception of the status of the application.

Suppose there is a given application that contains an Activity used to notify the user or to give to her some information about the status of the application itself. Suppose this Activity is just used by the application to display messages that it receives in input with an Intent message, i.e. it has not any kind of logic. An attacker could start this activity feeding it with arbitrary strings that will be prompted to the user.

To make this attack more effective, an attacker application may have been designed to launch a series of target application's Activities that exactly reconstruct and reproduce the normal navigation flow of the application. Doing so the user will be induced to think that she was the author of the flow of actions leading in this last screen, sometimes in the past.

Chapter 4

Design

4.1 Overview

As introduced in the previous chapter, Android does not explicitly force developers to design applications in a defensive way. Most of the problems described can be solved just by designing applications in a more conscious way, following the design guidelines indicated in the official developer guide and using Intent extra messages to exchange information only when strictly necessary.

For instance, thinking at the scenario described in Section 3.5.3, a different approach to avoid to transmit text to be displayed into Intent extras exists. We can think at two different use cases where Intents are convenient here. One where there is a set of pre-defined messages that are all eventually prompted in the same Activity, as a general information displaying screen, useful to reuse UI components. The other where the Activity is used to provide notifications to the user, and the text of the notification changes every time because, for example, it contains parts of an e-mail, or the event that required the user to be notified. In both of the cases can be convenient to design lightweight Activities that simply takes an arbitrary set of strings in input and place them on the screen. These Activities can be simply launched and populated through an Intent call.

A more clean and safe solution for the first case could be to directly hard code all the possible messages in the displaying Activity binding them to a meaningful ID. When there is the need to display a specific message other Activities can make use of this ID to command to the displayer which message has to be prompted. In this way we ensure that all the messages displayed to the user can not come from an untrusted source.

Unfortunately, if the messages to be shown are variable by nature, this approach does not apply. In this case the “clean” alternative to the one described above, could be to create a queue of messages to be dispatched in the local storage, that the notifying Activity consumes when notified. Here an Activity willing to use the notifying Activity communicate to the user can store the message in the local database and

command the second Activity to consume the last message in the queue, using it to populate the UI. If this part of the storage is reliable (encrypted, and not vulnerable to what described in section 3.5.2), is guaranteed that only notification generated from trusted sources will eventually be displayed.

This is just an example to show that a safe data handling strategy is possible.

4.2 Goals of the Analysis

The goal of this work is to design and provide a tool, that by statically analyzing the code, can tell to the developer, what kind of bad behavior is encoded in the application, and where exactly it occurs, so a responsible programmer can go back and immediately fix it.

The problem can be transposed to a statical code analysis problem, aimed to analyze the control flows that Activity extra parameter values take when received by an Activity. A demonstration of a vulnerability is obtained when there exist a control flow leading a parameter value to a set of API calls labeled as vulnerable. This set of API calls includes HTTP requests population (network), database non-prepared statements calls and UI elements text setters.

The static analysis should then be able to:

- **Detect Starting Points**: detect the code instructions after the parameter values are delivered to the Activity (the Intent management is delegated by the Activity Manager to the matching Activity)
- **Follow variables control flow**: follow and track all the statements that include or use the values for further computation
- **Deal with aliased variables**: follow and track also all the variable that aliases the variables containing the parameter values
- **Deal with branching**: effectively and smartly deal with branching and different control flow

- **Deal with different scopes**: deal with method calls that receive values as parameter, by following the control flow of these parameters

Moreover, in order to demonstrate that the results discovered by the static analysis are feasible at run time, a sound analyzer is needed. Such analyzer should be able to:

- *Reconstruct paths*: derive the set of values that the *entry point* variables must have so that execution reaches a specific *vulnerable statement* along a specific *computation path*.
- *Discover malicious values*: (try to) derive the set of values that the *entry point* variables must have, so that execution reaches a specific *vulnerable statement* with the desired values.

4.3 Entry Points Definition

As described in Section 3.1.2, we can consider a single entry point for the Intent to enter the Activity, i.e. the $onCreate()$ method. Every time an Activity needs to handle an Intent message, it is newly created and the corresponding creation handler method is invoked by the Activity Manager.

This ensures that defining entry points for the analysis for all the $onCreate()$ methods present in the application is an exhaustive assumption.

```
1. public void onCreate(Bundle savedInstanceState) {
2.  super.onCreate(savedInstanceState);
3.  String paramValue = getIntent().getStringExtra("A_PARAMETER");
4.  doSomethingWith(paramValue);
5. }
```

The variable $paramValue$ defined at line 3, is a suitable candidate for the analysis, so the specified $onCreate()$ method will be used as entry point.

Table 4.1: EXAMPLE OF RELEVANT STATEMENTS

Statement Type	Statement	Tracked
Assignment	$int\ b\ =\ doSomethingWith(a);$	Yes
Assignment	$int\ c\ =\ a;$	Yes
Assignment	$int\ c\ =\ a+\ 7;$	Yes
Method Call	$doSomethingWith(a);$	Yes
For	$for\ (int\ i\ =\ 0;\ i\ <\ a.size();\ i++);$	No
If	$if\ (a\ >\ 0);$	No

Suppose that only variable a is being tracked

4.4 Data Flow Analysis

In this section are described all the problems the analyzer has to deal with when analyzing the code.

4.4.1 Variable Path Following

We are interested in tracking every statement that uses one of the variables labeled as *entry point candidate*. The concept of using a variable, given a statement, can be seen as checking whether a given statement contains that variable on the right hand side of the assignment or not, and if no assignment is included in the statement, check id there is a method call that contains the variable in its argument list.

So we reduce the problem of following the flow paths taken by the data as tracking all the statements that uses the variable containing such data.

In Table 4.1 are listed a set of statements that can be relevant or not to track in the analysis.

In order to be able to communicate to the user which is the vulnerable piece of code, and more precisely, which is the parameter that causes the vulnerability, during the analysis, all the tracked statements must be related to the variable that

caused them to be tracked.

After this phase the tool should produce a data structure of the type $< parameter\_variable,\ [statements] >$ for each parameter that needs to be tracked.

4.4.2 Variable Aliasing

Since there is no guarantee of the fact that originally tracked parameters will not be identically assigned to new variables in the code, the analyzer, while executing, must dynamically add to its tracking set also all the variables that aliases the base ones. The algorithm of the analyzer should be general enough to treat generally both base parameters and the ones added during the execution.

In addition, for each aliasing variable, it has to be provided a reference to the original variable. For this reason we introduce a slightly modified version of the previous structure including the reference: $< aliasing\_variable,\ variable\_ref,\ [statements] >$

The reconstruction and the union of the statements that either directly or indirectly reference a parameter is left to further steps of the analysis.

4.4.3 Inter Procedural Analysis

In order to provide correct and exhaustive results, the analysis has to take care of methods calls and therefore of different variable scopes.

One approach might be to treat function parameters as special case of aliasing variables, where their traceability is limited to a specific call context occurred with a specific argument list. This means that for two different calls of the same method, with two different tracked variables passed, has to produce two completely separated statement lists. Similarly, two different methods that locally share the same variable name, called with the same value, must produce two completely separated sets: no overlap can occur.

To do so we need to augment the previously described structure with an additional information: a method identifier. The resulting structure has this shape: $< aliasing\_variable,\ variable\_ref,\ method\_id,\ [statements] >$.

The structure is instantiated when the method call occurs binding the argument ($variable\_ref$) with the corresponding parameter ($aliasing\_variable$). It has to be noticed that this approach is general enough to deal with nested method calls. The binding with the original parameter, when it is passed through multiple method calls can be obtained by navigating the $variable\_ref$ chain until a non-aliased variable is encountered.

4.5 Vulnerability Testing

As last step of the analysis, the remaining thing that has to be done is to check the statement lists of all the tracked variables against a *vulnerable list*. This list includes the methods signatures of all the API calls considered dangerous for our purposes.

Whereas a match is found in the list of statements corresponding to a parameter, that statement is marked as vulnerable and as consequence a warning presenting both the statement and the parameter is generated.

4.6 Example

Below is presented an example of the analysis performed on a small piece of code.

```
1.  public void onCreate(Bundle savedInstanceState) {
2.   super.onCreate(savedInstanceState);
3.   String paramValue = getIntent().getStringExtra("A_PARAMETER");
4.   doSomethingWith(paramValue);
5.  }
6.
7.  public void doSomethingWith(String value) {
8.   doSomethingElse(String value);
9.  }
10.
```

```
11. public void doSomethingElse(String v) {
12.  doSomethingBad(v);
13. }
```

The analysis, after line 3, will have a list of solutions containing $paramValue$ and an empty parameter statement list.

When the analyzer encounter line 4 the solution set will contain two elements, shown in Table 4.2:

Table 4.2: SOLUTIONS AFTER LINE 4

<paramValue, [doSomethingWith(paramValue);]>
<value, @paramValue, []>

Something similar will happen encountering line 8, in this case the binding is done between variables $value$ and v.

In Table 4.3 are listed the results obtained at the end of the analysis.

Table 4.3: RESULTS AT THE END OF THE ANALYSIS

<paramValue, [doSomethingWith(paramValue);]>
<value, @paramValue, [doSomethingElse(value);]>
<v, @value, [doSomethingBad(v);]>

In the end, supposing that $doSomethingBad()$ is a method contained in the vulnerable method list, such statement will be marked, so will be v. The marking will be propagated trough $value$ and eventually to $paramValue$ which is a *base* parameter value.

A report will be eventually produced, carrying $A\_PARAMETER$ as a vulnerable parameter and that the vulnerable call for such parameter occurred at line 12.

4.7 Vulnerability demonstration

In order to demonstrate the actual existence of vulnerable paths and their feasibility at run time we took advantage of a formal string solver. The goal of this step is to transform the data collected in the previous analysis into a problem formulation that the formal analyzer is able to solve.

4.7.1 Graph building

As first step of the process, we are interested in producing an unified graph of the original control flow graph.

Such graph should not contain the statements that does not involve any of the statements that operates or include either the *base variable* or one of its aliases but should only be built upon the statements tacked during the previous analysis step. For convenience and simplification of the parsing of the graph, it should be constructed as an unique, joint representation of all the method bodies captured during the analysis.

In order to better understand what the outcome of this step should be, let's take the example in Section 4.6 and replace $dosomethingElse()$ method body (from line 11 to line 13).

```
11. public void doSomethingElse(String v) {
12.  if (v !== "a string") {
13.   doSomethingBad(v);
14.  }
13. }
```

Figure 4.1 shows the outcome of the parsing. As we can see that it contains all the interesting nodes contained in all the captured method bodies.

It has to be noticed that in order not to loose information about the context method the statements belong to, the graph's nodes data structure must be labeled with such information.

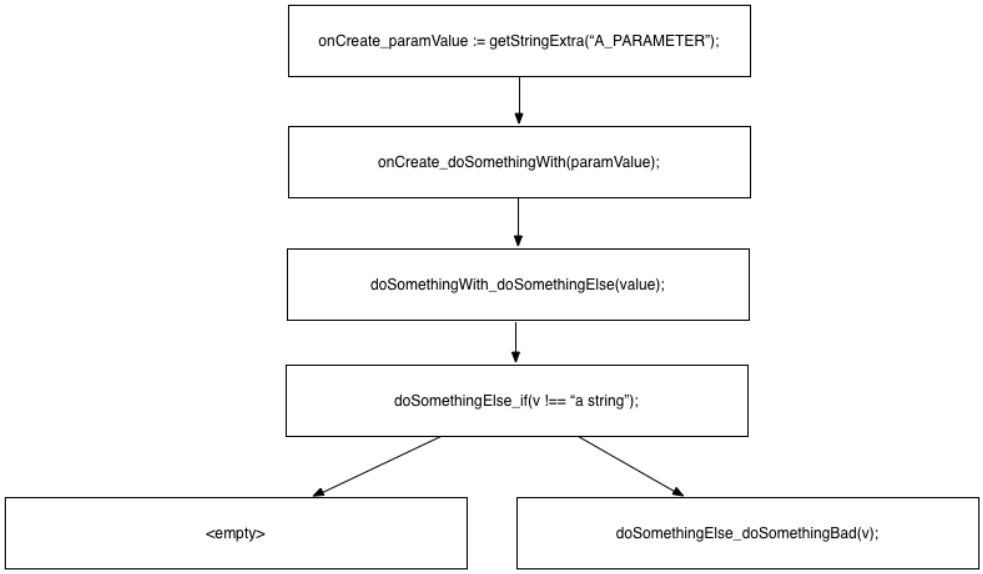

Figure 4.1: Demonstration - parsed graph

4.7.2 Formal problem formulation

The obtained graph is then traversed in order to produce the final problem formulation. For each conditional statement found in the graph we produce a string constraint, properly transforming the string expression (equality, inequality, inclusion). For each string operation, such as concatenation or replacement, a corresponding sequence of directives is produced.

The problem formulation is then used to query the string solver in order to obtain the solution for the string constraints. A solution for the constraints means that we have proven the vulnerability to be feasible at run time. If the solver is not able to find a solution, i.e. the given problem is unsatisfiable, the vulnerability is flagged as unfeasible and it is collected for further manual investigation.

Chapter 5

Implementation

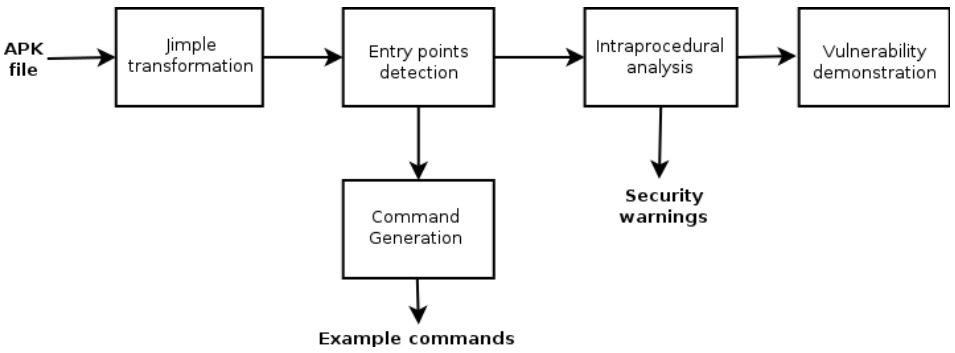

Figure 5.1: Analysis workflow

5.1 Overview

In this Chapter are described the tools and the techniques adopted to develop and implement the tool.

The tool is designed to be simple to use. It takes as input an Android application package (APK file) and passing through various analysis phases it produces as output security warnings, aimed to notify the user that a suspicious behavior has been encountered in the code. In addition the tool produces a set of example commands that can run on on the phone in order to effectively demonstrate the weakness in the code.

The code contained in the package, in form of Dalvik executable bytecode (dex), is decompiled and transformed into an higher level intermediate representation. This representation can be placed in between the bytecode and the actual Java since it is more human readable than the bytecode, but less complex and structured than the actual Java code.

This intermediate code, passes through a preliminary analysis aimed to detect the entry points, and collect informations used for further analysis. This step also includes a partial computation of the example commands.

The code is then sent to the last step of the analysis, where are extracted the sequence of operations that have a direct or transitive relation with the extracted entry points. Eventually, these sequences are checked against a list of potentially

dangerous operations. Every time a match between an element in the sequence and an element in the list is found, a report to the user is produced. For this analysis, the entry points are the values accepted as input by the activities and the list contains a set of methods concerning network request creation, database storage and UI presentation.

The tool is written in Java, and it makes use of both Soot[1] and Heros[2] frameworks to transform and analyze the code. The first is a well consolidated framework and the second was recently developed in order to complete some of the lacks of Soot. They are open source projects distributed under GNU LGPL.

The the vulnerability demonstration is performed with the help of Kaluza[3], a bounded-length string solver that generally used to solve real-world JavaScript constraints in web applications.

5.2 The Soot Framework

Soot was first presented between years 1999 and 2000 by members of Sable Research Group at McGill University as master thesis by Vallée-Rai[19] and with the more famous paper by the whole research group[20].

Soot is a Java static analyzer, that enables manipulation and optimization of the Java bytecode. The bytecode is transformed in a series of four intermediate representations designed for different objectives. These representations can be sorted from the ones closer to the bytecode to the furthest:

- **Baf** is a streamlined representation of the bytecode
- **Jimple** is a 3-addressed readable representation
- **Shimple** is a version of Jimple in *static single assignment form*
- **Grimp** is an aggregated version of Jimple

[1] http://www.sable.mcgill.ca/soot/
[2] http://sable.github.io/heros/
[3] http://webblaze.cs.berkeley.edu/2010/kaluza/

One of the main characteristic of Soot is its modularity. Soot is designed to be modular and highly adaptable to solve the set of problems it is aimed to. It relies on a singleton object, the Scene (accessible via $Scene.v()$), that is initially populated when the code is given as input, after being parsed. The Scene offers a large set of APIs to programmatically access classes and methods of the populated objects. A large set of tools is provided to analyze the Scene. Soot offers pre-implemented algorithms to perform call graph analysis, domination analysis or context-sensitive point-to point analysis. Soot also comes with a package manager that can be used to handle a chain of modules (as phases) which the code is eventually sent to. This allows to attach to the execution chain, external or custom transformations of analyzers.

Soot can run either as command line tool or as embedded Java library to satisfy needs that require more complex customizations.

5.2.1 Jimple Intermediate Representation

For the purposes of our analysis we adopted Jimple as intermediated representation. This choice was not driven by a special need, but we made it because of the wide support that this representation has obtained. Jimple has become the most popular intermediate representation and it can be considered as a kind of standard when performing static code analysis and optimization.

Here below is presented an example of method in its Jimple representation:

```
protected void onCreate()
{
    [...]
    $z1 == 0 goto label4;
    $r4 = $r0.<com.dropbox.client2.android.AuthActivity: java.lang.String consumerKey>;
    label4:
      virtualinvoke $r3.<android.content.Intent: android.content.Intent
        putExtra(java.lang.String,java.lang.String)>("CONSUMER_KEY", $r4);
    [...]
}
```

Jimple is a 3-addressed representation. It consists of a reduced set of statements (12), that limit the complexity of the analysis without reducing its expressibility in terms of readability. It also completely masks the underlining stack representation (as in the bytecode) by the use of local variables so not to cope with location of non-explicit variables in the stack. As can be noticed in the example above, it preserves the original Java types, complete with their namespace.

5.2.2 Soot with Dalvik Bytecode

It has recently been announced the integration in Soot of Dexpler[4] [3], adding to Soot the ability to transform the Dalvik bytecode into a Jimple representation and vice-versa.

This transformation may seem simple to perform, since Dalvik bytecode is register based, and there is a remarkable similarity between Dalvik registers and Jimple local variables. However, registers and constants in Dalvik are untyped, and their types has to be computed in order to complete the Jimple transformation (as seen before Jimple variables preserves the original Java types).

[4] Instrumenting Android Apps with Soot - http://www.bodden.de/2013/01/08/soot-android-instrumentation

Thank to this addition we were able to analyze the code without any further decompilation. This of course highly simplified the whole process. Unfortunately Dexpler has been pushed only to the development branch of Soot, so it is only included in the nightly build.

5.2.3 Heros Framework and IFDS

Unfortunately Soot does not provide a precise method for inter-procedural analysis. To overcome this lack, Eric Bodden, one of the Soot's current main contributors, has designed Heros[4].

Heros does not rely neither on Soot nor on any kind of Soot's intermediate representation to perform the analysis. It is also language-agnostic, in the sense that it can be used to analyze any language. It is written in Java and it accepts generics types representing statements, methods and data facts.

Heros is an IFDS/IDE general purpose solver. IFDS/IDE frameworks are class of algorithms to solve in polynomial time inter-procedural finite data-flow analysis.

IFDS is a general algorithm that allow to reduce a inter-procedural data-flow analysis into a graph reachability problem.[16]. It solves problems in which flow functions can be expressed as distributive functions. Many data-flow problems can be defined with distributive flow functions, and thus be solved with the IFDS/IDE framework such as defining truly-live variables, variables typestate and information-flow.

The IFDS algorithm extracts from the program's inter-procedural control-flow graph a, so called, "exploded super graph". In this graph, a node (s, d) is reachable from a selected start node $(s0, 0)$ if and only if the data-flow fact d holds at s, where s is a program statements. As "fact" is intended any logical statement, such as variable x has been declared, initialized or has passed through a series of defined computations. The algorithm makes use of data-flow functions to connect different nodes. As sketched in Figure 5.2 the function **id** is the identity function, mapping each data-flow fact before a statement into itself. The value **0** represents an empty

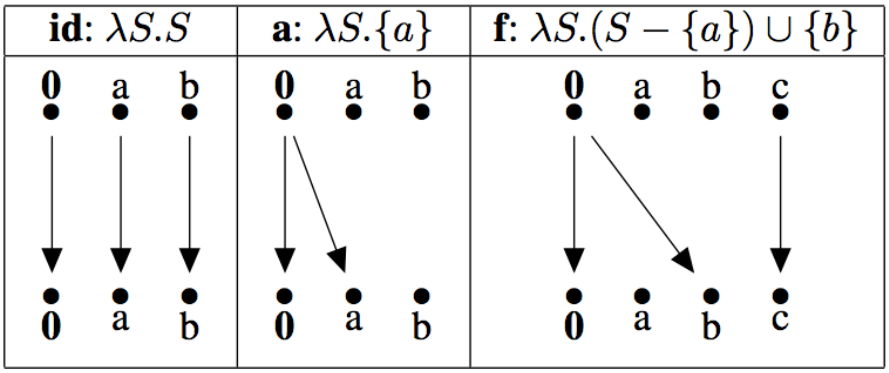

Figure 5.2: IFDS flow functions, reproduced from[16]

fact that is always valid. This **0** value is used to generate data-flow facts unconditionally. Function **a** adds a to the set of facts by connecting **0** to a, and at the same time removes b and other facts not explicitly connected. Function **f** adds b, removes a and leaves c untouched.

Heros solves arbitrary IFDS problems defined by implementing the `IFDSTabulationProblem<N,D,M>` interface, where `N` are node types (statements), `D` are data-fact types and `M` are method types. By implementing this interface the programmer has to define the chosen value for zero data-facts, provide a set of entry points for the analysis and define the behavior of the flow functions. Defining the flow functions means to implement four callback methods, one for each different inter-procedural behavior:

- `getNormalFlowFunction(N curr, N succ)`
- `getCallFlowFunction(N callStmt , M destinationMethod)`
- `getReturnFlowFunction(N callSite , M calleeMethod , N exitStmt , N returnSite)`
- `getCallToReturnFlowFunction(N callSite , N returnSite)`

For each of these functions (each of them representing a generic flow function) the programmer has to implement a inner callback method that will be called for

each data-fact present at a given computation moment: `Set<D> computeTargets(D source)`. In this way one or more facts can be generated specifically from a specific fact.

As can be noticed Heros is a precise and close implementation of the originally propose IFDS algorithm.

5.3 Entry Points Detection and Command Generation

IFDS algorithm has a worst-case complexity of $O(N^3)$ where N is the number of instructions in the analyzed domain. It is not reasonable to perform the analysis with data facts produced from any point in the program. To prevent this we need to collect a subset of methods (entry points) from which the analysis can begin.

To do so, a full scan of the Jimple representation of the program is performed. In this scan we are interested in detecting all the instructions that affect Intents extras, e.g. that both set or retrieve them. Since the set of API calls that Android provides to set and retrieve the extra parameters contained in the Intent message is quite limited and constant between different Android versions, the list of the signatures of this calls is explicitly declared. Whereas one these statements is interesting in the context of our search a data structure is populated wrapping together all the extras that are enclosed in the same Intent. To this data structure is also added other information that is not relevant to the real analysis but is instead necessary to generate the example commands, like class and package names.

In the application there may exists Intent messages targeted to other application's Activities (for example more than one application are included in the scope of the analysis), the tool also produces commands for Intent messages sent, but not received by the same application, as well as received, but never sent within the application. For this reason in this step is collected information both when a creation method callback is encountered ($onCreate()$) and when a request to start a new Activity is performed. Of course this causes duplicates that are removed afterwards.

As entry points for the real inter-procedural analysis, since we are tracking the programming behavior when receiving data, only the structures populated from receiving methods are considered.

5.4 Inter-procedural analysis

In this section is presented how the IFDS algorithm is exploited to perform the data flow analysis described in Section 4.4.

IFDS algorithm is well suitable to our analysis objectives, in fact this specific analysis is an instance of an information-flow problem easily describable as an IFDS Tabulation problem.

For our instance of the IFDS problem we use a simple data structure to represent data facts, similar to the one described in Section 4.4.3:

```
<Value trackedVariable,
  Value baseVariable,
SootMethod contextMethod,
List<Stmt> trackedStatements>
```

where `trackedVariable` is the variable to which the list of tracked statements corresponds, `baseVariable` is the reference to an aliased variable if any, `null` otherwise. `contextMethod` is the scope of validity of the variable and `trackedStatements` is the list of all the statements in which `trackedVariable` appears.

`Value` is a Soot interface that classes can implement to represent a variable, `SootMethod` is an utility class to access methods information and `Stmt` is an interface that represents general statements.

5.4.1 IFDS Problem

The problem is expressed as an instance of an Heros problem by implementing the interfaces described in Section 5.2.3. Below is described the logic encoded in the flow functions for each of the normal flow, call flow, return flow and call to return

flow callbacks.

As mentioned previously, as soon as the analysis starts is available the list of entry points complete with all the parameters treated in such entry points, namely *base parameters*.

- **Normal flow function** This callback is called when the statement does not include a method call or a return instruction.

  When this callback is called on behalf of a statement containing a base parameter reference, if the fact set is empty or it does not contain a fact yet having such base parameter as `trackedVariable` a new fact is added with the base parameter variable as `trackedVariable` and a new list of statements is initialized with the current statement.

  If the current statement is an assignment and one of the `trackedVariable` contained in the fact set is present in the right hand side of the assignment a new fact is generated. This fact will contain as `trackedVariable` the variable in the left hand side and the previous fact `trackedVariable` as `baseVariable`. Then the current statement is added to the new fact's statement list.

  Eventually, if a statement simply uses a variable contained in the `trackedVariable` field of a fact, such statement is added to the list, otherwise the previous facts are simply propagated.

- **Call flow function** This callback is called when a statements include a method invocation.

  If the fact set is empty is done something similar to what described for the normal flow callback, i.e. a new fact is generated if the current statement is an assignment that contains on its left hand side a variable containing a base value. Similarly, when the invocation contains as parameter a variable present in the `trackedVariable` field of a fact, if the returned value of the method is assigned to a new variable, a referencing fact is created containing the current statement in the statement list.

Also, whereas a variable in the fact list is passed a parameter, a new fact is generated with `trackedVariable` equal to the local variable, as `baseVariable` equal to the passed parameter and the statement list with the current statement added. In this way is explicitly created a parameter-argument binding for all the method invocations.

The fact list is killed in all the other cases so to avoid to analyze method not relevant in our analysis domain.

- **Return flow function**

  Since we are not propagating facts from the body to the method to the one enclosing it when the end of the body is reached, we simply kill all the paths passing by this callback.

- **Call to return function**

  We are not interested in propagating back facts binding the local returned variable to an external assignments, since we explicitly binding local and external scopes. This solutions is more robust since it allows to track internal operations performed on non primitive types (that are passed by reference). The flow function for this call back is simply an *identity function* that simply propagates all the facts valid so far.

This analysis will produce a set of all and only interesting variables in the context of the analysis, along with a list of all the statements that used them, or affected them for some reason.

5.4.2 Vulnerability check

At this point what remains to do is simply to query the IFDS facts for each captured $onCreate()$ method (entry points). The complete IFDS facts can be considered the ones corresponding to the last statement for each entry point method.

The list of facts can be used to reconstruct the variable dependencies (aliasing) in order to correlate all the statements to the variables enclosing the base parameter values and build a new list containing only base parameters in the `trackedVariable` field.

Then each of the entries in the statement lists is checked against a list of methods statements considered vulnerable. Every time a match is encountered a security warning is produced and outputted.

5.5 Vulnerability formal problem formulation

As stated in Section 4.7 in order to obtain a precise Kaluza formal problem formulation, two steps are required. First a inter-procedural unified graph, containing only *tracked statements* is extracted, then such graph is traversed in order to finally produce a Kaluza input string. The original control flow graph taken into consideration is not an exceptional control flow graph: for sake of simplicity exceptions are left out in this implementation. The algorithm presented in the following sections are executed independently *for each* captured vulnerable statement. A solution is considered acceptable is the solver can obtain a solution for all the vulnerable statements belonging to the same Intent payload.

5.6 Control flow graph extraction

Here below is presented a sketch of the algorithm used to obtain the desired graph. Where:

- $graph$ is the resulting graph variable
- $sink$ is the vulnerable statement
- $statementsSet$ is the list of tracked statements from the IFDS analysis

The algorithms works as follows: starting from the vulnerable statement point in the code the control flow graph is traversed backward (w.r.t. normal code execution order) jumping from callee to callers until the *entry point* variable is reached.

$extractMethod$ is the algorithm wrapper function taking care of managing inter-procedural calls. After adding the vulnerable statement the algorithms iterates over the callees until the newly generated graph has not been changed (convergence point).

```
graph;
sink;
statementsSet;
extractGraph() {
 context; // sink's method
 contextGraph;

 graph.addNode(sink);

 addPredecessors(contextGraph, sink, sink);

 repeat {
  headMethod = methodOf(lastUpdatedHead);
  caller = callerOf(headMethod);
  context = methodOf(callerStmt);
  contextGraph = extractGraph(context);

  if (caller in statementsSet) {
   addPredecessors(contextGraph, callerStmt, prevHead);
  }

 } until (graph.getHead() != lastUpdatedHead);
```

```
}

addPredecessors(contextGraph, currentStatement, graphHead) {
 for (predecessor, i : currentStatement.getPredecessors()) {
   if (!currentStatement in statementsSet) {
    addPredecessors(contextGraph, predecessor, graphHead);
    return;
   }
   if (current statemet branches) {
    if (i == 1) {
     addNodesAndEdge(graphHead, <Empty>);
    }

    addNodesAndEdge(predecessor, graphHead);

    if (i == 0) {
      addNodesAndEdge(graphHead, <Empty>);
    }
   } else {
     addNodesAndEdge(predecessor, graphHead);
   }

   addPredecessors(contextGraph, predecessor, newGraphHead);
 }
}
```

$addPredecessors$ iterates over statements inside a specific method body and it is in charge of reconstructing intra-procedural code points from the original method's

body to newly constructed graph. This method is invocated from the precise program point from which the previous method was invocated until the first statement in the method body.

The function first checks whether the current statement was tracked in the previous analysis step or not, i.e. such statement affects somehow the *base variable* (or one of its aliases) or not. If a match is encountered and the statements is not a merging point (the graph branches), the current statements is simply added to the graph and the next statement is taken in consideration. If the statement is a merging point, since the then-else branches are positional in the graph semantic, the right position for the next statement has to be preserved. This is obtained by checking the index of the current statement in the predecessors list (0 or 1).

It has to be noticed that:

- the first statement in a method body has no predecessors
- points in the code in which if-then-else blocks merges have two predecessors
- all other statements have only one predecessor

$addNodesAndEdges$ method (not presented here) is a simple utility method to add the edge between the two nodes in the right position.

Control flow graph parsing Eventually, in order to produce the desired Kaluza problem formulation, the previously constructed graph is parsed to obtain the set of string constraints and directives. The implementation works on a set of string operations such as concatenation, substring extraction, length equality, string equality. The subset of possible string operation chosen for the implementation was empirically determined by the code instruction found in the application taken in consideration for the proof of concept.

Kaluza has a wide set of pre-implemented string operations that are exploited to construct the problem formulation. For example:

- Concatenation of $S1$ and $S2$ ($S1$ and $S2$ strings) is $S3 := S1.S2;$

- Length equality of $S1$ and $S2$ ($S1$ and $S2$ strings) is $I1 := Len(S1); I2 := Len(S2); T1 := I1 == I2;$

- Equality/Inequality of $S1$ and $S2$ ($S1$ and $S2$ strings) is $T1 := S1 == S2;$ $T1 := S1! = S2;$

In order to avoid clashes, local variable names are transformed by prefixing to them their method name. For example variable v of method $doSomethingElse$ presented in the example 4.7.1 is transformed to the variable name $doSomethingElse_v$.

Chapter 6

Proof of Concept

6.1 Experimental Setup

In order to demonstrate the effectiveness of our method we choose a set of popular applications from the Google Play Store. We downloaded a sample consisting of 30 middle size applications. For middle size we mean applications with a limited, but significant, number of Activities (8-15) and Services (1-3). This choice was driven by the substantial execution time that the analysis takes. The IFDS solver takes between 2 and 9 minutes to complete the analysis of a single Activity. The variance in this time strongly depends on the number of methods present in the Activity, along with their complexity.

The execution time for the Kaluza problem formulation and solution is negligible w.r.t. IFDS solving time: the whole process always completes within a second.

6.2 Results

In Table 6.1 are present the number of paths the tool detected, divided into three sets. These three sets corresponds to the three set of vulnerabilities described in Section 3.4.

Below are listed some remarkable vulnerabilities which effectiveness have been demonstrated by manually creating an exploit that triggers them:

- **Mint** let an attacker load an arbitrary web page in the visual context of the application. Since this application deals with user's personal finance reports it is easy to imagine how an attacker could exploit this capability.

- **Poste Italiane/Postepay**: these two applications let an attacker communicate any kind of message to the application user through the applications' modal alert screen. An user may be deceived and induced to perform some dangerous action on its debit card account.

- **Airbnb** let open the modification of the house rules and the FAQ screen.

Table 6.1: EXPERIMENTAL ANALYSIS RESULTS

Application Name	Paths to Network	Paths to Database	Paths to UI
Airbnb	0	0	5
Airfrance	0	0	0
Blink	0	0	0
Booking	0	0	4
Craiglist	0	0	1
EF File Manager	0	0	0
Evernote	0	0	1
Expedia	0	0	2
Fancy	0	0	2
FriendCaster	0	0	0
GoChat	0	0	4
Hike	0	1	2
IM+	0	0	2
Imo	0	0	1
Mediolanum	0	0	0
Mint	1	0	1
OpenTable	0	0	1
Poste Italiane	0	0	5
Postepay	0	0	9
Readability	0	0	0
RocketTalk	1	0	8
Seesmic	0	0	1
Skype	0	0	0
Skyscanner	0	0	0
Snapchat	0	0	2
Swissquote	0	0	2
TripIt	0	0	0
Twitter	5	1	10
Wall Street Journal	0	0	0
Waze	0	0	0

- in **RocketTalk**, an attacker may specify a chat room (JSON resource) which URL could not belong to the RocketTalk default domain. This fact can be exploited by an attacker to actually manipulate user's chat history.

During the analysis was also found a noteworthy defense mechanism implemented by GOChat. This simple chat application exchanges verification codes for inter-Activity communication. This defense method exploits the Applications shared memory space (shared among all the Activities) to store a token generated from the Intent sender. The Intent receiver then matches the token received in the Intent payload with the one stored in the application shared memory. Every communication request without a valid token will then be discarded.

This simple approach was found to be very effective. In fact we were not able to manually perform any kind of operation on Activities protected by this mechanism. Of course this approach cannot be exploited when a given Activity or Service has to be world accessible by design.

6.3 Paths Distribution

As can be noticed in Figure 6.1 the number of UI paths detected heavily dominates the number of paths in the other two classes. Intuitively we can see the cardinality of these three sets decreasing with the increasing complexity of the threat model.

As can be intuitively deduced from an high level reasoning about the use cases and the underlining architectures that let these classes of vulnerability raise, network and database paths are less likely to appear.

To support this thought, we can also rank the three vulnerability classes in order of severity: network vulnerabilities are the most severe since they can impact remote user resources (maybe locally synchronized on many user devices). Then database set that affect the local status of the application and eventually the phishing set which (in theory) should only affect user's current session.

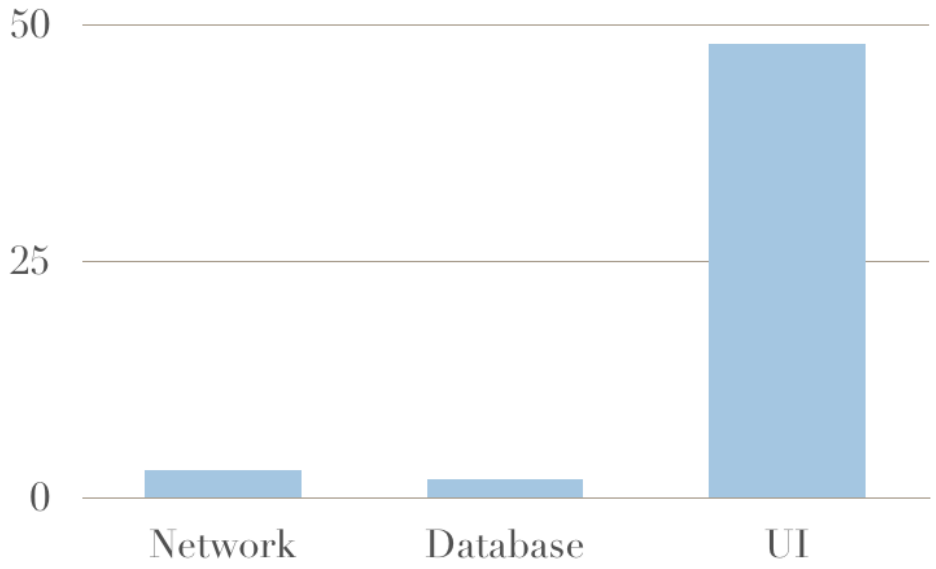

Figure 6.1: Path distribution histogram

Chapter 7

Conclusions and Future Work

To conclude, we identified a major source of vulnerabilities for Android applications, namely the lack of filtering and controls in *Intent* based inter-process communications. We explored the potential impacts, and identified the threat model.

Then, we proposed a method to automatically detect them in apps. With static analysis, we automatically identified data flows that could lead to (parts of) Intent payloads being used as (part of) arguments for framework method calls. We formulated the problem efficiently as an Interprocedural Distributive Environment one, and analyzed the flows to check for appropriate sanitization measures.

Where such controls seemed absent, we used a string solver to automatically generate Intent examples that trigger the detected behavior.

We tested our approach on 30 popular applications from the Google Play, finding 19 potential vulnerabilities and automatically exploiting 13 of these.

From our results it is clear that only a few applications implement appropriate security countermeasures for Intent communications.

The tool could be confidently used (along with manual validation) in real world development environments to test application's message passing implementation against the three class of vulnerabilities discussed in this thesis.

Unfortunately, since the tool is not able to cope with application semantic (common trough this class of approaches), it suffers of false positives. For example, just by looking at the paths it can not be determined if a phishing attach can be considered effective: we can not distinguish between slight modification in the UI (button label changing) and considerable modification that can actually deceive an user.

The main lack of this tool is a complete dynamic test suite that can demonstrate the severity of found vulnerabilities. This problem is, unfortunately, hard, because the analysis includes a wide set of Android system features, namely interprocess communication, network communication, databases and UI elements.

The only effective way to automatically prove the severity of a given vulnerability is to test it into the application execution context. This requires the ability to instrument the applications at run time, by artificially triggering events that lead

the application in the vulnerable state. To have a solid feedback is also needed to monitor the effective changes that occur in the resources, and to bind them to the event that provoked them.

Android debugging tools may not be enough for this purposes due to their inability to explicitly track cause-effect events. In addition, debugging tools does not provide mechanisms to analyze encrypted network traffic or encrypted data storage. Of course a exhaustive testing can be performed only by having clear access to all the resources of the system.

A solution could be to create a modified Android Open Source Project build, where informations are captured before being encrypted and (in case of the network), before leaving the system, and responses are captured after entering the system and after being decrypted. This, moreover, requires a deep knowledge of the Android system, because this approach requires to instrument all the system libraries such as the Activity Manager, the Apache HTTP Library included in Android, the SQLite database drivers and the component rendering library.

Bibliography

[1] Android: Security tips. `https://developer.android.com/training/articles/security-tips.html`, February 2013.

[2] David Barrera, H. Güneş Kayacik, Paul C. van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 73–84, New York, NY, USA, 2010. ACM.

[3] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, SOAP '12, pages 27–38, New York, NY, USA, 2012. ACM.

[4] Eric Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, SOAP '12, pages 3–8, New York, NY, USA, 2012. ACM.

[5] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011.

[6] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2011.

[7] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *20th USENIX Security Symposium*, San Francisco, CA, August 2011.

[8] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: user attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1–3:14, New York, NY, USA, 2012. ACM.

[9] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium*, volume 18, pages 19–31, 2011.

[10] Patrick Gage Kelley, Sunny Consolvo, Lorrie Faith Cranor, Jaeyeon Jung, Norman Sadeh, and David Wetherall. A conundrum of permissions: Installing applications on an android smartphone. In *Financial Cryptography and Data Security*, pages 68–79. Springer, 2012.

[11] Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, and Angelos Stavrou. A whitebox approach for automated security testing of android applications on the cloud. In *Automation of Software Test (AST), 2012 7th International Workshop on*, pages 22–28. IEEE, 2012.

[12] Nariman Mirzaei, Sam Malek, Corina S Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[13] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication

mapping in android: An essential step towards holistic security analysis, August 2013.

[14] Sarah Perez. Idc: Android market share reached 75% worldwide in q3 2012. Last accessed: March 2013.
"TechCrunch.com, November 2, 2012". http://techcrunch.com/2012/11/02/idc-android-market-share-reached-75-worldwide-in-q3-2012/.

[15] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. 2013.

[16] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[17] A-D Schmidt, Rainer Bye, H-G Schmidt, Jan Clausen, Osman Kiraz, Kamer A Yuksel, Seyit A Camtepe, and Sahin Albayrak. Static analysis of executables for collaborative malware detection on android. In *Communications, 2009. ICC'09. IEEE International Conference on*, pages 1–5. IEEE, 2009.

[18] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. "andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.

[19] Raja Vallée-Rai. Soot - a java bytecode optimization framework. Master's thesis, School of Computer Science, McGill University, Monreal, 2000.

[20] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pages 13–. IBM Press, 1999.

Appendix A

IFDS implementation

```
1
2  public class IFDSFollowFlow
3      extends DefaultJimpleIFDSTabulationProblem<IFDSFact, InterproceduralCFG<Unit, SootMethod>> {
4
5    private final Set<Unit> initialSeeds;
6    private final Set<Value> baseValues;
7    private final SootMethod baseContext;
8
9    public IFDSFollowFlow(InterproceduralCFG<Unit, SootMethod> icfg, AbstractTrappedAction receiver) {
10     super(icfg);
11     Set<Unit> seeds = new HashSet<Unit>();
12     baseValues = new HashSet<Value>();
13     baseContext = Scene.v().getMethod(receiver.getMethodSignature());
14     seeds.add(baseContext.getActiveBody().getUnits().getFirst());
15     baseValues.addAll(receiver.getAllValues());
16     this.initialSeeds = seeds;
17   }
18
19   private boolean valueInUseBoxes(Value val, Stmt statement) {
20     for (ValueBox dBox : statement.getUseBoxes()) {
21       if (dBox.getValue().equivTo(val)
```

```
22            && interproceduralCFG().getMethodOf(statement).equals(baseContext)) return true;
23      }
24      return false;
25    }
26
27    @Override
28    protected FlowFunctions<Unit, IFDSFact, SootMethod> createFlowFunctionsFactory() {
29      return new FlowFunctions<Unit, IFDSFact, SootMethod>() {
30
31        @Override
32        public FlowFunction<IFDSFact> getNormalFlowFunction(Unit curr, Unit succ) {
33          if (interproceduralCFG().isStartPoint(curr)
34              && interproceduralCFG().getMethodOf(curr).equals(baseContext)) {
35            return new FlowFunction<IFDSFact>() {
36
37              @Override
38              public Set<IFDSFact> computeTargets(IFDSFact source) {
39                if (source == zeroValue()) {
40                  HashSet<IFDSFact> res = new HashSet<IFDSFact>();
41                  for (Value v : baseValues) {
42                    IFDSFact fact = new IFDSFact(v, baseContext);
43                    res.add(fact);
44                  }
45                  return res;
46                } else {
47                  return Collections.emptySet();
48                }
49              }
50            };
51          }
52
53          if (curr instanceof DefinitionStmt) {
54            final DefinitionStmt assignment = (DefinitionStmt) curr;
```

```
55          return new FlowFunction<IFDSFact>() {
56            @Override
57            public Set<IFDSFact> computeTargets(IFDSFact source) {
58              // facts have already been initialized
59              if (source != zeroValue()) {
60                // source base value is assigned
61                if (source.getValue().equals(assignment.getRightOp())
62                    && interproceduralCFG().getMethodOf(assignment).equals(
63                        source.getContextMethod())) {
64
65                  source.addStatement(assignment);
66                  source.addAlias(assignment.getLeftOp(),
67                      interproceduralCFG().getMethodOf(assignment));
68                  return Collections.singleton(source);
69                }
70                // source alias is assigned
71                if (source.hasAlias(assignment.getRightOp(),
72                    interproceduralCFG().getMethodOf(assignment))) {
73                  source.addAlias(assignment.getLeftOp(),
74                      interproceduralCFG().getMethodOf(assignment));
75                  source.addStatement(assignment);
76                  return Collections.singleton(source);
77                }
78
79              }
80              return Collections.singleton(source);
81            }
82          };
83        }
84
85        return Identity.v();
86
87      }
```

```
88
89      @Override
90      public FlowFunction<IFDSFact> getCallFlowFunction(final Unit callStmt,
91          final SootMethod destMethod) {
92        if (interproceduralCFG().isStartPoint(callStmt)
93            && interproceduralCFG().getMethodOf(callStmt).equals(baseContext)) {
94          return new FlowFunction<IFDSFact>() {
95
96            @Override
97            public Set<IFDSFact> computeTargets(IFDSFact source) {
98              if (source == zeroValue()) {
99                HashSet<IFDSFact> res = new HashSet<IFDSFact>();
100               Set<Stmt> statements = new HashSet<Stmt>();
101               for (Value v : baseValues) {
102                 IFDSFact fact = new IFDSFact(v, baseContext);
103                 res.add(fact);
104               }
105               return res;
106             } else {
107               return Collections.emptySet();
108             }
109           }
110         };
111       }
112       final Stmt stmt = (Stmt) callStmt;
113       final InvokeExpr invokeExpr = stmt.getInvokeExpr();
114       final List<Value> args = invokeExpr.getArgs();
115       final List<Value> localArguments = new ArrayList<Value>(args.size());
116       for (Value value : args) {
117         if (value instanceof Value)
118           localArguments.add(value);
119         else
120           localArguments.add(null);
```

```
121         }
122
123         return new FlowFunction<IFDSFact>() {
124
125           @Override
126           public Set<IFDSFact> computeTargets(IFDSFact source) {
127             if (source == zeroValue()) return Collections.singleton(source);
128             if (localArguments.contains(source.getValue())) {
129
130               int paramIndex = args.indexOf(source.getValue());
131               source
132                   .addAlias(
133                       new EquivalentValue(Jimple.v().newParameterRef(
134                           destMethod.getParameterType(paramIndex), paramIndex)),
135                       invokeExpr.getMethod());
136               source.addStatement(stmt);
137               source
138                   .addAlias(
139                       new EquivalentValue(Jimple.v().newParameterRef(
140                           destMethod.getParameterType(paramIndex), paramIndex)),
141                       invokeExpr.getMethod());
142               if (callStmt instanceof DefinitionStmt) {
143                 DefinitionStmt defStmt = (DefinitionStmt) callStmt;
144                 if (defStmt.getLeftOp().getType().toString().equals("java.lang.String")) {
145                   source.addAlias(defStmt.getLeftOp(), baseContext);
146                 }
147               }
148               return Collections.singleton(source);
149             }
150             SootMethod contextMethod = interproceduralCFG().getMethodOf(callStmt);
151             Value aliasedArg = getValueFromFact(source, contextMethod, localArguments);
152             if (aliasedArg != null) {
153               source.addAlias(aliasedArg, invokeExpr.getMethod());
```

```
154                 source.addStatement(stmt);
155                 if (callStmt instanceof DefinitionStmt) {
156                   DefinitionStmt defStmt = (DefinitionStmt) callStmt;
157                   if (defStmt.getLeftOp().getType().toString().equals("java.lang.String")) {
158                     source.addAlias(defStmt.getLeftOp(), contextMethod);
159                   }
160                 }
161                 return Collections.singleton(source);
162               }
163               // do not propagate trough non-interesting functions
164               return Collections.emptySet();
165             }
166           };
167         }
168
169         @Override
170         public FlowFunction<IFDSFact> getCallToReturnFlowFunction(Unit arg0, Unit arg1) {
171           return Identity.v();
172         }
173
174
175         @Override
176         public FlowFunction<IFDSFact> getReturnFlowFunction(Unit arg0, SootMethod arg1, Unit arg2,
177             Unit arg3) {
178           // no need to propagate local fact
179           return KillAll.v();
180         }
181       };
182     }
183
184     @Override
185     public Set<Unit> initialSeeds() {
186       return initialSeeds;
```

```
187   }
188
189   @Override
190   protected IFDSFact createZeroValue() {
191     return new IFDSFact(new JimpleLocal("<<zero>>", NullType.v()), baseContext);
192   }
193
194 }
```